Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 101–116
www.elsevier.com/locate/entcs

Model Based Testing of a Network-on-Chip Component

Leonidas Tsiopoulos [1] Manoranjan Satpathy [2,3]

Department of Information Technologies
Abo Akademi University
Joukahaisenkatu 3-5, FIN-20520 Turku, Finland

Abstract

We discuss the problem of model based test case generation and that of automatic testing of a component of an asynchronous Network-on-Chip (NoC). We start with a model of the component in *B Action System*, which is a state based formalism based on *Action Systems* and the *B Method*. We construct a finite state space graph by executing the model, and next, generate a test driver from the abstract test cases. This test driver can be used to test a matching implementation automatically. The important contribution of our work is that we consider hierarchical models for test case generation and automatic testing, whereas the previous approaches considered flat models. In addition, we also highlight the issue due to non-determinism in hierarchical models.

Keywords: Model Based Testing, Network-on-Chip, B Method, Action Systems

1 Introduction

The primary aim of model based testing is the automatic generation of test cases from a model of a system. A model of software is usually a specification of the system which is developed from the requirements early in the development cycle. A (state based) formal model can be subjected to symbolic execution to obtain a coverage graph in which nodes represent states and edges are labeled with operation applications (or transitions). Based on a test adequacy criterion, one can select a finite set of finite behaviours (i.e., test cases) from this graph and test if the implementation is consistent with these behaviours.

In this paper, we consider models (specifications) in state based formal notations like Z [28], VDM [17] , B [1], Event-B [2], ASM [15], and Action Systems [3,19]. Behavior of such systems are described using an explicit model of the system state

[1] Email: ltsiopou@abo.fi
[2] Email: manoranjan.satpathy@gm.com
[3] Currently at GM India Science Lab, Bangalore

doi:10.1016/j.entcs.2009.09.054

along with operations which modify the state. B Action Systems [7,31], the modeling notation of this paper, is a state based formalism based on Action Systems and the B Method [1]. Note that a model in B Action Systems is a valid model within the B Method; therefore, tool support for the B Method [9,5,18] can be used to analyze models in B Action Systems. The existing model based testing approaches [4,24] to handle B models consider only flat models; i.e. models without any hierarchy. Hierarchy plays an important role in modular design and modular implementations. Hierarchy also provides a natural mechanism for parallel and distributed design. In addition, refinements in B Method may also introduce hierarchy.

The paradigm of Systems-on-Chip (SoC) requires complex communication structure which can be addressed by the Network-on-Chip (NoC) technology [6,10]; NoCs offer higher bandwidth and support concurrent communications on a chip. NoCs can provide well-defined interfaces which can decouple computation from communication. It is important that the NoCs are reliable. A NoC consists of two primary components - the routers and the network interfaces (NI). A router can be connected to another router or to the NIs. The routers transport data packets from one NI to another. The authors of [29] have discussed a model of an asynchronous NoC router, specified using B Action Systems [7,31]. We generate functional test cases from this model and a test driver to perform automatic testing; in particular, we generate model based test cases to test the routing and data propagation functions of the NoC component.

The main contributions of this paper are: (a) automatic test case generation from a hierarchical model specified in B Action Systems; the previous works discussed only flat models, (b) generation of a test driver from the abstract test cases assuming that the implementation is also hierarchical, and (c) an approach to handle non-determinism in a hierarchical model.

The organization of this article is as follows. Section 2 discusses the related work. In Section 3, we present a short introduction to the B Action Systems and then discuss the model of a NoC component. Section 4 presents the problem that we address in this paper. Section 5 discusses the test generation procedure. In Section 6, we outline a solution to the problem of non-determinism. In Section 7, we present an analysis of our approach. Finally, Section 8 concludes this paper.

2 Related Work

Conformance testing is concerned with the assessment of the extent to which an implementation or system conforms to its specification [13]. A *testing criterion* is a set of requirements on test data which reflects a notion of adequacy on the testing of a system. A test adequacy criterion determines whether sufficient testing has already been done, and in addition, it provides measurements to obtain the degree of adequacy obtained after testing stops [33]. A *test oracle* is a mechanism to determine the correctness of test executions. A *test driver* is a tool which activates a system, provides test inputs and reports test results. A *representation mapping* is a mapping which maps the abstract name space of the model with the concrete

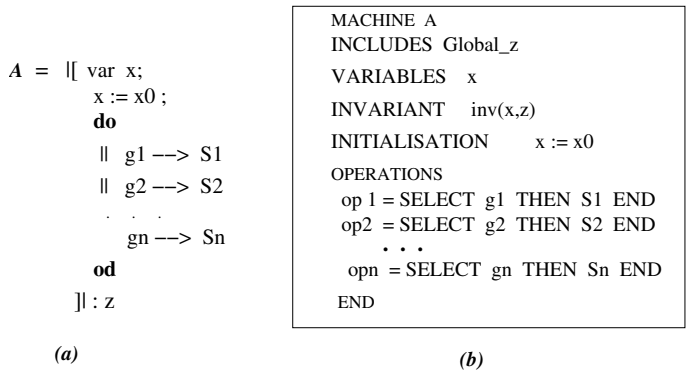

Fig. 1. Structure of a B Action System

name space of the system under test (SUT) [14].

