Université de Montréal

Compositional Verification Using Interface Recognizers/Suppliers (IRS)

Par

Mohammad-Sadegh Jahanpour

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiae Doctor (Ph. D.)
en informatique

Juillet 2001

Compositional Verification Using Interface Recognizers/Suppliers (IRS)

Mohammad-Sadegh Jahanpour

Département d'informatique et de
recherche opérationnelle
Faculté des arts et des sciences

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée

Compositional Verification Using Interface Recognizers/Suppliers (IRS)

présenté par

Mohammad-Sadegh JAHANPOUR

a été évaluée par un jury composé des personnes suivantes:

1. Michel BOYER

(président-rapporteur)

2. Eduard CERNY

(directeur de recherche)

3. Ivo ROSENBERG

(membre du jury)

4. Gert SABIDUSSI

(membre du jury)

5. Radu NEGULECU

(examinateur externe)

Acknowledgment

I am grateful to my advisor professor Eduard Cerny for encouraging and guiding me during my Ph.D. study. He teaches, listens and trusts his students, and my biggest fortune was in having him as my supervisor. I thank him for guidance over the years.

I would like to give my gratitude to the enthusiastic friends in lab LASSO at the University of Montreal for creating such a wonderful environment. In particular, I would like to thank professors E. M. Aboulhamid and S. Tahar for their encouragements and colaborations. I thank our friendly administrator Michel Reid for his quick and fast maintenance of lab equipments.

Iranian Ministry of Research and Higher education and my parents were instrumental in my coming Canada for graduate education. My wife, Rita supported me during my stay in Montreal. Without her love, affection and understanding, this thesis would have not been possible.

Sommaire

Les blocs pre-conçus et pre-vérifiés, appelés les blocs de propriété intellectuelle (IP), sont de plus en plus utilisés dans la conception des systèmes microélectroniques. D'une part, ils facilitent la conception des systèmes et, d'autre part, ils contribuent de façon significative à l'évolution du produit.

Cependant la vérification de ces systèmes, intégrant les blocs IP devient un véritable défi et une tâche très laborieuse. En effet, la difficulté réside dans le fait qu'un bloc IP fonctionne correctement seulement dans son propre environnement et il est peu probable qu'il maintient ses propriétés dans un environnement arbitraire.

Afin de garantir le bon fonctionnement de ces systèmes, un bloc IP non seulement doit être certifié mais son environnement doit être aussi mis en exergue. Dans cette direction, l'approche supposition/garantie[1] est un support idéal aussi bien pour spécifier le comportement attendu du système que pour prouver certaines de ses propriétés. Ainsi, la spécification des blocs IP se subdivise en deux parties. La première partie décrit les hypothèses que le composant suppose sur son environnement tandis que la deuxième partie spécifie les propriétés du composant. D'une manière intuitive, cette décomposition est justifiée par le fait que le composant garantit ses propriétés lorsque son environnement satisfait les hypothèses que le composant suppose.

L'inconvénient du paradigme supposition/garantie est que les hypothèses et les propriétés sont souvent exprimées en logique temporelle ou bien dans un langage

propre des outils qui supportent ce type de spécification, comme MOCHA à titre d'exemple [37]. Cette dépendance de la logique ou d'un outil particulier n'est pas acceptable à la spécification des blocs IP car il limite généralité, réutilisablilité et portabilité des blocs. C'est pourquoi, nous introduisons les automates acceptants/imposants pour la spécification supposition/garantie des blocs IP. Ensuite, nous proposons un paradigme formel pour la vérification compositionnelle des systèmes intégrants de ces blocs.

Cette thèse est constituée de 6 chapitres :

Dans le premier chapitre, nous introduisons le problème d'intégration des blocs IP. Nous discutons aussi les objectives des travaux présentés dans les chapitres suivants.

Dans le deuxième chapitre, nous présentons une synthèse de la littérature récente reliée à la vérification formelle des composants matériels[2]. Ce chapitre introduit les notions mathématiques utilisées dans la thèse et regroupe un ensemble de méthodes supposition/garantie et les méthodes de vérification sémantique compositionelles[3]. Nous classifions ces méthodes par deux critères : (1) Est-ce que la méthode en question permet un raisonnement/composition circulaire des propriétés des composants de système? (2) Est-ce que cette méthode peut être appliquée aux compositions des propriétés de sûreté ainsi qu'aux propriétés de vivacité?

Dans le troisième chapitre, nous proposons d'utiliser les automates acceptants/imposants[4] pour spécifier les hypothèses d'environnement de composant, par exemple, un bloc IP. Un automate acceptant/imposant est un automate qui a été augmenté par certaines contraintes booléennes. Ces contraintes décrivent ce que doit être fourni à l'entrée de l'automate dans ses états. Les contraintes restent satisfaites tant que le comportement de l'environnement de l'automate est conforme au

---

[1] Assume-guarantee
[2] Hardware
[3] Compositional model checking
[4] Interface Recognizers/Suppliers (IRS)

comportement décrit par ces contraintes. Ainsi, on représente le comportement par l'automate acceptant. Supposons que nous modélisons les hypothèses d'environnement d'un composant par un automate acceptant. Nous composons cet automate avec le composant et nous imposons une condition que les contraintes d'automate soient maintenues (vrais) pendant toute la vérification du composant. Le vérificateur sémantique[5] doit appliquer toute la séquence de données qui satisfait les contraintes de l'automate. De cette façon, l'automate acceptant caractérise les séquences de données du composant. Ce dernier nous permet de vérifier le composant dans un environnement propre. Nous décrivons la syntaxe et la sémantique de ce type d'automate dans ce chapitre.

En guise d'exemple, nous décrivons un modèle d'un commutateur ATM[6] dans les chapitres 4 et 5. Notre objectif est d'illustrer l'application de l'automate acceptant/imposant à la spécification et vérification d'un système complexe. Ce commutateur est comosé d'un ensemble de contrôleurs et un commutateur. Nous décrivons les hypothèses des environnements du contrôleur et du commutateur. Dans le chapitre 4, nous démontrons que les hypothèses des contrôleurs sont respectées par le commutateur et vice-versa, les hypothèses du commutateur sont respectées par le commutateur. De la même façon, nous illustrons dans le chapitre 5 que ce type d'automate peut être utilisé pour la spécification et la vérification compositionelle des propriétés de sûreté et de vivacité du commutateur ATM.

Nous concluons ce travail en chapitre 6 en mettant en évidence les aspects pratiques de l'automate proposé, en particulier son application à la vérification des processeurs ainsi qu'aux systèmes de protocoles de télécommunication multicouche.

---

[5] Model checker
[6] Asynchronous Transfer Mode

Abstract

In this thesis, we review recent developments in compositional and assume guarantee verification. We discuss whether each method supports circular/non circular reasoning and whether it can be used when proving safety/liveness properties.

We formulate interface recognizers/suppliers (IRS), which are recognizers augmented with Boolean constraints. The constraints specify what values may occur on IRS inputs at each state. In other words, IRS can constrain its inputs.

We discuss a composition theorem for circular reasoning using IRS. In this way, IRS framework extends non-circular constraint model checking [25] to a circular constrainted model checking.

We demonstrate an application of IRS in (1) specifying environment assumptions and in (2) modeling pre conditions / post conditions of properties of an ATM switch. Using IRS, we specify and then verify the switch.

Abreviations

A/G Assume guarantee

ATM Asynchronous Transfer Mode

CMC Constraint Model **Checking**

CTL Computation Tree Logic

CV Compositional Verification

FIFO first in first out

FSM Finite State Machine

IP Intellectual Property

IRS Interface Recognizer/Supplier

LTL Linear Temporal Logic

RTL Register Transfer Level

TBL tableau

TLA Temporal Logic of Actions

WFC Well-foundedness Conditions

List of Symbols

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Predesigned, preverified silicon building blocks or cores are finding increasing use in microelectronic system designs [50]. Examples of such cores or hardware intellectual properties (IP) are microprocessors, DSP, PCI, MPEG and JPEG cores. Integration of these application specific components into complex system-on-chip (SOC) designs is a new challenge for system-level designers. As the complexity and the density of ICs increase, verification becomes even more important than before. Traditionally, simulation has been used for design verification, but the increasing design complexity makes it very difficult, if not impossible, to create sufficient test vector sets. Even with partial vector sets, simulation usually takes too long for each iteration. International Technology Roadmap for Semiconductors (ITRS) [10] has identified test and verification of complex systems as the challenges of the system design in the next decade. ITRS has warned that these challenges are soon becoming crises. There are currently 2 to 3 times more verifications engineers than designers on microprocessor teams. Overall Cost of design threatens continuation of the semiconductor roadmap [10].

Formal verification may offer a viable approach to the verification of these complex systems. Instead of applying stimuli to a design and comparing its responses with expected results, formal verification tools examine a design and mathematically prove or disprove its functional properties. The huge effort needed to create functional test

vectors can be avoided except for those parts of the design where simulation is still necessary. Putting both methods together, a thorough verification can be achieved in a much shorter time than purely using simulation. Unlike nonexhaustive simulation, which only aims to show the presence of bugs, formal verification can prove their absence. When errors are found, formal verification tools can also generate counter examples to demonstrate the error conditions. Formal verification has become an essential technology for solving today's verification problems [48].

Most formal verification methods fall into one of two classes [31]; (1) proof based methods which use theorem provers, and (2) state-exploration methods which use model checkers to automatically search the state space of the design. Theorem provers use the full power of mathematics, so they are very flexible and can proof properties of entire classes of systems [31]. The main drawback of such methods is that they require a large amount of interaction from the users. In contrast, state-exploration methods restrict the model to be finite-state, and use state space search algorithms to check automatically that the specification is satisfied. The most serious drawback of the state-exploration methods is the state explosion problem [31]. This problem mostly arises in systems composed of multiple components operating in parallel. Composing finite state machines in parallel leads to an exponential explosion of states in the worst case, which imposes a strong limit on the size and complexity of systems that can be verified by state enumeration methods [14].

Two main techniques have been proposed to avoid state explosion problem [31]: (1) compositional verification and (2) abstraction. In compositional verification, the specification of the system is separated into properties of its components. Then, the properties of the components are separately verified. Finally, it is proven that the components specifications imply the specification of the entire system. In abstraction, the models are simplified by hiding details of the design. Then the simplified model is verified. Finally, a relation between the abstract model and the original one is established to assure that the correctness at the abstract level implies the correctness at the detailed or refined level.

Typically, a component works properly only in a given environment in the system. It is unlikely that the component satisfies any useful property in an arbitrary environment. This is called "the environment problem" [14]. In compositional verification, properties or abstract models of the other components constitute a constraining environment for verifying the given component.

The environment problem appears in a similar way when reusing IP cores in new applications. Given that the core works properly under specific environment assumptions, it is unlikely that it works in any arbitrary application. For a successful core reuse, a formalism is needed to specify these assumptions.

In this thesis, we study formal requirements for (reusable) component integration and propose a formalism for the specification of the properties and the environment assumptions of these cores. We use compositional verification methodologies to enssure correct integration of cores in systems.

Our objective is also to illustrate compositional reasoning on a relatively complex system. As a case study, we target telecommunication hardware systems. We study an asynchronous transfer mode (ATM) switch module that consists of port controllers and a switch fabric. ATM is a cell based switching and multiplexing technology designed to be a general purpose transfer mode for a wide range of services [52]. It is particularly well suited for the exchange and transfer of media intensive data such as real time audio, video, and high quality images.

The ATM switch includes a complex control path and handles large data structures like ATM cells. The switch is quite complex for current model checking tools, e.g., Formal Check [3], and consequently a compositional approach must be adapted for the overall verification. Although, there have been earlier efforts in the verification of switch fabrics [32], there is no published work on (the formal or informal) verification of port controllers. As data rates of networks increase and more services

are offered by network service providers, these port controllers are becoming more complex since they have to handle packets in a shorter time budget and with different qualities of service. For instance, with an OC-192 data link, the switch has a maximum time of 51 microseconds to read, convert the header, and route each ATM cell [26]. Becoming more complex, the switch module requires a parallel architecture to perform its functions. Currently, major semiconductor manufacturers are starting to sell a new type of integrated circuit, the network processors [26]. Network processors are programmable chips like general purpose microprocessors, but are optimized for packet processing required in network devices. This (network processor) industry is currently at its early stage. In this thesis, we implement a relatively simple switch module which (is not as complex as a network processor, however) performs basic operations, e.g., cell buffering, routing, header transformation, and cell prioritization. We aim to develop the following ideas:

1. If a switch fabric or a port controller is provided as an IP core, what would be the environment assumptions for each of them and which specification formalism can describe those assumptions?

2. Having specified the environment assumptions, what properties have to be specified for switch components? Although a set of well-defined, well-established properties is available for microprocessors, such a specification does not exist for port controllers of an ATM switch.

3. Having the assumptions and the properties, how could one verify a correct integration of switch components in an application? How could one show that each component satisfies the assumptions made by its neighbors?

1.2 Contributions

The principal contributions of this thesis are as follows:

1. We classify recent (and relatively poorly documented) compositional verification methods. In chapter 2, we describe assume guarantee reasoning implemented by theorem provers and non-circular and circular compositional verification methods implemented by state of the art model checkers.

2. We introduce interface recognizers/suppliers (IRS) as a practical mechanism to represent environment assumptions and interface properties of the components. IRS is a recognizer[7] augmented with a set of Boolean constraints. By forcing the constraints to be always true, IRS constrains its inputs. By checking the constraints of IRS, we verify that a component satisfies the specification modeled by the IRS. IRS can thus equally act as a recognizer of a property and as a supplier of assumptions on its inputs.

3. We extend the application of IRS from modeling environment assumptions to compositional verification. Given that the reasoning with IRS can be circular, we develop well-foundedness conditions to avoid erroneous conclusions.

4. We model an existing port controller and switch fabric modules of an ATM switch. (Port controllers are modeled in VHDL [9] and the fabric is translated from Verilog to VHDL. The controller and the fabric models are about 3500 and 500 lines of VHDL code respectively. (See Appendix 4.)) In a 4x4 setting, there are 4 in port controllers and one 4x4 switch fabric. This 4x4 model is about 14500 lines of VHDL code and includes 1500 state variables, which is far beyond the capacity of a commercial model checker. We use IRS to model environment assumptions of the switch fabric. Using the same IRS, we show that the in port controllers satisfy these assumptions. We also show that the fabric satisfies its properties when operating under the environment IRS.

5. We propose a specification and verification methodology for switch-type systems, e.g., for this class of network devices. We specify 5 properties for in port controllers

[7] Recognizer is defined in Chapter 3.

and then use IRS to model and verify these properties. Due to the large complexity of ATM cells, we use the data-independence assumption and cell size reduction techniques to carry out successful verification.

We organize the thesis as follows: Chapter 2 introduces the mathematical notation needed to describe and classify compositional verification methods. Assume/guarantee reasoning and compositional model checking methods are also discussed there.

In Chapter 3, we propose interface recognizers/suppliers (IRS) as a means to specify environment assumptions and properties of (reusable) components.

Chapters 4 and 5 contain our case study: The ATM switch design is introduced and its properties are formally verified using the IRS methodology.

Chapter 6 presents conclusions and discusses possible extensions or improvements to the proposed compositional verification method.

The methodology proposed in this thesis has been published in [28][29][2]. We have also submitted a comprehensive article on the compositional reasoning using IRS to Formal methods in system design journal.

Chapter 2

Compositional verification rules

In this chapter, we present the computation models and the formal notation that are used to describe compositional verification rules. In particular, we represent the models that have been used in compositional verification of hardware systems.

Definition 2.1 [31]: A Moore machine $M = <S, Init, I, O, T, L>$ is a tuple of the following form:

1. S is a finite set of states.
2. $Init \subseteq S$ is a nonempty set of initial states.
3. I is a finite set of input propositions.
4. O is a finite set of output propositions.
5. $T \subseteq S \times 2^{I} \times S$ is a transition relation.
6. L: $S \rightarrow 2^{O}$ is a function that maps each state to the set of output propositions true in that state.

It is required that $I \cap O = \varnothing$ and for every $s \in S$ and $v \subseteq I$, there exists some $t \in S$ such that $T(s, v, t)$.

Moore machines that have disjoint sets of outputs can be composed in a natural way. The composition, for instance at the circuit level, corresponds to wiring outputs of one machine to the appropriate inputs of the other. Each machine receives some of its

inputs from the other machine and some of its inputs from an environment. The next definition is a formal definition of the composition.

