Vdm++TesK: Testing of VDM++ programs

Alexander A. Koptelov, Victor V. Kuliamin, and Alexander K. Petrenko

Institute for System Programming of Russian Academy of Sciences (ISPRAS),
B. Communisticheskaya, 25, Moscow, Russia
{steve,kuliamin,petrenko}@ispras.ru
http://www.ispras.ru/~RedVerst/

Abstract. This article presents the Vdm++TesK technology – a technology of test development for programs written in VDM++. To support this technology, the demo version of Vdm++TesK tool was developed.

1 Introduction

This article is dedicated to Vdm++TesK test development technology. Vdm++TesK is a technology based on the UniTesK [6] test development methodology, and intended for testing programs that are written in VDM++.

UniTesK is a successor of KVEST test development technology [11] developed by RedVerst [5] group of ISP RAS for Nortel Networks. KVEST was used in Nortel Networks projects for over 5 years and had gained both positive and negative experience. UniTesK is an attempt to overcome KVEST problems.

UniTesK is based on utilization of the software contract of the tested software (its constraint specifications) to generate the test suite for this software.

2 UniTesK test suite

Before we begin to describe the process of the test development in Vdm++TesK, let's look shortly at the UniTesK methodology. The full description of the UniTesK architecture can be found in [6].

2.1 The basic concepts of UniTesK

One of the main goals of testing is to demonstrate that behavior of the tested system (or *target system*) conforms to its requirements. To do this automatically, requirements should be written in rigorous, clear, and unambiguous way (in the form of *formal specifications*). UniTesK implements approach of *conformance testing* — the formal specification allows generating an oracle, program that checks the result of the target system's work against constraints given in its specification, and assigns a verdict on their correspondence.

Since in most cases a system can't be tested in all of the test situations (due to their very huge amount), in UniTesK the testing is performed only in a finite set of equivalence classes of such situations. This approach is called *partition testing*.

The percent of situations (from some set of testing situations) that are tested during some test is called *test coverage*. Measures of different coverages of the target component's domain are usually called *test coverage criteria*. The coverage chosen as a test coverage criterion for a test we call *the target coverage* of this test. Since we want to be able to test the component with different target coverages, we need to generate from its specifications a kind of a *universal oracle*, which can check the correctness of the component's behavior for an arbitrary input (see [11,10,9] for more details on automatic generation of such kind of oracles).

To test a system in its different states, the system is modeled by some *automaton*. An *automata based testing* is one of the main concepts of UniTesK.

Since the amount of the system's states is huge or even infinite, an equivalence relationship is defined over them. So, we can convert the initial automaton to the one with less set of states. In UniTesK, the particular case of this conversion is used, that is called factorization technique, and described in [8].

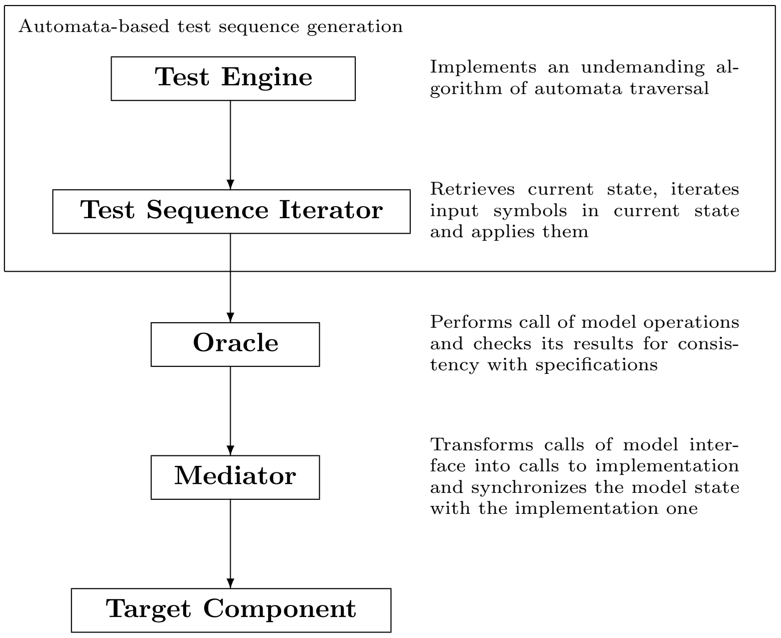
2.2 Details of UniTesK Test Suite Architecture

