Formal Verification of Fault Tolerant NoC-based Architecture

Manamiary Bruno Andriamiarina, Hayat Daoud, Mostefa Belarbi, Dominique Méry, Camel Tanougast

Formal Verification of Fault Tolerant NoC-based Architecture

Manamiary Bruno Andriamiarina[‡], Hayat Daoud[*], Mostefa Belarbi[*], Dominique Méry[‡], Camel Tanougast[†]
[*]IBN Khaldoun University, LIM
hayat.daoud@hotmail.fr
master.dept.inf@gmail.com
[†]Université de Lorraine University, LICM, ISEA
camel.tanougast@{univ-metz, univ-lorraine}.fr
[‡]Université de Lorraine University, LORIA
Vandœuvre-lès-Nancy, France
dominique.mery@{loria, univ-lorraine}.fr
manamiary.andriamiarina@loria.fr; manamiary-bruno.andriamiarina6@etu.univ-lorraine.fr

***Abstract*—Approaches to design fault tolerant Network-on-Chip (NoC) for System-on-Chip(SoC)-based reconfigurable Field-Programmable Gate Array (FPGA) technology are challenges on the conceptualisation of the Multiprocessor System-on-Chip (MPSoC) design. For this purpose, the use of rigorous formal approaches, based on incremental design and proof theory, has become an essential step in a validation architecture. The Event-B formal method is a promising formal approach that can be used to develop, model and prove accurately the domain of SoCs and MPSoCs. This paper gives a formal verification of a NoC architecture, using the Event-B methodology. The formalisation process is based on an incremental and validated *correct-by-construction* development of the NoC architecture.**

***Keywords*-Network on chip, Switch, Adaptive-routing, machine, context, Model, specification, refinement, Formal proof, Correct-by-construction.**

I. Introduction

Designs are usually verified by simulation with created stimuli. This allows the detection of the coarse errors in a design. However, simulation can not find all possible errors in a design. This is why we use formal methods, such as Event-B, and especially the *correct-by-construction* paradigm [8] for specifying hardware systems. The *correct-by-construction* paradigm offers an alternative approach to prove and derive *correct* systems and architectures, through the reconstruction of a target system using stepwise refinement and validated methodological techniques [2, 4, 9]. Our goal is to complement the time consuming simulations in the design flow with a formal proof method. The prerequisites for the formal development of a given microelectronic architecture are the description and/or the design of the architecture.

The dynamic reconfigurable NoC are adequate and appropriate for FPGA-based systems, where the main problem arises when components IPs (Intellectual Property) must be set dynamically at runtime. Given the rapid changes and increasing complexity of MPSoCs (Multiprocessor System on Chip), constraints of cost and performance, related to the complexity and the increasing number of modules or IPs interconnected, must be solved. Current on-chip communication networks implement data packet transmissions between interconnected nodes. Sometimes, communications in these networks are difficult, even impossible. This is the main reason why fault-tolerant XY routing algorithms (for these networks) have been introduced [6]. Routers can control if previous switches have made routing errors (e.g. packet out of the XY path, etc.). Moreover, new adaptive and fault-tolerant routing techniques, with error detection and based on the well known XY and turn model routing schemes [7], have been introduced. Usually, these designs are verified by simulation, which allows the detection of coarse errors. However, simulation alone is not sufficient to improve such architectures [5].

In this article, we use Event-B to specify, verify and prove the behaviour of NoC architectures.

The paper is organized as follows. Section 2 presents an overview of the Event-B approach. Section 3 introduces the studied NoC architecture. Section 4 describes the formal development of the NoC architecture. Section 5 concludes this paper along with the future work.

II. Event B: Stepwise Design of Systems

We choose Event B [1] as a modeling language, mainly because of the refinement, which allows a progressive development of models. Event B also is supported by a complete toolset RODIN [10] providing features like refinement, proof obligations generation, proof assistants and model-checking facilities.