Definition 2.2. The composition of Moore machines $M1 = <S_1, Init_1, I_1, O_1, T_1, L_1>$ and $M2 = < S_2, Init_2, I_2, O_2, T_2, L_2 >$ (denoted $M1 \parallel M2$) is defined when $O_{M1} \cap O_{M2} = \varnothing$ and it is the Moore machine M defined by:

1. $S = S_1 \times S_2$, where $\times$ represents the Cartesian product. For instance, $(s_1, s_2) \in S$ only if $s_1 \in S_1$ and $s_2 \in S_2$.
2. $Init = Init_1 \times Init_2$.
3. $I = (I_1 \cup I_2) - (O_1 \cup O_2)$.
4. $O = O_1 \cup O_2$.
5. $T[(s_1, s_2), v, (s'_1, s'_2)]$ iff $T_1[s_1, (v \cup L_2(s_2)) \cap I_1, s'_1]$ and $T_2[s_2, (v \cup L_1(s_1)) \cap I_2, s'_2]$.
6. $L(s_1, s_2) = L(s_1) \cup L(s_2)$.

This definition of $M1 // M2$ has the following properties:

1. Each machine must make a transition[8], and
2. The inputs that each machine sees are the inputs from the overall environment plus the outputs from the other machine in the composition. Finally, the union of the outputs of the modules gives the outputs of the composed system. (It is possible to restrict the global outputs to a subset of this union, e.g., $O \subseteq O1 \cup O2$. However, we have reported the original definition in [31] .)

Kripke structures [41] are usually used in model checking of hardware systems. We study the relationship between Moore machines and Kripke structures. A Kripke structure $M = <S, Init, T, A, L>$ is defined as follows [31]:

1. S is a finite set of states.
2. $Init \subseteq S$ is a nonempty set of initial states.

[8] In this thesis, we have only assumed synchronous hardware systems.

3.$T \subseteq S \times S$ is a transition relation.

4. A is a finite set of atomic propositions.

5. L is a function that maps each state to the set of output propositions true in that state.

A Kripke structure or structure for short does not distinguish inputs from outputs. Moore machines, on the other hand explicitly define inputs to interact with their environments. By incorporating inputs of a Moore machine to its states, it is possible to obtain the corresponding Kripke structure. Figure 2.1 shows a machine M and the corresponding structure struct(M).

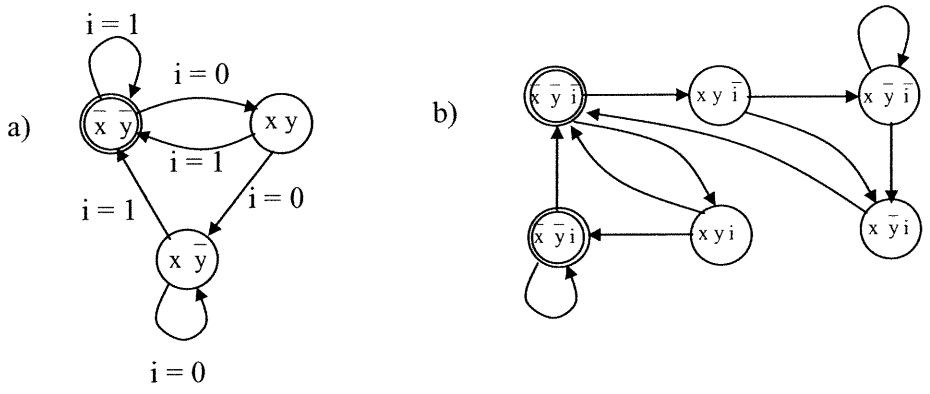


Figure 2.1: a) A Moore machine M and b) the corresponding structure struct(M) [31]

Next, we review a temporal logic that is commonly used for property specification in the literature. Temporal logic is a logical language used in the formal verification of concurrent systems. We will use standard linear temporal logic (LTL) [51]to reason about the composition of properties in Chapter 5. A model for an LTL formula is an infinite sequence $s_0, s_1, \ldots$ of states, representing consecutive time instants. A formula is either an atomic proposition or one of $\neg p$, $p \wedge q$, $p\mathrm{U}q$, Xp, where p and q are formulas. "$\neg$" and "$\wedge$" represent propositional operators *not* and *and*, respectively. The Until operator "U" and the Next operator "X" are defined as follows:
Each formula is either true or false in a given state.

We use the notation (M, s_i) $\models p$ to indicate that M satisfies p at the ith state, i.e., s_i.

1. $(M, s_i) \models \neg p$ iff (M, s_i) $\not\models p$.
2. (M, s_i) $\models p \wedge q$ iff (M, s_i) $\models p$ and (M, s_i) $\models q$.
3. (M, s_i) $\models p$ U q iff (M, s_i) $\models q$ or there exists $j>i$ such that (M, s_j) $\models q$, and for all $i \leq k < j$, (M, s_k) $\models p$.
4. (M, s_i) $\models X p$ iff (M, s_{i+1}) $\models p$.

The formula $F\ p$ (eventually p) is an abbreviation to (true $U\ p$) and it predicts the eventual occurrence of p becoming true. $G\ p$ (globally p) is equivalent to $\neg F \neg p$ indicating that p is true from now on.

We will use finite state automata to specify interface behaviors. A finite state automaton (FA) is a 5-tuple (*S, Init, A, T, F*), where [30]

1. S is a finite set of states.
2. $Init \in S$ is a start state.
3. A is a finite set called the alphabet.
4. T: $S \times A \rightarrow S$ is the transition function.
5. $F \subseteq S$ is the set of accepting (or final) states.

The input alphabet contains the allowed input symbols. If the automaton receives an allowed input symbol a in a state s, it moves to the next state indicated by the transition function T. When a machine accepts a string, it ends up in an accepting state. If L is the set of strings that machine M accepts, we say that L is the "language" of machine M [30].

A finite state machine (FSM) is a model similar to FA. An FSM generates outputs in each state of the machine. (In contrast to FA, outputs rather than final states are considered and emphasized for FSMs.) The outputs are determined from the current

state and the inputs of the machine [5]. When the output values depend only on the state of the FSM, we get a Moore machine (Definition 2.1).

FSMs can model synchronous circuits. Suppose a machine M implements a property p under the environment assumptions e. This is denoted by [4]

$$< e > \mathrm{M} < p > \tag{2.1}$$

The environment assumptions e can be supplied as a set of linear temporal logic formulas, or equivalently by an FSM E that models these temporal formulas. (However, to model eventuality formulas, fairness assumptions sould be added to these FSMs.) In [6] a practical method called tableau construction is proposed to build a maximal model for a given temporal logic formula. The tableau or the maximal model is the one that can simulate all the models that satisfy the formula[9]. We denote TBL(p), the tableau for a formula p. Assertion (2.1) can then be implemented using model checking algorithms. For instance, one can verify that

$$\mathrm{E} \parallel \mathrm{M} \models p \tag{2.2}$$

Equivalently, it can be verified that the composed machine E||M can be simulated by the machine TBL(p) [6]

$$\mathrm{E}\|\mathrm{M} \leq \mathrm{TBL}(p) \tag{2.3}$$

Assertion (2.1) can also be implemented using deduction-based algorithms, i.e, theorem prover systems. The formulas are expressed with first order or higher order logics and they get an explicit time parameter t. For instance, (2.1) is implemented as:

$$\forall \mathrm{t}.\ e(t) \Rightarrow (\ \mathrm{M}(t) \Rightarrow p(t)) \tag{2.4}$$

[9] The simulation relation is defined in Appendix 2.

Having defined computational models, we proceed to study compositional verifications methods.

2.1 Assume guarantee reasoning (A/G)

Abadi and Lamport [22] presented an assume guarantee formalism for the specification of open systems, i.e., the systems that interact with their environment. Suppose that the specification of a component M is represented with an assume guarantee (A/G) specification $e \xrightarrow{+} p$, where p specifies the component commitments and e describes its environment assumptions. This A/G specification asserts that M maintains its commitments p if the environment satisfies the specification e. (More precisely, formula $e \xrightarrow{+} p$ asserts that (for all i,) p is true up to point i of the computation if e holds up to point $j < i$. This means that p holds at least one step more than e does. The formula $e \Rightarrow p$ is weaker than $e \xrightarrow{+} p$ in that it only asserts that p holds as long as e holds.)

Lamport [23] states the principles of the composition as follow. Suppose a system M is composed of components $M_1, \ldots, M_n$. Each component guarantees its specification under a specific environment assumption. If the following conditions are met, then the principles of the composition infer that the global system M guarantees the global specification p under the global environment assumption e. These conditions are:

1- Every component M_j guarantees its specification p_j under the environment assumption e_j. ($e_j \xrightarrow{+} p_j$, for $0 \leq j \leq n$)

2- The environment assumption e_j of each component is satisfied under the global assumption e and the inout properties of all components. That is
$(e \wedge p_1 \wedge \ldots \wedge p_n \Rightarrow e_j)$, for $0 \leq j \leq n$.

3- M guarantees p if each component M_j guarantees p_j. This means that the global system specification p is implied by the component properties p_j, i.e.,

$(e \wedge p_1 \wedge \ldots \wedge p_n \Rightarrow p)$.

A composition rule based on the composition principles has been implemented in a special linear temporal logic, called TLA [21]. TLA uses a theorem proving approach. Conditions (1) to (3) of the composition are manually verified using a proof assistant. However, Lamport and Kurshan [33] presented a hybrid approach to compute Steps (2) and (3) of the rule by a theorem prover, and Step (1) by a model checker.

The composition rule, as expressed in TLA can be applied only to safety properties [15]. A safety property holds in all states of a model. Safety properties assert that bad things "never" happen. A liveness property on the other hand talks about eventual occurrences of events. For example, a liveness property in a communication protocol can be as follows: if a good message is sent by the transmitter, it is eventually received by the receiver.

In the next section, we review the compositional methods which use model checking algorithms (rather than deduction-based reasoning) to implement compositional verification. We also discuss whether each method can prove liveness properties in addition to the safety ones.

2.2 Compositional verification

The recent developments in the compositional verification (CV) originated from the assume guarantee (A/G) reasoning[10]. Although very similar, the major difference is in the way they treat the environment model. In A/G approaches, the environment assumptions are stated explicitly, from the beginning, with the component specifications. (That is what we wish to require for the specifications of reusable components.) In compositional verification methods, the environment assumptions do

[10] These compositional approaches attempted to implement structural induction by using model checking algorithms [31][14]. A more complete survey of compositional methods, e.g., the methods that generally infer system properties from component properties can be found in COMPOS97 proceedings [49]

not explicitly exist. They are subsequently obtained by the abstract models of the components of the system, surrounding a given module. These so obtained environment assumptions create an abstract context for the verification of the component(s). It is thus verified that the component satisfies its properties in the abstract environment.

2.2.1 Non circular compositional verification

Pnueli [4] presented an assume guarantee rule for the temporal logic model checking. The rule states that if a module *M1* satisfies a property p_1 and then assuming this formula, a module *M2* satisfies a property p_2, then the system *M1* // *M2* satisfies the property p_2.

$$\frac{\begin{array}{ll} <> & M1 <p_1> \quad (2.5\_1) \\ <p_1> & M2 <p_2> \quad (2.5\_2) \end{array}}{<> M1 \,//\, M2 <p2>} \quad (2.5)$$

In the rule, (2.5_1) discharges property p_1 which is assumed in (2.5_2). This kind of reasoning has been implemented in model checkers such as SMV [16]. When the properties p_1 and p_2 are known, this rule provides the benefit that direct reasoning about the composed system *M1* // *M2* is avoided. In practice, nevertheless, determining such properties may be highly non-trivial [20].

2.2.2 Circular compositional verification

The inference rule (2.5) presented in Section 2.3.1 is not circular. The first subgoal of the rule states that *M1* satisfies p_1 without any further assumption. Generally, this is not the case and *M1* may make certain assumption p_2 about *M2* in order to maintain p_1. The rule thus becomes circular. *M1* satisfies p_1 if *M2* satisfies p_2 and vice-versa, *M2* satisfies p_2 if *M1* satisfies p_1. From propositional logic point of view, this circular reasoning may conclude wrong results. One cannot deduce $p_1 \wedge p_2$ from $p_1 \Rightarrow p_2$ and $p_2 \Rightarrow p_1$. For instance, both predicates p_1 and p_2 may be false and yet $p_1 \Rightarrow p_2$ and

$p_2 \Rightarrow p_1$ are true. We illustrate this by an example.

Example 2.1 (*cycle-of-gates*) Consider a cycle of NOT gates *N1* and *N2* in Figure 2.2a. Let $p_1 := (i = 0)$ and $p_2 := (o = 1)$. Under the assumption G p_1, i.e., G $(i = 0)$, *N1* satisfies G p_2, i.e., G $(o = 1)$. Similarly, assuming G $(o = 1)$, *N2* satisfies G $(i = 0)$. However, the conclusion $G(i = 0) \wedge G(o = 1)$ in $N1 \parallel N2$ is wrong, since we may have $G(i = 1) \wedge G(o = 0)$.

$$
\begin{aligned}
&<> \quad N1 \ < G\,[(i = 0) \Rightarrow (o = 1)] > \quad (2.6\_1)\\
&<> \quad N2 \ < G\,[(o = 1) \Rightarrow (i = 0)] > \quad (2.6\_2)\\
&\text{---------- ---------- } \quad (2.6)\\
&<> \quad N1 \parallel N2 \ < G\,(i = 0) \wedge G\,(o = 1) > \quad \{\text{wrong conclusion}\}
\end{aligned}
$$

If there is at least one unit time delay[11] in the cycle, the conclusion is valid, however, based on induction in time [14]. Let X represent the next state operator of linear temporal logic. We have added a register in Figure 2.2b and the following reasoning is then sound.

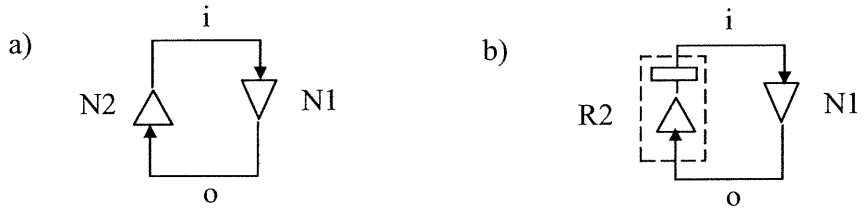


Figure 2.2: a) Cycle-of-gates. (b) The cycle is broken by a register

$$
\begin{aligned}
&<> N1 < G\,[(i = 0) \Rightarrow (o = 1)] > \quad (2.7\_1)\\
&<> R2 < G\,[(o = 1) \Rightarrow X\,(i = 0)] > \quad (2.7\_2)\\
&<> R2 < (i = 0) > \quad (2.7\_3)\\
&\text{---------- } \quad (2.7)\\
&<> N1 \parallel R2 < G\,(i = 0) \wedge G\,(o = 1) >
\end{aligned}
$$

Subgoal (2.7_3) in (2.7) asserts that R2 satisfies (i=0) in the initial state. Let $p_1(\tau)$ denote that p_1 := (i = 0) holds true at time τ. Let p_2 := (o = 1). $p_1(0)$ holds true by (2.7_3). $p_2(0)$ holds true by (2.7_1) at the output of *N1* which is the input to R2. If p_2 holds true at the input of (register) R2 at the current time, then p_1 holds true at its output at the next time, by definition of register R2. Hence, by $p_2(0)$, $p_1(1)$ holds true at R2 output (which is the input to *N1*). Continuing this way, p_1 and p_2 hold at all times, i.e., G $p_1 \wedge$ G p_2. We represent this inductive reasoning by the following general rule:

$\forall t.\ p_1(t) \Rightarrow p_2(t)$	(2.8_1)	$<> M1 <$ G $[p_1 \Rightarrow p_2] >$
$\forall t.\ p_2(t) \Rightarrow p_1(t+1)$	(2.8_2)	$<> M2 <$ G $[p_2 \Rightarrow$ X $p_1]>$
$p_1(0)$	(2.8_3)	$<> M2 < p_1 >$
----------	(2.8a)	----------(2.8b)
$\forall t.\ p_1(t) \wedge p_2(t)$		$<> M1 \,//\, M2 <$G $(p_1 \wedge p_2)>$

Rule (2.8b) implements (2.8a) using temporal logic operators.

McMillan [14] proposed the following rule to implement compositional model checking. Let $p|^{\tau}$ denote that p holds up to (including) time $t = \tau$.

$p_1\|^{\tau} \Rightarrow p_2(\tau)$	(2.9_1)
$p_2\|^{(\tau-1)} \Rightarrow p_1(\tau)$	(2.9_2)
----------	(2.9)
$\forall t.\ p_1(t) \wedge p_2(t)$	

This rule is also sound based on induction on τ. When evaluated at time [14], (2.9_2) asserts that $p_1(0)$ holds true. Let $p_1|^0 = p_1(0)$. Then, $p_2(0)$ holds by (2.10_1), $p_1(1)$ holds by (2.10_2) and so on.