The core of UniTesK test suite is the traversal mechanism for finite automata. To provide additional flexibility, it is divided into two parts: test engine component encapsulating an algorithm of traversal of a finite automaton from some class, and test sequence iterator component, which embodies all the details of particular automaton. Test engine and test sequence iterator interact through well defined interface consisting of the following three operations defined in test sequence iterator.

– `State getState().` This operation returns the identifier of the current state of the automaton. State identifiers can be stored by test engine to facilitate a traversal, but the only thing it can do with them is comparison, which shows whether two identifiers designate one state of the automaton under traversal or two different states.
– `Input next().` This operation seeks for the next input symbol in the current state, which has not been applied yet during this traversal. If there are some, it returns anyone of such symbols, otherwise it returns `null`. The objects of `Input` type are identifiers of input symbols. As state identifiers, they also may be stored by test engine and can be compared with each other.
– `void call(Input param).` This operation applies the input symbol identified by the `param` object in the current state. It actually performs the corresponding transition in the automaton under traversal.

Notice, that the definition of the automaton is implicit – only sequences of input symbols for each state are specified. There is no need to specify all of its states and transitions between them.

One more important point of UniTesK test suite architecture is the use of adapter pattern (see [7] on detailed description of this pattern) to bind specification and implementation of the target component. For historical reasons, we call such adapters, which have specification interface and implement it on the base of the implementation under test, *mediators*.



Automata-based test sequence generation
Test Engine
Implements an undemanding algorithm of automata traversal
Test Sequence Iterator
Retrieves current state, iterates input symbols in current state and applies them
Oracle
Performs call of model operations and checks its results for consistency with specifications
Mediator
Transforms calls of model interface into calls to implementation and synchronizes the model state with the implementation one
Target Component

Fig. 1. Complete architecture of UniTesK test suite

Fig. 1 demonstrates the complete set of the main components of UniTesK test suite architecture.

Let us say some words on the origin of the components presented. Test engine component is a predefined part of a test suite. There are several such components, but the test developer does not need to write them himself. Instead, one of the existing ones should be used. Oracles are supposed to be generated automatically from specifications, which, in turn, are always developed by hand. Mediators are always developed by hands.

Now we consider the structure of the test sequence iterator component in more details paying most attention to the mechanism of iteration of applicable input symbols.

Test sequence iterator should provide all possible input symbols for each state of the automaton under traversal. Input symbol of the automaton under traversal corresponds to some class of possible inputs for the system under test.

An arbitrary coverage can be described by a number of predicates depending on the target operation identifier and a list of its input parameters. Each of these predicates determines one element of the coverage. UniTesK technology requires from specification designer to emphasize the basic partition of the specified operation domain by means of special constructs, `branch` operators. The elements of this basic partition correspond to subdomains where the specified operation has a substantially different functionality.

It is impossible to automatically obtain input symbols for the particular element of target coverage. So, test designer should provide handmade components called *iterators*. To filter input symbols that correspond to coverage elements already covered by the test, the components called *coverage trackers* are used.

Other functionality of test system iterator is represented by methods `getState()` and `call()`. As our experience shows, in some simple cases both of them can be generated automatically from the specification of the target component. When we consider the coverage criteria based only on coverages of component's operations domains, the method `call()` can be generated in general case. For most testing tasks it is enough, but sometimes, when we need to cover some specific sequences of calls of target operations, we should write part of this method by hands, and for this reasons this possibility exists in UniTesK technology.