The Event B modeling language can express *safety* properties, which are either *invariants*, *theorems* or *safety properties* in a machine corresponding to the system. The two main structures available in Event B are:

- Contexts express static informations about the model.
- Machines express dynamic informations about the model, invariants, safety properties, and events.

An Event B model is defined either as a context or as a machine. A machine organises events (or actions) modifying state variables and uses static informations defined in a context. The general form of an event is expressed as follows *ANY*

x *WHERE* $G(x,u)$ *THEN* $u :|(P(u,u')$ END and corresponds to the transformation of the state variable u, which is set to a value u' satisfying the formula $\exists\, x\, .G(x,u) \wedge P(u,u')$, where u is the value of u before the observation of the event. If the set of events is denoted E, then the *before–after* predicate $BA(e)(x,x')$, where e is in E, is the previous formula. Proof obligations (INV 1 and INV 2) are produced by the RODIN tool, from events, to state that an invariant condition $I(x)$ is preserved. Their general form follows immediately from the definitions of the before–after predicate $BA(e)(x,x')$ of each event e of E and $grd(e)(x)$, which is safety of the guard $G(t,x)$ of event e: (INV1) $Init(x) \Rightarrow I(x)$; (INV2) $I(x) \wedge BA(e)(x,x') \Rightarrow I(x')$; (FIS) $I(x) \wedge \mathsf{grd}(e)(x) \Rightarrow \exists y.BA(e)(x,y)$.

The proof obligation FIS expresses the feasibility of the event e, with respect to the invariant I. By proving feasibility, we achieve that $BA(e)(x,y)$ provides an after state whenever $grd(e)(x)$ holds. This means that the guard indeed represents the enabling condition of the event.

These basic structures are extended by the refinement of models which provides a mechanism for relating an abstract model and a concrete model by adding new events or variables. This feature allows to develop gradually Event-B models and to validate each decision step using the proof tool. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P *simulates* M. The final concrete model is close to the behaviour of real system that executes events using real source code. The relationships between contexts, machines and events are illustrated by the next diagrams, which consider refinements of events and machines.

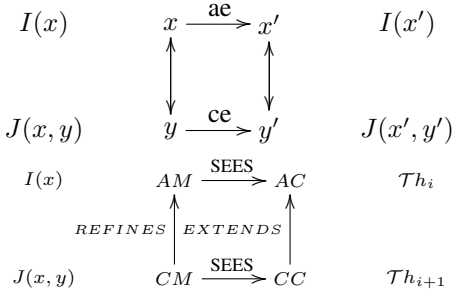

Fig. 1. Machines and Contexts relationships

The refinement of a formal model allows us to enrich the model via a *step-by-step* approach and is the foundation of our correct-by-construction approach [8]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description. This is done by extending the list of state variables (possibly suppressing some of them), by refining each abstract event to a set of possible concrete versions, and by adding new events.

We suppose that an abstract model AM with variables x and invariant $I(x)$ is refined by a concrete model CM with variables y and gluing invariant $J(x,y)$. Event e is in abstract model AM and event f is in concrete model CM. Event f refines event e. $BA(e)(x,x')$ and $BA(f)(y,y')$ are predicates of events e and f respectively; we have to prove the following statement, corresponding to proof obligation (1):

$$I(x) \wedge J(x,y) \wedge BA(f)(y,y') \Rightarrow \exists x' \cdot (BA(e)(x,x') \wedge J(x',y'))$$

We have shortly introduced the Event B modeling language and the structures proposed for organising the development of state-based models. In fact, the refinement-based development of Event B requires a very careful derivation process, integrating possible *tough* interactive proofs for discharging generated proof obligations, at each step of development.

III. NoC Architecture Overview

A. NoC Architecture Topology

The topology of a NoC architecture is usually a *Mesh*. The network has a *grid-like* form (see Fig.2): boundary switches are connected to two or three neighbours, whereas other nodes are connected to four neighbours.

