A Tool for Equivalence Verification of TLMs using Model Algebra

Lochi Yu, Samar Abdi, Daniel Gajski

UCI Technical Report TR-08-04
Feb. 28, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8919

{lochi.yu,sabdi,gajski}@uci.edu

Abstract

This report presents a tool for functional equivalence verification of transaction level models (TLMs). The tool is based of the theory of Model Algebra which provides the objects and composition rules needed to abstract TLMs into symbolic expressions. Synthesis and verification of TLMs is made possible by the manipulation of these symbolic expressions using the transformation rules of Model Algebra. We define the verification problem in the context of platform based design. Functional TLMs that are refined as a result of design optimization are verified against original TLMs for consistency. We describe the Application Program Interface for the Model Algebrabased verification tool. The tool is composed of two main parts: the model creation component and the verifier component. The first API allows the user to abstract TLMs into model algebraic expressions, while the second API allows the application of different transformations on the expressions as well as algorithms for automatic equivalence checking for certain refinements.

Contents

List of Figures

A Tool for Equivalence Verification of TLMs using Model Algebra

L. Yu, S. Abdi, D.Gajski
Center for Embedded Computer Systems
University of California, Irvine

Feb. 28, 2008

1 Introduction

With the growing complexing of modern systems, the abstraction level for specifying such systems has moved above RTL. The so called transaction level modeling (TLM) approach is being increasingly adopted to cope with the amount of software and heterogeneous cores in current systems. However, using TLM exclusively for modeling without giving thought to synthesis and verification is a trap that must be avoided. For TLMs to be synthesizable and verifiable, well defined TLM semantics are required. Existing formalisms for RTL design such as FSDM or boolean algebra are insufficient to express TLMs. Therefore, new formalisms for TLM based design are needed. Furthermore, the TLM semantics must allow simulation models written in languages like SystemC [1] to be easily abstracted into mathematical expressions for symbolic manipulation.

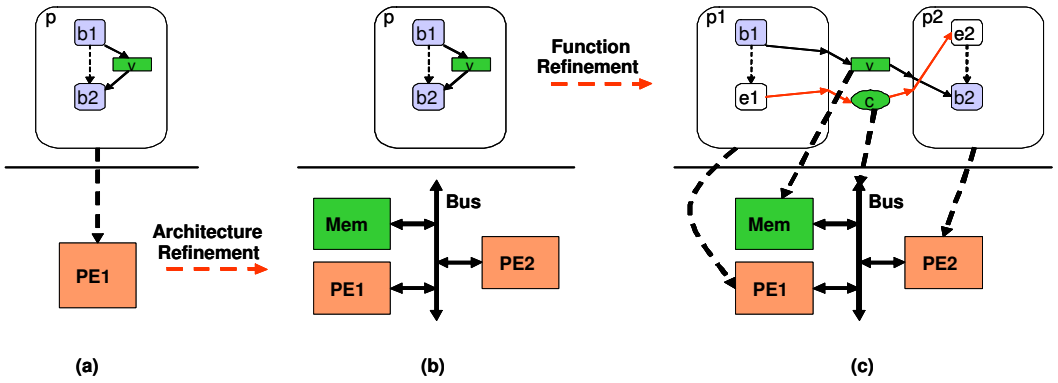

Figure 1: Architecture and functional refinement in platform based design.

Model Algebra [3][2] is one such formalism that can be used for refinement and verification of TLMs. As part of the GSRC initiative on core design technologies, Model Algebra based TLM

verification is being applied to the Platform based Design [5] paradigm as implemented in the Metropolis II [4] framework. Figure 1 shows a simple design optimization step in the platform based approach. The basic concept is the separation of functional and architectural modeling. Executable models are defined in the functional space which platform netlists are defined in the architecture space. Subsequently, a mapping is defined from the functional to architecture space, that allows designers to evaluate useful metrics. A key assumption is a many to one mapping from the function space to the architecture space. This constraint is neccessary to produce an unambiguous design.

Figure 1(a) shows a simple mapping of a sequential composition of two functional objects (called behaviors) onto a processing element (PE) in the architecture. Now, assume that the designer figures that this mapping does not produce a satisfactory execution time. So, he or she may select another PE (PE2) that is optimized for behavior *b2*. The new architecture is shown in Figure 1(b). However, in this new function and architecture specification, there is no feasible mapping. This is because a sequential composition cannot be mapped to a concurrent architecture. Therefore, the function must now be refined to the one shown in Figure 1(c) by isolating *b2* into a concurrent process. A synchronization channel is added to keep the execution order between *b1* and *b2*. This new refined functional model is now mappable to the refined architecture.