The work by Dick and Faivre [11] is a major contribution to the use of formal methods in software testing. A VDM [17] specification has state variables and an invariant (Inv) to restrict the variables. An operation, say OP, is specified by a pre-condition (OP_{pre}) and a post-condition (OP_{post}). The expression $OP_{pre} \wedge$ $OP_{post} \wedge$ Inv is converted into its Disjunctive Normal Form (DNF); each disjunct, unless a contradiction itself, represents an input sub-domain of OP. Next, as many operation instances are created as the number of valid disjuncts in the DNF. An attempt is then made to create a Finite State Automaton (FSA) in which each node represents a possible machine state and an edge represents an application of an operation instance. Test cases are then generated by traversing the FSA.

The BZ Testing Tool (BZ-TT) [4] and the ProTest approach [24] also partition the operation input space in a manner similar to the method of Dick and Faivre. Both methods also use variants of the Dick and Faivre technique to generate test cases. In addition, both consider flat B models.

Hamon et al. [16] have used model checking to generate test cases from models in the SAL [23] formal language. Keeping a coverage criterion in mind, the SAL model is so instrumented with *trap variables* [16] that reachability of a trap variable implies reachability of a model element; reachable traces then become the test cases.

Although there is considerable amount of work on formal methods applied to NoC system design [32,26,29], to the best of our knowledge, there is hardly any work on Model Based Testing relating to formal models of NoC frameworks.

3 B Action Systems

B Action Systems (BAS) [7,31] is a state based formalism based on Action Systems and the B Method. The BAS formalism was created in order to be able to reason about parallel and distributed systems, like Action Systems, within the B Method. BAS is related to Event-B [2] in the sense that both can model reactive systems; however, BAS is hierarchical but Event-B is not. The structure of an Action System A is shown in Figure 1(a) in which z and x are respectively the global and local state variables. The Action System interacts with the environment through the

global variables. State variables have types and the set of possible assignments to them constitutes the state space. The statement $x := x_0$ assigns initial values to the local variables. Each action has the form $g_i \rightarrow S_i$, where $i = 1..n$, in which g_i is the guard and S_i is a statement on the state variables. An action is enabled only if its guard evaluates to *true*. As regards to the behaviour of the Action System [3,19], the initialization statement is executed first, and thereafter as long as there are enabled actions, one of the enabled actions is selected non-deterministically for execution. When there are no enabled actions, the system terminates.

An Action System can be translated to a B abstract machine – an abstract machine is the basic unit of specification in B – as shown in Figure 1(b). This is an Action system within B, and is called a *B Action System* (BAS). The system is identified by a unique name. The local variables of the system are given in the VARIABLES-clause. The INVARIANT-clause defines the types of the local variables and gives their guaranteed behaviour. Initial values are assigned to the local variables in the INITIALISATION-clause. The operations (or actions) in the OPERATIONS-clause are of the form *op* = SELECT g THEN S END, where g is said to be the guard and S the body. The guard g is a predicate on the variables, and when g holds the action *op* is said to be enabled. Only enabled actions are considered for execution and if there are several actions enabled simultaneously they are selected for execution in a non-deterministic manner, analogous to the behaviour of an Action System.

B Action Systems can be composed to model parallel systems [7,20]. Structuring mechanisms such as SEES, INCLUDES and PROMOTES can be used to express B Action Systems as a composition of subsystems [1]. The SEES-mechanism allows read access to the seeing system. The INCLUDES-mechanism allows write access to the variables of the included system. Actions of the included system can also be made available by promoting them into the including system within a PROMOTES-clause. The structuring mechanisms provide an efficient way to model system hierarchy.

Communication can occur between two BAS in two ways: by use of global variables, or by global procedures; this paper uses global procedures for communication purposes. Actions do not take any parameter and they execute autonomously; procedures may or may not have parameters, and they are to be invoked in a certain context. Consider the action A_i = SELECT g_i THEN S_i || Proc END; this action is enabled only if the procedure Proc is also enabled. The action and the procedure in its body are executed as an atomic entity. Communication using global procedure will be explained in the next section in more detail.

3.1 Our example

Tsiopoulos and Waldén [29] have specified a formal routing scheme relying on the request and acknowledge phases of the asynchronous communication. Within this asynchronous NoC routing scheme, data packets are received at the input channels of the routers and they are distributed to their output channels for subsequent propagation to the neighbouring routers. The routers acknowledge their input channels so that new data packets can be received. Controlling components of routers propagate the data packets to routers that have not received them yet, and prohibit the

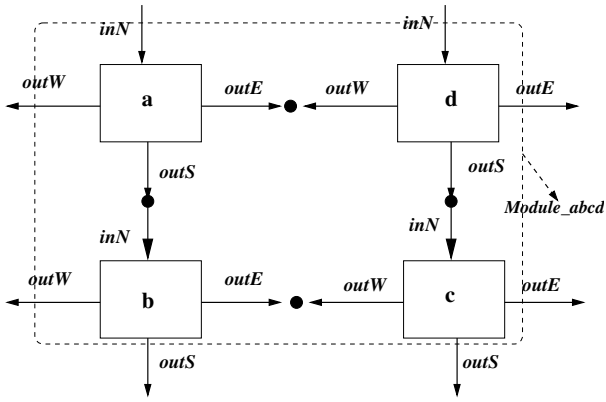

Fig. 2. Module_abcd: part of a 2D mesh network

cycling of data packets to occur. This is because cycling of data packets back to routers which have received them already, reduces on-chip interconnect efficiency and increases power consumption.

