Model Based Testing of Large-scale Software: How Can Simple Models Help to Test Complex System*

Victor V. Kuliamin

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

Abstract. The article is concerned with an approach to model based test development for large software systems. The approach presented is a part of UniTesK test development technology, which is developed on the background of 10-year experience of ISP RAS in verification and test development for complex industrial software [1]. The article states that the well-known software engineering principles underlying the approach and aimed at coping with complexity makes possible its application in software projects of real-life size and complexity.

1 Introduction

Revolutionary changes in software engineering during last decades have already become a commonplace. We see that modern methods, technologies, and tools make development of software more complex than can be even imagined 20 years ago an everyday practice. Functionality and complexity of modern systems grows exponentially. But this evolution has one big problem – quality control techniques lag behind the development ones and cannot provide the same level of quality for complex systems with manageable growth of effort.

What can help software development community in this situation? One promising candidate is *model based testing*. This approach has serious advantages in comparison with traditional testing techniques.

- Models and further tests can be prepared on the base of requirements and design decisions before the other parts of software are ready. Thus, the total time of production cycle can be decreased.
- Model based testing can help to find unimplemented features and requirements, which cannot be detected by implementation-based testing.
- Model based testing helps to raise the abstraction level of systematic testing and to evaluate quickly the correctness of large subsystems or system as a whole. Implementation-based testing can get this result only after the long

* 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.

way through the smallest units and larger components to integration and system testing. The analogous results of ad-hoc functional testing are much less reliable due to its unsystematic character.
– Model based testing has a potential to make manageable testing of extremely complex systems, which cannot be adequately tested by implementation-based techniques with an acceptable effort.

Unfortunately, most model based testing case studies deal with small and simple systems. Such a situation is caused partially by current organizational patterns of industrial development, but it is also related with the background formalisms of most part of model based testing techniques. They are based usually on automata models of various kinds (FSMs, LTSs, EFSMs, etc.). Automata models of real life software with complex functionality often have large numbers of states and transitions making such model based testing methods unmanageable (this is so called 'state explosion problem'). So, the most important statement that model based testing helps to cope with increasing complexity still needs to be corroborated.

It seems that the mankind has no other tools to cope with complexity than the well-known software engineering principles: abstraction, modularization, and separation of concerns. So, the search for scalable model based testing techniques should pay attention on the following two issues.

– The models used by appropriates techniques should admit smooth integration with this principles.
– The well-scalable structure of the test system should become one of the main concerns of the technique. There is no way to test an actually complex system both in an adequate and simple way. But maybe we can find an organizational approach that makes much more easy to manage the complex work we should do in the case. Thus, simple models can help to test complex systems only if we know how to organize them in complex structures in manageable way.

This article provides a possible approach to manageable test construction for complex software based on the principles stated. The approach presented is a part of UniTesK test development technology designed in ISP RAS on the base on 10 year experience in test development for complex industrial software. It uses contract specifications that provide componentwise description of the functionality of the system under test and more abstract automata models that are used to create tests for different aspects and different parts of the system.

The next section of the article deals with main issues of the approach. The third section presents the results of a case study. The fourth one gives a brief overview of related works and compares them with our approach. The conclusion summarizes the main results of the article.

2 The Approach

The approach presented here was developed for model based testing of large systems in the context of UniTesK test development technology. It is used to

develop functional tests checking conformance between the system and its model and consists on the following main steps.