---

[11] In this thesis, we have considered synchronous haradware systems. Gates have no delay and

Rule (2.9) replaces subgoals (2.8_2) and (2.8_3) with one subgoal (2.9_2). Moreover, (2.9_1) and (2.9_2) can be verified using linear temporal logic formulas [15]. For instance, we (reproduce a proof to) show that $p_2|^{(\tau-1)} \Rightarrow p_1(\tau)$ can be computed by $\neg(p_2 \text{ U } \neg p_1)$. By definition, a module M satisfies $(p_2 \text{ U } p_1)$, if p_1 holds true at the initial state or if there exists a state s_j in which M satisfies p_1 and in all states s_k before s_j, M satisfies p_2.

$$p_1(0) \vee [(\exists j>0).\ (\forall(0\leq k<j).\ s_k \models p_2) \wedge s_j \models p_1\ ] \tag{2.10}$$

For $(p_2 \text{ U } \neg p_1)$, this definition becomes:

$$\neg p_1(0) \vee [(\exists j>0).\ (\forall(0\leq k<j).\ s_k \models p_2) \wedge s_j \models \neg p_1\ ] \tag{2.11}$$

By negating (2.11), i.e., $\neg(p_1 \text{ U } \neg p_2)$, we obtain the following expression. (Note that $\neg\ [(\exists j>0).\ Q(j)] \equiv [(\forall j>0).\ \neg Q(j)].$)

$$p_1(0) \wedge [(\forall j>0).\ \ \neg(\forall(0 \leq k < j).\ s_k \models p_2) \vee s_j \models p_1\ ] \tag{2.12}$$

or

$$p_1(0) \wedge [(\forall j>0).\ (\forall(0\leq k<j).\ s_k \models p_2) \Rightarrow s_j \models p_1\ ] \tag{2.13}$$

A module satisfies $p_2|^{(\tau-1)} \Rightarrow p_1(\tau)$ if it satisfies p_1 in the initial state and (for all j) when module satisfies p_2 up to (and including) *state* s_{j-1}, then it satisfies p_1 at state s_j.

$$p_1(0) \wedge [(\forall j>0).\ (\forall(0\leq k<j).\ s_k \models p_2) \Rightarrow s_j \models p_1\ ] \tag{2.14}$$

That is,

$$[p_2|^{(\tau-1)} \Rightarrow p_1(\tau)] \equiv [\neg(p_2 \text{ U } \neg p_1)\ ] \tag{2.15}$$

---

registers have one clock period delay. The clock period represents one unit "time" delay.

Now, we show that $p_1|^{\tau} \Rightarrow p_2(\vec{\tau})$ can be computed by $\neg(p_1 \text{ U } (\neg p_2 \wedge p_1))$. By definition, we have that

$$[p_1|^{\tau} \Rightarrow p_2(\vec{\tau})] \equiv [(p_1|^{(\tau-1)} \wedge p_1(\vec{\tau})) \Rightarrow p_2(\vec{\tau})] \tag{2.16}$$

Using the relation $[(p \wedge q) \Rightarrow r] \equiv [p \Rightarrow (q \Rightarrow r)]$, we get

$$\begin{aligned} [p_1|^{\tau} \Rightarrow p_2(\vec{\tau})] &\equiv [(p_1|^{(\tau-1)} \wedge p_1(\vec{\tau})) \Rightarrow p_2(\vec{\tau})] \\ &\equiv [p_1|^{(\tau-1)} \Rightarrow (p_1(\vec{\tau}) \Rightarrow p_2(\vec{\tau}))] \\ &\equiv [p_1|^{(\tau-1)} \Rightarrow (\neg p_1(\vec{\tau}) \vee p_2(\vec{\tau}))] \end{aligned} \tag{2.17}$$

Rewriting (2.17) using (2.15) gives the following result:

$$\begin{aligned} [p_1|^{\tau} \Rightarrow p_2(\vec{\tau})] &\equiv [\neg(p_1 \text{ U } \neg(\neg p_1 \vee p_2))] \\ &\equiv [\neg(p_1 \text{ U } (p_1 \wedge \neg p_2))] \end{aligned} \tag{2.18}$$

Using (2.15) and (2.18), the circular model checking rule (2.9) is computed by checking the following two temporal logic formulas on *M1* and on *M2*.

$$\begin{gathered} <> \quad M1 <\neg(p_1 \text{ U } (\neg p_2 \wedge p_1)> \\ <> \; M2 \quad <\neg(p_2 \text{ U } \neg p_1)> \\ \text{----------} \\ <> \; M1 \,//\, M2 < \text{G } (p_1 \wedge p_2) > \end{gathered} \tag{2.19}$$

In summary, we conclude that if there is no cycle-of-gates in the system, i.e., every cycle is cut by at least one unit delay, the circular reasoning is sound, based on the induction over time introduced by the delay element.

Note that p_1 and p_2 can represent safety and liveness properties. Rule (2.19) (unlike other approaches) can then be used to prove liveness properties. We will use (2.8b)

(which is our approximation of (2.19)) to prove a liveness property of an ATM switch in Chapter 5. In Appendix 2, we prove that (2.8b) $\Rightarrow$ (2.19), i.e., (2.8a) is a conservative approach to (2.19).

Next, we review the assume guarantee reasoning in reactive modules [35]. With reactive modules, the specifications of components are not described using temporal logic formulas. Instead, the specification is a higher-level design of the component, thus another reactive module. This framework uses trace-containment relation to show that a component implements a specification.

2.2.3 Reactive modules [35]

A reactive module M (or module M, for short) has a finite set of variables, denoted V_M. A "state" of M is a valuation for V_M. The module represents a system that interacts with an environment. V_M is partitioned into three sets; input variables I, output variables O, and privates variables P. While O and P are updated by M, I is updated by the environment. M contains two predicates to assign values to the variables in $(O \cup P)$; an initial action that assigns initial values to $(O \cup P)$, and an update action that assigns updated values to them. For every state s of M, and for every valuation of I, there exists a finite number of next states for s. In other words, the update predicate is always executable, and the system is prepared to respond to all possible environment moves. M does not constrain the behavior of I variables and thus interacts with the environment in a nonblocking way.

A module M consists of one or more atoms that control $(O \cup P)$ variables of the module. Each atom controls one or more variable, however, every variable is controlled by one and only one atom. Let X_a be a finite set of variables of an atom a. X_a contains three sets of variables; a set of controlled variables $ctrX_a \subseteq X_a$, a set of read variables $readX_a \subseteq X_a$, and a set of awaited variables $waitX_a \subseteq \{X_a \setminus ctrX_a\}$. A controlled variable of an atom may depend sequentially on a read variable of the atom, much like a register output that depends on register input. A controlled variable

y of an atom may depend combinationally on an awaited variable x of the atom. This is denoted $x <_a y$ to indicate that atom a can update y only after x has been updated.

A module M consists of its atoms which have the following properties. (1) Controlled variables of atoms are disjoint, i.e., for every atom a and b of M, $ctrX_a \cap ctrX_b = \varnothing$. (2) The set $(O \ U \ P)$ of M equals the set $(U_{a \in atoms} \ ctrX_a)$ of the atoms. (3) The transitive closure of $<_M = (U_{a \in atoms} <_a)^+$ is asymmetric. The third condition ensures that the await dependencies among the variables of M are acyclic, and consequently, there exists a consistent ordering for updating all atoms of M [35].

The execution of a module results in a trace of observation. For two states s and t of M, the state t is a successor of s if t can be obtained from s, by executing updating actions of M. A trajectory of M is a finite sequence $s_0 ... s_n$ of states such that (1) s_0 is an initial state and (2) for all $0 \leq i < n$, the state s_{i+1} is a successor of s_i. If s is a valuation to variables V_M of M and $W \subseteq V_M$, then $[s]_W$ denotes the valuation restricted to W. Let IO denote the variables in $(I_M \cup O_M)$. If $\bar{s} = s_0 ... s_n$ is a trajectory of M, then its projection sequence $[\bar{s}]_{IO} = [s_0]_{IO} ... [s_n]_{IO}$ is a trace of M. The trace language L_M of M is the set of traces of M.

A module M implements a module N, written $M \leq N$, if the following conditions are met: (1) $O_N \subseteq O_M$;(2)$I_N \subseteq (I_M \cup O_M)$; (3) for $\forall x \in (I_N \cup O_N)$ and $\forall y \in (O_N)$ such that we have $x <_N y$, then $x <_M y$; and (4) if $\bar{\bar{s}}$ is trace of M, then the projection $[\bar{s}]_{IO}$ is a trace of N, where $IO = (I_N \cup O_N)$.

Two modules *M1* and *M2* are "compatible" if (1) $O_{M1} \cap O_{M2} = \varnothing$, and (2) the transitive closure $(<_{M1} \cup <_{M2})+$ is asymmetric, i.e., the await dependencies among IO variables of *M1* and *M2* are acyclic.

If *M1* and *M2* are two compatible modules, then the composition *M1* // *M2* is the module with $P_{(M1 \parallel M2)} = P_{M1} \cup P_{M2}$, $O_{(M1 \parallel M2)} = O_{M1} \cup O_{M2}$, $I_{(M1 \parallel M2)} = I_{M1} \cup I_{M2} \setminus O_{(M1 \parallel M2)}$, and $Atoms_{(M1 \parallel M2)} = Atoms_{M1} \cup Atoms_{M2}$.

Assume guarantee [35]: Let *M1* and *M2* be two compatible modules, and let *N1* and *N2* be two compatible modules such that $I_{(N1 \parallel N2)} \subseteq IO_{(M1 \parallel M2)}$. If $M1 \parallel N2 \leq N1$ and $M2 \parallel N1 \leq N2$, then $M1 // M2 \leq N1 \parallel N2$.

The steps of the assume guarantee rule are illustrated in Figure 2.3. The proof of the rule, based on induction on the trace length is given in Appendix 1.

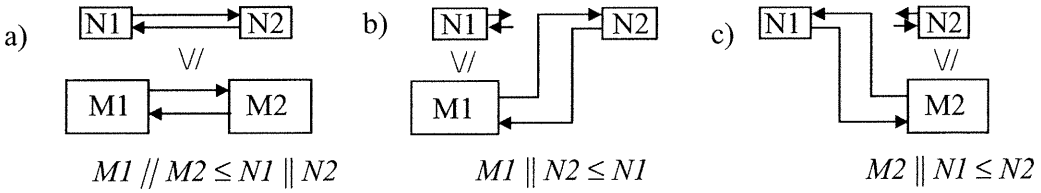


Figure 2.3: The assume guarantee rule. Steps b) and c) prove a)

2.3 Summary

We have presented recent works in assume guarantee (A/G) reasoning and its descendant, compositional model checking (CMC). In the original A/G reasoning [22], the environment assumptions of the modules are explicitly stated, a priori along the component specification. In this way, a closed system, (i.e., the component and its environment assumptions) is specified, as the specification of each component. In CMC approaches, the environment assumptions do not explicitly exist. However, an abstract environment is obtained by the properties or abstract models of the components of the system, surrounding a given module. These so obtained assumptions create an abstract context for the verification of components. If we want to apply compositional verification methods to designs constructed using reusable

components (e.g., intellectual property (IP) blocks), the environment assumptions of these components must be supplied with the component specification so that the components could be safely used in any application.

The following classification represents the works reviewed in this chapter.

1. (Non-circular compositional model checking) Long [31] presented a composition rule that non-circularly verifies a composed system using model checking algorithms. For instance, it is first verified that a component satisfies its properties under certain assumptions. It is then verified that the other components of the system satisfy those assumptions, without any assumption about the first component. Long showed how to transform a temporal logic formula to a Kripke structure so that to compose components with formulas. This composition allowed the assumtion/guarantee be implemented using model checking systems. This framework supports both safety and liveness property verifications.

2. (Assume guarantee reasoning) Abadi and Lamport [22] assuming an interleaving model of concurrency, proposed a circular compositional rule to verify safety properties of the composed systems. The rule is originally implemented by theorem provers and supports only safety properties.

3. (Assume guarantee with synchronous communications) Alur and Henzinger [35] extended the interleaving unit-delay model of the components in [22] to synchronous Mealy machines. These machines may contain zero-delay communication from machine inputs to machine outputs. This framework imposes a well-foundedness condition that there must be no-cycle-of-gates in the composed Mealy machines. This method supports only safety properties.

4. (Cicular compositional model checking) McMillan [14][15][17][19] contributed in two ways to the compositional model checking literature. First, he relaxed the condition that only one component (or one atom) constrain any output of the system.

To resolve the conflicts, he introduced a refinement relation among those modules that constrain one output [14]. Second, he introduced an inductive rule and verified liveness properties of composed systems, in addition to the safety properties.

Compositional methods use abstract models of the components, either in the form of temporal properties or in the form of higher-level designs to compute subgoals of the composition rule efficiently. A rule is efficient if the abstract models are logically sufficient to carry out the proof obligations. However, when either of them is not strong enough, the rule cannot yield the desired results. The abstractions constitute both the strength and the weakness of the methods. They make the computation efficient. But in practice the problem remains how to obtain the appropriate abstract models.

In the next chapter, we propose a compositional rule based on the interactions observed at the interfaces of components. This method uses interface interactions as the formalism of the specification and abstraction. We present interface recognizers/suppliers (IRS) which enable us to symmetrically verify a property or supply an assumption on components inputs. IRS can be used to organize an end-to-end verification of modular systems.

Chapter 3

Constraints in model checking

In chapter 2, we mentioned that during model checking, environment assumptions of components could be provided as temporal logic formulas. Industrial model checkers, e.g., Formal Check [3] or Verdict [25] have mechanisms to specify these as verification constraints. In this chapter, we study the requirements that the constraints and the components must satisfy so that the reasoning about them is valid. We then present a methodology based on interface recognizers/suppliers (IRS) to implement constrained model checking. We propose a composition rule and discuss its well-foundedness.

3.1 Constrained model checking

Developing environment models during formal verification of components in modular systems is a time-consuming and error-prone activity. Kaufmann et al. [25] suggest using "constraints" as a simple way to model the environment. A constraint is a Boolean formula involving any signals in the design. Constraints appear at three levels of granularity: (1) At the first level, they involve only input signals of the component. Suppose A and B are inputs of a module M. Then, $constraint[$\neg(A \wedge B)$] specifies a constraint in Verdict [25] that restricts the inputs of M so as to always satisfy $\neg(A \wedge B)$. (2) At the second level, constraints may also depend on the internal state of the design. For instance, $constraint[$(state = s_1) \Rightarrow (A \vee B)$] defines a constraint that depends on state s_1 of M. Implicitly, it is assumed that the design by itself contains the information necessary to determine what its (next) input should be. Consequently, the inputs of the design are combinationally computed from the state

information of the design. (3) At the third level, the inputs may depend not only on the current state of the design but also on the history of reactions of the design to its inputs. In this case, a finite state machine called "monitor" is defined to watch and record the information needed to determine the next inputs to the design. With the addition of the monitors, constraints become as expressive as environment models [25].

Constrained model checking performs reachability analysis over those computations that globally satisfy the constraints. (A computation path is an infinite sequence of states [37].) It has to be ensured that (1) this state space is not empty, i.e., the design has at least one initial state that satisfies the constraints. It also has to be ensured that (2) the constrained model does not contain any "dead-end" state, i.e., every reachable state of the constrained model should have at least one successor state that satisfies the constraint (a dead-end state is a reachable state that does not have a next state that satisfies the constraints [25]). Although traditional hardware modules satisfy these conditions, no-dead-end condition may fail in the presence of constraints. These conditions can be verified using temporal logic model checking. Suppose (M, s_0) $\models$ $AG\ p$ indicates that a property p holds true in all states of all computation paths starting from s_0. Similarly, (M, s_0) $\models EG\ p$ asserts that there exists a computation path on which all states satisfy p. Let C denote a constraint, e.g., a Boolean formula. Suppose that M satisfies C in some initial state $s0$, i.e., (M, $s0$) $\models C$. Suppose that assuming M satisfies C in the current state, we can prove that there exists some next state that satisfies C, i.e., (M, $s0$) $\models AG(C \Rightarrow EX\ C)$. Then, M composed with C does not have any dead-end state. It is also said that the constrained model M_C of M is "model checkable" [25].

The semantics of the constrained model checking is defined as follows [25]. Let S be a set, and R be a binary relation. Let Img(S, R) denote the image of S by R, i.e., $\{s' \mid \exists s \in S.\ R(s, s')\}$. Consider a Kripke structure M with states S, transition relation T, and initial states Init. Let C be a Boolean constraint over S and the signals of the design. Let $CInit \neq \varnothing$ be the initial states of M that satisfy C. States CS of M_C are

obtained from S by restricting the reachable states of M to the states that respect C. In terms of the fix point computations, CS is the least fix point of the following monotone functional F [25].

$$F(Y) = CInit \cup C(Img(Y, T)),$$

where $C(Z)$ represents the set of states in Z that satisfy C. CS is the least set Y of states containing Cinit such that for every $(s, s') \in T$, for which $s \in Y$ and s' satisfies C, $s' \in Y$.