A hierarchical and compositional development method was used [29] for this purpose. Starting with the simplest subsystem of the routing framework, an asynchronous channel component called $PushChannel$ (data propagation upon request) was specified as a BAS. The channel's inputs and outputs were modeled as global variables. Of these global variables, the necessary input and output asynchronous control handshakes were modeled with boolean variables named *cstart* and *cend* respectively, and the input and output data were captured with variables named *dstart* and *dend* of the generic type $DATA$. These variables together with simple procedures to update their state were defined in a separate machine named $ChannelData$ which was included in $PushChannel$. The latter had two operations: one to propagate data and the request from its input to its output and the other to acknowledge its input after the output data was taken, so that new data and request could be received. The interface of $PushChannel$ has two global procedures, $ProcChangeCstartAndDstart$ and $ProcChangeCendFalse$; the first to update its input with a new request and data and the second to acknowledge its output after the transfer of the output data.

A BAS named *Router* including eight instances of $PushChannel$ was then created. Four actions were specified for controlling the channels and propagating the data along them. For example, action $TransferData\_N$ copies the request value of the communication and the data from the output of the input northern channel inN to the inputs of the output channels, $outE$, $outS$, and $outW$ so that the data can be propagated further towards the neighboring routers (refer to Figure 2). Simultaneously it sends acknowledgment to the output of channel inN to indicate that *Router* is ready to receive new data via this channel.

Finally, four instances of this router (*a.Router*, *b.Router*, *c.Router* and *d.Router*) were composed into a controlling system named $Module\_abcd$ which controlled the data distribution between the routers. In order to do so, some of the global interface procedures of $PushChannel$ were *promoted* to the interface of the

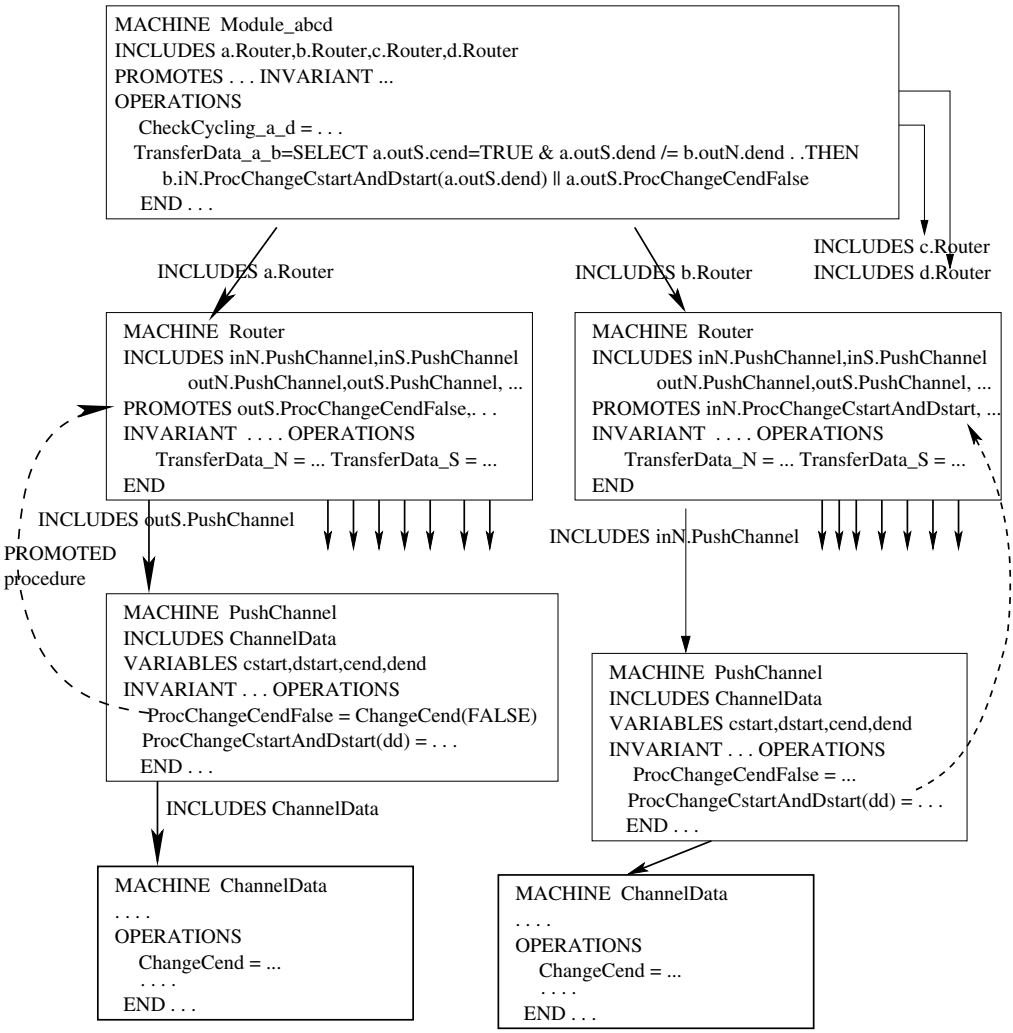

Fig. 3. Machine hierarchy in the NoC specification.

Router.