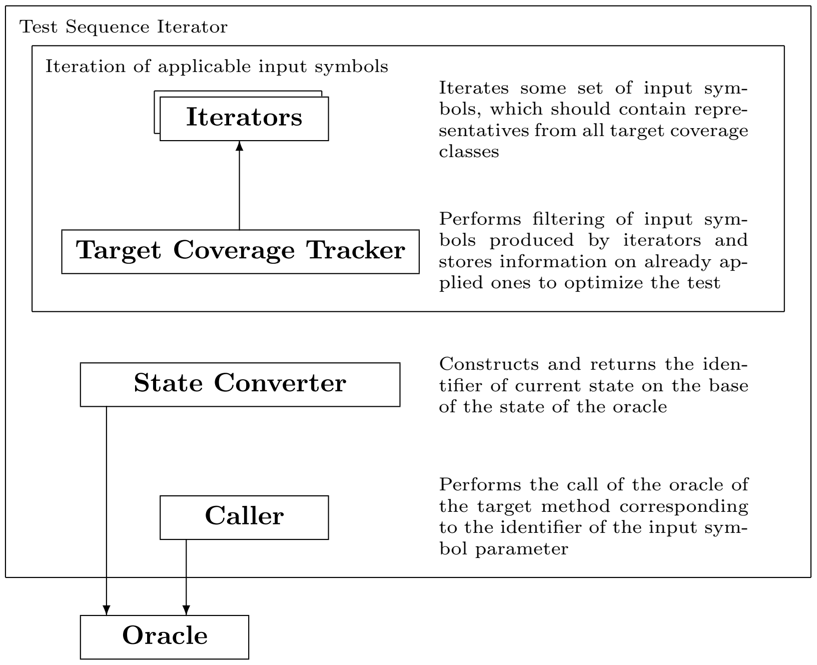

Test Sequence Iterator
Iteration of applicable input symbols
Iterators
Iterates some set of input symbols, which should contain representatives from all target coverage classes
Target Coverage Tracker
Performs filtering of input symbols produced by iterators and stores information on already applied ones to optimize the test
State Converter
Constructs and returns the identifier of current state on the base of the state of the oracle
Caller
Performs the call of the oracle of the target method corresponding to the identifier of the input symbol parameter
Oracle

Fig. 2. Typical structure of test sequence iterator component

Fig. 2 shows the structure of mechanism iterating applicable input symbols used in UniTesK.

3 Vdm++TesK – UniTesK implementation for VDM++

One of the main concepts of UniTesK lies in fact that tests are developed in the specification extension of the programming language of the target system (*target language*).

In Vdm++TesK, we deal with the target language that is a formal specifications language, so no language extentions are needed to support constraint specifications.

But, as usual, there are both good and bad sides in it.

The advantage is that in Vdm++TesK a test developer familiar with VDM++ would deal exactly with VDM++, not with its extension.

The disadvantage lies in the fact that without such language extension Vdm++TesK developers restrict their abilities to provide usable constructs that are not supported by VDM++, but very handy in test development. Examples of such constructs are declaration of additional coverage criteria (not supported by Vdm++TesK) and test scenarios (more compact and handly notation for the test sequence iterator). Implementation of these constructs in J@va can be viewed at [2].

3.1 Test coverage criteria

The current version of Vdm++TesK supports the only test coverage criterion, which is based on coverage of method's different functionality (*branch* coverage criterion).

This criterion is specified by user in a postcondition by dividing it in different branches basing on pre-values of the object's instance variables and the method's input values.

3.2 Implicit automaton definition

To test target methods' behavior in different states of their objects, in Vdm++TesK, as in UniTesK, automata-based testing is used.

The test automaton is specified in an implicit form – there is no need to describe all of its states and transitions between them. It's needed only to develop the operation for acquiring the identifier if the tested object's current state, iteration of automaton input symbols, and operation that performs test impact on the target system according to the input symbol.

In opposite to UniTesK, concept of the test scenarios is not implemented in Vdm++TesK. Test scenario is a shorthand way to specify the test sequence iterator (the tested automaton). In Vdm++TesK, part of the iterator's components (required classes, data and methods) are generated basing on the specifications of the target methods, and the rest are developed by hands.

3.3 Mediators

In UniTesK, mediadors bind the model described in specification and the implementation, and bind models of different abstraction levels.