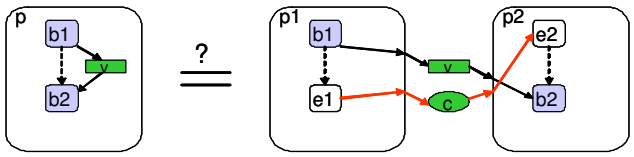

Figure 2: TLM equivalence verification problem.

As we saw in the optimization example above, functional models may need to be refined every time the architecture netlist is modified. It is imperative that each such refinement be functionality preserving. This poses the TLM functional equivalence problem as illustrated in Figure 2. The problem is to verify that any functional refinement produces a TLM that is functionally equivalent to the original TLM. In this paper we propose such a tool based on Model Algebra that verifies equivalence of two well formed TLMs. By well formed, we mean that the TLMs must follow the semantics of the objects and composition rules of Model Algebra. The theory of Model Algebra has been published in [3]. Here, we present the specific application programming interfaces (APIs) that have been developed to facilitate construction of algebraic TLMs and to perform symbolic transformations on them in conformance with the rules of Model Algebra. We also present the usage of the verification tool and an explanation of the tool's graphical user interface (GUI).

2 Application Program Interfaces

2.1 Class Structure

Our models are composed of objects and the relationships between them: Behaviors, Control Dependencies, Data Dependencies, Ports, Variables, Links and Channels. For each of these objects, there exists a class with different attributes and methods. In order to construct the relationships between them, and set those attributes, the tool offers the class *Design*. Once the model is constructed, the class *Verifier* is used to transform it and check its equivalency with another model.

2.2 Model Algebra's objects

Each of the following classes contain a set of attributes which are written and read by the following way:

- Class: ABC
- Instance name: _abc
- Attribute: xyz
- Write attribute: _abc→set_xyz(char * value)
- Read attribure: char * value = _abc→get_xyz(void)

In addition to these attributes there can be a sequence of other objects, which can be accessed by the following way:

- Sequence: mno
- Reset counter: _abc→set_mno_index(0)
- Get next object: mno * _mno = _abc→get_mno_next()
- Get a particular object: mnp * _mno = _abc→get_mno_idx(int number)
- Add an object: _abc→add_mno_element(class mno *)
- Delete an object: _abc→del_mno_element(int number)

These objects must be added to an instance of the class *Design*.

2.2.1 Behavior

Attributes:

- id: name of the instance
- parent: name of the parent behavior
- type: type of behavior (hierarchical, identity, leaf)
- vsp: virtual starting point identity behavior
- vtp: virtual terminating point identity behavior

Sequences:

- behavior: subbehaviors inside this behavior
- variable
- condition
- channel
- link
- port
- cd: control dependencies
- dd_var_nb_read: data dependency non-blocking read from a variable
- dd_var_nb_write: data dependency non-blocking write to a variable
- dd_port_nb_read: data dependency non-blocking read from a port
- dd_port_nb_write: data dependency non-blocking write to a port
- dd_port_b_read: data dependency blocking read from a port
- dd_port_b_write: data dependency blocking write to a port
- dd_ch_nb_read: data dependency non-blocking read from a channel
- dd_ch_nb_write: data dependency non-blocking write to a channel

Methods: All objects should be constructed inside a 'top' behavior. In order to save the design, the following function should be used:

- void print_to_file(FILE * filename, int ident): the file pointer should be provided, as well as the starting indentation of the behavior (use 0). The extension of the filenames used in this tool is *.mag*.