Figure 3 outlines part of the hierarchical structure of the specification of the NoC routing scheme and presents one of the communications between the subsystems using global procedures. Action $TransferData\_a\_b$ in $Module\_abcd$ transfers data from the southern output channel of $a.Router$ to the northern input channel of $b.Router$. The guard checks whether channel $a.outS$ is ready to transmit and channel $b.inN$ is ready to receive. If so, the data value of channel $b.inN$ gets the data value of channel $a.outS$. And the flags of both the channels are modified to indicate that the transmission has taken place. The global procedures $ProcChangeCstartAndDstart$ and $ProcChangeCendFalse$ defined in machine $PushChannel$ do this modification. See in the figure, how this global procedures has been *promoted* to the interface of the router.

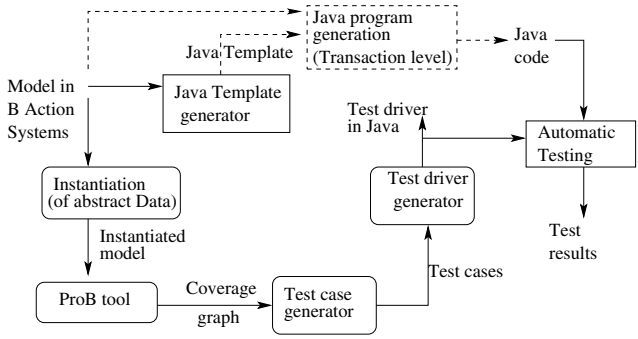

Fig. 4. The implementation framework. The implementer refers to the template.

4 The problem

The modeling of the NoC routing scheme as a hierarchical BAS is at a higher level of abstraction. Languages like SystemC, Java are usually used for Transaction Level Modeling of the NoC systems (also called programmer's view) [22,32]. Thereafter timing and other parameters are added and hardware/software partitioning is performed for subsequent implementation. The conformance between the transaction level modeling and the actual implementation needs another round of testing, which is beyond the scope of this paper. For the present, we assume that the specification of the router component in BAS is used to obtain a transaction level model in Java. In such a case, one should ask whether the Java program is faithful to the model? We rely on model based testing to answer such a question.

Figure 4 illustrates the testing approach. Given the hierarchical model of the NoC router, we first instantiate the abstract data in the model. The data objects which need to be propagated are elements of an abstract set $DATA$. We restrict this set and fix its elements in order to avoid state space explosion. Then we use the ProB tool [18] to obtain a finite state space graph – more about this later. Some traces in the graph become test cases. A test driver – a Java program – is obtained from the coverage graph. The expected outputs are available when model execution is performed; in order to compare such outputs with the outputs produced during program execution, the test driver includes assertions at appropriate control points.

We assume that the transaction level model of the Router component in Java has a similar structure – more about this later. To help the implementer in preserving the model structure, we generate a Java template and the implementer uses it to write the Java program. Now when this Java program runs with the test driver (also in Java), automatic testing is performed. In the following, we discuss these steps in more detail. We explain the concepts of our method by assuming Java as the language of implementation. However, any other programming language like C, SystemC or C++ will not pose any problem.

4.1 Pre-processing

Each procedure or action in the BAS hierarchy has its guard already in DNF. Furthermore, since there is no control branching within any operation, if we use

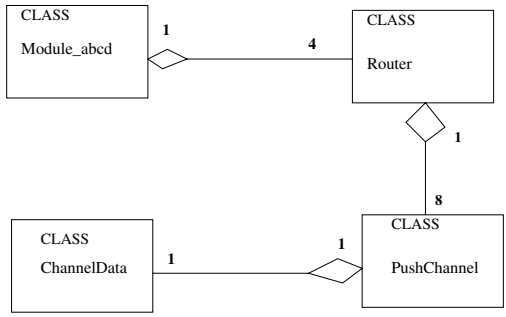

Fig. 5. Machine inclusion becomes class nesting

the method of Dick and Faivre [11], we obtain a single instance for each operation or action. Thus there is no need to instrument the models in terms of operation (action) instances.

We consider a simple test criterion. Generate a set of test cases such that each operation (or action) in the model is invoked at least once in test suite. A test case is a sequence of action or procedure invocations. Note that we can also consider any other complicated test criterion, say transition-pair coverage.

Let us instantiate the generic set $DATA$ to $\{d1, d2, d3, d4\}$. This small size of $DATA$ will make it easy to obtain the full state space.

4.2 Implementation Recommendations

- From a BAS, we derive a Java template automatically. Such a template has a similar hierarchical structure as in the model in the sense of the following. Each basic model in BAS becomes a Java object. Furthermore, an INCLUDES relationship becomes class nesting in Java (aggregation). Refer to Figures 3 and 5. *Module_abcd* contains four instances of *Router*; so, *class Module_abcd* also contains four instances of class *Router*. Similarly, class *Router* contains eight instances of class *PushChannel*, and class *PushChannel* contains a single instance of *ChannelData*. Figure 6 shows the correspondence between the model *Router* and the *class Router*.
- An action or a procedure in a machine becomes a method in the corresponding Java class. Further, the action or procedure signature is preserved in the class implementation in the following sense.
  · If a model parameter type is either numeric or boolean it becomes *int* and *boolean* in the implementation respectively.
  · For any other model parameter of type PP, we implement it as an object of a Java class PP.