Vdm++TesK is aimed at testing of VDM++ models, so it's supposed that the implicit specification is written exactly for the tested model. So, mediators are not needed for this propose.

Support for different levels of abstraction is absent in Vdm++TesK. Also, Vdm++TesK does not support test components reusing.

4 Vdm++TesK example

In this section, application of Vdm++TesK will be described. The full description of test development using Vdm++TesK can be viewed in [1]

The test development process in Vdm++TesK consists of two basic steps:

– Development of the target class specification.
– Development of the test sequence iterator.

Specification of the target class consists of definition of its invariants and description of pre- and postconditions of its methods. In Vdm++TesK, it's supposed that pre- and postcondition of the method are written jointly with its explicit definition using the *extended implicit operation definition* VDM++ construct.

Basing on the specification, oracle class is generated, which methods are intended to assign a verdict about conformance between behavior of the corresponding target method and its specification.

Basing on the list of jointly tested methods, the following classes are automatically generated: class with components of test sequence iterator, and base classes for automaton input symbols, one for each tested method.

Test developer should define handmade components by inheriting generated and predefined classes and declaring there the needed data and overriding virtual methods.

Finally, there will be a class that implements interface required by the predefined component that traverse the automaton of the target system's states.

4.1 Target system description

The target system is the class that implements a priority queue. An item is placed in the queue with a particular value of priority. An item with a higher priority will be removed from the queue earlier than item with a lower one. Items with equal priorities will be removed in the order they were put in the queue.

A queue item is an object of a `QueueItem` class inheritor.

```
class QueueItem
end   QueueItem
```

The queue is implemented in the `PriorityQueue` class. The `PriorityQueue` class has the following functionality:

`enqp(item: QueueItem, priority: int)` places an item `item` with priority `priority` into the queue.
`enq(item: QueueItem)` places an item `item` into the queue with the lowest priority.
`deq() item: [QueueItem]` takes an item with the highest priority from the queue. If the queue is empty, `nil` value is returned.
`size() res: nat` returns the length of the queue.
isEmpty() checks whether the queue is empty.

```
class PriorityQueue

values

  public
    MIN_PRIORITY : int = 0;

  public
    MAX_PRIORITY : int = 255;

instance variables

  public
    -- sequence of priorities
    priorities : seq of int := [];

  public
    -- sequence of the queue items
    -- queue(i) is the item with priority priorities(i)
    queue      : seq of QueueItem := [];

functions

  protected
    insertBefore[@item] : seq of @item * @item * nat +> seq1 of @item
    insertBefore(list, elem, idx) ==
        [list(i) | i in set {1, ..., idx-1} ]
      ^ [elem]
      ^ [list(i) | i in set {idx, ..., len list} ];

operations

  public
    enqp(item: QueueItem, priority: int)
    == ( dcl i : int := 1;
         while i <= len priorities and priorities(i) >= priority
         do i := i+1;
         priorities := insertBefore[int](priorities, priority, i);
         queue := insertBefore[QueueItem](queue, item, i);
       );

  public
    enq(item: QueueItem)
    == ( priorities := insertBefore[int]( priorities,
                                          MIN_PRIORITY,
                                          len priorities + 1
                                        );
         queue := insertBefore[QueueItem](queue, item, len queue + 1);
       );

  public
    deq() item: [QueueItem]
    == if len queue > 0
         then ( dcl item : QueueItem := hd queue;
                priorities := tl priorities;
                queue      := tl queue;
                return item;
              )
         else return nil;

  public
    size() res: nat
    == return len queue;

  public
    isEmpty() res: bool
    == return size() = 0;

end PriorityQueue
```

4.2 Implicit specification of the target class

The Vdm++TesK technology, it's required that specification class would be a successor of the `Specification` class.

```
class PriorityQueue
  is subclass of Specification

...
```

```
end   PriorityQueue
```

Let's specify invariants of the instance variables.

```
  inv forall i in set inds priorities &
            priorities(i) >= MIN_PRIORITY
        and priorities(i) <= MAX_PRIORITY;
  inv len priorities = len queue;
  inv forall i in set {2, ..., len priorities} &
        priorities(i) <= priorities(i-1);
```

Now the behaviour of the methods will be defined in the form of constraint specifications (pre- and postconditions).

