Verification of Equivalence of Policy-selected Software Components in a Cognitive Radio

Leszek Lechowicz (Department of Electrical and Computer Engineering, Northeastern University, Boston, MA; llechowi@ece.neu.edu); Mieczyslaw M. Kokar (Department of Electrical and Computer Engineering, Northeastern University, Boston, MA; mkokar@ece.neu.edu)

ABSTRACT

In this paper we consider a scenario in which a declaratively specified policy on a cognitive radio recommends to use a specific type of functionality for communication with another peer radio. The requested functionality is given in terms of a specification in a formal policy language. The specification includes some behavioral aspects, e.g., delays introduced by particular sub-functions. The peer composes a module out of components it has in its library according to its component composition policies. To ensure that the composed functionality will perform correctly, the module needs to be verified against the specification. In this paper we look at two ways of achieving this goal – through direct inference (derivation) within the policy language, and through checking a model automatically constructed for this purpose. For instance, instead of trying to use a theorem prover on a collection of facts in the knowledge base, one could construct a semantic model based on those facts and then check the model to see if a given logical proposition holds. The specific goal of this paper is to discuss advantages and disadvantages of the two approaches to verification mentioned above. One of the aspects to compare is the time complexity of verification. It is known that the inference within a first-order language is undecidable, while checking a specific model is time linear in the size of the data. In this paper we describe our experiments with both approaches.

1. INTRODUCTION

A previously described ([1],[2]) interoperability scenario assumes that cognitive radios (CRs) are able to negotiate the use of software components based on specific policies and communication parameters. In the process of negotiation one of the CR nodes might be given a description of the software component that is not readily available in its knowledge base. Since it is assumed that CRs share a common *base ontology*, the description of the new software component might be given at an arbitrary level of complexity as long as all of its subcomponents decompose at some level to components directly defined in the base ontology. In case the node that received the specification of the new software module encounters a subcomponent it is not familiar with, it can send a query to the radio for the description of such a subcomponent. The querying can repeat iteratively until the node "knows" all the subcomponents, which in the worst case happens when the description of the component uses only the concepts from the base ontology.

Once the CR node "understands" the description of the new software module it can assemble it from the available components, both software and hardware. Since the composition in a general case may have a different structure than the one given by the description, the verification of composition is an important step in the interoperability scenario and is discussed in this paper in some detail.

2. FUNCTIONAL EQUIVALENCE AND VERIFICATION

Two software modules are functionally equivalent if all sequences of valid input values produce identical sequences of output values for both modules. Such defined *input-output equivalence* treats the software module as a black box with sets of inputs and outputs. As long as we cannot differentiate two modules by observing their responses to stimuli, they are functionally equivalent.

The simplest example of functionally equivalent modules might be two sorting algorithms that take an array of numbers as the input and return an array of those numbers in a non-descending order. Such two algorithms might differ substantially in non-functional properties. For example, the bubble sort algorithm has O(n^2) complexity, while the merge sort O(n*log(n)), so their execution time will significantly differ for large sizes of input arrays. But from the functional point of view they behave the same way.

Software Defined Radio algorithms are not confined to purely software solutions. With the advent of powerful and relatively inexpensive FPGAs, it is very likely that CR hardware will include specialized hardware units for the benefit of signal processing algorithms running in the radio.

Those units can be used in the signal processing modules (assuming that getting data in and out of such units is not a problem).

A simple example presented in Figure 1 shows how a specialized unit (in this case Multiply-Add Unit) can be used to make a particular module implementation more efficient by selecting a different structure to implement the same functionality.

The upper diagram in Figure 1 depicts a structure of a quadrature modulator, which consists of two multipliers, an adder and a 90 deg phase shifter. A structure-preserving implementation of such a module would instantiate appropriate software modules and would establish data paths among them exactly as in that figure. If a specialized multiply-add hardware unit is available it could be substituted for the multiplier and the adder, cutting the number of modules that have to be instantiated in half, and potentially making the composite module more efficient.

Note that those two different implementations have different structures but are functionally equivalent. Even though we concentrate mainly on the functional equivalence in this paper, it should be emphasized that non-functional concerns have to be resolved before a freshly composed software module can be inserted in the data-processing path.