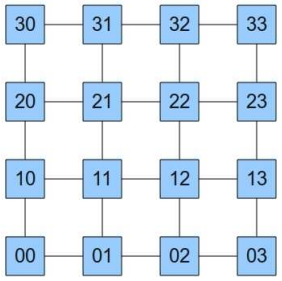

Fig. 2. A Mesh Topology

B. Structure of a Switch

The role of a switch is to pass data packets between elements (routers) of a NoC architecture.

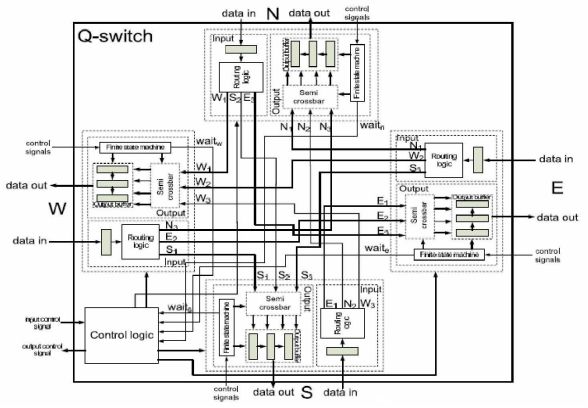

Fig. 3. Structure of a Switch

The structure of a switch (see Fig.3) is as follows:

- *Input Register:* Each incoming packet is stored in an *input register*. A specific component, called *Routing logic*, computes the next direction of the packet (whether N, E, S or W; see Fig.3). A maximum of three packets is allowed per direction. The packets are transmitted to the *output logic*. An arbitration policy can be adopted

to define priorities between packets stored in the input registers of a switch, according to the next direction of the packets. This policy is based on the rules of *right priority* (see Fig.4).

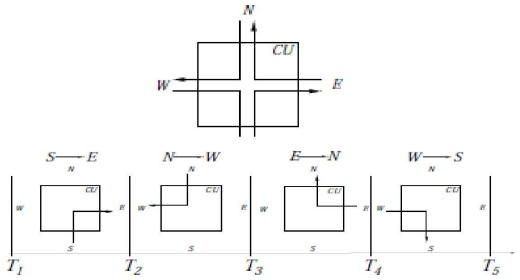

Fig. 4. Right Priority

- The *Output Logic* is made up of a *semi crossbar*, an *output buffer* and a *finite state machine*. The *semi crossbar* is composed of three inputs and four outputs. Incoming packets are stored into inputs according to priorities. If the neighbours of a switch are not busy, the first output of the *semi crossbar* is one of the adjacent switches. The *output buffer* consists of registers. These registers store packets, in the case where more than one packet choose the same output (direction). The *output buffer* is also used when the selected output (direction) is busy (*occ* signal). A maximum of three messages can be stored in a *output buffer*. The *finite state machine* (FSM) manages control signals and its role is also to avoid packets collisions. Moreover, the *finite state machine* (FSM) provides a *central logic* with informations about the states of adjacent switches (*wait* situation, *out* signal, etc.).
- The *Control Logic* manages connections between the input and output ports of a switch. The *Control Logic* also handles the storage of packets that can not be transferred to next directions, due to occupation signals from neighbouring switches. Moreover, if the switch can not store more incoming packets, the *Control Logic* informs the neighbours (which have sent the switch packets) that the switch can not accept any other packets.

C. Routing Process

The XY routing algorithm defines packets transmission:

- Let the source (s) and destination (d) of a packet (p) be defined by 2D coordinates: (x_s,y_s) for the source (s) and (x_d,y_d) for the destination (d).
- The packet (p) travels first along x dimension, until x_s=x_d. Then, the packet (p) travels along y dimension, until y_s=y_d.
- If the packet (p) encounters elements unable to transmit data in x dimension, the routing temporary switches to y dimension.
- It should be noted that the network can evolve (deletion of some links, isolation of some switches, etc.), and data transmission can be disrupted. However, a *reconfiguration* mechanism ensures that for each transiting packet, either a path leading to the destination of the packet always exists or, if the packet is stored in some node unable to transmit data, the link between this node and the destination of the packet will eventually be restored.