Vdm++TesK supposes to combine specification of method's behaviour and partition of it's input space in the single notation. Firstly, it reduces the value of specifications and other sources of tests. Secondly, is simplifies the synchronization between specifications and requrements to tests completeness.

Technically, this combination is achieved in the following way. The postcondition should be written as if-elseif-else expression with each branch explicitly marked. To mark these branches, the following function is used.

```
  protected
    branch : seq1 of char +> bool
    branch(-) == true;
```

This function always returns true value. It's only intention is to grant an information for the static analysis of equivalence classes of the input space, and to generate trace about calls to target methods and equivalence classes covered by these calls.

```
  public
    enqp(item: QueueItem, priority: int)
    == ...
    ext wr priorities, queue
    pre priority >= MIN_PRIORITY and priority <= MAX_PRIORITY
    post if priorities~ = []
           then
                 branch("Empty queue")
             and priorities = [priority]
             and queue = [item]
         elseif priority not in set elems priorities~
           then
                 branch("Priority is not in queue")
             and priority in set elems priorities
             and exists1 i in set inds priorities
                 &     priorities(i) = priority
                   and queue(i) = item
```

```
              and   [ priorities(k)
                    | k in set inds priorities
                    & k <> i
                    ]
                  = priorities~
              and   [queue(k) | k in set inds queue & k <> i]
                  = queue~
      else
            branch("Priority is in queue")
        and priority in set elems priorities
        and exists1 i in set inds priorities
            &     priorities(i) = priority
              and queue(i) = item
              and   [ priorities(k)
                    | k in set inds priorities
                    & k <> i
                    ]
                  = priorities~
              and   [queue(k) | k in set inds queue & k <> i]
                  = queue~
              and forall k in set elems [ k
                                        | k in set
                                            inds priorities
                                        &   priorities(k)
                                          = priority
                                        ]
                  & k <= i;
```

In the `enqp()`, three branches are selected:

Empty queue. Placing an item into the empty queue.
Priority is not in queue. Placing an item with priority that doesn't exist in the queue.
Priority is in queue. Placing an item with priority that already exists in the queue.

```
public
  enq(item: QueueItem)
  == ...
  ext wr priorities, queue
  pre true
  post     branch("single branch")
       and priorities = priorities~ ^ [MIN_PRIORITY]
       and queue = queue~ ^ [item];

public
```

```
    deq() item: [QueueItem]
    == ...
    ext wr priorities, queue
    pre true
    post if len queue~ > 0
           then     branch("non-empty queue")
                and priorities = tl priorities~
                and queue = tl queue~
                and item = hd queue~
           else     branch("empty queue")
                and priorities = priorities~
                and queue = queue~
                and item = nil;

  public
    size() res: nat
    == ...
    ext rd queue
    pre true
    post     branch("single branch")
         and res = len queue;

  public
    isEmpty() res: bool
    == ...
    ext rd queue
    pre true
    post     branch("single branch")
         and res = (len queue = 0);
```

This specification should be translated using the Vdm++TesK tool to obtain the oracle class which methods perform the checking of the conformance between methods of the target class and their specifications.

4.3 Test sequence iterator

The test sequence iterator developed for the particular group of methods can be divided into three parts:

Predefined. Components that are independent of particular group of tested methods.
Generated Components that can be automatically generated basing on the specifications of the tested methods.
Handmade. Components that should be developed manually.

Predefined components exist in Vdm++TesK toolbox as VDM++ classes. Generated components are classes with methods, a few of which are abstract.

Handmade components are developed as classes that inherit predefined and generated ones.

Test state identifier It's impossible to test the system in all its states. So, an equivalence relationship is defined over them. Testing will be performed in each of these equivalence classes. The test state identifier is built basing on the current state of the system and is used to define to which equivalence class this state belongs.

A class that inherits the `Node` class is used as the test state identifier. This class contains data that determine the equivalence class of the corresponding state, and the comparison method. This method applied with another test state identifier object should return true, if both identifiers correspond to equivalent states.