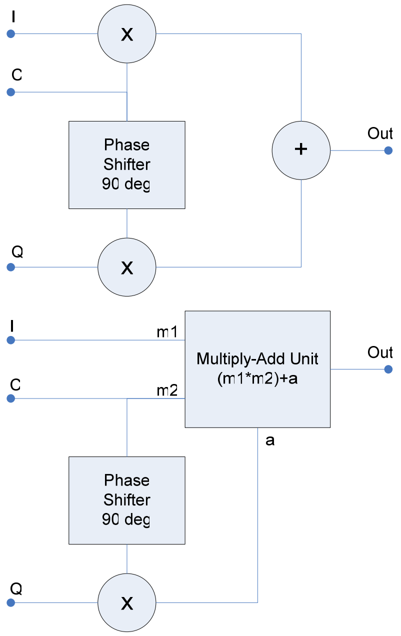

Figure 1. Functionally equivalent modules.

As an example, in the context of software defined radio (similarly as in other real-time systems) one of the most important concerns is to be able to finish computation on the current set of data before another set of data arrives. Thus in that case algorithms of lower computational complexity might be preferred. On the other hand, in the systems with very limited amounts of available memory, the memory footprint of the algorithm will be of utmost importance.

Those kinds of non-functional constraints have to be taken into consideration during the search phase in which the specification of the candidate for the composite software module is defined. The same constraints have to be later fed into the algorithm that validates the module to make sure that no hard requirements, like real-time deadline or memory size, are violated.

3. VERIFICATION THROUGH THEOREM PROVING.

As we mentioned in our previous work [2], there might be commonalities between different software modules that are visible only at the abstract functionality level. For example, a multiplier for complex numbers encoded as two single-precision float values significantly differs in the implementation from a multiplier for real numbers encoded as fractional numbers in 32-bit integers. The high level functionality (multiplication), however, is still the same; it's just multiplication after all.

Specware, a software design framework developed by Kestrel Institute, supports systematic construction of software from abstract specifications to executable code through a series of refinements [3]. An abstract specification written in Metaslang – the language of Specware – is refined through a series of category theory operations like morphism and colimit until the desired level of implementation detail is achieved. Specware distribution integrates Snark – a first-order logic theorem prover. Snark can be used to prove conjectures regarding functional equivalence of modules.

In the following snippet of a Metaslang code spec, QM_cand describes a composite module equivalent to the lower diagram in Figure 1. The conjecture QM_eq defines an input-output equivalence condition for that module with respect to the reference specification QM (which is equivalent to the structure in the upper diagram).

```
Adder= spec
  import Samples
  op Adder.Func: Sample*Sample -> Sample
  def Adder.Func(x,y) = Sample.add(x,y)
endspec

Multiplier = spec
  import Samples
  op Multiplier.Func: Sample*Sample -> Sample
  def Multiplier.Func(x,y) = Sample.multiply(x,y)
endspec

PhShifter90Deg = spec
  import CplxIntSamples
  op  PhShifter90Deg.Func: Sample -> Sample
  def PhShifter90Deg.Func(x) = Sample.conj(x)
endspec

MAC = spec
  import Samples

  op MAC.Func: Sample*Sample*Sample -> Sample
  def MAC.Func(m1,m2,a) = Sample.add(
Sample.multiply(m1, m2), a )
endspec

QM = spec
  import Adder
  import Multiplier
  import PhShifter90Deg

  op QM.Func: Sample*Sample*Sample -> Sample
  def QM.Func(I,Q,C) = Adder.Func(
Multiplier.Func( I, C ), Multiplier.Func(
PhShifter90Deg.Func(C), Q ) )
endspec

QM_cand = spec
  import Adder
  import Multiplier
  import PhShifter90Deg
  import MAC
  import QM

  op QM_cand.Func: Sample*Sample*Sample -> Sample
  def QM_cand.Func(I,Q,C) = MAC.Func( I, C,
Multiplier.Func( PhShifter90Deg.Func(C), Q ) )
```

```
  conjecture QM_eq is
    fa( I,Q,C )
        QM_cand.Func(I,Q,C) = QM.Func(I,Q,C)
endspec

p0 = prove QM_eq in QM_cand options "(use-
resolution t) (use-paramodulation t)"
```

A major weakness of first-order theorem provers (such as Snark) is the fact that first-order logic is undecidable in a general case. In other words, there exist valid first-order logic formulas that theorem provers are unable to prove or disprove in any arbitrary amount of time. And even those formulas that can be decided might require prohibitive amounts of time and memory to carry out the proof.