  Consider the B procedure $ProcChangeCstartAndDstart()$ in Figure 3 whose only parameter is an element of set $DATA$. Its signature in Java will be: $void\ ProcChangeCstartAndDstart(class\ DATA\ dd)$. Here, parameter dd is an object of class $DATA$.
- Machine inclusion is transitive in B. Let an operation $Op()$ in a machine occurring down in a hierarchy be promoted to the interface of a parent machine.

```
d_pre←probe_aoutSdend,
  TransferData_a_b,
d_post←probe_TD_binNdstart,
  assert(d_pre = d_post)
  (a)
```

```
d_pre = MM.probe_aoutSdend();
MM.TransferData_a_b();
d_post = MM.probe_TD_binNdstart()
assert(d_pre == d_post);
(b)
```

Table 1
probe template: an example; here MM is an object of class $Module\_abcd$.

To make it possible the appropriate methods and attributes are to be declared *public*. In Figure 7(a), global procedure $ProcChangeCstartAndDstart$ of machine $PushChannel$ has been promoted to the interface of machine $Router$. Since, method $ProcChangeCstartAndDstart()$ can be used by an object of class $Module\_abcd$, the declarations $Router\ b$, $PushChannel\ inN$ and $void\ ProcChangeCstartAndDstart()$ are all to be declared *public*. This has been shown in Figure 7(b).

- To show that a test case is correct we need oracle information for which we introduce, for each operation, a set of probe operations to the model [24]. For instance, consider the operation *TransferData_a_b* in Figure 3. When this operation gets executed, the data object at the south output channel of router a propagates to the north input channel of router b. Before this operation is invoked, the probe operation *d_pre* ← *probe_aoutSdend* checks the data object at the south output channel of router **a** and produces its value as the result. And the probe operation *d_post* ← *probe_TD_binNdstart* checks, after the invocation of *TransferData_a_b*, the data object at the north input channel of router b and outputs this as its result. Here *d_pre* and *d_post* are result parameters of the respective operations and BAS supports them.

  Our method associates a *probe template* with each (non-probe) operation of the model. Observe the probe template of operation *TransferData_a_b* in Table 1(a). It tells to invoke *d_pre* ← *probe_aoutSdend* before the above operation and to invoke *d_post* ← *probe_TD_binNdstart* after the operation. And it should be checked that the results are equal. This probe template is translated to its Java counterpart as shown in Table 1(b). In the table, $Module\_abcd$ is the name of a class corresponding to the B machine $Module\_abcd.mch$, and MM is an object of this class.

  In summary, some of the probe operations – called pre-state probe operations – gather state information prior to the operation call and the rest – called post-state probe operations – gather the same after the call. Thereafter a sequence of assertions check the inter- and intra-relationship between the results of the pre- and post-state probe operations.

  It is to be noted that these probe operations are to be added to the BAS model, possibly by the model- or the test-engineer. Thus such operations are present in the Java template, and therefore, the implementer also implements them with matching signatures.

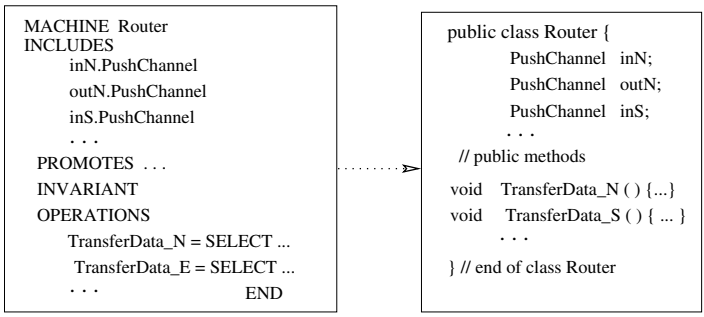

Fig. 6. Correspondence between a B machine and its class in implementation.

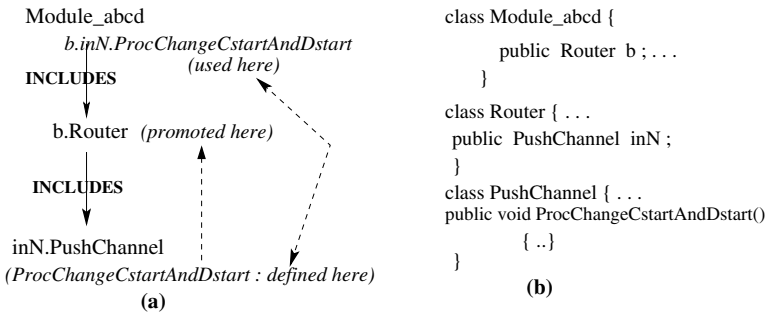

Fig. 7. (a) Proc. *ProcChangeCstartAndDstart* of machine *PushChannel* is promoted to interface of *b.Router*, (b) Appropriate methods become public in the corresponding Java template

5 Test case generation

ProB is a model checking and animation tool for B machines [18]. ProB includes a fully automatic animator written in SICStus prolog [27]. ProB also contains a test case generation facility called ProTest [24]. ProTest takes an instantiated model in B in which the generic set $DATA$ has been instantiated to the set $\{d1, d2, d3, d4\}$. ProB takes the instantiated BAS model and generates a finite coverage graph (in our case, it is 10,000 nodes). ProTest then uses the Dijkstra's shortest path algorithm to generate a set of test cases with the aim of covering all the operations. Each trace is a sequence of B action invocations, and the very first call originates from the initial state. ProTest obtained 8 traces in this graph, the average path length being 20. The test cases were abstract because they just contained operation invocations. One such abstract test case has been shown in Table 2. Figure 8 shows a part of the state space graph produced by the ProB animator.