2.2.2 Variable

Attributes:

- id: name of the instance
- parent: name of the parent behavior

2.2.3 Condition

Attributes:

- id: name of the instance
- parent: name of the parent behavior
- type
- name

2.2.4 Channel

Attributes:

- id: name of the instance
- parent: name of the parent behavior

Sequences:

- link

2.2.5 Link

Attributes:

- id: name of the instance
- parent: name of the parent behavior
- sender
- receiver

2.2.6 Port

Attributes:

- id: name of the instance
- parent: name of the parent behavior
- direction: direction of the data flow
- portmap: mapping to a variable or another port

2.2.7 Control Dependency

Attributes:

- id: name of the instance
- parent: name of the parent behavior
- condition: condition to which this control dependency maps
- predecessors: behaviors that precede this control dependency
- successor: behavior that follows this control dependency

2.2.8 Data Dependencies

The data dependencies classes are differentiated by the direction of the data flow, type and object: read/write, blocking/nonblocking, variable/port/channel. Attributes:

- id: name of the instance
- parent: name of the parent behavior
- address
- source
- destination

2.3 Model Input API

2.3.1 Basic functions

ERROR addBehavior(behavior *be, char *id, behavior *parent) Adds a previously instantiated behavior to the design. If it's added as a subbehavior, the parent must be specified; otherwise, it should be NULL.

ERROR addHierBehavior(behavior *be,char *id,behavior *parent) Same as addBehavior, except that it calls add_vsp_vtp (see below) by default.

ERROR behaviorExists(char *id) Returns NO_ERROR if the specified behavior exists in the design. Returns ERROR_BEHAVIOR_NOEXISTS otherwise.

behavior* getBehavior(char *id) Returns the pointer to a behavior given its ID.

ERROR addIdentity(behavior *be,behavior *parent) Adds an Identity Behavior as a subbehavior to a parent. The ID is chosen by using an internal counter.

ERROR addCD(behavior*be, char *id, list⟨behavior*⟩ predecessors, behavior *successor) Creates a Control Dependency object, given the parent behavior, its ID, a list of predecessing behaviors, and a successor behavior.

ERROR addCD(behavior*be, list⟨behavior*⟩ predecessors, behavior *successor) Creates a Control Dependency object, given the parent behavior, a list of predecessing behaviors, and a successor behavior. Its ID is chosen using the character 'q' followed by a number taken from an internal counter.

ERROR addCD(behavior*be, char *id, list⟨behavior*⟩ predecessors, behavior *successor,variable *var) Creates a Control Dependency object which depends on a variable, given the parent behavior, its ID, a list of predecessing behaviors, a successor behavior and the variable which it depends on.

ERROR addCD(behavior*be, list⟨behavior*⟩ predecessors, behavior *successor,variable *var) Creates a Control Dependency object, given the parent behavior, a list of predecessing behaviors, a successor behavior and the variable which it depends on.

ERROR addCD(behavior*be, char *id, list⟨behavior*⟩ predecessors, behavior *successor, port *pt) Creates a Control Dependency object which depends on a port, given the parent behavior, its ID, a list of predecessing behaviors, a successor behavior and the port which it depends on.

ERROR addTrueCD(behavior*be,behavior *predecessor, behavior *successor) Creates a Control Dependency object which is always true, given the parent behavior, a list of predecessing behaviors, and a successor behavior.

void add_vsp_vtp(behavior *be) Adds the Virtual Starting Point and Virtual Terminating Point to a given behavior.

behavior *getvsp(behavior *be) Returns the pointer to the VSP behavior, given its parent.

behavior *getvtp(behavior *be) Returns the pointer to the VTP behavior, given its parent.

ERROR addPort(behavior *be,char *portname,char *direction, char *map) Adds a port to a behavior, given its name, direction and if it's mapped to another port or variable.

port *getPort(behavior *be, char *portname) Returns the pointer to a port given its name and parent.

variable * addVar(behavior *be,char *varname) Adds a varialbe to a behavior, given its name and parent.

variable *getVar(behavior *be,char *varname) Returns the pointer to a variable given its name and parent.

ERROR addDDVNBW(behavior *be, port *sourceport,variable *destvar) Adds a data dependency - non blocking write to a variable, given its parent, source port and destination variable pointers.

ERROR addDDVNBW(behavior *be,char *id, port *sourceport,variable *destvar) Adds a data dependency - non blocking write to a variable, given its parent, ID, source port and destination variable pointers.

ERROR addDDVNBR(behavior *be, variable *sourcevar, port *destport) Adds a data dependency - non blocking read from a variable, given its parent, source port and destination variable pointers.

ERROR addDDVNBR(behavior *be,char *id, variable *sourcevar, port *destport) Adds a data dependency - non blocking read from a variable, given its parent, ID, source port and destination variable pointers.

ERROR addDDPBW(behavior *be,char *id, port *source, port *destination) Adds a data dependency - blocking write to a port, given its parent, ID, source port and destination port pointers.