4. VERIFICATION THROUGH MODEL CHECKING

Clarke and Emerson [4] recognized the difficulties of the formal program correctness verification through logical inference and proposed a different approach based on model checking. Model checking is a technique based on a finite-state model of a system. A set of logical properties defining the system's behavior and constraints have to be defined and the model is automatically checked against those properties to see if they hold in all states.

One of the most popular software verification systems based on model checking is SPIN. SPIN (Simple Promela INterpreter) analyzes models of the system written in PROMELA (PROcess MEta LAnguage). In addition to verifying that all logical conditions hold in all states it also checks for deadlocks, race conditions, assertion violations and similar situations, which are results of errors in system design. SPIN has been successfully used for the verification of a variety of mission critical systems, including control algorithms of the flood control barrier system in Netherlands, telephone and data switch software and software for several space missions at NASA [5].

5. EXHAUSTIVE STATE SPACE SEARCH

In order to prove that the system is free from errors, an exhaustive search over its whole state space has to be performed. Only then the system is *proven* to be error-free. Coverage less than 100% can give some level of confidence that the system is correct (especially if the coverage percentage is high), but only the full state space exploration proves that beyond any doubt.

It should be emphasized that the number of states in the composite system is usually orders of magnitude bigger than the number of states in all state-machines that define the behavior of the system. This is because the overall system state depends not only on in what states particular state machines are at any given moment, but also on states of all variables and message channels.

In order to show how big a difference it makes let's look at the example given by Holzmann [6]. In that example he considers a protocol with two processes, each of them having 100 states. Each process accesses 5 local variables - each variable can have only 10 distinct values. Each of the two processes also has a message queue associated with it. Message queues have 5 slots each and the number of distinct message types is limited to 10.

This seemingly simple system potentially could have the number of system states in the order of 10^{24}. This number has been determined in the following way: two processes, each of them in one of 10^2 states, yield 10^4 potential combinations. Each of the processes has 5 local variables, each of them can have any of the 10 possible values – that gives additional $10^{5*2} = 10^{10}$ possible system states for each of the 10^4 states induced by the processes' state machines. Finally, each of the queues can hold from zero to five messages, each of them having one of ten possible values. So in the worst case the system could have the number of system states equal to

$$10^{10} \cdot 10^{4} \cdot \left(\sum_{i=0}^{5} 10^{i} \right)^{2}$$

Even if a computer system could analyze 1 million states per second, such a big system state space would require approximately 10^{11} years to complete the exhaustive analysis. Fortunately in practical systems, the number of reachable states is usually lower.

6. USING SPIN TO VERIFY FUNCTIONAL EQUIVALENCE

Conceptually, the verification of a composite software module is simple – create a model of the module in Promela and use SPIN to verify that its behavior is identical to that of the model created for the reference design.

Let's consider an example of a very simple composite module consisting of an instance of an adder and an instance of a multiplier (the lower diagram in Figure 2). We want to verify that this module is functionality equivalent to a reference multiply-add module, which is defined as a black box with inputs, outputs and the mathematical equation describing their relationship (the upper diagram).

The following snippet of the Promela code defines models for the adder, multiplier and a reference Multiply-Add unit.

```
proctype Adder(chan chin1, chin2, chout)
{
  SAMPLE a1, a2;
  do
    :: chin1?a1; chin2?a2 -> chout!(a1+a2)
  od;
}

proctype Multiplier(chan chin1, chin2, chout)
{
  SAMPLE m1, m2;
  do
    :: chin1?m1; chin2?m2 -> chout!(m1*m2)
  od
}
proctype MAdd_Reference(chan chm1, chm2, cha, cho)
{
  SAMPLE m1, m2, a;
  do
    :: chm1?m1; chm2?m2; cha?a -> cho!(m1*m2+a)
  od
}
```

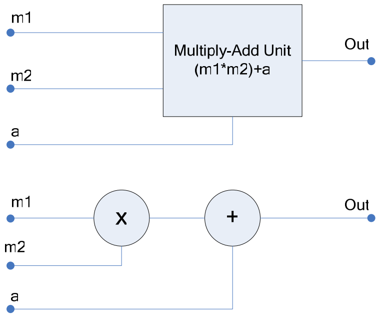

Figure 2. Equivalent Multiply-Add modules.