3.2 Assume guarantee in constrained model checking

Let a constraint C be a simple propositional logic formula, i.e., free of temporal logic operators (G, F, X, U). Suppose that a model M satisfies a property p under constraint C. This is denoted

$$<GC> M <p>,$$

where G represents the "global" operator of temporal logic. M preserves p whenever it works in an environment E that satisfies GC. Therefore, there is an obligation to prove that the environment E satisfies GC. However, the verification that E satisfies GC may fail, since E may in turn make some assumption about its inputs from M. Kuafmann et al. [25] suggested to verify that $E \parallel M$ satisfies GC (Figure 3.1b). However, when the state space of $E \parallel M$ becomes larger than the capacity of the model checker, this subgoal is informally verified by a simulation. For instance, it is verified that the constraint is not violated in $E \parallel M$ during simulation. In Section 3.4, we show that by abstracting the component it is possible to formally discharge the environment assumptions using a less complex subsystem.

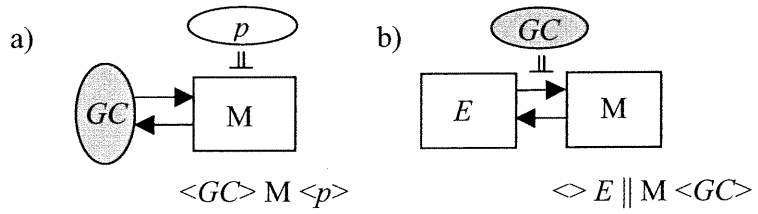


Figure 3.1: Assume guarantee reasoning in constrained model checking.
a) Module M under constraint *GC* satisfies property *p*.
b) *E*||M discharges assumption *GC*.

Monitors can be added to the design to provide history variables for constraint definition. This type of verification is reviewed in the next section.

3.3 Constrained model checking with monitors

The inputs of a design may depend on the current state of the design as well as on the history of interactions that occurred between the module and its environment. In this case, a finite state machine called monitor is defined to record such information. A monitor has multiple inputs and one output [25]. It watches the inputs to ensure that they are behaving as expected. It could monitor, for instance, that the interactions follow a given protocol.

Monitors like auxiliary variables [19] may provide extra signals for constraint definitions. The output of a monitor can be used in a constraint to form a "sequential" constraint (Figure 3.2a). Similarly, the same monitor can be used when discharging the constraint (Figure 3.2b). However, the system *E* || M may become very complex. Then, M should be abstracted away from this verification. We study this kind of verification in the next section in the context of constrained model checking using interface recognizers/suppliers.

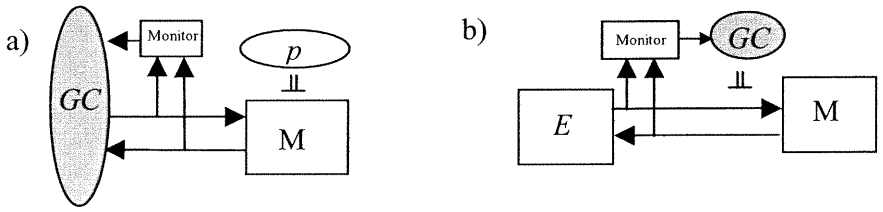


Figure 3.2: Constrained model checking using monitors.
a) < *GC* > M || Monitor <*p*>. b) <> *E* || M || Monitor <*GC*>.

3.4. Interface recognizer/suppliers

We extend assume guarantee reasoning in constrained model checking in two ways. First, we propose an interface-based verification methodology, assuming that the internal states of the components are not accessible (or visible) to the environment. This is the case, for instance, when the state information of the design is not provided. In that case, whenever a sequential property (of the interface) is concerned, the use of monitors becomes inevitable. We provide a tighter connection between monitors and the constraints by using recognizers. Moreover, we make an abstraction of the component when discharging the environment assumptions. This second abstraction makes the methodology symmetric, i.e., the environment is abstracted (using the constraints) to model check the component and the component is abstracted (using the constraints) to model check the environment. This circular reasoning is not sound in general. We must avoid propositional circularity, by implementing an appropriate framework within the compositional model checking methodologies presented in Chapter 2.

Before presenting the formal foundation of the methodology, we give an example to introduce interface recognizers/suppliers (IRS).

Example 3.1 (*Interface recognizers/suppliers*) Suppose we want to design a generator to produce pulses of unit length on signal f such that they are at least 5 clock cycles apart (Figure 3.3a). A recognizer can be developed for the generator (Figure 3.5b). The recognition is encoded by a set of constraints in each state of the recognizer. For instance, consider the automaton in Figure 3.3b that recognizes the sequences that could be produced by the pulse generator. The signal f can be zero or one in state *s1*. After becoming 1 in *s1*, f becomes zero in *s2*. In *s3*, the recognizer uses a counter that counts modulo 5. f should remain zero in *s3* until state *s1* is reached. Note that if f gets a value other than the ones specified in the specification (in Figure 3.3a), the recognizer will not change its state.

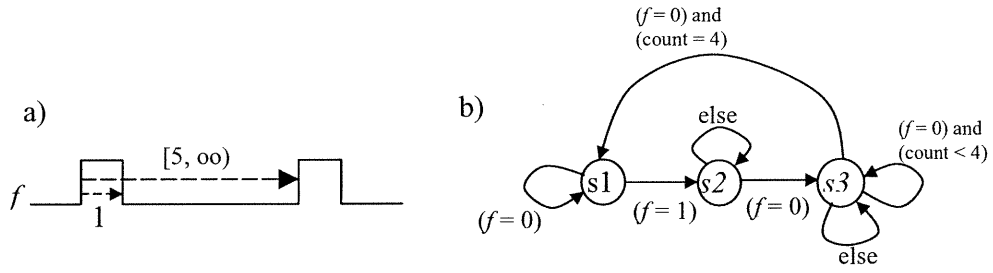


Figure 3.3: A recognizer for the pulse generator f.
a) Specification. b) The recognizer. The "variable" count is zero in state *s1*, one in state *s2*, and is incremented in state *s3* until reaching 5.

We define a Boolean signal C_f to monitor f:

$$C_f := [\ (\ (state = s2) \text{ or } (state = s3)\ ) \Rightarrow (f = 0)\ ] \tag{3.1}$$

$C_f = true$ asserts that f is zero in states *s2* and *s3* of the recognizer. (Note that it does not assert that *s2* and *s3* are ever reached. In fact, if *s2* and *s3* are not reached, then C_f = true will still hold.) $C_f = false$ asserts that f is not zero in either of those states. C_f acts like an acceptance condition of the recognizer R such that, when always true, it verifies that the generator F respects the unit cycle as well as the 5 clock-cycle period constraints on f. Figure 3.4a shows the generator F and the recognizer R. Suppose we verify the recognizer (without using the generator) with a model checker like Formal Check [3] and we set a constraint that C_f be always true (Figure 3.4b). $C_f = true$ is a constraint of the recognizer which observes the primary input signal f. By forcing

C_f = *true*, the model checker must supply only those combinations of f which respect the constraint C_f = *true*. The net result is that f can be 1 (or 0) at state *s1* and 0 at states *s2* and *s3*, thus generating exactly the pulses required by the specification. In this way, R supplies constraints on its primary input f. **This *input-shaping* feature of the recognizer enables us to produce almost any signal characterized by the constraints of the recognizer using any model checker. In this configuration, the recognizer forms a *supplier,* i.e., it supplies constraints or assumptions on the (otherwise free) inputs. This type of a recognizers that is augmented by Boolean constraints is called an "interface recognizer/supplier" (IRS).**

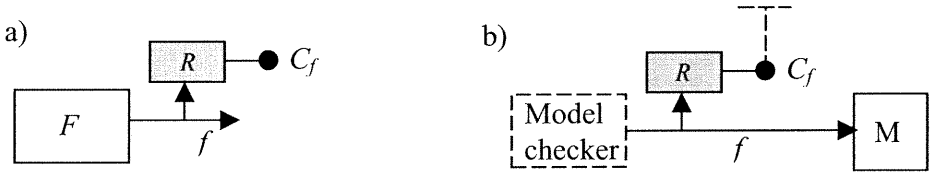


Figure 3.4: Interface recognizer/supplier.
a) Recognizer R for pulse generator F. b) When C_f = *true*, R shapes f for a module M.

Definition 3.1 An IRS machine R = <*S, Init, I, C, T, O*> is a tuple of the following form:

1. S is a finite set of states.
2. $Init \subseteq S$ is a nonempty set of initial states.
3. $I = \{i_1, i_2, \ldots, i_n\}$ is a finite set of inputs.
4. $C = \{c_1, c_2, \ldots, c_m\}$ is a finite set of constraints, where each constraint
   $C_i : S \times 2^I \rightarrow$ {false, true} is a function that determines the constraints value (either false or true) from the current state and the inputs.
5. $T : S \times I \rightarrow S$ is a transition relation.
6. $O : S \times I \rightarrow C$ is a function that determines the constraint value (either *FALSE* or *TRUE*) from the current state and the inputs.

We require that for every state $s \in S$ and input $i \in I$, there exists some $t \in S$ such that $t = T(s, i)$. By this definition, IRS accepts any inputs at any state, i.e., it is receptive to all inputs. As in the automaton in Chapter 2, accepting states F can be added to the IRS definition. We shall address the use of accepting states when we verify liveness properties of a switch fabric in Chapter 5, otherwise, all states may be considered as accepting.

IRS can be used in compositional model checking. Consider a system *M1* // *M2* in Figure 3.5. We want to verify a property or a specification *spec* about the interactions that occurr at the interface of *M1* and *M2*. An IRS *I* is first developed to represent a model for *spec.* Then, two constraints C_x and C_y are defined to recognize *spec,* i.e., to determine what values may happen on interface signals x and y at each state of *I.* Now, *M1* and *M2* can be separately verified using C_x and C_y as follows.

Convention: (*Activating a constraint*) IRS *I* and its constraints C_x and C_y are denoted as $I(C_x, C_y)$. We use the expression "activate a constraint" to indicate that the constraint is set always true during model checking. The formula
$<GC_x>$ M || I $<GC_y>$ asserts that module M composed with *I* where C_x is activated satisfies the property that C_y is always true on *I* (Figure 3.5b). We represent $<GC_x>$ M || *I* $<GC_y>$ by an IRS model checking assertion $I(C_x) \wedge$ M $\models I(C_y)$. As in *TLA* [21], the conjunction *M1* $\wedge$ *M2* represents the composition *M1* || *M2*.

We activate C_x to verify *M1* against C_y (Figure 3.5b), and we activate C_y to verify *M2* against C_x. We study the problem whether it is sound to conclude *M1* // *M2* $\models I(C_x) \wedge I(C_y)$ from the subgoals $I(C_x) \wedge$ *M1* $\models I(C_y)$ and $I(C_y) \wedge$ *M2* $\models I(C_x)$. The following example demonstrates that such circular reasoning may not be sound.

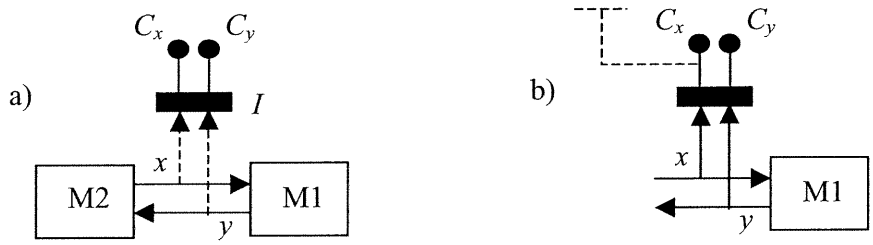


Figure 3.5: A drawing convention for interface recognizer/supplier.
a) Modules M1, *M2*, and an interface recognizer $I(C_x, C_y)$. b) C_x is activated and C_y is checked.

Example 3.2 (*Cycle-of-gates problem in IRS*) Let *M1* and *M2* be two registers as shown in Figure 3.5a. The IRS I that contains two states *s0* and *s1* and its constraints C_x and C_y are defined in Figure 3.6. We want to verify that $I(C_x) \wedge M1 \models I(C_y)$ (e.g., Figure 3.5b). Register *M1* has an initial value, say $y = 0$. Then, by C_x in *s0*, x becomes 1. In the next cycle, i.e., in *s1*, *M1* assigns 1 to y. Then, x becomes 0 in *s1* and so on. Figure 3.6b illustrates this verification in $I(C_x) \wedge$ M1. Note that, although C_x and C_y are similar, the intent in C_x is to restrict x (i.e., the input of M1) and the intent in C_y is to verify a property of y (i.e., the output of M1). The subgoal $I(C_x) \wedge M1 \models I(C_y)$ is thus successfully verified. The subgoal $I(C_y) \wedge M2 \models I(C_x)$ is also verified in *M2* with $x = 0$ initially (Figure 3.6c). But because of the initial values, we have $(x = 0) \wedge (y = 0)$ at all times in $M1 \parallel M2$. Consequently, the conclusion $M1 // M2 \models I(C_x) \wedge I(C_y)$, i.e., $M1 // M2 \models G(x = not(y))$ is not valid. In this example, although there is no cycle-of-gates in $M1 \parallel M2$, there does exist a "cycle-of-gates" in the specification, i.e., in the IRS. From $I(C_x) \wedge M1 \models I(C_y)$ and $I(C_y) \wedge M2 \models I(C_x)$, one cannot deduce $M1 // M2 \models I(C_x) \wedge I(C_y)$.

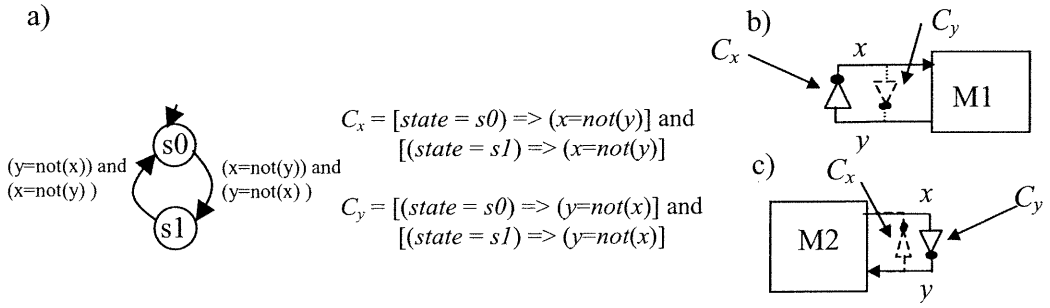


Figure 3.6: Cycle of gates problem in IRS.
a) An IRS *I* and its constraints C_x and C_y. b) C_x is activated and C_y is checked in M1. c) C_y is activated and C_x is checked in *M2*.

To avoid erroneous conclusions, we must define a set of conditions for compositional verification within the IRS framework to be well-founded. For instance, we define what it means to have no "cycle-of-gates" in an IRS and what it means to declare that a constraint and a module are compatible thus the constrained module is model-checkable.

3.5 Logical foundation of composition using IRS

Following the requirements that are proposed for assume guarantee reasoning in reactive modules (Section 2.2.3 in Chapter 2), we propose well-foundedness conditions for compositional verification in the IRS framework. These conditions mainly concern the "cycle-of-gates" and "disjoint-outputs" properties of the subsystems involved in the subgoals of the rule. We define those properties as follows:

Definition 3.2: (*Zero-delay dependency (or zero-delay path)* [14]) We write $x \xrightarrow{0}_M y$ to denote that there exists a gate in module M with y as its output and x as one of its inputs. For instance,

$x := y$ or z, (3.2)

introduces $y \xrightarrow{0}_M x$ and $z \xrightarrow{0}_M x$. The zero-delay dependency relation $\xrightarrow{0}_M$ is transitive, that is, from $x \xrightarrow{0}_M y$ and $y \xrightarrow{0}_M z$ we get a zero delay path $x \xrightarrow{0}_M z$.

Definition 3.3 (*Cycle-of-gates*) Any two variables (x, y) for which $x \xrightarrow{0}_M y$ and $y \xrightarrow{0}_M x$ hold, introduce zero-delay paths from x to y and from y to x. Consequently, they form a cycle-of-gates (or a zero-delay cycle) in M.