ERROR addDDPBW(behavior *be, port *source, port *destination) Adds a data dependency - blocking write to a port, given its parent, source port and destination port pointers.

ERROR addDDPBR(behavior *be,char *id, port *source, port *destination) Adds a data dependency - blocking read from a port, given its parent, ID, source port and destination port pointers.

ERROR addDDPBR(behavior *be, port *source, port *destination) Adds a data dependency - blocking read from a port, given its parent, source port and destination port pointers.

ERROR addChannel(behavior *be, channel *ch, char *id) Adds a channel given its parent and ID.

ERROR addDDCHW(behavior *be,char *id,port *sourceport, channel *destinationch) Adds a data dependency - write to a channel, given its parent, ID, source port and destination channel.

ERROR addDDCHR(behavior *be,char *id,channel *sourcech, port *destport) Adds a data dependency - read from a channel, given its parent, ID, source channel and destination port.

ERROR addLink(behavior *be,char *id,channel *ch,port *source, port *destination) Adds a link between a source and destination ports to a given channel.

ERROR addLink(behavior *be,channel *ch,port *source, port *destination) Adds a link between a source and destination ports to a given channel.

ERROR addDDPNBW(behavior *be,char *id, port *source, port *destination) Adds a data dependency - non blocking write to a port, given its parent, ID, source port and destination port.

ERROR addDDPNBW(behavior *be, port *source, port *destination) Adds a data dependency - non blocking write to a port, given its parent, source port and destination port.

ERROR addDDPNBR(behavior *be,char *id, port *source, port *destination) Adds a data dependency - non blocking read from a port, given its parent, ID, source port and destination port.

ERROR addDDPNBR(behavior *be, port *source, port *destination) Adds a data dependency - non blocking read from a port, given its parent, source port and destination port.

2.3.2 Complex functions

ERROR addHierBehaviorReturn(behavior *be,char *be_name,behavior *parent) This will add a hierarchical behavior, add the port that will write its return value and add the variable. The variable will be named 'behaviorname_result', and the corresponding data dependency will also be added. The output port will be named 'behaviorname_return_port'.

ERROR addBehaviorReturn(behavior *be,char *be_name,behavior *parent) This will add a behavior, add the port that will write its return value and add the variable. The variable will be named 'behavior_result', and the corresponding data dependency will also be added.The output port will be named 'behaviorname_return_port'.

void addReturnBeh(behavior *be,variable *result) Adds a behavior that reads a given variable, writes to its parent output port ('behaviorname_return_port') and goes to the VTP.

behavior * addReturnBeh(behavior *be,port *pt) Adds a behavior that will write to given port and goes to the VTP of its parent. It is basically a behavior that returns a constant.

void addB2Vrw(behavior *be, behavior *parent, variable *var) Will take a behavior and a variable and create the two ports, and two data dependencies (variable non blocking read and write).

void addB2Vr(behavior *be, behavior *parent, variable *var) Will take a behavior and a variable and create a port and a non blocking read.

void addB2Prw(behavior *be, behavior *parent, port *ptin, port *ptout) Will take a behavior and two ports, and create the two data dependencies (port non blocking read and write).

void addB2Pr(behavior *be, behavior *parent, port *ptin) Will take a behavior and a port and create a data dependency - non blocking read.

void addB2Pw(behavior *be, behavior *parent, port *ptout) Will take a behavior and a port and create a data dependency - non blocking write.

behavior * addRWId(behavior *parent, char *varread, char *varwrite) Will create an identity behavior that will read from a variable and write into another

void addCase(behavior *parent, behavior *initial, behavior *final) Adds a CASE statement: will create a behavior reading from a variable, write into another and proceed if true to another behavior. Will create the behavior, the CD, the DD and the second variable

void addIf(list⟨behavior*⟩ funcs,behavior *parent,behavior *startbeh, behavior *ifbeh, behavior *elsebeh) IF statement with just one function call //if (func(any parameter)==12) else //assumes the return variable named: func_return

void addFor(behavior *parent, behavior *be, behavior *loopstart, behavior *loopend, behavior *exitb) FOR statement, need to manually set the variable DDs

2.4 Verifier API

2.4.1 Control functions

int transform(behavior *, int enable) Takes a behavior and opens the graphical interface. The enable int can be 0 for no graphics displayed in the canvas, 1 for displaying just behaviors and control dependencies (no data dependencies nor variables), and 2 for full graphics.

int verify(behavior *, behavior *, int enable) Takes two behaviors and opens the graphical interface. The 'Check Equivalency' icon will be enabled (see below). The enable int can be 0 for no graphics displayed in the canvas, 1 for displaying just behaviors and control dependencies (no data dependencies nor variables), and 2 for full graphics.

void printStats(behavior *) Prints the statistics of rules applied. See below in 'Statistics Icon'.