Each of the processes above executes an infinite `do … od` loop. The processes wait for data from the input channels (the `chan?var` statement) and once it arrives, execute an appropriate calculation and send the resulting data to the output channel (the `chan!expr` statement).

The composite Multiply-Add module simply instantiates Multiplier and Adder processes and connects them through a communication channel (see Promela code below). Note that in this particular implementation, the MAC_Composed process terminates after the Multiplier and Adder processes are created. So effectively it substitutes those two processes for itself. It is not required in general case – both parent and child processes can exist at the same time.

```
proctype MAdd_Composed(chan chm1, chm2, cha, cho)
{
  chan chMulOut = [1] of { SAMPLE };

  run Multiplier(chm1, chm2, chMulOut);
  run Adder( chMulOut, cha, cho);
}
```

In order to run both simulation and verification we need to add some stimuli to the inputs of the module and somehow evaluate the output data. In our little experiment quasi-random data sources are used to drive inputs of both Multiply-Add modules.

It is very important to understand the difference between the simulation and validation modes of SPIN. Let's consider the following PROMELA statement:

```
if
    :: x = 1;
    :: x = 2;
    :: x = 3;
    :: x = 4;
    :: x = 5;
fi
```

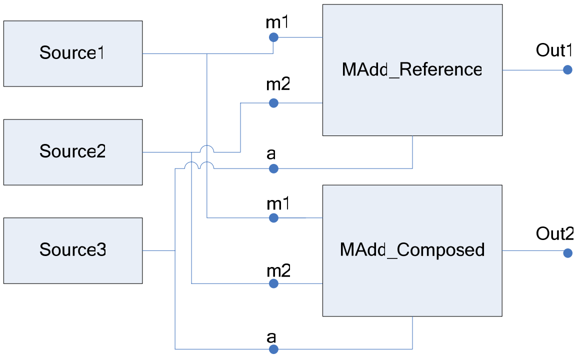

Figure 3. Simulation/Verification setup.

In the simulation mode, the if … fi statement works in such a way that if more than one of the options (statements starting with double colon) can be executed when the flow of execution enters it, one of those options is selected randomly in a non-deterministic way. Since in the code above no option has a logical condition, all of them are executable. That means that in the simulation mode one of the five values will be assigned to the variable x. So in essence this code works as a single-shot, limited range random number generator.

In order to validate the models of the system, SPIN is used to generate C code that actually implements the exhaustive search algorithm. The above if .. fi statement in the worst case increases five-fold the number of states the search has to cover.

The validation code requires that system invariants (i.e. Boolean conditions that are true in all reachable states) are established. Their negated forms are used to create the so called *never claim* – i.e., a composite statement that is false in all reachable states of the system.

For the Multiply-Add equivalence model discussed here the system invariant is simply the condition that if two sets of output values have been received from MAdd_Reference and MAdd_Composed, they are always equal. In the LTL (Linear Temporal Logic) syntax that condition can be expressed as below:

```
[]( recv -> outEq )
```

where outEq is defined as

```
#define outEq  (output1 == output2)
```

and `recv` is a Boolean flag set to true when `output1` and `output2` have been received from the modules.

The negated LTL formula is used to automatically generate the never claim for the system.

```
never {    /* !([]( recv -> outEq )) */
T0_init:
        if
        :: (! ((outEq)) && (recv)) ->
```

```
                                goto accept_all
        :: (1) -> goto T0_init
        fi;
accept_all:
        skip
}
```

During the validation, the never claim is evaluated after every single system state transition. That, together with the exhaustive search, guarantees that if the validation passed the system, the invariants hold for all reachable states.

7. PROBLEMS WITH FUNCTIONAL EQUIVALENCE VERIFICATION

From the earlier discussion of the size of the system state space (Section 5) it is obvious that the described approach is not feasible from the computational point of view. The input parameters alone multiply the size of the state space by factor 2^{48} if the SAMPLE type is defined as a 16-bit integer or 2^{96} if it is a 32-bit integer. Even if the width of the data type is reduced to 8-bit, the number of states is still so large that the validation could not be executed on a notebook computer with 1.25 GB of RAM due to insufficient memory.

The experiment with the 8-bit data type width however revealed another issue that has been ignored during the creation of the model – the 8-bit data type is not wide enough to accommodate the results of addition and multiplication of two 8-bit numbers. So the model in which inputs and outputs of the Multiply-Add unit have the same width is wrong, unless there exist some additional restrictions as to what ranges of values can be passed on the inputs of the system.