IV. Modeling NoC Architecture

This section presents the formal development of the NoC Architecture. However, due to space limitations, we have given sketch of the modeling. A detailed formal development is available[1]. It should be noted that *refinement* allows us to break the complexity of the NoC Architecture and perform our formalisation with different levels of abstraction, step-by-step (see Fig.5).

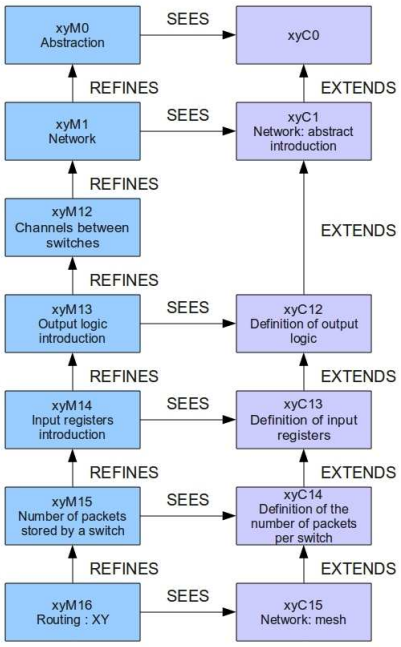

Fig. 5. Step-by-step Modeling of NoC Architecture

A. Abstract Specification: xyM0

The first model xyM0 is an abstract description of the service offered by the NoC Architecture: the sending of a packet (p) by a switch source and the receiving of (p) by a switch destination.

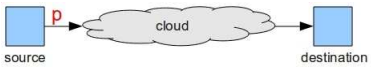

Fig. 6. Abstraction

A set of switches ($NODES$), a set of packets (MSG), a function src, associating packets and their sources, a function dst, coupling packets and their destinations, are defined in context xyC0. The machine xyM0 uses (*sees*) the contents of context xyC0, and with these, describes an abstract view of the service provided by the NoC Architecture:

- An event SEND presents the sending of a packet (m), by its source (s), to a switch destination (d).

[1] http://www.loria.fr/~andriami/noc-pdf/project.html

- An event RECEIVE depicts the receiving of a sent packet (m) by its destination (d).

Moreover, the model xyM0 allows us to express some properties and invariants:

$$ran(received) \subseteq ran(sent)$$

This invariant expresses that each packet received by a switch destination has been sent by a switch source.

B. First Refinement (xyM1): Network Introduction

The machine xyM1 refines xyM0 and introduces a network (a *graph*) between the sources and destinations of packets. Some properties on the *graph* are defined in context xyC1: *graph* is non-empty, non-transitive and is symmetrical.

Fig. 7. Adding Network

The events in xyM0 are refined:

- Event SEND: When a source sends a packet, the packet is put in the network.
- Event RECEIVE: A packet is received by its destination, if the packet has reached the destination.

New events are also introduced by xyM0:

- Event FORWARD (see Fig.8): in the network, a packet (p) transits from a node (x) to another node (y), until the destination (d) of packet (p) is reached.

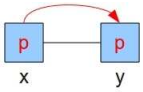

Fig. 8. Transfer of a Packet (p) between Switches

- Event DISABLE: A node is *disabled*. The node is not allowed to communicate with its neighbours (*failure*, etc.). During the *disabling* of some nodes, we ensure that the packets transiting in the network will eventually reach their destinations (either after a reconfiguration of the network or by always letting a path to destinations available).
- Event RELINK: This event models the reconfiguration of the network. *Disabled* nodes are re-*enabled*: the links between them and their neighbours are restored, therefore allowing communications and packets transfers. The reconfiguration of the network helps in demonstrating the safety of data transmission between a switch source and a switch destination.