Next a concretization operation uses the operation templates (refer to Table 1) and obtains concrete test cases; concretization amounts to substituting each operation invocation with corresponding Java template in this table.

The last phase of ProTest takes the concrete test cases and obtains a test driver in Java. The test driver for the single test case in Table 2 has been shown in Appendix A. This test driver is made to run along with the implementation, and if it runs without any assertion violation, then it would mean that the SUT has passed the test case.

```
a.inN.TransferData; a.TransferData_N;a.outE.TransferData;
a.outW.TransferData; a.outS.TransferData; d.inN.TransferData;
d.TransferData_N; d.outS.TransferData; d.outW.TransferData;
d.outE.TransferData; TransferData_a_b; TransferData_d_c;
b.inN.TransferData; b.TransferData_N; c.inN.TransferData;
c.TransferData_N; b.outE.TransferData; c.outW.TransferData;
CheckCycling_b_c;
```

Table 2
A sequence of B action invocations as a test trace.

Refer to the test driver shown in Appendix A. This Java program has a manual part and an automatic part. The automatic part is generated automatically by the test driver generator. Everything cannot be generated automatically because in that case, we must have knowledge of the signatures of the class constructors even before the code is written. In the manual part essentially the global objects are created. These issues have been discussed in [25]. Here, we have considered automatic testing of a single test case. The same can be repeated for the remaining test cases.

5.1 Coverage analysis

The BAS model of the Router component has altogether 23 operations. All such operations were covered by the eight test cases. Note here that, we have considered a simple testing criterion. For a complex test criterion, achieving full coverage could be a challenging task.

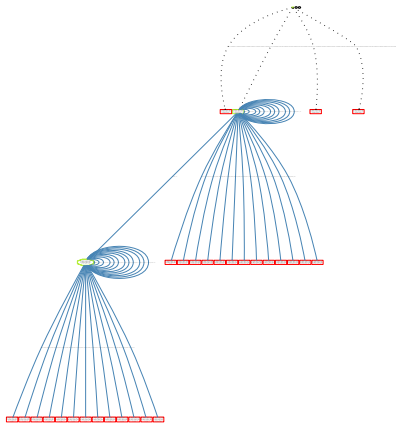

Fig. 8. Part of the coverage graph generated by ProB; only transitions for 5 states have been shown.

6 Problem of non-determinism

Consider the test case in Table 2. This is a trace from the initial state and the last operation in the sequence is *CheckCycling_b_c*. The function of this operation can be explained as follows. If only data variables *a.inN.dstart* and *d.inN.dstart* of the northern input channels of routers *a* and *d*, respectively, are initialized to equal

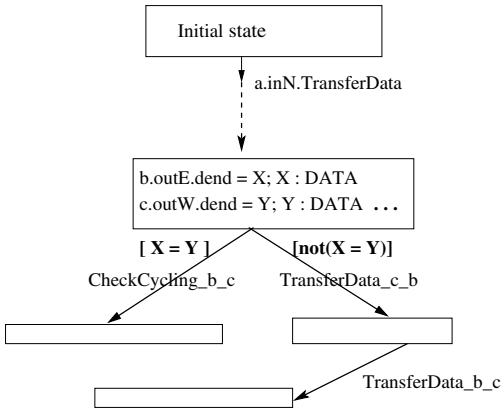

Fig. 9. Handling of the non-determinism

```
DATA X,Y;
X = a.inN.get_dstart(); Y = d.inN.get_dstart(); ...
if (X = Y){
   MM.CheckCycling_b_c(); ... }
else{
   MM.TransferData_b_c();MM.TransferData_c_b();
   ... }
```

Table 3
An adaptive test case

values then when control arrives at operation $CheckCycling\_b\_c$, its guard will be true. Therefore, unless $a.inN.dstart$ and $d.inN.dstart$ are given equal initial values, operation $CheckCycling\_b\_c$ will not occur in the state space graph and so it cannot be tested. A better way to specify is to give non-deterministic values by the non-deterministic assignments: $a.inN.dstart :\in DATA$ and $d.inN.dstart :\in DATA$.

So if $DATA = \{d1, d2, d3, d4\}$ then each of $a.inN.dstart$ and $d.inN.dstart$ can receive any value in the above set. It is not known in advance which values the implementation would receive and since test cases are derived from the specification, the test driver must prepare itself for all possibilities. Testing in this case can use the solution discussed in [25]. Refer to Figure 9. While generating the state space graph, variables $a.inN.dstart$ and $d.inN.dstart$ are given unconstrained values denoted by the variables X and Y, and additionally the restrictions that $X, Y \in DATA$. So when control reaches operation $CheckCycling\_b\_c$ in the implementation, both the possibilities that $(X = Y$ or $not(X = Y))$ are possible and that is why we create two branches for the two possibilities. Table 3 shows the corresponding test driver code fragment. The two temporary variables X and Y capture the values of $a.inN.dstart$ and $d.inN.dstart$ respectively. Before the call to $CheckCycling\_b\_c$, the if statement checks whether the values are equal or unequal. Depending on the case, the appropriate if-branch is selected; thus the test case is an *adaptive* one.