Transitive closure of $\xrightarrow{0}_M$ in M is denoted by $\xrightarrow{0}^*_M$. When there are no cycle-of-gates in the module, the relation $\xrightarrow{0}^*_M$ is irreflexive. The predicate $\xrightarrow{0}^*_M$, being irreflexive, anti-symmetric, and transitive becomes a well-founded partial order $\prec_M$ on the variables of the module [19]. This means that there is a consistent ordering for updating all variables of M.

In order to formulate a zero-delay dependency relation in IRS machines, we follow a convention when defining the constraints. Suppose that each constraint restricts only one signal. A constraint that restricts a signal x in some state(s) of an *IRS I* is denoted by C_x. In the simplest case, C_x = *true* assigns a value a to x in some states s_i, ..., s_j of I. Let $P(s) := ((s = s_i)$ or ... or $(s = s_j))$ denote a predicate on states s of I, i.e., given a state s, it returns either *false* or *true*. C_x may define x in terms of other inputs y, ..., z of I. We denote this by a predicate $R(x, y, ..., z)$. In general C_x has the following form:

$$C_x := [\bigwedge_i (P_i(s) \Rightarrow R_i(x, y, \ ..., z))] \quad (3.3)$$

We assume that all constraints of IRS are non-conflicting with each other. So, whenever the constraints are activated, there exists a possible valuation in each state of the IRS for all IRS inputs.

For instance,

$$C_x := [\ ((state = s_0) \Rightarrow x = not(y))\ \wedge\ (\ (state = s_1) \Rightarrow x = 0)\ )\ ] \qquad (3.4)$$

is such a non-conflicting constraint for the IRS shown in Figure 3.6a.

The intention of the predicate $R_i(x, y, \ldots, z)$ of C_x in (3.3) is to restrict x in terms of other signals $y, \ldots, z$. We assume that when C_x is activated, the IRS will be composed with a module M such that M accepts x as input and has $y, \ldots, z$ as outputs. C_x= *true* then constrains the value of x in certain states of I in relation to other variables of the interface. Consider $R1(y, x) \equiv (x = not(y))$ in (3.4). We may call the predicate $R1(y, x)$ of C_x a *conditional* zero-delay dependency or *conditional* zero-delay path (from *M1* output y to *M1* input x, i.e., $y \xrightarrow{0}_{Cx} x$). It is conditional since it is only effective in the appropriate *state*(s) when the constraint is activated. Otherwise (constraint not activated), no restriction is introduced. The conditional path $y \xrightarrow{0}_{Cx} x$ introduced by $x = not$(y) is like a "static" zero-delay path $y \xrightarrow{0}_{M} x$ in $x := not(y)$ which constrains x in terms of y. Similarly, C_x= *true* in (3.3) introduces conditional zero-delay paths between x, y, ..., z, i.e., $y \xrightarrow{0}_{Cx} x, \ldots, z \xrightarrow{0}_{Cx} x$.

Conditional zero-delay dependencies are symmetrical. If a signal x is constrained by a signal y via a conditional path $R(y, x)$, then y is also constrained by x via $R(y, x)$. For instance, in $x = not(y)$, x restricts y and y restricts x. Given a predicate $R(y, x)$, one cannot deduce whether x restricts y or y restricts x. However, when composing a constraint and a module, depending on the application context, it will be fixed which variable is actually constrained and which is the constraining one. This is different than a zero-delay path $\xrightarrow{0}_{M}$, which is inferred from the gates that clearly distinguish causal dependency from input to output. The predicate $\xrightarrow{0}_{M}$ must be anti-symmetric to have no-cycle-of-gates in M. A cycle-of-gates in IRS must be defined differently to take into account the symmetrical conditional zero-delay paths that are introduced by the constraints of the IRS. For instance, the direction of each

conditional path must be determined after composing the constraint and the component. Then, it must be examined whether or not these paths can form a cycle.

Example 3.3: (*Conditional cycle-of-gates*) Let the constraints C_x and C_y of an IRS I be given as $C_x := [(state = s_i) \Rightarrow (x = not(y)]$ and $C_y := [(state = s_j) \Rightarrow (y = x)]$. By C_x, there is a conditional binary predicate $x = not(y)$ over y and x in state s_i, and by C_y, there is a conditional binary predicate $x = y$ over x and y in state s_j. Assume that the naming convention is respected, i.e., C_x is used to restrict x as input to one module and C_y is used to restrict y as input to the other module. C_x and C_y introduce the paths $y \xrightarrow{0}_{Cx} x$ and $x \xrightarrow{0}_{Cx} y$, respectively. If $s_i = s_j$, then $y \xrightarrow{0}_{Cx} x$ and $x \xrightarrow{0}_{Cx} y$ form a conditional (conflicting) cycle-of-gates in the IRS. However, if $s_i \neq s_j$, C_x and C_y form no cycle-of-gates in I, since states s_i and s_j of the IRS are at least one clock cycle apart. (We will proof this by Theorem 1 in this section.)

Example 3.3 illustrated that when conditional paths co-exist in a state of IRS a cycle is formed. We define conditional cycle-of-gates as follows:

Definition 3.4 (*Conditional cycle-of-gates*) For an *IRS I*, let $C_x := [P1(s) \Rightarrow R_1]$ and $C_y := [P_2(s) \Rightarrow R_2]$ be two constraints where $P1$, P_2 , R_1, and R_2 are some predicates and x and y are IRS inputs. When the following conditions (*C1*, *C2*, and *C3*) hold, a conditional cycle-of-gates $y \xrightarrow{0}_{Cx} x \xrightarrow{0}_{Cy} y$ is formed in $I(C_x) \wedge I(C_y)$.

C1 (*Non unary predicates*) R_1 and R_2 are defined over x, y, and possibly some other variables.

In this way, R_1 (of C_x) establishes a "conditional" zero-delay path $y \xleftrightarrow{0}_{Cx} x$ in I while R_2 (of C_y) establishes a conditional zero-delay path $x \xleftrightarrow{0}_{Cy} y$ in I. (Figure 3.7 shows an example with $R_1(x, y) := (x = not(y))$ and $R_2(x, y) := (y = not(x))$.)

C2 (*Common states*) $P1(s)$ and $P_2(s)$ are defined on some common states of I, i.e., there exists a state t such that $P1(t) \wedge P_2(t)$ is true. Consequently, the conditional zero-delay paths $y \xleftarrow{0}_{Cx} x$ and $x \xleftarrow{0}_{Cy} y$ will co-exist in t.

C3 (*Directions*) C_x is activated when verifying C_y in *M1* (e.g., $I(C_x) \wedge M1 \models I(C_y)$ in Figure 3.7) and C_y is activated when verifying C_x in *M2* (e.g., $I(C_y) \wedge M2 \models I(C_x)$ in Figure 3.7).

The components in the design determine the actual direction of the conditional zero-delay paths. Without considering the design components, the direction of the conditional paths cannot be known. Practically, when there exist some actual paths in the design (with zero-delay or more) from x to y, C_x is activated when proving C_y. Otherwise (no path from x to y in the design), C_x cannot help in proving C_y. Therefore, the direction of a path in the design will choose the direction of conditional paths. A conditional cycle-of-gates can be visualized when both constraints C_x and C_y are simultaneously activated and more importantly the direction of the conditional paths are known. To test for the existence of a cycle of gate, the following has to take place:

(1) Activate all constraints of the IRS to detect all the conditional zero-delay paths.
(2) Determine the directions of these paths from the way the constraints are composed with the components in the subsystem verifications.
(3) Consider the states of the IRS that these constraints are defined on, in order to verify that the paths co-exist in some states of the IRS.

The fact that a conditional path has no direction (prior to the component verifications) is a unique property of the IRS that enables them to symmetrically recognize a property or supply a constraint without further code modifications. This feature will be used in Chapter 5 where we prove end-to-end properties of a switch module.

Example 3.4. (*Unary and binary predicates*) Let $C_x := [(state = s_1) \Rightarrow (x = not(y)]$ and $C_y := [(state = s_1) \Rightarrow (y = not(x)]$ be two constraints in Example 3.2. With these constraints, a conditional cycle-of-gates $y \xrightarrow{0}_{Cx} x \xrightarrow{0}_{Cy} y$ is formed in the IRS. However, if C_y is modified to become $C_y := [(state = s_1) \Rightarrow (y = 0)]$, then no cycle is created since C_y does not introduce a zero-delay path from x to y (i.e., Condition *C1* of Definition 3.4 does not hold.). C_y in the latter case will contain a unary predicate (i.e., $y = 0$) while C_x still contains a binary predicate (i.e., $y = not(x)$). A unary predicate like a component that has one output but no input does not introduce any (input-output) path. Then, C_x and C_y create no cycle.

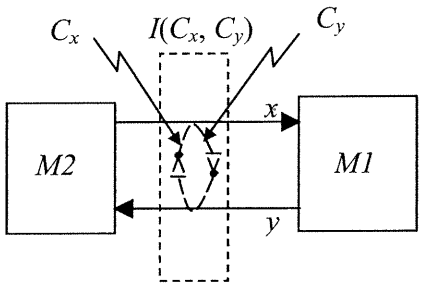


Figure 3.7: Conditional cycle-of-gates in IRS $I(C_x, C_y)$.

Having defined a conditional cycle-of-gates, we proceed to specify the well-foundedness/compatibility conditions for the compositional verification using IRS machines.

3.5.1 Well-foundedness/compatibility conditions (WFC)

We present the well-foundedness conditions for compositional verification in IRS framework following the conditions presented for assume guarantee reasoning in reactive modules (Section 2.2.3).

Let *M1*, *M2* be two modules and $I(C_x, C_y)$ be their interface recognizer/supplier machine as illustrated in Figure 3.8.

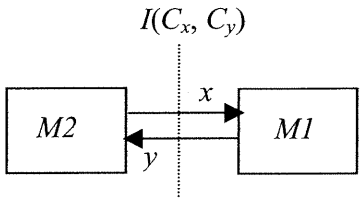


Figure 3.8: Modules *M1* and *M2* and the *IRS* $I(C_x, C_y)$

The following is a list of conditions that are required for a sound composition of modules *M1* and *M2* and IRS constraints.

***W1* (No conflicting outputs in the implementation)** The outputs of *M1* and *M2* are disjoint. Only one module assigns values to a signal of the interface.

***W2* (No cycle-of-gates in the implementation)** *M1* // *M2* contains no cycle-of-gates; In other words, the transitive closure $(\xrightarrow{0}_{M1} \cup \xrightarrow{0}_{M2})^*$ is irreflexive.

Assuming that *M1* and *M2* are receptive, i.e., each one accepts all possible inputs, and the conditions *W1* and *W2* hold true, then *M1* and *M2* are compatible. I.e., there exists a consistent assignment to input/output signals of *M1* and *M2* in *M1* || *M2*. If there were conflicting outputs or conflicting cycle-of-gates, then *M1* // *M2* could deadlock (Section 3.1).

***W3* (No conditional cycle-of-gates in IRS)** The IRS machine I contains no conditional cycle-of-gates (Definition 3.4).

***W4* (No output-constraining)** When IRS is composed with a module, no activated IRS constraint can constrain an output of that module.

We explain when an IRS does not constrain outputs of a module. This condition in fact imitates *W1* that requires that *M1* should not constrain an output of *M2* and vice-

versa. Similarly, an activated constraint can only restrict the input of the module. Consider the subgoal $I(C_x) \wedge M1 \models I(C_y)$ where x is an input and y is an output of $M1$. Let $C_x := [P1(s) \Rightarrow R_1(y, x)]$. C_x must compute a value for x and should not constrain y in $I(C_x) \wedge M1$. More precisely, for any y in $M1$, there must exist a value for x such that the predicate $R1$ of C_x can be satisfied. We express Condition *W4* using characteristic functions. Characteristic function f_Z of a set Z is a function that returns 1 (or true) for any element s of Z and 0 otherwise.

$$s \in Z \Leftrightarrow f_Z(s) = 1 \tag{3.5}$$

Let $y = Y(s)$ be an output function of $M1$ that computes y, for given states of $M1$. Let $f_Y(s, y)$ be the characteristic function of Y, RS denote the set of reachable states of $M1$, and let $f_{RS}(s)$ be the characteristic function of RS. In $I(C_x) \wedge M1$, for $R_1(x, y)$ of C_x not to constrain the output y of $M1$, for any reachable value a of y (via $M1$), there must exist a valuation b of x such that $R_1(b, a)$ holds.

$$\exists s.\ (f_{RS}(s) \wedge f_Y(s, y)) \Rightarrow \exists x.\ R_1(y, x) \tag{3.6}$$

Similarly, $I(C_y) \wedge M2$ must be checked to satisfy the no-output-constraining property (*W4*).

***W5* (No cycle-of-gates in the subsystems)** The compositions $I(C_x) \wedge M1$ and $I(C_y) \wedge M2$ contain no cycle-of-gates. If there is a predicate $R1(y, x)$ in C_x (that introduces a conditional zero-delay path $y \xrightarrow{0}_{Cx} x$), then there has to be no (static) zero-delay path $x \xrightarrow{0}_{M1} y$ in $M1$[12].

$$R1(y, x) \Rightarrow [\neg(x \xrightarrow{0}_{M1} y)] \tag{3.7}$$

Similarly for $R2(y, x)$ of C_y and $M2$.

Note that by $(P \Rightarrow \neg Q) \equiv (Q \Rightarrow \neg P)$ in (3.7), one can infer that when $M1$ contains a zero-delay path from x to y, predicate $R1$ of C_x should not contain any conditional zero-delay path from y to x to prevent forming a cycle-of-gates in $I(C_x) \wedge M1$.

After presenting the well-foundedness/compatibility conditions $W1$-$W5$, we can now define the composition rule as follows:

Assume guarantee with IRS: Let $M1$ and $M2$ be two modules and $I(C_x, C_y)$ be their IRS machine (Figure 3.8). Suppose that the systems (or subsystems) $M1 \parallel M2$, $I(C_x) \wedge M1$, $I(C_y) \wedge M2$, and $I(C_x) \wedge I(C_y)$ that are involved in this assume guarantee reasoning satisfy all of the well founded/compatibility conditions $W1$-$W5$. It follows that:

$$\frac{\begin{array}{c} I(C_x) \wedge M1 \models I(C_y) \\ I(C_y) \wedge M2 \models I(C_x) \\ W1\text{-}W5 \end{array}}{M1 \wedge M2 \models I(C_x) \wedge I(C_y)} \quad (3.8)$$

Proof: We provide a proof sketch based on the same theorem in reactive modules [36]. First, the interface machine $I(C_x)$ can be replaced by a non-deterministic model $N1$ as follows. $I(C_x)$ restricts x in terms of y. module $N1$ has y as its input and generates an x equal to that allowed by $I(C_x)$, as its output. The input/output sequences of $N1$ are exactly those accepted by $I(C_x)$. Similarly, $I(C_y)$ can be replaced by an equivalent module $N2$. Let $N1 \models I(C_x)$ assert that every trace of $N1$ is an

[12] This condition may be too strong. For instance, $M1$ may also contain a conditional path, which is not activated at the same time as the IRS. It would be more precise to examine zero-delay paths of the module and the IRS at each state of the IRS. This would then need more complex computation.

accepted trace of $I(C_x)$. (So, the satisfaction relation $\models$ represents trace containment when left and right hand sides of $\models$ are both modules [44]).
Now, we recall the assumption/gurantee rule in reactive modules from Chapter 2. Reactive modules *M1* and *M2* are "compatible" if (1) their outputs are disjoint and (2) the transitive closure $(<_{M1} \cup <_{M2})+$ is asymmetric, i.e., they form no cycle-of-gates. Let *M1* and *M2* be two compatible modules, and let *N1* and *N2* be two compatible modules such that every input of $N1 \parallel N2$ is an input or an output of $M1 \parallel M2$. If $M1 \parallel N2 \models N1$ and $M2 \parallel N1 \models N2$, then $M1 // M2 \models N1 \parallel N2$.

For comparison, let the reactive modules *M1*, *M2*, *N1*, and *N2* correspond respectively to *M1*, *M2*, $I(C_y)$, and $I(C_x)$ in this theorem. All of the conditions of the assume guarantee theorem in reactive modules have equivalent conditions in rule (3.8), e.g., disjoint outputs (*W1*, *W4*) and no cycle of gates (*W2*, *W3*, *W5*). Moreover, every input of $I(C_x) \wedge I(C_y)$ is an input or an output of $M1 \parallel M2$, by construction of the IRS. Given that $I(C_x)$ and *N1*, and $I(C_y)$ and *N2* are trace-equivalent, by $I(C_x) \wedge M1 \models I(C_y)$ and $I(C_y) \wedge M2 \models I(C_x)$ in the rule (3.8), we have that $N2 \parallel M1 \models N1$ and $N1 \parallel M2 \models N2$, respectively. Then, $(M1 \parallel M2) \models (N1 \parallel N2)$, by the theorem of reactive modules. From $(M1 \parallel M2) \models (N1 \parallel N2)$, we get $(M1 \parallel M2) \models (I(C_x) \parallel I(C_y))$ which using IRS notation is denoted by $M1 \wedge M2 \models I(C_x) \wedge I(C_y)$.
(We recalled the proof of the assume/guarantee theorem in reactive modules in Appendix 1. The theorem is proven by induction on trace length).