In this example, all queues with equal number of items with equal priority are equivalent.

```
class PQueue_Node is subclass of Node

instance variables

  pris : map int to nat := {|->};

operations

public
initPQueue_Node : map int to nat ==> PQueue_Node
initPQueue_Node(p_pris) == (pris := p_pris; return self;);

public
compare: Node ==> bool
compare(node) ==
  return if isofclass(PQueue_Node, node)
           then let n : PQueue_Node = node in n.pris = pris
           else false;

end PQueue_Node
```

Automaton input symbols For each tested method, the class that defines input symbol is generated. Each input symbol corresponds to a set of parameters of the tested method.

To check the functionality of the `enqp()` method, it is not important, which item will be placed in the queue. Instead, the priority with which an item is placed is significant. So, an input symbol for this method should designate the priority of an item to be put in the queue.

```
class PriorityQueue_enqp_Arc is subclass of PriorityQueue_enqp_arc
instance variables
  public pri : int;
operations
  public
  initPriorityQueue_enqp_Arc : int ==> PriorityQueue_enqp_Arc
  initPriorityQueue_enqp_Arc(p_pri) == (pri := p_pri; return self);
end PriorityQueue_enqp_Arc
```

The `enq()` method has the only parameter, `item`, and the other methods have no parameters at all. So, there's no need to develop classes that represent input symbols corresponding to these methods.

Test sequence iterator A test sequence iterator is a class with well-defined set of methods. The handmade part of the iterator is developed in the class that iherits the generated one.

```
class SCRD_PQueue is subclass of SCRD_PQueue_generic

...

end SCRD_PQueue
```

The following operations should be defined in this class:

- The method that constructs the test state identifier basing on the current state of the target system.
- Methods that iterate automaton input symbols.
- Methods that convert automaton input symbols into a set of parameters of the corresponding target method.

The method that constructs the test state identifier will be as follows:

```
  public
    getNode: () ==> Node
    getNode() ==
      return
        new PQueue_Node()
          .initPQueue_Node(
            {   p
            |-> len [ oracle_of_PriorityQueue.priorities(i)
                    | i in set
                        inds oracle_of_PriorityQueue.priorities
                    & oracle_of_PriorityQueue.priorities(i) = p
                    ]
            | p in set elems oracle_of_PriorityQueue.priorities
            }
          );
```

To iterate values of the priority, it is convenient to define the following function:

```
functions
  nextPri : int +> int * bool
  nextPri(pri)
  == if pri = PriorityQueue`MIN_PRIORITY
       then mk_( (   PriorityQueue`MIN_PRIORITY
                   + PriorityQueue`MAX_PRIORITY
                 ) div 2,
                 true
               )
      elseif pri = PriorityQueue`MAX_PRIORITY
       then mk_( PriorityQueue`MIN_PRIORITY,
                 false
               )
       else mk_( PriorityQueue`MAX_PRIORITY,
                 true
               );
```

Now, let's write the operation for iteration of input symbols corresponding to the `enq()` and `enqp()` methods. For each method, the operation of obtaining the first input symbol, and the operation that construct the next symbol basing on the previous one.

```
operations
  PriorityQueue_enq_arc_init:
    () ==> PriorityQueue_enq_arc * bool
  PriorityQueue_enq_arc_init()
  == return mk_( new PriorityQueue_enq_arc(),
                 len oracle_of_PriorityQueue.priorities < 10
               );

  PriorityQueue_enqp_arc_init:
    () ==> PriorityQueue_enqp_arc * bool
  PriorityQueue_enqp_arc_init()
  == return mk_( new PriorityQueue_enqp_Arc()
                   .initPriorityQueue_enqp_Arc
                      (PriorityQueue`MIN_PRIORITY),
                 len oracle_of_PriorityQueue.priorities < 10
               );


  PriorityQueue_enq_arc_next:
    PriorityQueue_enq_arc ==> PriorityQueue_enq_arc * bool
  PriorityQueue_enq_arc_next(cur_arc)
```

```
    == return mk_(new PriorityQueue_enq_arc(), false);

    PriorityQueue_enqp_arc_next:
      PriorityQueue_enqp_arc ==> PriorityQueue_enqp_arc * bool
    PriorityQueue_enqp_arc_next(cur_arc)
    == let curArc : PriorityQueue_enqp_Arc = cur_arc,
           mk_(p,v) = nextPri(curArc.pri)
       in return mk_( new PriorityQueue_enqp_Arc()
                          .initPriorityQueue_enqp_Arc(p),
                        v
                      );
```