The problem like covering the call to $CheckCycling\_b\_c$ can also be solved by following the under-constrained execution of Engler & Dunbar [12]. The symbolic

constraint of a desired path can be solved, and appropriate initial assignments can be derived. Such initialization can take control to the desired path. For the present, it could be derived that $a.inN.dstart$ and $d.inN.dstart$ both need identical values.

7 Discussion

- In this paper, we have considered models in BAS which can be hierarchical. Previous works [4,24] considered only flat B models. For our testing approach to succeed, we expect that the implementer must not deviate from the structure of the test template derived from the BAS. This means the implementer must preserve the hierarchy and the action (and procedure) signatures; in addition, the recommendations associated with a promoted operation must also be preserved. Furthermore, the implementation must have the probe operations with matching signatures. Note that the implementation is treated as a black box.
- We have discussed automatic testing; however a test engineer has to do some manual tasks. Appropriate probe operations are to be added to the model if the existing ones are not adequate. The only other manual part is the creation of a global context (the marked top portion in Appendix A). This part is trivial since only the global objects are created. Note that the class constructors are defined by the implementer but not by the model engineer.
- Following the principle of the B Method, one would expect that the development process is completely formal and then no testing will be necessary. However, it is not the case that a formal refinement process is always followed. In many cases, the specification and possibly the first few refinements undergo the formal refinement process and for the rest, model based testing is recommended. Moreover, sometimes model based testing is recommended before performing any refinement proof between the abstract and refined models. For testing purposes, we can view the refined model as the implementation. This activity can potentially cut down the time for performing proofs considerably.
- Functional test cases can also be generated by using model checking [16]. ProB tool can generate a finite state space of the hierarchical B model. A test criterion like state or transition coverage can mean reachability of the state or transition; trap variables can be attached to the state or transition, and reachability of the trap variables means satisfying a LTL formula. Next LTL model checking [8] can be used to find an appropriate trace which can be used as a test case.
- We have assumed that our implementation language is Java. However, similar testing activities can be used when the implementation languages is C, C++ or SystemC. If the language is C, what was a class declaration in Java becomes a `structure` in C. Nested classes now become nested structures. But the temporary variables which were object references in Java become pointers to structures in C. Refer to the first few lines of the automatically generated portion of the test driver in `Appendix A`. Those lines in relation to a C implementation become:

```
struct DATA *temp1,*temp2,*temp3,*temp4;
temp1 = probe_TD_dstart(MM.a.inN); TransferData(MM.a.inN);
temp2 = probe_TD_dend(MM.a.inN); assert(temp1 == temp2);
```

- We have considered a simple testing criterion like testing each operation instance at least once. However, our method can handle other complicated testing criteria like transition-pair coverage or MC/DC coverage.
- In case of non-determinism in the specification, the test cases must be adaptive, and hence a test case must be in the form of a decision tree. A decision branch is translated to an if-branch in the test driver code.

8 Conclusion

Tsiopoulos and Waldén in [29] have presented the specification of a router component as a hierarchical BAS. We have analysed this system from Model Based Testing point of view. By bounding the state variables in the specification, we discussed how a state space graph is generated by using the ProB model checking and animation tool. Depending on a test criterion, a set of traces are generated as test cases. The test cases are converted to a test driver code in a programming language. The major contributions of our work are: (a) handling of hierarchy in the model, and (b) automatic derivation of a test driver in presence of hierarchy and non-determinism. In the process, we have presented an approach to Model Based Testing of formal NoC frameworks.

Formal methods with adequate tool support are important for the design of complex NoC systems in order to correct errors in the early design phases and reduce the involved costs of NoC system design and development. For ensuring that a possible NoC implementation is consistent to its formal specification, Model Based Testing could play a prominent role.

Acknowledgement

We would like to thank Professor Kaisa Sere, Marina Waldén and Pontus Boström for fruitful discussions.

References

[1] Abrial J.-R. (1996). *The B–Book*, Cambridge University Press.

[2] Abrial, J.-R., Hallersted, (2006). Refinement, Decomposition and Instantiation of Discrete Models, Fundamentae Informatica.

[3] Back, R.J.R., Kurki-Suonio, R. (1983). Decentralization of Process Nets with Centralized Control, Proc. of the 2nd Symposium on Principles on Distributed Computing, pp. 131-142.

[4] Bernard E., Legeard B., Luck X., Peureux F.(2004). Generation of test sequences from formal specifications *Software Practice and Experience*, Volume 34 (10) , pp. 915 - 948.

[5] B-Core(UK) Ltd, B-Toolkit, http://www.b-core.com/btoolkit.html

[6] Bjerregaard, T., Mahadevan, S. (2006). A Survey of Research and Practices of Network-on-Chip, ACM Computing Surveys (CSUR), Volume 38 (1).

[7] Butler, M., Waldén, M. (1996). Distributed System Development in B, Proc. of the 1st conference on the B Method, Nantes, France, pp. 155-168.

[8] Clarke, E.M., Grumberg, O., Peled, D.A. (2000). *Model Checking*, MIT Press.

[9] ClearSy, Atelier B, http://www.atelierb.societe.com/

[10] Dally, W. J. and Towles, B. Route packets, not wires: On-chip interconnection networks. In Proc. of the DAC'01, pp. 681 - 689, 2001.