2.4.2 Transformation functions

STATUS flatten(behavior *be)

STATUS flattenChannels(behavior *be)

STATUS identityElimination(behavior *be)

STATUS controlElimination(behavior *be)

STATUS controlRelaxation(behavior *be)

2.4.3 Examples

3 Tool Usage

3.1 Calling the Tool

There are several options when calling *tlmver*: either using the graphical interface to visualize and decide each transformation, or automatically call the transformation rules with no graphical output. The option *-display* will invoke the graphical tool (*uDrawGraph*); if not given, the default is to print the final results in the standard output. The options are:

$ tlmver -display file.mag Will load the model saved as *file.mag* and display it using *uDrawGraph*. The transformation rules can be applied using the enabled icons.

$ tlmver -reduce file.mag -o reduced.mag Will load the model saved as *file.mag* and reduce it using all the transformation rules. When the transformation rules can no longer reduce the model, it will be saved as *reduced.mag*.

$ tlmver -model file.mag -ref reference.mag Will load two models, take the one in *file.mag* and apply the transformation rules automatically. When the transformation rules can no longer reduce the model, it will check for equivalency with the model in *reference.mag*. Note: the model in *reference.mag* will not be transformed before the equivalency check.

$ tlmver -display -model file1.mag -ref file2.mag Will load both models and display them both using *uDrawGraph*. The reference model will be shown in a smaller window, and the first model will be the only one that can be transformed using the icons.

$ tlmver -help This will display the help message with the different options of the tool.

3.2 Using the Graphical Interface

The graphical interface is handled by *uDrawGraph* which is released under LGPL.

When it opens, the screen will show as Figure 3.

The graphical user interface is composed of basically 2 parts: the canvas and the control panel on the left.

Control Panel This panel is composed of these icons:

AUTO This applies the previous rules iteratively until all of them fail to produce a transformation.