The machine xyM1 also presents some properties of the system:

$$ran(received) \cap ran(store) = \varnothing$$

This invariant demonstrates that a packet (p) sent by a source is either traveling in the network ($store$) or is received by a destination.

C. Second Refinement (xyM12): Channels Introduction

This second refinement decomposes the event FORWARD of xyM1 into two events:

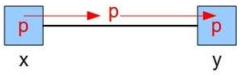

Fig. 9. Channel Introduction

- A refinement of the event FORWARD depicts the passing of a packet (p) from a switch (x) to a channel (ch), leading to a neighbour (y).
- An event FROM_CHANNEL_TO_NODE models the transfer of a packet (p) from a channel (ch) to a connected switch (n).

The machine xyM12 also defines some properties:

$$ran(c) \cap ran(switch) = \varnothing$$

The invariant expresses that each sent packet is either in a channel or in a switch. A sent packet can not be in a channel and in a switch at the same time.

D. Third Refinement (xyM13): Output logic Introduction

This refinement allows us to introduce the structure of a switch gradually. We express, in xyM13, that switches possess output ports (see Fig.10). The abstract event FORWARD is further decomposed:

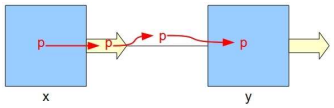

Fig. 10. Adding Output Ports

- The refinement of event FORWARD adds the fact that a packet (p), which is leaving a switch (x) and heading for a neighbour (y), first enters the output logic (op) of the switch (x) leading to (y).
- A new event OUTPUT_BUFFER_TO_CHANNEL models the transition of a packet (p) from an output port (op) to a channel (ch) leading to a target switch (n).

Moreover, new properties and invariants are defined in xyM13:

$$\begin{array}{l} inv1 \ : \ ran(chan) \subseteq ran(sent) \\ inv2 \ : \ ran(outputbuffer) \subseteq ran(sent) \\ inv3 \ : \ ran(outputbuffer) \cap ran(chan) = \varnothing \end{array}$$

The invariant $inv1$ expresses that each packet transiting in a channel (ch) has been sent by a source (s); $inv2$ demonstrates that each packet transiting in an output port (ch) has been sent by a source (s); $inv3$ presents the fact that a packet is either in an output port or in a channel, the packet can not be in an output port and a channel between two switches at the same time.

E. Fourth Refinement (xyM14): Input register Introduction

This refinement (xyM14) adds input ports to the structure of a switch.

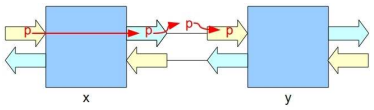

Fig. 11. Adding Input Ports

- The event SEND is refined: when a switch source (s) sends a packet (p), the packet (p) is put in an input port (ip) of the switch (s).
- The actions described by the abstract event FORWARD are decomposed:
  - The event SWITCH_CONTROL, a refinement of FORWARD, models the passing of a packet (p), from an input port (ip) of a switch (x), to an output port (op) leading to a switch (y).
  - The event OUTPUT_BUFFER_TO_CHANNEL presents the transition of a packet (p), from an output port (op), to a channel (ch) leading to a target switch (n).
  - The event FROM_CHANNEL_TO_INPUT_BUFFER demonstrates the transition of a packet (p) from a channel (ch) to an input port (ip) of a target switch (n).

The machine xyM14 also presents properties and invariants:

$$\begin{array}{l} inv1 \ : \ ran(inputbuffer) \subseteq ran(sent) \\ inv2 \ : \ ran(outputbuffer) \cap ran(inputbuffer) = \varnothing \\ inv3 \ : \ ran(inputbuffer) \cap ran(chan) = \varnothing \end{array}$$

The invariant expresses that each packet transiting in an input port (ip) has been sent by a source (s); $inv2$ demonstrates that each packet is transiting either in an output port (op) or an in input port (ip); $inv3$ presents the fact that a packet is either in an input port or in a channel, the packet can not be in an input port and a channel between two switches at the same time.