[11] Dick, J.; Faivre, A. (1993). Automating the Generation and Sequencing of Test Cases from Model-based Specifications, Proc. of the FME'03, LNCS 670, 1993, pp. 268–284.

[12] Engler, D., Dunbar, D. (2007). Under-constrained execution: making automatic code destruction easy and scalable, ACM ISSTA'07, pp. 1-4.

[13] ETSI.(1995). ETS 300 406: Methods for Testing and Specification (MTS); *European Telecommunication Standard*.

[14] Gannon, J.D., Hamlet R.G., Mills, H.D. (1987). Theory of modules, IEEE Transactions on Software Engineering, 13(7):820–829.

[15] Gurevich, Y. (2000). Sequential Abstract-State Machines Capture Sequential Programs, ACM Transaction on on Computational Logic, Vol 1(1): 77–111.

[16] Hamon, G., de Moura, L, Rushby, J.(2005). Automated Test Generation with SAL, CSL Technical Note, January 2005.

[17] Jones, C.B. (1990). Systematic Software Development using VDM (2nd Edn), Prentice Hall.

[18] Leuschel, M., Butler M. (2005). ProB: A Model Checker for B, Proc. FME'03, LNCS Volume 2805, Springer, pp. 855–874.

[19] Misra J. (2001). A Discipline of Multiprogramming, Springer.

[20] Plosila, J., Sere, K., Waldén, M. (2005). Asynchronous System Synthesis, Science of Computer Programming, Vol. 55(1–3), pp. 259–288.

[21] RODIN. RODIN project homepage, http://rodin.ncl.ac.uk/.

[22] Safrr, S.H., Bennour, I.E., Tourki, R. (2006). Transaction Level Modeling of an OSI-like layered NoC, Proc. of Design and Test of Integrated Systems in Nanoscale Technology (DTIS'06).

[23] SRI International. SAL home page http://sal.csl.sri.com

[24] Satpathy, M., Leuschel, M., Butler,M. (2005). ProTest: An Automatic Test Environment for B Specifications, *ENTCS*, Vol. 111, pp: 113–136.

[25] Satpathy M., Butler M., Leuschel M., Ramesh S.(2007). Automatic Testing From Formal Specifications, Tests and Proofs (TAP-07), LNCS Volume 4454, Zurich, Feb 2007.

[26] Schmaltz, J. and Borrione, D. A Formal Approach to the Formal Specification of Networks on Chip. In Proc. of the FMCAD '04, pp. 52 - 66, 2004.

[27] SICS. (2006). SICStus Prolog, website: http://www.sics.se/sicstus

[28] Spivey, J.M. (1988). *Understanding Z*, Cambridge University Press.

[29] Tsiopoulos, L., Walden, M. (2006). Formal Development of NoC Systems in B, Nordic Journal of Computing, Vol. 13 (2006). pp. 127–145

[30] Utting, M., Legeard, B. (2007). *Practical Model-Based Testing*, Morgan Kaufmann.

[31] Waldén, M., Sere, K. (1998). Reasoning about Action Systems using the B Method, Formal methods in System Design, Vol. 13(1), pp. 5–35.

[32] Yound J.S., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., Newton, A.R. (1998). Design and Specification of Embedded Systems in Java using Successive, Formal Refinement, Design Automation Conference (DAC'98), Hune 1998.

[33] Zhu, H., Hall P.A.V., May J.H.R. (1997). Software Unit Test Coverage and Adequacy, *ACM Computing Surveys*, 29(4):366–427.

Appendix

Outline of the test driver for test case in Table 2. *temp1,.., temp4* are temporaries to hold probe results. Code up to the `global context` is written by test engineer; rest is generated automatically.

```
import java.io.*;
public static lass MainClass{
  DATA d1,d2,d3,d4;
  PushChannel ainn,ains,aine,ainw,bine,binw,binn,bins . . .
  Router ar,br,cr,dr; Module_abcd  MM;
  public static void main(String[] args){
  d1=new DATA(); d2=new DATA(); d3=new DATA(); d4 =new DATA();
  ainn = new PushChannel(d2,d1); ains = new PushChannel(d1,d1);
            . . .
  ar=new Router(ainn,ains,aine,ainw,aoutn,aouts,aoute,aoutw);
  br=. . .; cr= ...; dr= . . .;
  MM=new Module_abcd(ar,br,cr,dr);
/** global context ends  **/

/* the following part is autometically generated */
DATA  temp1,temp2,temp3,temp4;  // temporary variables
temp1 = MM.a.inN.probe_TD_dstart();
        MM.a.inN.TransferData();
temp2 = MM.a.inN.probe_TD_dend();
        assert(temp1 == temp2);
temp1 = MM.a.probe_TD_N_inN_dend();
        MM.a.TransferData_N();
temp2 = MM.a.probe_TD_N_outS_dstart();
        assert(temp1 == temp2);
           . . .
           . . .
temp1 = MM.c.outW.probe_TD_dstart();
        MM.c.outW.TransferData();
temp2 = MM.c.outW.probe_TD_dend();
        assert(temp1 == temp2);
        MM.CheckCycling_b_c();
flag1 = MM.b.outE.probe_cend();
flag2 = MM.c.outW.probe_cend();
        assert(flag1 == flag2);
        System.out.println("Testcase succeeded.");
} }
```