This assume guarantee rule shows that the IRS framework is a special case of the reactive modules. However, detecting a "conditional" cycle-of-gates (Definition 3.4) in $I(C_x) \wedge I(C_y)$ is not as intuitive as detecting a "static" cycle-of-gates in $N1 \parallel N2$. Interestingly, this resemblance indicate that $I(C_x) \wedge I(C_y)$, i.e., the IRS when its both constraints are activated, plays the same role as the abstraction modules *N1* and *N2* in reactive modules. In this regard, by considering interactions at the interface of components, IRS suggests a practical approach to the abstraction module development. IRS represents a model for the "joint" behavior of the modules in the composition (unlike to the separate models, e.g., *N1* and *N2* in reactive modules) and

the constraints then adapt the IRS to the specific needs of the left hand side module or the right hand side module of the interface.

Similarly as in reactive modules, the IRS methodology as presented so far is applicable to safety properties only. However, in Chapter 4, we present a verification approach using IRS to verify liveness properties of a switch module.

In the well-foundedness/compatibility conditions, we required that $I(C_x) \wedge I(C_y)$ contains no conditional cycle-of-gates (*W3*). In Definition 3.4, we have illustrated how a conditional cycle-of-gates is formed in an IRS. For instance, when appropriate conditional zero-delay paths occur in the same state of the IRS, i.e., they occur at the same "time", a conditional cycle is formed. We have implicitly assumed that if such conditional zero-delay paths happen on disjoint sets of IRS states, no cycle is formed. In the following, we formally prove this assumption.

Theorem 1 (Non zero-delay cycles) Consider a module *M1* with input x and output y and a module *M2* with input y and output x. *M1* and *M2* and their IRS $I(C_x, C_y)$ is depicted in Figure 3.9a. Let C_x contain one binary predicate $R_{1x}(x, y)$ and a finite number of unary predicates $R_{2x}(x), \ldots R_{nx}(x)$. (We separate the unary predicates from non unary predicates since a non unary predicate can introduce a zero-delay path while a unary predicate cannot. Example 3.4 illustrated this.)

$$C_x := [P1_x(s) \Rightarrow R_{1x}(y, x)] \wedge \bigwedge_k (P_{kx}(s) \Rightarrow R_{kx}(x)] \quad , \ k \geq 2 \tag{3.9}$$

where $P_{kx}(s)$ are predicates on states s of the IRS. C_y is defined similarly.

$$C_y := [P1_y(s) \Rightarrow R_{1y}(y, x)] \wedge \bigwedge_l (P_{ly}(s) \Rightarrow R_{ly}(y)] \quad , \ l \geq 2 \tag{3.10}$$

Suppose that the systems $I(C_x) \wedge M1$, $I(C_y) \wedge M2$, and $M1 \ // \ M2$ satisfy conditions *W1-W5* that apply. If conditional zero-delay paths introduced by $R_{1x}(x, y)$ (of C_x) and

$R_{1y}(x,y)$ (of C_y) occur on disjoint sets of IRS states, then no conditional zero-delay cycle is formed by C_x and C_y, i.e., the set of constraints is well formed.

Proof: Constraint C_x defines one binary predicate over x and y and a finite number of unary predicates over x. The proof does not change if C_x or C_y contains more than one binary predicate, since only the closest one to the initial state of the IRS is involved in the proof.

Consider the subgoal $I(C_x) \wedge M1 \models I(C_y)$ in Figure 3.9b. Since there is a binary predicate $R_{1x}(x, y)$ in C_x, there has to be no zero-delay path $(x \xrightarrow{0}{}^{*}_{M1} y)$ within $M1$ for $M1 \wedge I(C_x)$ to satisfy the no cycle-of-gates condition (*W5*). We have thus shown a register from x to y in $M1$. Similarly, there has to be no zero-delay path within $M2$ from y to x, because of $R_{1y}(x, y)$ (Figure 3.9c).

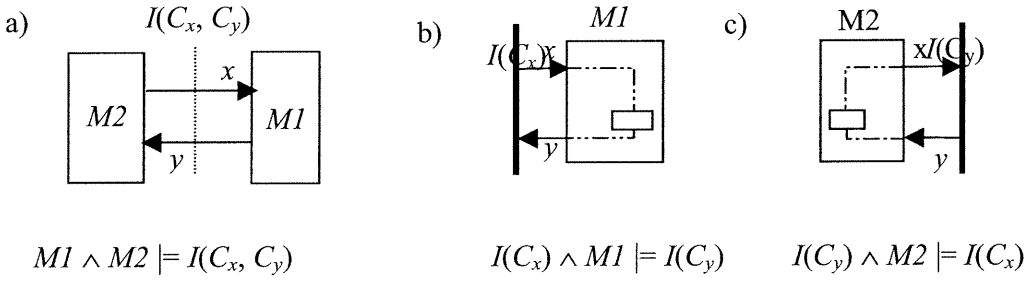


Figure 3.9: Non zero-delay cycles in IRS $I(C_x, C_y)$.

Consider $P1_x(s)$ in (3.9). Let s_i be the first state of the IRS that makes $P1_x(s)$ true.

$$P1_x(s_i) = true$$

Similarly, let s_j be the first state of the IRS that makes $P1_y(s)$ in (3.10) true.

$$P1_y(s_j) = true$$

We have that $s_i \neq s_j$, since otherwise there will be a conditional cycle-of-gates at s_i (Definition 3.4). Suppose, without loss of generality, that $s_i < s_j$, i.e., s_i is reached before s_j in the IRS.

$$s_i < s_j$$

From $(P1_x(s_i) \Rightarrow R_{1x}(x, y))$ in C_x, $R_{1x}(x, y)$ restricts x before $R_{1y}(x, y)$ from $(P1_y(s_j) \Rightarrow R_{1y}(x, y))$ in C_y restricts y, since $s_i < s_j$. While x is restricted by $R_{1x}(x, y)$ in s_i, y is not restricted by $R_{1y}(x, y)$ at that state. Consider the projection of C_y over s_i. First, we show that this projection is not empty and therefore it provides a constraint for y at s_i. Consider the verification $I(C_y) \wedge M2 \models I(C_x)$ at s_i. We denote this by

$$I(C_y) \wedge M2 \models [I(C_x)]_{si} \tag{3.11}$$

Since there is a register in the path y to x via $M2$, x has a value at s_i, for instance, $x = a$ (Figure 3.9c). In subgoal $I(C_y) \wedge M2 \models I(C_x)$, C_y is activated in order to verify C_x. Suppose that y is not restricted at s_i, i.e., $[C_y]_{si} = \varnothing$. (3.11) can then be rewritten as follows:

$$[M2]_{si} \models [I(C_x)]_{si} \tag{3.12}$$

Which can be simplified to:

$$[M2]_{si} \models R_{1x}(a, y) \tag{3.13}$$

If y is not restricted at s_i, then (3.13) fails since $R_{1x}(x, y)$ cannot be proven true with $x = a$ and y free in $M2$. (If (3.13) holds true with $x = a$ and y free, then $R_{1x}(a, y)$ is independent of y, i.e., it is not a binary predicate over y and x. However, $R_{1x}(x, y)$ is a binary predicate by assumption and introduces a zero-delay path from y to x.) Consider the second case that y is restricted at s_i, by a predicate $R_{py}(y)$ of C_y.

$$[C_y]_{si} = R_{py}(y) \tag{3.14}$$

$R_{py}(y)$ is unary predicate and assigns a value b to y such that $R_{1x}(b, a)$ holds at s_i.

$R_{py}(y) \wedge M2 \models R_{1x}(a, b)$

$R_{py}(y)$ must, in turn be discharged by verifying $I(C_x) \wedge M1 \models I(C_y)$ (Figure 3.9b).

$[I(C_x) \wedge M1]_{si} \models R_{py}(y)$

Suppose that to discharge $R_{py}(y)$ at s_i, an extra constraint must be assumed on (the primary input) x at a state $s_q \leq s_i$, e.g, $[C_x]_{sq} = R_{qx}(x)$[13].

$$R_{qx}(x) \wedge [M1]_{si} \models R_{py}(y) \tag{3.15}$$

Figure 3.10 illustrates the sequence of the constraints and the states of the IRS. Consider the verification (3.15). *M1* contains a register in the path from x to y. When proving a property about the output of a register at time t, one cannot use an assumption about values on its input at time t, since, the output at t is independent of the input at t. However, if necessary, one could make an assumption about the input at time (t-1). In order to prove (3.15), i.e., to prove $R_{py}(y)$ (on *M1* output in Figure 3.9b) using $R_{qx}(x)$ (on *M1* input), $R_{qx}(x)$ has to be declared in a state $s_q < s_i$ in IRS (Figure 3.10). We can trace back the chain of dependencies of constraints on the states of the IRS to find out how each constraint is assumed when proving the other constraint. (Note that because of the registers in the design, this tracking is guaranteed to move backward.) However, there has to be a first state s_b in the IRS, where the first constraint is ever declared. This first state exists, since at the earliest, it could be the initial state.

[13] This is similar to proving a property (y = b) at the current state of a register with input x and output y that requires that (x = b) to be assumed in the previous state of the register.

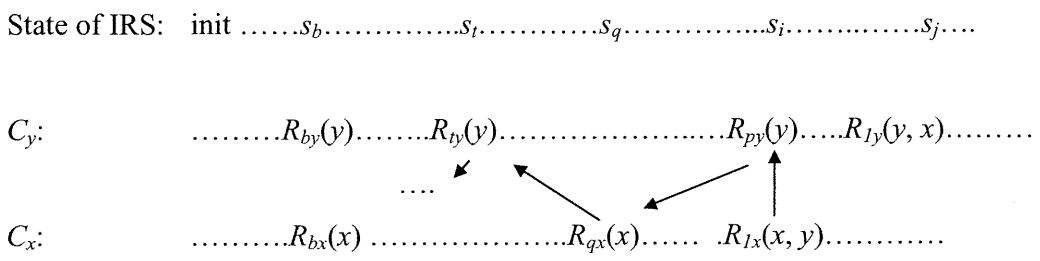


Figure 3.10: Projections of the constraints C_x and C_y over states of the IRS.

There is no constraint before s_b, and every constraint at that state has to hold true without further assumption/constraint on the other part. At s_b, however, we may have two constraints, e.g., $[C_x]_{sb} = R_{bx}(x)$ and $[C_y]_{sb} = R_{by}(y)$. Given that *M1* in $[I(C_x) \wedge M1 \models I(C_y)]_{sb}$ contains a register, $[C_x]_{sb}$ restricts the input of the register at s_b. $[C_y]_{sb}$ defines the output of the register at s_b. The output of the register at each time t is independent of its input at time t. Thus *M1* has to satisfy $[C_y]_{sb}$ without assuming $[C_x]_{sb}$.

$$M1 \models [I(C_y)]_{sb} \tag{3.16}$$

In the same way, *M2* has to satisfy $[C_x]_{sb}$, without using $[C_y]_{sb}$.

$$M2 \models [I(C_x)]_{sb} \tag{3.17}$$

After that, $[C_x]_{sb}$ and $[C_y]_{sb}$ become root assumptions. The set of assumptions/properties is thus well-founded, i.e., there exist verified root assumptions and the set of assumptions is not (zero-delay) circular.

■

By this theorem, non-zero-delay cycles, as opposed to the conditional cycle-of-gates could not introduce invalid results since the dependencies in the non-zero-delay cycles is not (zero-delay) circular. The theorem concludes that when non-unary

predicates happen in disjoint sets of IRS states, no zero-delay cycle is created. We may call such "cycles" non-zero-delay or sequential cycles.

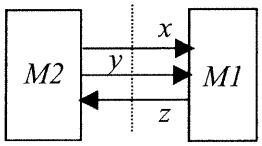
This theorem provides an intuitive justification for the assume guarantee rule (3.8). The correctness of that rule can be inferred from the time delays introduced in no-cycle-of-gates conditions (*W2*, *W3*, *W5*) so that they break every cycle by at least one unit delay [19]. Disjoint-outputs (*W1*), no cycle-of-gates (W2, W3, *W5*) and no-output-constraining (*W4*) conditions assure that all the compositions involved in the rule are model-checkable, i.e., they could not deadlock. These requirements together disallow any zero-delay conditional or static cycles and provide a well-founded non-circular ordering for evaluating all variables and constraint/assumptions in the system.

3.5.2 Generalization

We consider three (schematic) examples of the assume guarantee rule using IRS.

Example 3.5 (*More interface signals*) Figure 3.11 represents a case where *M1* and *M2* communicate using signals *x*, *y*, *z*. The subgoals of the composition rule are given in Figure 3.11b.

a)



b)

$$I(C_x) \wedge I(C_y) \wedge M1 \models I(C_z)$$

$$I(C_z) \wedge M2 \models I(C_x) \wedge I(C_y)$$

Well-foundedness

$$M1 \parallel M2 \models I(C_x) \wedge I(C_y) \wedge I(C_z)$$

Figure 3.11: Application example of the compositional rule using IRS.
(a) Modules *M1* and *M2* and their IRS $I(C_x, C_y, C_z)$. (b) Assume guarantee reasoning.

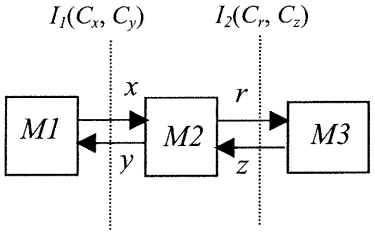
To ensure the well-foundedness of the composition, we have to verify the no-output-constraining (*W1*, *W4*) and the no cycle-of-gates conditions (*W2*, *W3*, *W5*) for subsystems involved in the compositional reasoning. Consider $I(C_x) \wedge I(C_y) \wedge M1$, for instance. To check that this system meets *W4*, we verify that (1) $I(C_x, C_y)$ does not

restrict the output of *M1* and (2) $I(C_x, C_y)$ computes some values for the inputs of *M1*. Let $z = Z(s)$ represent the output function of *M1* that computes z, for a given state s. Let *RS* represent the set of reachable states of *M1*. We have to verify that for any valuation of z in *M1*, there exist some values for x and y such that the predicates *R1* of C_x and *R2* of C_y can be satisfied. Using the characteristic functions, we verify the following.

$$(\exists s.\ f_{RS}(s) \wedge f_Z(s, z)) \Rightarrow (\exists x, y.\quad R1 \wedge R2)\,, \tag{3.18}$$

where, f_{RS} and f_Z represent the characteristic functions of *RS* and *Z*, respectively. Similar condition exists for *R3* of C_z. Other conditions of the well foundedness/compatibility are similar to the basic case in Figure 3.7.

Now, consider our second example in Figure 3.12. The compositional reasoning is given in the figure as well. We verify that all systems involved in the subgoals satisfy the well-foundedness/compatibility conditions.



$I(C_x) \wedge M2 \wedge I(C_z) \models I(C_r)$
$I(C_r) \wedge M3 \models I(C_z)$
$I(C_z) \wedge M2 \wedge I(C_x) \models I(C_y)$
$I(C_y) \wedge M1 \models I(C_x)$
Well-foundedness

$M1 \parallel M2 \parallel M3 \models I(C_x) \wedge I(C_y) \wedge I(C_r) \wedge I(C_z)$

Figure 3.12: Three modules with IRS machines I_1 and I_2.

Consider $I(C_x) \wedge M2 \wedge I(C_z)$. This system contains a cycle-of-gates if (1) C_x contains a predicate $R_x(x, y)$ over x and y (defined in any state of IRS I_1), (2) *M2* contains a zero-delay path from x to r and a zero-delay path z to y, and (3) C_z contains a predicate $R_z(r, z)$ over z and r (in any state of IRS I_2). Similarly, if C_x contains a predicate $R_x(x, y)$ and *M2* contains a zero-delay path from x to y, a cycle-of-gates is formed in the subsystem. Therefore, if the following condition hold, there is "no" cycle-of-gates in $I(C_x) \wedge M2 \wedge I(C_z)$.