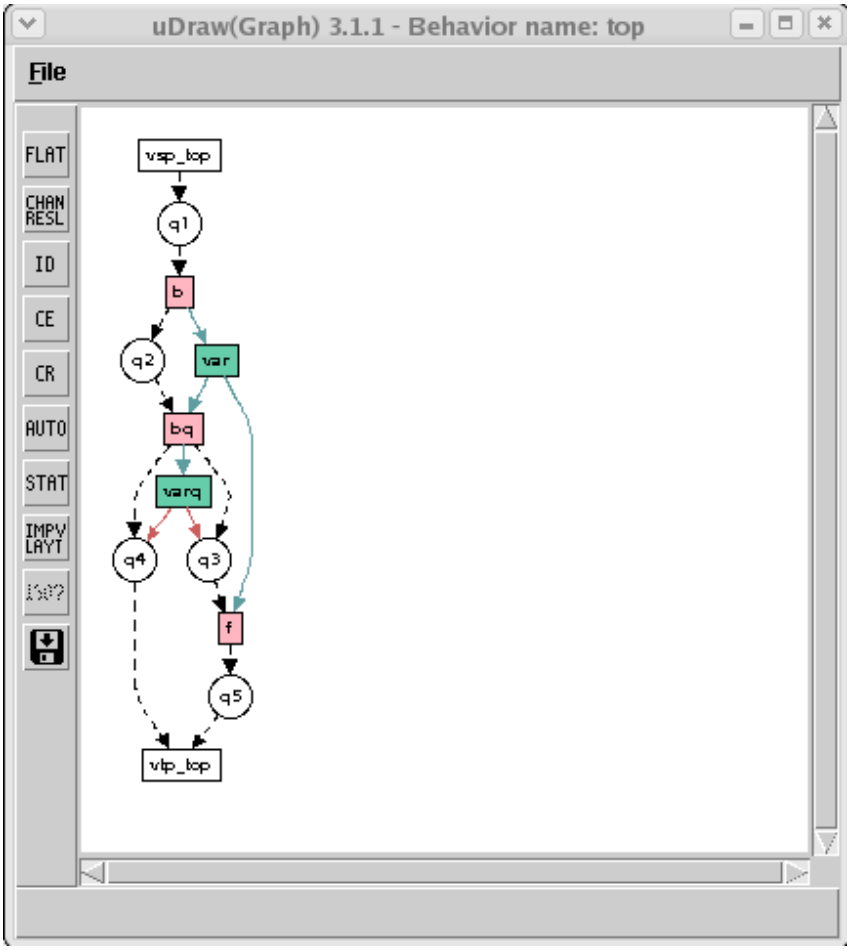

Figure 3: Initial screen

Figure 4: Flattening Rule Icon

Figure 5: Channel Resolution Icon

Figure 6: Identity Elimination Rule Icon

Figure 7: Control Elimination Rule Icon

Figure 8: Control Relaxation Rule Icon

Figure 9: Automatic Rule Application Icon

Figure 10: Print Statistics Icon

Statistics Prints the summary of all rules applied, the number of times it was applied, the order of the transformations, the total transformation time, and graphical statistics (number of objects and relationships).

Figure 11: Improve Layout Icon

Layout Since objects and edges can be dragged and the transformation rules themselves will delete objects and create new ones, the canvas may be disorganized after a few transformations, especially in complex models. In order to adjust the canvas to further optimize the use of its space and be able to visualize better the model, this icon will reorganized all objects and edges the best way it can.

Figure 12: Check for Isomorphism Icon

Isomorphism This icon is only enabled when a model and a reference model are loaded into the design. Will check both models for isomorphism. Does not apply any transformation on any model.

Figure 13: Save and Quit Icon

Saving and Quitting This will take the model being transformed and save it under the name of the top behavior's name plus the string "_reduced.mag".

Canvas The canvas is the whitespace where all objects and relations are displayed. All objects and edges in the canvas can be dragged, but no addition nor deletion can be made. All modifications of the model should be made either in its creation phase (*design* class) or using the transformation icons (which uses the *verifier* class).

The representation of the objects is shown in the Table 1.

Table 1: Legend for canvas' objects

Object	Representation
Behavior	Single Frame Rectangle
Hierarchical Behavior	Double Frame Rectangle
Identity Behavior	Red Rectangle
Control Dependency	Circle
Channel	Light Blue Oval
Link	Bold Blue Line
Data Dependency	Green and Red Lines
Variable	Green Rectangle
Control Flow	Dashed Line

3.3 Creating Models: an example

Starting from C code, the user is able to create a representation in Model Algebra. For the following C code, we need to construct the model.

```
int v;
f1(&v);
if (v<CONSTANT){
  f2(&v);
}
```

The corresponding model is created by the following code, which is in the file *model1.cc* in the directory *examples/api_tests/model1.cc*.

```
//create new behavior
design *des = new design();
behavior *top = new behavior;
des->addBehavior(top,"top",NULL);
des->add_vsp_vtp(top);

list<behavior*> predecessors;

behavior *b = new behavior;
des->addBehavior(b,"b",top);
predecessors.push_front(des->getvsp(top));
des->addCD(top,"q1",predecessors,b);
predecessors.clear();
des->addPort(b,"b_out","OUT","port");
behavior *bq= new behavior;
des->addBehavior(bq,"bq",top);
predecessors.push_front(b);
des->addCD(top,"q2",predecessors,bq);
predecessors.clear();
des->addPort(bq,"bq_in","IN","port");
```

```
des->addPort(bq,"bq_out","OUT","port");
des->addVar(top,"var");
des->addDDVNBW(top,"bwr",des->getPort(b,"b_out"),des->getVar(top,"var"));
des->addDDVNBR(top,"bqr",des->getVar(top,"var"),des->getPort(bq,"bq_in"));

des->addVar(top,"varq");
des->addDDVNBW(top,"bqw",des->getPort(bq,"bq\_out"),des->getVar(top,"varq"));
behavior *f = new behavior;
des->addBehavior(f,"f",top);
des->addPort(f,"f_in","IN","port");
predecessors.push_front(bq);
des->addCD(top,"q3",predecessors,f,des->getVar(top,"varq"));
des->addCD(top,"q4",predecessors,des->getvtp(top),des->getVar(top,"varq"));
predecessors.clear();

des->addDDVNBR(top,"rd",des->getVar(top,"var"),des->getPort(f,"f\_in"));

//add control dependencies
predecessors.push_front(f);
des->addCD(top,"q5",predecessors,des->getvtp(top));
predecessors.clear();

FILE *output;
output=fopen("model1.mag","w");
top->print_to_file(output,0);
fclose(output);
```