In order to better illustrate the problem of state space explosion we tried to verify the functionality of an Adder ($c = a+b$) and a Multiply-Add unit ($z = v*u+y$) composed of an Adder and a Multiplier for different bit widths of the input data. The results are shown in Table 1.

Table 1. State explosion in functional verification through model checking.

Word width in bits	Adder		Multiply-Add	
	States	Memory [MB]	States	Memory [MB]
3	6678	0.299	325778	21.977
4	28078	1.195	2746930	177.867
5	115038	4.774	22538354	1431.150
6	465598	19.086	Out of	memory
7	1873278	76.329		
8	7514878	305.282		
9	30103038	1221.065		

It is clear that direct application of model checking is ill suited to functional equivalence verification. As a matter of fact, Promela Reference Manual states directly that "*Spin* targets the verification of process interaction and process coordination structures, and not internal process computations. Abstraction is then best done at the process and system level, not at a computational level." [7]

8. VERIFICATION OF THE COMPUTATION THROUGH SYMBOLIC EXECUTION

Siegel et al. found an interesting novel approach to dealing with state explosion in the verification of numerical algorithms [8]. They were trying to solve the problem of functional equivalence between sequential and parallel versions of the same numerical algorithm. The serial algorithm was the reference; the parallel version was being verified for equivalence.

In their solution the computations are modeled symbolically. An input is considered to be a vector of symbolic constants and the output is a vector of symbolic expressions. The numerical operations in the program are replaced by appropriate symbolic operations in the model. Moreover, each symbolic expression is represented by a single index into a table, which prevents the state space explosion and makes it possible to use model checkers, such as SPIN, for verification. The numerical expressions undergo reduction through the application of appropriate reduction theorems. In case the numerical algorithm contains conditional branches, the models not only have to explore all possible execution branches, but also must record the path condition for each branch execution. The parallel program is equivalent to the sequential one iff the outputs of both versions are deterministic functions of their inputs (i.e. they cannot change from one run to another depending on the values of data) and iff the symbolic functions are equivalent for all possible paths of executions.

The results of the experiments cited in their paper show a large reduction of the state space size for numerical algorithms (see Table 2).

Table 2. Experimental data from Siegel et al. [8]

	matmat	gauss	jacobi	monte
states ($x10^3$)	4443	16114	6295	3112
Memory (MB)	217	801	362	279
time (seconds)	506	3224	9846	738

matmat – 6x6 matrix multiplication
gauss – 6x6 matrix Gaussian elimination
jacobi – 17x17 Linear equation solving through Jacobi iteration
monte – π approximation through Monte Carlo simulation

Comparing those results with the results we gathered during our experiments makes the reduction of the space state though symbolic method impressive.

9. BEHAVIORAL ASPECTS IN FUNCTIONAL VERIFICATION METHODS

As we indicated in our earlier paper [2], systems based on functional languages (e.g. Specware) are not well suited to describing behavioral aspects of cognitive radios. The gist of the problems is in that behaviors require mechanisms (e.g. global variables) that are considered side effects in functional languages and as such are not directly supported. We also suggested that even though a particular side effect is not supported in the Specware itself, if we create a spec for that specification and a set of axioms, we still can reason about it in an abstract way.

The UnitDelay module is an example of a behavioral functionality – it requires a memory element (such as register) to store the value of the argument until the next time the function is called. Since it uses a side effect (a memory location) it cannot be expressed directly in Specware, instead the properties of UnitDelay are described through the axioms.

```
UnitDelaySpec = spec
  import Samples
  op     UnitDelay.Func: Sample -> Sample

  axiom UnitDelay_commutativity is
    fa( f:( Sample->Sample ), x:Sample )
      UnitDelay.Func(f(x)) =
                    f( UnitDelay.Func(x) )

  axiom UnitDelay_commutativity2 is
    fa(f:(Sample*Sample->Sample),
       x:Sample, y:Sample )
    UnitDelay.Func( f(x,y) ) = f(
UnitDelay.Func(x), UnitDelay.Func(y) )
endspec
```

The UnitDelay specification can be used in other Specware modules to model behavior within the confines of the functional language.

```
 MACSpec = spec
  import UnitDelaySpec
  op MAC.Func: Sample*Sample*Sample -> Sample
  def MAC.Func(m1, m2, a) =
    UnitDelay.Func(
        Sample.add(
          UnitDelay.Func(Sample.multiply(m1, m2)),
          UnitDelay.Func( a ) ) )
endspec
```

In the above example MACSpec models the functionality of a Multiply-Add module with a two-unit delay between the inputs and the output. It should be emphasized that even though the behavioral aspects of functionality are not supported in Metaslang directly, it is possible to create Specware specs describing them and to use them to prove conjectures, which was also shown in [2].