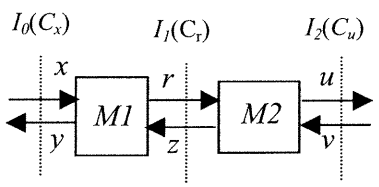
$$R_x(x, y) \Rightarrow \neg[\ (x \xrightarrow{0}{}^{*}_{M2}\ y) \vee\ ((x \xrightarrow{0}{}^{*}_{M2}\ r) \wedge R_z(r, z) \wedge (z \xrightarrow{0}{}^{*}_{M2}\ y)\ ]$$

To respect the no-output-constraining property (*W4*) in $I(C_x) \wedge M2 \wedge I(C_z)$, a predicate R_x of C_x cannot constrain the output y of *M2*, if for any reachable value of y via *M2*, R_x can compute a value for x.

$$\exists s.\ f_{RS}(s) \wedge f_Y(s, y) \ \Rightarrow\ \exists x.\ R_x\ ,$$

where f_{RS} and f_Y denote the characteristic functions of reachable states *RS* of *M2* and output function $Y(s)$ of *M2*, respectively. In the same way, it is checked that C_z does not constrain output r of *M2*. The other systems involved in the reasoning in Figure 3.12 are verified in a similar way to meet the no-output-constraining (*W1*, *W4*) and the no-cycle-of-gates conditions (*W2*, *W3*, *W5*).

Example 3.6 (*Transitivity rule*) Consider the system and compositional reasoning in Figure 3.13. When all the subsystems involved in the reasoning satisfy the well-foundedness/compatibility conditions, compositional reasoning infers an end-to-end property for the composed system. This rule resembles the transitivity rule in propositional logic. We will apply this rule in Chapter 5 to prove end-to-end properties of an ATM switch.



$I_0(C_x) \wedge M1 \models I_1(C_r)$

$I_1(C_r) \wedge M2 \models I_2(C_u)$

Well-foundedness/compatibility

$I_0(C_x) \wedge M1 \wedge M2 \models I_2(C_u)$

Figure 3.13: IRS machines I_0, I_1 and I_2 allow us to prove a property for $M1 \parallel M2$.

3.6 Summary

Kaufmann et al. [25] presented a framework for asymmetric constrained model checking. They suggested (1) to model the environment assumptions with simple constraints of the form $G\ p$ (globally p) and (2) to use the complete system (i.e., the

environment and the module) to discharge the constraints, e.g., Environment || Module /= $G\,p$.

We have extended this framework to a symmetric constrained model checking. We presented IRS machines that specify the "joint" behavior of the environment and the module at their interfaces. An IRS defines a set of constraints that specifies what values may happen on its inputs, i.e., on the signals of the interface. These constraints enable us to verify the environment and the modules separately. Compared to the monitors in asymmetric constrained model checking, the IRS in symmetric constrained model checking is more compact. A Monitor has an output to assert that for instance, a component correctly follows a protocol. Unlike a monitor, an IRS through its constraints shapes/verifies its "inputs", i.e., it does not generate any output signal. This key feature enables IRS to symmetrically recognize a property or supply assumptions on inputs. Using transitivity rule, it can be used to organize end-to-end property verification, which is important for modular systems.

Reasoning with IRS is circular. To avoid erroneous results, we adapted the well-foundedness conditions from reactive modules. We defined and formulated the conditional cycle-of-gates and the no-output-constraining conditions for the systems composed with IRS. In this way, (1) the constrained systems remain model-checkable, as requested by Kaufmann et al. [25], and (2) the overall reasoning is sound, as defined by assume guarantee reasoning in reactive modules [35]. These properties together characterize the IRS methodology.

In the next chapter, we report on the verification of an ATM switch module to illustrate the applicability of the IRS methodology to model checking of large systems that involve complex control path and large data structures. We illustrate how IRS can be used to conduct compositional verifications in modular systems.

Chapter 4

ATM Switch Specification

In this chapter, we specify properties and environment assumptions of an Asynchronous Transfer Mode (ATM) switch. The Fairisle ATM switch [32] was developed at Cambridge University for an experimental network. It consists of a 4x4 switch fabric that performs the actual switching, and four port controllers that handle cell queueing, prioritizing and transfer to the fabric.

There have been earlier efforts in ATM switch verification [32][46][42]. For example, Curzon developed a detailed model of the fabric using HOL (Higher Order Logic). He showed that an RTL (Register Transfer Level) implementation of the fabric implements a higher-level specification written in HOL. Tahar et al. [46] used MDG (Multiway Decision Graph) to automate this verification. Lu et al. [13] using the model checker VIS [48] verified certain safety and liveness properties of the fabric. These works focused on the fabric verification and none of them addressed the port controllers of the switch. Rajan et al [42] introduced a parametric high-level model of an ATM switch and used a combination of formal verification methods (e.g., theorem proving and model checking) and informal verification methods (e.g., simulation) to verify the design. This (validated) high-level design was then synthesized with concrete values for the generic parameters. The idea was to avoid verifying the synthesized concrete designs by verifying only the parameterized higher level ones. Although this work emphasizes the integration of formal techniques in

early design phases, their high-level model left out certain functions, e.g., cell prioritization and data-flow regulation to a central switch controller [42].

Our model of the port controllers is more complex than the one used in the earlier works. (Its VHDL code is about 4000 lines i.e., 7 times bigger than that of the fabric. We provided VHDL models of the port controller and the fabric in Appendix 4.) The controller includes features such as data-flow control and cell prioritization. The switch module (i.e., the fabric and the port controllers) uses acknowledgements from the destination buffers to control the transmission of cells to the buffers. Moreover, the in port controllers contain separate queues for storing and handling the cells of different priorities. This division splits the main cell flow inside the controller into parallel paths. Formal verification of such systems, especially for liveness properties is more complex than in the earlier works that do not include this cell prioritization feature.

Our methodology for the formal verification of ATM switch is compositional model checking using interface recognizer/supplier (IRS), thus different from the previous works which used theorem proving or model checking or a combination of them to verify (somewhat simplified/high level) models. IRS is a formalism for modeling assumptions and safety properties of components. It allows us to subsequently (and easily) discharge these assumptions, i.e., to show that other modules satisfy the assumptions that the component makes about them.

In the next section, we first use IRS to model the environment assumptions of the fabric. Then, we prove that these assumptions are respected by the port controllers.

4.1 Fairisle ATM switch [32]

The switch consists of three types of components: in port controllers, out port controllers and a switch fabric (Figure 4.1). It switches ATM cells from the input transmission lines to the output transmission lines. In a 2x2 switch, there are 2 loop back First-In-First-Out (FIFO) buffers inside the switch that serve to return the cells

back into the switch for special routing, e.g., for multistage switching. An ATM cell is composed of 48 bytes of data (payload) and 5 bytes of header (containing information about the channels). The in port controllers synchronize incoming cells, convert cell headers according to a routing table, append two routing tags *h1* and *h2* in front of each cell, and send them to the fabric. The fabric does not use the information in the header; it is only used by the port controllers. The header is treated by the fabric as additional data of cell. Routing tags *h1* contain all the control information that the fabric needs for arbitration. There is a synchronization signal called frame start (fs) which begins a cycle of a cell transmission. When a new cycle starts, the fabric watches the first bit of each byte on its in ports, e.g., dIn0(0) in Figure 4.1. This is the active bit of the cell routing tag (Figure 4.2). As soon as one such bit on an in port becomes 1, that event marks the start of cells on all the in ports. On receiving a set of routing tags in a particular frame, the fabric processes them and arbitrates among the in ports for accesses to the requested out ports. The cells are then forwarded to the appropriate out ports as determined by the routing tags *h1*. When sending out the cells, the fabric removes *h1* and forwards *h2* with the cell to the out port. *h2* is used internally by the out port controllers to choose either out buffers or loop buffers to deposit the cell, and is subsequently stripped off. Out port controllers forward the ack signal from the selected out buffers to the fabric. The port controllers and the switch fabric all use the same clock. They use a frame start signal to ensure that the in port controllers inject cells synchronously, i.e., the routing tags arrive at the same time to the fabric in ports. If no in port raises the active bit throughout a frame then the frame is inactive - no cells are processed. Otherwise, the frame is active.

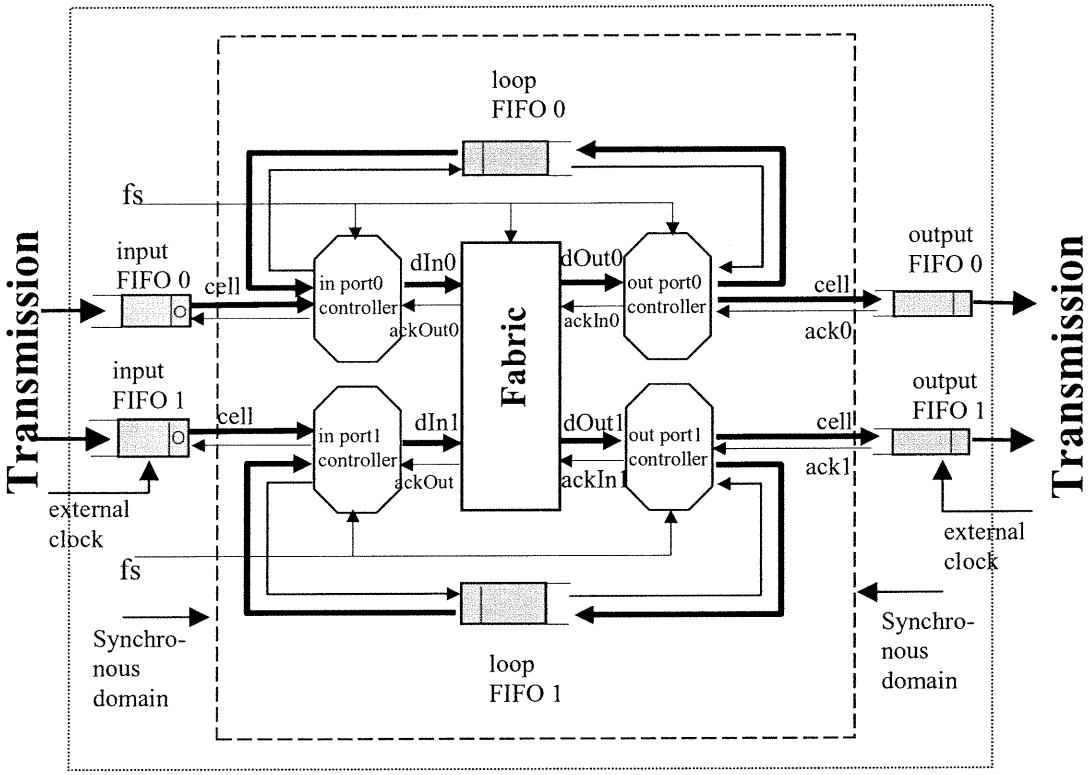


Figure 4.1: A 2x2 Fairisle switch (the original switch is 4x4 [32]).

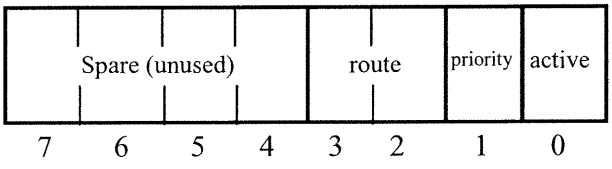


Figure 4.2: Routing *tag h1* for a 4x4 Fairisle switch [32]

4.2 Global specification

We specify global properties and assumptions that the switch makes about its (global) environment. Then, we specify the local components, e.g., the in port controllers and the fabric each one in its own specific environment. Finally, we use a compositional approach to show that the local properties imply the global specification. Depending on the complexity of a component, it may require further utilization of compositional reasoning. For instance, the local properties of the in port controllers will be in turn verified by a compositional approach.

We introduce the following global properties as the most basic operations of the switch module. (We will formally state and verify these properties using interface recognizers/suppliers in Chapter 5.)

G1 (*Extraction*) The switch should correctly dequeue cells from the input FIFO buffers. It should discard incomplete cells and deposit complete ones in cell memory.

G2 (*Header conversion*) The switch should convert cell headers according to the routing table. The headers contain the channel numbers needed to route cells to correct destinations.

G3 (*Data integrity, no duplication, no loss*) The switch should transfer cells to the destination output buffers with no duplication or loss or corruption. Under certain assumptions (e.g., acknowledgement reception and a limited number of high priority cells), it should eventually transfer cells to the output buffers. In the case of no-ack, it may discard them after several attempts.

G4 (*Prioritization*) The switch should deliver cells based on their priorities, e.g., it should deliver the high priority cells first.

G5 (*Order preservation*) The order of cells should be respected, i.e., there should be a first-in-first-out order between cells of the same priority (that are arriving from the

same in port and are destined to the same out port). The order between the bytes of a single cell has also to be preserved.

As an example, suppose we want to send two very short cells "hello" and "peter" to destination Peter. No duplication property (*G3*) asserts that, we are not delivering for instance "hello", "pepeter" to Peter. *G3* also guarantees that the cells are not cut to "hel" and "pet", for example. Order preservation (*G5*) requires that the message not be transferred as "peter", "hello" or "olleh", "retep" etc. The correct header transformation (*G2*) i.e., correct destination channel number ensures that the messages will be delivered to Peter and not to John.

Next, we define environment assumptions of the switch. The switch reads the cells from the input FIFO buffers and sends them to the appropriate output FIFO buffers. It uses a frame start signal *fs* to synchronize its components. For a correct operation of the components, the switch requires that the cell frame (which is delimited by the frame start pulses) be greater than the cell size:

E_g: The cell frame should be greater than the cell size[14].

4.3 Specification of the components

We (informally) defined the global properties and the environment assumptions of the switch in Section 4.2. In the same way, we specify the switch components, i.e., the in port controllers and the fabric.

4.3.1 In port controller specification

The in port controller extracts the cells from the input FIFOs and sends them to the fabric. The following is a list of local properties that characterize the desired behavior of the in port controllers: (Formal specification using IRS will be given in Chapter 5.)

[14] We will consider a (fairness) assumtion on acknowledgement when we verify liveness property (*G3*) of the fabric in Section 5.7 in Chapter 5.

C1 (*Extraction*) The controller should drop incomplete cells and deposit complete ones in a cell memory.

C2 (*Header conversion*) The controller should convert cell headers (in the memory) according to the routing table. It should add two routing tags *h1* and *h2* to each cell; *h1* should indicate the start of the cell and the cell priority to the fabric. *h2* should indicate the start of the cell to the out port controllers.

C3 (*Data integrity, no loss, no duplication*) Upon receiving positive acknowledgment during cell transmission, the in port controllers should forward the cells to the fabric without duplication or loss. (Low priority cells can be transferred if (1) ack is received and (2) the number of high priority cells is limited.)

C4 (*Prioritization*) Cells should be forwarded to the fabric based on their priorities, i.e., the high priority cells should be transmitted first.

C5 (*Order preservation*) The order of cells should be respected, i.e., there should be first-in-first-out order between cells of the same priority. Similarly, the order of the bytes in a single cell should be preserved during the transfer.

The in port controllers use the frame start signal *fs* to synchronously inject cells to the fabric. They assume that the frame size (supplied by the local environment) is bigger than the cell length.

E_c The frame length has to be greater than the cell size.

4.3.2 Fabric specification

The fabric receives cells on its in ports and forwards them to its out ports. It does not store the cells, and if there is contention among the incoming cells for an out port, the fabric arbitrates between them and sends negative acknowledgments to the unsuccessful in ports. The successful ports however are forwarded the (positive or negative) acknowledgment received from the destination out port.

F1 (*Data transfer*) After dropping routing *tag h1* from the cells, the fabric should forward the successful cells to the requested out ports.

F2 (*Ack transfer*) The fabric should forward the acknowledgement from the out ports to the successful in ports.

F3 (*Prioritization*) High priority cells should be given precedence over the low priority ones during arbitration in the fabric.

F4 (*Order preservation*) The fabric should preserve the order of cells. If a *cell1* in an in port *i* is to go to destination out port *j* and a *cell2* arrives to the same in port after *cell1* and is also to go to the destination port *j* and *cell1* has a high priority (while *cell2* can have high or low priority), then *cell1* may reach the out port only before *cell2* does. In other words, either *cell1* is discarded or it reaches the out port before *cell2* does.

Having specified the properties, we next define the assumptions that fabric (implementation) makes about its environment. These assumptions concerns the relative time distance between the frame start and cell reception from the in ports. A timing diagram (specification) for the in *port0/fabric* interface of the switch is shown in Figure 4.3. After sending *h1*, *h2* and the first byte of the cell on *dIn0*, the controller on in *port0* expects to receive an acknowledgement (*ackOut0*) from the

fabric for each transmitted byte of the cell. With a positive ack, a cell (of length *ATMlength*) is completely transmitted to the fabric.

For proper operation, the fabric makes the following assumptions about *h1* and *fs* that it receives from its environment. These assumptions are going to be discharged on the in port controllers and the frame pulse generator *fsGenerator*, as their properties.