- **Step 1. Interface definition.** We define clearly the interface under test and the abstraction level of the future testing. After this step we should have a list of operations and events in the system under test, which should be tested or tracked by the tests, and an understanding what should be checked in tests and what should not, because it is considered as implementation specifics.
- **Step 2. Model development.** After definition of the interface we should provide a strict definition of its properties in the form of *software contracts* – preconditions and postconditions of operations and events, data integrity constraints for all observable data types (parameters and results of operations and events). To write meaningful preconditions and postconditions we may need the correspondingly abstracted *model state* also represented as data with some integrity constraints. The model state and software contracts provide *a behavior model* of the system under test on the abstraction level chosen.
- **Step 3. Adapter development.** Thus we have the model and the modeled system and wish to test their conformance. We need first to provide some links between their components. Usually this task is solved by means of *adapters.* In UniTesK they are called *mediators* for historical reasons. Mediator of an operation implements some model operation by means of transformation of its calls into corresponding calls of the operations of the system under test. Mediator of an event, on the contrary, waits for the corresponding event in the system under test and transforms it to the model representation.
- **Step 4. Test scenario development.** The description of the test is represented as *a test scenario.* While the main goal of behavior model is to describe what the software under test is to do, the goal of scenarios is to provide data for test input and test sequence generation. Each scenario has its specific objective usually represented as a coverage level to be achieved according to some *coverage criterion.* Coverage criteria are defined in terms of the behavior model and can be considered as sets of specific situations called *coverage goals.* A coverage goal corresponds to a situation where some predicate on the model state and parameters of the model operations called becomes true. Coverage goal can be defined explicitly by such a predicate or implicitly by pointing out the corresponding location in the code of operation's postcondition. In the second case the corresponding predicate is extracted automatically.

  Test scenario represents another model of the system's component under consideration. This model is a *finite state machine* (FSM) or *input/output finite state machine* (IOFSM) having a set of states S and a set of transitions T. Each transition corresponds to a sequence of calls of model operations (let us denote all possible such calls as $Calls$) and occurrences of model events. Test scenario describes the generic state machine in implicit form, namely, instead of explicit description of all SM's states and transitions (each having

starting state and end state), test scenario consists of the following data (in the following *Obs* denotes all the observable data of both the behavior model and the system under test).

- The data type of possible SM states S.
- The function mapping the observable data into a state $st : Obs \to S$. Only those states, which are actually reached during the test execution (that is, returned by this function on some testing step) are considered.
- The function returning admissible calls of model operations in the current state $act : S \times \mathbb{N} \to Calls^*, (s, i) \mapsto (c_1, ...c_n)$. The sequence of calls $(c_1, ...c_n) \in Calls^*$ should be executed when the test comes to the state s on i-th time. We require that for each $s \in S$ there exists some $n \in \mathbb{N}$ that $act(s, n)$ is empty. Such empty sequence means that the test has already performed all the necessary operations in this state during the previous visits to it.

States and operation calls in particular scenario are defined in such a way that the test objective stated can be achieved by performing a transition tour on the state machine described.

The approach presented consistently applies three fundamental principles of software engineering – modularization, abstraction, and separation of concerns, to test system construction.

– **Modularization.**
  - The system under test is considered as a number of interacting components, which should be modeled separately. The partitioning of the system into components is performed on several levels of abstraction and on the highest one it can be represented as the single component. Such an approach makes possible to achieve high quality testing of components of one level of abstraction on the base of hypotheses on reliability of lower-level components tested previously.
    Such an approach requires from the modeling technique used to be well modularizeable and useful on large range of abstraction levels. *Contract specifications* seem to fit both these requirements more than any other modeling formalism. After testing the conformance of a component to its formally specified contract we can rely on this component in construction of higher-level ones.
  - Each component is tested separately from the others on the same abstraction level, but only after testing of lower-level subcomponents. For the convenience of test developers operations rather close conceptually, but without dependence on states can be composed in *stateless components* and tested together. An example of such a component is group of operations implementing primitive mathematical functions.
  - The tests are also modularized. Each test consists of
    * *A test engine* implementing particular algorithm of test sequence construction for general scenario
    * A test scenario, which provides data for test sequence construction

  - A number of *iterators* providing data for test input generation for separate operations under test
  - *An oracle* for each operation under test and each event tracked in this test. Oracle is generated from behavior model and checks the conformance between the actual behavior of the operation or event and its contract.
  - Mediators for each operation under test and each event tracked.

– **Abstraction.**

- Functionality of operations under test is described in the form of software contracts on the desired abstraction level, which hide implementation specifics and make more clear essential properties of the software under test. This makes possible construction of reusable tests, which can be applied to any system equivalent to the original one on the chosen abstraction level and to other versions of the system under test in particular.
- Software contracts are rather flexible modeling technique capable to capture the properties required on different levels of abstraction, starting from the particular algorithm used.