3.3.1 Program explanation

In lines 2 to 5, a new design is created and a top behavior is created and added into the design. The Virtual Starting and Terminating Points are added to the top behavior as well. In lines 9 to 12, a behavior 'b' is created and inserted into the top behavior. Control flow is specified by the use of a list of predecessor behaviors (in this case only the VSP of 'top'); so that the VSP flows into the newly created behavior 'b'. A port is inserted into 'b'. Another behavior 'bq' is created, inserted into 'top' , its ports and control flow specified (lines 15 to 20). The control flow would go from the VSP, to 'b' and then to 'bq'. A variable 'var' is created, and data will be written from the output port in 'b' and read from the input port in 'bq' (lines 23 and 24). Another variable 'varq' will be written by 'bq' (lines 26,27). There are two more Control Dependencies 'q3' and 'q4' which direct the flow depending on the value in 'varq'; it may go either to the behavior 'f' or the VTP of 'top' (lines 32 and 33). Finally, a behavior 'f' will read 'var' and finish in the VTP of 'top'(named *vtp_top* by the function *add_vsp_vtp*). The resulting model is shown in Figure 1 below.

3.3.2 Example: Modeling a CASE statement

The previous example showed how to model an IF statement. For a Case statement, there are some complex functions that can be used to speed up the model creation. For instance, for the following code:

```
switch (error(error_data, stream, frame)) {
   case MAD_FLOW_STOP:
      goto done;
   case MAD_FLOW_BREAK:
      goto fail;
   case MAD_FLOW_IGNORE:
      break;
   case MAD_FLOW_CONTINUE:
   default:
      continue;
}
```

The API calls could be:

```
behavior *mad_recoverable = new behavior;
d->addBehaviorReturn(mad_recoverable,"mad_recoverable",run_sync);
d->addB2Vrw(mad_recoverable,run_sync,stream);
pred.push_front(mad_recoverable);
d->addIf(pred,run_sync,id2,error,end);
pred.clear();
d->addCase(run_sync,error,mad_synth_finish);
d->addCase(run_sync,error,fail);
d->addCase(run_sync,error,error);
d->addCase(run_sync,error,end);
```

The tool would represent this model as shown in Figure 14.

4 Acknowledgments

This work was supported in part by the Gigascale Systems Research Corporation (GSRC) under its Heterogeneous Systems Design pillar (Task 1.4.3.2).

References

[1] SystemC, OSCI[online]. Available: http://www.systemc.org/.

[2] S. Abdi and D. Gajski. A formalism for functionality preserving system level transformations. In *Proceedings of the Asia-Pacific Design Automation Conference*, pages 139–144, 2005.

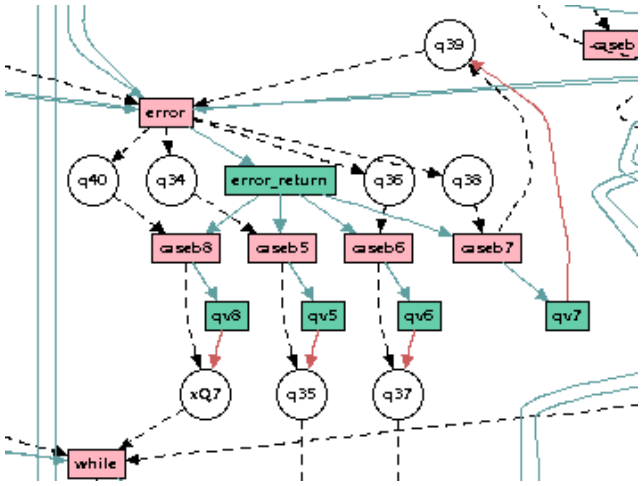

Figure 14: Representation of a CASE statement

[3] S. Abdi and D. Gajski. Verification of system level model transformations. *International Journal of Parallel Programming*, 34(1):29–59, 2006.

[4] A. D. et al. A next-generation design framework for platform-based design. In *Conference on Using Hardware Design and Verification Languages (DVCon)*, February 2007.

[5] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.