Now we should define two operations that convert input symbols corresponding to the `enqp()` and `enq()` methods to values of their parameters. The values of method's parameters should be returned via the value of the composite type, which fields have the names, types and order equal to the method's parameters ones.

```
  operations
    form_pvt_PriorityQueue_enq: Arc ==> pvt_PriorityQueue_enq
    form_pvt_PriorityQueue_enq(cur_par)
    == return mk_pvt_PriorityQueue_enq(new QueueItem());

    form_pvt_PriorityQueue_enqp: Arc ==> pvt_PriorityQueue_enqp
    form_pvt_PriorityQueue_enqp(cur_par) == (
       dcl curArc: PriorityQueue_enqp_Arc := cur_par;
       return mk_pvt_PriorityQueue_enqp( new QueueItem(),
                                         curArc.pri
                                       );
    );
```

4.4 Test execution

The developed test can be executed using the following VDM++ statement:

```
new SCRDEngine().init(new SCRD_PQueue()).main()
```

During the test execution, a test trace is generated. The information about absence or presence of an error and about the test coverage can be obtained from this trace. The test trace can be directed to both a file and the console of the VDM++ interpreter.

5 Conclusion

UniTesK has shown itself as technology that facilitate the development of high-quality tests for complex software. The examples of application of UniTesK (and

KVEST, it's successor) are projects of testing the following software: OS kernels, protocols, request brokers, etc. Vdm++TesK allows to carry the experience of UniTesK usage into both the VDM++ models and their programm implementations (due to the facility of translation the VDM++ test suite to programming languages).

Vdm++TesK can be evolved in conjunction with the UniTesK common line. At the time present, besides the traditional "black box" testing, UniTesK supports testing of distributed software components (e.g., protocols, see CTesK [3]), exceptions (implemented in J@va [2]), and specific interfaces like EJB and SOAP. The perspective goals are multithreading, load and real-timetesting. All of these means are opened for Vdm++TesK.

The demo version of the Vdm++TesK tool is available for public downloads at citeVdm++TesK.

References

1. http://redverst.ispras.ru/RedVerst/White Papers/vdmtesk/Main.html
2. http://www.ispras.ru/groups/rv/downloads/jatva.zip
3. http://www.ispras.ru/groups/rv/downloads/CTesK-Superlite.zip
4. http://www.ispras.ru/groups/rv/downloads/VDM++TesKDemo-1.3.zip
5. http://redverst.ispras.ru/
6. Igor B. Bourdonov, Alexander S. Kossatchev, Victor V. Kuliamin, Alexander K. Petrenko. *UniTesK Test Suite Architecture. In proc. of FME'02.* To be printed.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.
8. I. B. Burdonov, A. S. Kossatchev, and V. V. Kulyamin. Application of finite automatons for program testing. *Programming and Computer Software,* 26(2):61–73, 2000.
9. M. Obayashi, H. Kubota, S. P. McCarron, L. Mallet. The Assertion Based Testing Tool for OOP: ADL2, available via http://adl.xopen.org/exgr/icse/icse98.htm
10. D. Peters, D. Parnas. Using Test Oracles Generated from Program Documentation. *IEEE Transactions on Software Engineering,* 24(3):161–173, 1998.
11. I. Bourdonov, A. Kossatchev, A. Petrenko, and D. Galter. KVEST: Automated Generation of Test Suites from Formal Specifications. *FM'99: Formal Methods. LNCS,* volume 1708, Springer-Verlag, 1999, pp. 608–621.