PROMELA supports global and local variables so modeling of the behavior is easy to implement. The following fragment of code for example defines a unit delay in PROMELA.

```
proctype UnitDelay( chan chi, cho)
{
  SAMPLE reg = 0;
  SAMPLE tmp = 0;
  do
    :: chi?tmp -> atomic { cho!reg; reg = tmp;}
  od
}
```

Such defined process type can be used to define the Multiply-Add unit equivalent to the one created in Metaslang.

```
proctype MAC_2Delay( chan chm1, chm2, cha, cho)
{
  chan chMulOut = [1] of { SAMPLE };
  chan chD1Out = [1] of { SAMPLE };
  chan chAddOut = [1] of { SAMPLE };
  chan chAddDelOut = [1] of { SAMPLE };

  run Multiplier(chm1, chm2, chMulOut);
  run UnitDelay(chMulOut,chD1Out);
  run UnitDelay( cha, chAddDelOut);
  run Adder( chAddDelOut, chD1Out, chAddOut);
  run UnitDelay( chAddOut, cho);
}
```

Behavioral elements obviously introduce additional complexity to the system with respect to comparable purely functional specifications.

In model checking each UnitDelay instance introduces a memory element (register) whose presence increases the state space times 2^w, where *w* – the width of the memory element in bits.

In theorem proving the increased complexity is more difficult to gauge and will depend on the number of additional axioms inserted into the theorem prover and number of instances of UnitDelay in the composite module under verification.

10. CONCLUSION

We looked into two methods of functional equivalence verification. The theorem proving method yielded positive results in our experiments. The fact that first-order logic is undecidable in a general case, the experimental character of theorem proving software available at the moment, as well as significant computational and memory resources required make this method might not be suitable for low-power, portable devices.

In our limited experimentation we didn’t experience the undecidability of the first-order logic even after introduction of behavioral elements (UnitDelay modules). SNARK was able to prove functional equivalence conjectures in a fraction of second. It is difficult however to draw any conclusions as to how SNARK would behave with more complicated specifications. The ongoing research into the theorem proving theory and algorithms might make this technology feasible in the feature, especially if additional restrictions are placed on the types of first-order logic expressions allowed in the prover.

In the context of cognitive radio, model checking is not well suited for functional equivalence verification due to the system state space explosion. Novel ideas of combining symbolic execution with model checking can bring an algorithm’s state space down to a small fraction of the original size so that verification of a not-too-complex computation like the 6x6 matrix multiplication is feasible on an average computer workstation. Still, due to the very nature of this method, it is required that all reachable states in the system are explored, and thus the feasibility of application of this method in the cognitive radio is questionable at this time.

11. REFERENCES

[1] L. Lechowicz, M. Kokar. *Achieving Dynamic Interoperability of Communication: Transfer of Ontology and Rules Between Nodes.* In Proceedings of the Software Defined Radio Technical Conference SDR'06, 2006.

[2] L. Lechowicz, M. Kokar. *Composition, Equivalence and Interoperability: An Example.* In Proceedings of the Software Defined Radio Technical Conference SDR'07, 2007.

[3] Y. V. Srinivas, R. Jullig. *SPECWARE: Formal Support for Composing Software.* Tech. Rep. KES.U.95.5, The Kestrel Institute, Palo Alto, CA, 1995.

[4] Clarke E., Emerson E. A. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic.* Logics of Programs, Workshop, Yorktown Heights, New York, May 1981.

[5] SPIN website, http://spinroot.com/.

[6] Holzmann G. J. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[7] Promela Reference Manual, http://spinroot.com/spin/Man/promela.html#section0.

[8] Siegel S. F., Mironova A., Avrunin G. S., Clarke L. A., *Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs.* ACM Transactions on Software Engineering and Methodology, Vol. 17, No. 2, April 2008.