  Let us consider the component implementing the functionality of abstract map mapping integers to objects (in reality it can have more specific form, like indexed records in database or pool of objects used by Web-server). The main operations of such a map are addition of a key-object pair `bool Add(int key, object value)` (returns true iff the pair is added, false iff the key is already used in the map), check whether a key value is already used `bool ContainsKey(int key)`, search of an object by its key `object Get(int key)` (can be called only for used keys), and removal of a key along with the corresponding object `void Remove(int key)` (can be called only for used keys). Fig. 1 in Appendix demonstrates the detailed contract of this component in the extension of C# language used by one of UniTesK tools, Ch@se.

  We can either add more details to that specification by refining the nature of objects stored in map, or skip some details, for example, do not check *frame conditions* stating that the parts of state not touched by the operation are preserved. Or we can abstract from stored objects at all reducing a map described to the set of its keys.
- Test scenarios provide more abstract view on the corresponding aspect of the system functionality. Namely, they abstract from all the details that are not needed to construct a test sequence achieving the coverage desired.

  Test scenario can abstract the component state in different ways according to its goals. For example, Fig. 2 in Appendix demonstrates the test scenario for testing the map described above. It defines the state machine with states corresponding to different numbers of key-value pairs in the map under test. The scenario shown on Fig. 3 defines the state as the set of integers used as keys, so it provides much more detailed testing of

the component. Note that we need only to override `State` property of test scenario to make this refinement.

– **Separation of Concerns.** Separation of concerns principle is implemented both between the components that user has to develop to construct fully functional test and between the components of the resulting tests.
  - Contract specifications provide strict description of functional requirements to corresponding operations and events of the system under test. They give means to check the conformance between this requirements and the system's actual behavior.
  - Coverage criteria provide description of test adequacy criteria, which are used to measure the adequacy of resulting tests and to aim test scenarios at achieving high levels of testing quality.
  - Mediators provide binding between model and the system under test during test execution. Mediator transforms calls of model operations into corresponding calls of implementation operations and transforms the result of implementation call or event on model level.
  - Iterators provide data for generation of operations' parameters values during test execution.
  - Test scenario's concern is to provide data for test sequence generation during test execution.
  - Test engine's concern is to construct a needed path on the FSM implicitly described by the test scenario used in test construction. To solve this task it can use only the data on the already executed transitions and visited states, so UniTesK test engines are based on algorithms for unknown graph exploration.

This combination of fundamental software engineering principles and special attention on modularization aspects make the approach useful for manageable test development for large-scale software. This statement can be confirmed by successful use of UniTesK in testing of large and complex systems.

3 The Case Study

The origin of UniTesK technology is related with telecommunication software verification project conducted by ISP RAS for Nortel Networks in 1994-1996 [2]. The project goal was to develop a regression test suite for kernel of the operating system of telecommunication switch. The total size of the system under test was about 250 K lines of code. The interface under test consisted of more than 500 operations and included such functionality as arithmetic operations with basic and extended number types, time data conversion, messaging management, process management, synchronization management, process resources management, and so on.

The resulting test suite partitioned the interface under test into 44 components (in this case it was groups of operations dependent from each other), from which 29 components were statefull. Total size of the specification written is

about 60 K lines of code. Total size of code of mediators developed is about 50 K lines. 372 test scenarios were developed, 304 for state-independent operations and 68 for statefull components. Their total size is about 80 K lines of code. The total effort of the project was about 10 man-year (the total effort of development of the system under test is about ten times more).

The test suite developed was used for regression testing of the OS kernel till 2000. It was useful not only the regression test suite, since more than two hundred errors were found in the field version of the OS with its help and several dozens of them were showstoppers – they required cold restart of the system to restore it in the operating state.

4 Related Works

Author knows only several works concerned with model based testing of large software. The bibliography analysis shows that the most part of model based testing society tends to work with toy and small-size examples. This is really unfortunate for propagation of model based approach to testing in the industry, since most significant problems of traditional testing techniques are related with large-scale and complex systems. One cause of such a situation is V-model stereotype for software development, which seems to prescribe test development in the inverse order to the main development activity – unit tests first, then tests for larger units and components, then integration tests, and only them system tests. The author thinks that complexity and gigantic size of modern systems urgently requires new test development techniques that can be started on high or intermediate design level.