F. Fifth Refinement (xyM15): Number of Messages per Switch

This refinement introduces the storage of packets in a switch: each output port of a switch can store a number of packets up to a limit ($outputplaces$) of three messages. Packets can be blocked in a switch, because of *wait* or *occupation* signals from neighbours.

The event SWITCH_CONTROL is refined, and adds the fact that following the transition of a packet from an input port of a switch (x) to an output port, if the switch (x) is not busy anymore, it sends a release signal to the previous switch linked to the input port. A new event RECEIVE_BUFFER_CREDIT models the receiving of a release signal by a switch (n).

G. Sixth Refinement (xyM16): Algorithm XY

The last model xyM16 describes the architecture of the network (*graph*): *graph* has a *mesh* topology (see Fig.12). A numerical limit ($nsize$) is introduced to bound the number of routers in the dimensions x and y of the network topology; the network will be a regular *2D-Mesh*, with a size ($nsize \times nsize$); each switch is coupled with unique coordinates (x, y), with $x \in [0..nsize-1]$ and $y \in [0..nsize-1]$.

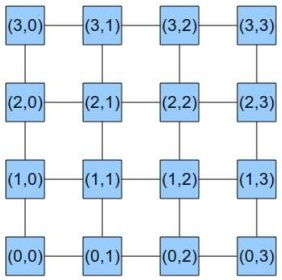

Fig. 12. A regular Mesh with 2D-coordinates

This coordinate system allows to be more precise on the neighbours of each switch, as seen in figure 12. This model also gives a fine-grained description of the structure of a switch (see Fig.13):

- A switch has generally four output ports and four input ports (usually labelled N, S, E and W), used for communication with neighbours.
- However, two more cases are distinguished:
  - Boundary switches in the corner have only two output ports and two input ports (N-E, N-W, S-E, S-W).
  - Other boundary switches have three output ports and three input ports (N-S-E, N-S-W).

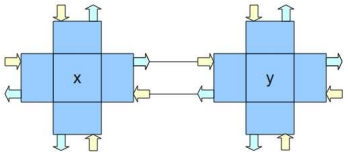

Fig. 13. Switches: Structure and Links

Moreover, this concrete model also introduces the XY routing algorithm:

```
D : destination. Coordinates (Dx, Dy)
C : current node. Coordinates (Cx, Cy)

if (Cx > Dx) :
     return W; (Case 1)
if (Cx < Dx) :
   return E; (Case 2)
if ((Cx = Dx) ∨ ((Cx > Dx) ∧ W is blocked) ∨
   ((Cx < Dx) ∧ E is blocked)) :
   if (Cy < Dy) :
      return N; (Case 3)
   if (Cy > Dy) :
      return S; (Case 4)
```

The cases of the XY routing algorithm are matched with refinements of event SWITCH_CONTROL:

- SWITCH_CONTROL_LEFT models **Case 1**: a packet (p) is transmitted, from an input port of a switch (x), to an output port, leading to a neighbour (y), located at W. This event is triggered if the x-coordinate of the destination (d) (of the packet(p)) is inferior to the x-coordinate of the current node (x).

- SWITCH_CONTROL_RIGHT models **Case 2**: a packet (p) is transmitted, from an input port of a switch (x), to an output port, leading to a neighbour (y), located at E. This event is triggered if the x-coordinate of the destination (d) (of the packet(p)) is superior to the x-coordinate of the current node (x).
- SWITCH_CONTROL_UP models **Case 3**: a packet (p) is transmitted, from an input port of a switch (x), to an output port, leading to a neighbour (y), located at N. This event is triggered if the y-coordinate of the destination (d) (of the packet(p)) is superior to the y-coordinate of the current node (x), and either, if the x-coordinate of the destination (d) is equal to the x-coordinate of the current node (x), or if the packet (p) can not transit along the x-axis.
- SWITCH_CONTROL_DOWN models **Case 4**: a packet (p) is transmitted, from an input port of a switch (x), to an output port, leading to a neighbour (y), located at S. This event is triggered if the y-coordinate of the destination (d) (of the packet(p)) is inferior to the y-coordinate of the current node (x), and either, if the x-coordinate of the destination (d) is equal to the x-coordinate of the current node (x), or if the packet (p) can not transit along the x-axis.