The first group of works related with the large-scale model based testing is concerned with architecture-based testing. The works of A. Bertolino, P. Inverardi and H. Muccini [3–6] and related article of D. Richardson and A. Wolf [7] use informal architecture-level description of software under test to construct LTS models for different aspects of the software and then derive tests from them. Their approach mostly focused on integration testing and checking the interaction of large components of the system under test. UniTesK approach presented in this article can be used both for unit testing and for integration testing. One more difference of the two approaches lies in the models used – UniTesK uses contract specifications for low-level descriptions and FSMs (along with IOFSMs) for higher-level modeling, and the authors cited use LTSs for both detailed and abstracted descriptions.

The other group of works on testing complex software systems known to the author is related with *Côte de Resyste* project [8–11]. The participants of this project developed the techniques and tools (TorX and TGV) capable to construct model based tests from the LTS models and user defined test purposes. The tools were successfully used in testing real-life software applications. The most prominent is using TorX in testing the Oosterschelde Storm Surge Barrier control software, which code size is about 80 K lines. The approach used in *Côte de Resyte* tools is close to the one presented in this paper. Main differences

are related with use of contract specifications and coverage criteria based on them in UniTesK and use of only LTS models in *Côte de Resyste.* The works of J. Tretmans are mostly focus on formal definition of tested conformance relation between model and implementation. The relation used in UniTesK is more complex, since it includes contract specifications. Some details on this topic can be found in [12].

5 Conclusion

The UniTesK approach for model based testing was designed on the base of a well-known set of principles that are aimed at coping with complexity. This helps to use it for large-scale software testing with effort comparable with traditional test development, but with much higher quality of resulting software. It also includes techniques for modeling concurrency and timing properties of tested software in an environment familiar for industrial developers [13]. In addition UniTesk tools are based on extensions of widely used programming languages. That fact facilitates introduction of the approach into industrial processes.

The results of case studies [14] of using the approach to test industrial software of various domains show its applicability for a large variety of contexts. Now its case study database includes, besides kernel verification project presented above, test development for several different IPv6 implementations, banking client management software system based on J2EE and Web technologies, messaging systems, device drivers, etc.

Of course, studies on the approach applicability to various project types and comparative evaluation of effort needed to develop tests using different techniques should be continued. But the data we already have demonstrate that model based testing can provide actual help in development of modern software systems having complex functionality and intractable by means of traditional testing methods.

References

1. http://www.ispras.ru/groups/rv/rv.html
2. 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.
3. A. Bertolino, P. Inverardi. Architecture-based Software Testing. In joint ACM proceedings of the 2-nd International Software Architecture workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96) on SIGSOFT'96 workshops, pp. 62–64, October 1996.
4. A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An Approach to Integration Testing Based on Architectural Descriptions. In Proc. Third IEEE International Conference on Engineering of Complex Computer Systems (ICECCS97), pp. 77–84, Como, Italy, September 1997.
5. A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving Test Plans from Architectural Descriptions. In Proc. 22-nd International Conference on Software Engineering (ICSE'2000) , pp. 220–229, Limerick, Ireland, June 2000.

6. H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. IEEE Transactions on Software Engineering, Vol. 30, No. 3, pp. 160–171, March 2004.
7. D. J. Richardson, A. L. Wolf. Software testing at the architectural level. In joint proceedings of the 2-nd International Software Architecture workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96) on SIGSOFT'96 workshops, pp. 68–71, October 1996.
8. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. Software – Concepts and Tools, 17(3):103–120, 1996. Also: Technical Report No. 96-26, Centre for Telematics and Information Technology, University of Twente, The Netherlands.
9. J. Tretmans. Testing Concurrent Systems: A Formal Approach. In J. C. M. Baeten and S. Mauw, editors, CONCUR'99 – 10-th Int. Conference on Concurrency Theory, volume 1664 of Lecture Notes in Computer Science, pp. 46–65. Springer-Verlag, 1999.
10. A. Belinfante. Timed Testing with TorX: The Oosterschelde Storm Surge Barrier. In M. Gijsen, editor, Handout 8^e Nederlandse Testdag, Rotterdam, The Netherlands, November, 2002, CMG.
11. J. Tretmans, E. Brinksma. TorX: Automated Model Based Testing. Côte de Resyste. In Proc. of ECMDSE'2003, Nuremberg, Germany, December 2003.
12. V. Kuliamin. Multi-paradigm Models as Source for Automated Test Construction, In Proc. of Model Based Testing workshop on ETAPS'2004, Barcelona, Spain, March 2004.
13. 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.
14. http://www.unitesk.com

6 Appendix

Appendix contains the example of specifications of a component implementing map mapping integers to objects. Specification is written in the extension of C# used by one of UniTesK tools. Two examples of test scenarios for the same component are also presented here.

```
namespace Chase.Examples {
  specification public class IntToObjectMapSpecification {
    public Hashtable items = new Hashtable();

    invariant I( "Keys are integers" ) {
      foreach(object o in items.Keys)
       if(!(o is int)) return false;
      return true;
    }

    specification public bool Add( int key, object value )
      reads key, value
      updates items
    {
      post {
        if(!items.Keys.Contains(key)) {
          branch UnusedKey( "The key is not used" );
          Hashtable old = (Hastable)(items.Clone());
          old.Remove(key);
          return $this.Result == true && old.Equals(pre items.Clone());
        } else {
          branch UsedKey( "The key is used" );
          return $this.Result == false && items.Equals(pre items.Clone());
        }
      }
    }

    specification public bool ContainsKey( int key )
      reads key, items
    {
      post {
        branch Single( "Single branch" );
        return   $this.Result == items.Keys.Contains(key)
               && items.Equals(pre items.Clone());
      }
    }

    specification public object Get( int key )
      reads key, items
      updates items
    {
      pre { return items.Keys.Contains(key); }
      post {
        branch Single( "Single branch" );
        return   $this.Result == items[key]
               && items.Equals(pre items.Clone());
      }
    }

    specification public void Remove( int key )
      reads key
      updates items
    {
      pre { return items.Keys.Contains(key); }
      post {
        branch Single( "Single branch" );
        Hashtable old = (Hastable)(pre items.Clone());
        old.Remove(key);
        return old.Equals(items);
      }
    }
  }
}
```

Fig. 1. Example of specifications in C# extension

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

    IntToObjectMapSpecification target = new IntToObjectMapSpecification();
    int maxNumber = 10;

    protected virtual void configureMediators() {
      target = mediator IntToObjectMapMediator( new IntToObjectMap() );
      target.AttachOracle();
    }

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

    public override Chase.Lang.ModelObject State {
      get { return new Chase.States.IntState( target.items.Count ); }
    }

    scenario Add() {
      if( target.items.Count < maxNumber )
        iterate( int i = 0; i < maxNumber; i++; )
          target.Add(i, new object());
      return true;
    }

    scenario Contains() {
      iterate( int i = 0; i < maxNumber; i++; ) target.ContainsKey(i);
      return true;
    }

    scenario Get() {
      iterate( int i = 0; i < maxNumber; i++; )
        if(target$Get.pre(i)) target.Get(i);
      return true;
    }

    scenario Remove() {
      iterate(int i = 0; i < maxNumber; i++; )
        if(target$Remove.pre(i)) target.Remove(i);
      return true;
    }
  }
}
```

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

```
namespace Chase.Examples {
  scenario public class DetailedMapTestScenario : MapTestScenario
  {
    public static void Main() {
      Tracer.Init();
      MapTestScenario test = new DetailedMapTestScenario( 5 );
      test.Run();
      Tracer.Finish();
    }

    public DetailedMapTestScenario( int maxNumber ) {
      base(maxNumber);
    }

    public override Chase.Lang.ModelObject State {
      get {
        IntSetState state = new IntSetState();
        foreach(object o in target.items.Keys)
          state.Add( o as int );
        return state;
      }
    }
  }
}
```

Fig. 3. Example of more detailed test scenario in C# extension