V. Conclusion

This paper presents an incremental development of a Network-on-Chip Architecture, using the Event B formalism. The formalization of the architecture is presented from an abstract level to a more concrete level in a hierarchical way. The complexity of the development is measured by the number of proof obligations which are automatically/manually discharged (see table I).

Model	Total	Auto		Interactive	
xyC0	3	3	100%	0	0%
xyC1	6	6	100%	0	0%
xyC12	0	0	100%	0	0%
xyC13	0	0	100%	0	0%
xyC14	1	1	100%	0	0%
xyC15	5	0	0%	5	100%
xyM0	26	25	96.15%	1	3.85%
xyM1	38	28	73.68%	10	26.32%
xyM12	72	45	62.5%	27	37.5%
xyM13	74	37	50%	37	50%
xyM14	67	23	34.33%	44	65.67%
xyM15	24	14	58.33%	10	41.67%
xyM16	26	18	69.23%	8	30.77%
Total	342	200	58.48%	142	41.52%

TABLE I
SUMMARY OF PROOF OBLIGATIONS

We remark that for context xyC15 and machine xyM14, there are more interactive proofs that automatic ones. This is explained by the fact that a majority of these interactive proofs are *quasi-automatic*: the proofs did not need tough efforts (neither importing hypotheses or simplifying goals, etc.), the mere usage/running of provers (provided by the RODIN platform) allowed us to discharge these obligations. Contrary to the verification by simulation only, our work provides a framework for developing the Network-on-Chip Architecture and the XY routing algorithm using essential safety properties together with a formal proof that asserts its correctness.

As a part of our future efforts, we consider the translation of the most concrete (detailed and close to algorithmic form) model into an intermediate language, from which hardware description (e.g. in *VHDL*) can be extracted. Moreover, we note that the first levels of the Event B design of the NoC Architecture express general cases of routing methologies and fall in the interesting domain of reusable and generic refinement-based structures [3, 9]. We plan to investigate further on this domain of genericity and reusability of proof-based models.

Acknowledgment

This work has been financed by the joint project STIC-Algérie 2011-2013 between the Université de Tiaret and the research group MOSEL/VERIDIS on the formal developent of FPGA circuits.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. 2010.
[2] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.
[3] M. B. Andriamiarina, D. Méry, and N. K. Singh. Revisiting Snapshot Algorithms by Refinement-based Techniques. In *PDCAT*. IEEE Computer Society, 2012.
[4] R.-J. Back and K. Sere. Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.
[5] D. Cansell, C. Tanougast, and Y. Beviller. integration of the proof process in the design of microelectronic architecture for bitrate measurement instrumentation of transport stream program mpeg-2 dvb-t. 2004.
[6] S. Jovanovic, C. Tanougast, and S. Weber. A new high-performance scalable dynamic interconnection for fpga-based reconfigurable systems. pages 61–66, 2008.
[7] C. Killian, C. Tanougast, F. Monterio, and A. Dandache. Online routing fault detection for reconfigurable noc. In *International Conference on Field Programmable Logic and Applications*, 2010.
[8] G. T. Leavens, J.-R. Abrial, D. S. Batory, M. J. Butler, A. Coglio, K. Fisler, E. C. R. Hehner, C. B. Jones, D. Miller, S. L. P. Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE*, pages 221–236. ACM, 2006.
[9] D. Méry. Refinement-based guidelines for algorithmic systems. *Int. J. Software and Informatics*, 3(2-3):197–239, 2009.
[10] Project RODIN. Rigorous open development environment for complex systems. http://www.eventb.org/, 2004-2010.