E_{f1} Cells must not arrive sooner than three cycles after the frame start signal *fs*. The first byte of a cell is *h1*. If t_s and t_{h1} denote respectively the time that *fs* and *h1* arrive, we have

$$t_{h1} \geq t_s + 3 \tag{4.1}$$

E_{f2} Frame start cannot be asserted before three cycles, after t_{h1}. If t_e denotes the end of the current frame or the beginning of the next frame, then the following must hold.

$$t_e \geq t_{h1} + 3 \tag{4.2}$$

E_{f3} The frame length has to be greater than the cell size.

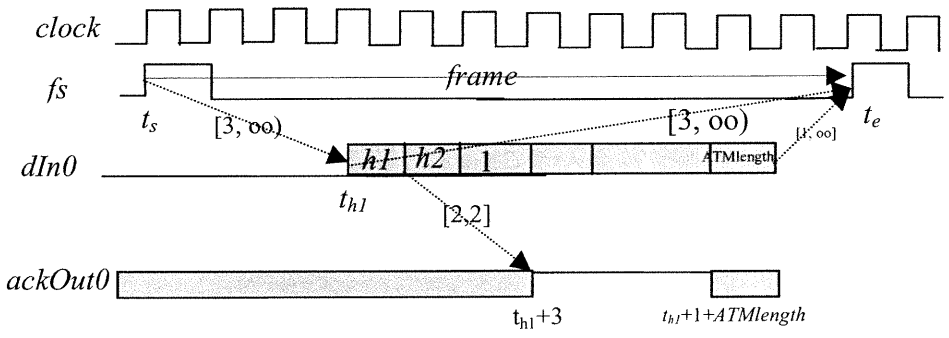


Figure 4.3: Timing diagram specification for the in *port0/fabric* interface in Figure 4.1

Next, we model fabric environment assumptions using interface recognizers/suppliers (IRS). Frame start pulses that are generated by the global environment can be supplied (as assumptions) by an IRS in Figure 4.4 (similar to the one presented in Figure 3.3 in Chapter 3). By setting its constraint C_{fs} to be always true, frame start pulses can be generated respecting the given cell size *ATMlength*. As mentioned in Chapter 3, we will use the expression "*activate a constraint*" to indicate that the constraint is set always true while model checking a component. By activating C_{fs}, *fs* pulses are always generated with a period bigger than *ATMlength*. Since *fs* is non-deterministically generated in *s1*, *fs* can take any period bigger than *ATMlength*. Although, the lower limit of frame sizes accepted by IRS I_{fs} is bounded, its upper limit is not restricted.

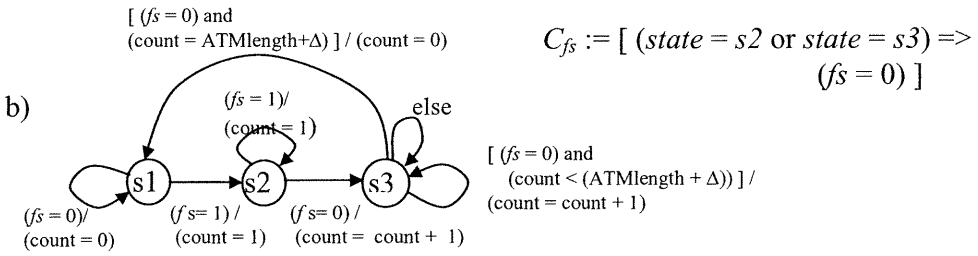


Figure 4.4: IRS and its constraint C_{fs} for frame start pulse *fs*. A parameter (Δ>2+t_{h1}-t_s) determines the frame size in Figure 4.3

We have an obligation to show that an actual component (called *fsGenerator*) that generates *fs* pulses with a fixed period satisfies C_{fs}.

$$fsGenerator \models I_{fs}(C_{fs}) \tag{4.3}$$

After representing assumption E_{f3} by I_{fs}, we next model assumptions E_{f1} (4.1) and E_{f2} (4.2) by an IRS I_{env}. Figure 4.5 shows I_{env}, which recognizes the sequence of events that is authorized by E_{f1} and E_{f2}. For instance, suppose that a frame start occurs at state *p1*. The routing *tag h1* can then arrive at state *p4* (up to *p11*), i.e., at least 3 clock

cycles after frame start. Similarly, after *h1* occurs (at one of the states *p4* to *p11*), the next frame start may occur at *p7*, i.e., 3 clock cycles later. The switch fabric begins the arbitration between ports as soon as it receives one *h1* on one of its in ports. If no *h1* is received up to state *p11*, the frame is considered late or inactive, since a complete cell could not be transferred within a given frame size. We added the transition *p11* to *p1* to distinguish inactive frames traced by *p1-> ... ->p11->p1* from active frames traced by *p1->...->p9->p5->...->p7*. This distinction helps us to express properties of active and inactive frames.

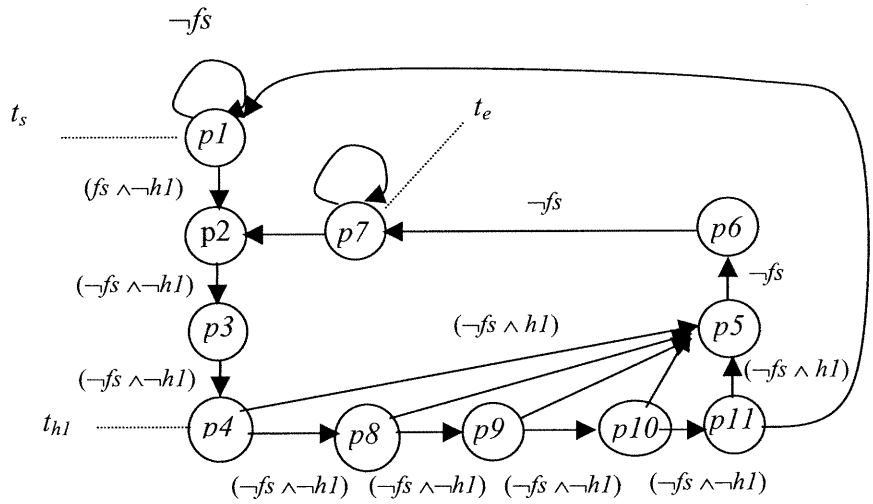


Figure 4.5: Interface machine I_{env} of the environment assumptions on the in ports of the fabric.

Next, we define a constraint C_{h1p0} for I_{env} to state that (the active bit of) *h1* on in *port0* should be zero at states *p2*, *p3*, and at *p1* and *p7* when *fs* is 1.

$$C_{h1p0} := [\ (\ (state = p2) \text{ or } (state = p3) \text{ or } (\ (state = p7 \text{ or } state = p1) \text{ and } fs = 1\ )\ ) \Rightarrow (dIn0(0) = 0)\ ] \quad (4.4)$$

where *dIn0* indicates the fabric input on in *port* 0. C_{h1p1} is similarly defined for in *port* 1.

$$C_{h1p1} := [\ (\ (state = p2)\ \text{or}\ (state = p3)\ \text{or}\ (\ (state = p7\ \text{or}\ state = p1)\ \text{and}\ fs = 1\ )\ )$$
$$\Rightarrow (dIN1(0) = 0)\ ] \tag{4.5}$$

After developing an IRS, the machine has to be validated to ensure that it correctly models the interface interactions. For instance, we activate C_{fs} and C_{h1p0} to verify that after a frame starts (i.e., state *s2* of I_{fs} is reached), *h1* on in *port0* remains zero (i.e., inactive) until state *p4* (i.e., t_s + 3) is reached. This verification is denoted by the following formula:

$$I_{env}(C_{h1p0}) \wedge I_{fs}(C_{fs}) \models \text{AFTER: } I_{fs}(state = s2) \tag{4.6}$$
$$\text{ALWAYS: } dIn0(0) = 0$$
$$\text{UNLESS: } I_{env}(state = p4)$$

where AFTER, ALWAYS, and UNLESS denote the temporal operators of the model checker Formal Check [3].

We have an obligation to prove that the in port controllers respect the environment assumptions (4.4) of the fabric. To verify this, we compose I_{env} with an in port controller to check that C_{h1p0} is always true on I_{env}. The verification can be denoted by the following formula:

$$I_{fs}(C_{fs}) \wedge inPortController \models I_{env}(C_{h1p0}) \tag{4.7}$$

Formula (4.7) indicates that the in port controller composed with the pulse generator interface I_{fs} with the constraint C_{fs} activated satisfies the property that C_{h1} is always true on I_{env} that observes the fabric interface of the controller.

It is possible to discharge the original subgoal that the transmitter respects the environment assumptions of the fabric directly using the proper language of the model checker, i.e., without any use of I_{env} or C_{hlpo}. However, *by introducing IRS I_{env} which is coded in VHDL or Verilog, we get a portable and uniform representation of these environment conditions*. This representation by IRS is portable, since it does not use the model checker (temporal) operators. It is uniform because IRS can uniformly verify a property or supply an assumption using the constraints. In any new application of the fabric, the components in the application should satisfy these conditions to use the fabric safely. IRS should thus accompany the specification of any reusable component to represent its environment assumptions.

4.4 Summary

In this chapter, we (informally) specified properties and environment assumptions of the in port controllers, the fabric, and their composed system - the switch module. We used IRS to model these environment assumptions. We described that the in port controllers and the frame pulse generator *fsGenerator* should meet environment assumptions of the fabric. We illustrated how we can discharge these obligations using constraints of IRS I_{env} and I_{fs}.

In the next chapter, we will use IRS to specify and verify properties of the switch components. We will then compose these local properties to prove the global ones.

Chapter 5

Formal Verification of an ATM Switch

In Chapter 4, we specified the in port controller and the switch fabric. In this chapter, we prove that the in port controllers and the fabric implementation satisfy their properties. Then, we compose them to show that the switch module (composed of the in port controllers and the fabric) satisfies the global properties. During these verifications, we culd observe that model checking of the in port controller for 53-byte ATM cells is very expensive in time and memory. Therefore, we first decompose the in port controller into its components and use a compositional approach to verify the global properties of the controller. Two abstraction techniques will also be used to reduce the verification problem. First, we assume a data independent model [34] and second, we scale down the cell size.

5.1 In Port controller implementation

The port controller of Fairisle ATM switch consists of a receiver, a dispatcher, a scheduler, a transmitter, an arbiter, and a cell memory (Figure 5.1). In addition, there are 5 queues (*R, P1, P2, T,* and *F*) that contain addresses of the memory locations where the cells are stored. These queues are implemented in the same memory, which stores the cells. When a cell is present in one of the input FIFO buffers (e.g., FIFO *L* or *I*), the receiver allocates an address from the free address queue F to the cell. The receiver then transfers the cell one word at a time from the FIFO to the memory. After transferring a complete cell, the address is inserted into the queue *R*. The dispatcher detects the presence of a cell address in queue *R* and proceeds to update

the cell header in the memory. It transforms the header according to a routing table and, moreover, it adds a fabric tag (*h1*) and an out port tag (*h2*) to the cell. The priority (high, low) of a cell is indicated by a bit in *h1* (Figure 4.2). The dispatcher checks that bit and depending on the priority, it inserts the cell address (dequeued from *R*) to either queue *P1* or queue *P2*. The scheduler transfers cell addresses from *P1* and *P2* to the transmission queue *T*, giving priority to the cell addresses in *P1*. The transmitter sends the ATM cells from the cell memory to the fabric one byte at a time. After a cell has been transmitted to the fabric, its address (dequeued from *T*) is returned to the free address queue *F*. The arbiter regulates the accesses to the memory shared between the receiver, the dispatcher, the scheduler and the transmitter, giving always priority to the transmitter. Although we use independent queue models, we assume that they are implemented in the shared memory, hence only one cell word or one address in a queue can be accessed at a time. This is enforced by the arbiter.

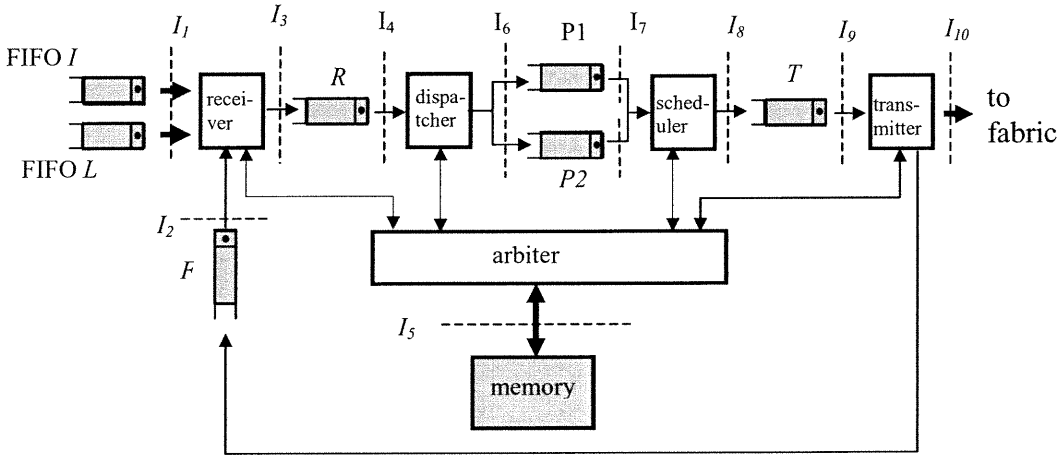


Figure 5.1: An ATM switch in port controller

5.2 Arbiter abstraction

The receiver, the dispatcher, and the scheduler require a permission from the arbiter to access the memory. The transmitter has the highest priority, however, and can access the memory as soon as it requests. We first verify that if the transmitter releases the memory infinitely often, the arbiter can grant memory accesses to the

receiver, the dispatcher, and the scheduler infinitely often. We shall show in Section 5.8 that the transmitter does release the memory infinitely often, thus we abstract the arbiter and decompose the port controller into receiver, dispatcher, scheduler, and transmitter subsystems where each subsystem may assume to receive memory accesses infinitely often.

5.3 Receiver subsystem

The receiver is the most complex module among the components of the port controller. We verify the following properties of the receiver.

R1 (*Extraction*) Incomplete cells are dropped and the complete cells are transferred from the input FIFOs to the cell memory.

R2 (*Order preservation*) When depositing the cells in the memory, the order of the bytes inside a cell is preserved. Similarly, the ordering between cells is preserved. For instance, if a complete *cell1* arrives to the receiver before a complete *cell2*, *cell1* is deposited to the memory before *cell2* is deposited.

R3 (*Address path*) After depositing a cell in the memory, its address that is taken from queue F is enqueued in queue R.

R4 (*Address order preservation*) While transferring the addresses from F to R, the order among addresses is preserved.

In the input FIFOs, the bytes are made of 9 bits where the 9^{th} bit indicates the start of a cell (when equal to 1). When reading the cell bytes, if a start-of-cell bit is set or if the buffer becomes empty, the cell is discarded.

Model checking of the receiver subsystem containing the receiver, queues F and R, and the cell memory is practically not feasible because of the state explosion problem. To reduce the problem, we use the data independence assumption: the

queue, the memory, and the receiver are data-independent. The data-independence assumption infers, for instance, that, a queue preserves the order among its elements, if it does so for a reduced data set {0, 1, 2} that contains three elements (Appendix 2).

The receiver has a scalable architecture, parameterized on the cell length. This allows us to reduce the cell to the minimum of 3 elements <*start-of-cell*, *data0*, *data1*>. With the data-independence assumption and the cell-size reduction, it is possible to verify the receiver subsystem.

5.3.1 FIFO/receiver interface machine I_1

We use IRS machines to verify the properties *R1-R4* of the receiver described in Section 5.3. First, we develop an interface machine I_1 to recognize the interactions between the FIFO *I* and the receiver (Figure 5.1). We will ultimately remove FIFO *I* and use I_1 to supply an abstract model of FIFO to the receiver as receiver environment. I_1 as shown in Figure 5.2 depends on the following Boolean variables:

- *socI*: the 9th bit of the FIFO *I* top element, indicating start of a cell.
- *deqNeeded*: when true, it indicates that an address will be dequeued from *F* for a new cell.
- *fifoIempty*, *qFempty*: indicate whether FIFO *I* respectively queue *F* is empty.
- *deqI*, *deqF*: dequeue commands on the corresponding queue interfaces.
- *oneCellFromI*: indicates whether FIFO *I* contains a cell, i.e., 1 + 53 bytes of data where the first byte represents the start-of-cell byte.
- *rok*: true, indicates that the receiver has the right (from the arbiter) to access the shared memory.

Following the data independence principle, IRS I_1 should recognize the language (*cell0*)*(*cell1*)(*cell0*)*(*cell2*)(*cell0*)* and will subsequently be used to constrain the inputs of the receiver to this language. *cell0*, *cell1*, and *cell2* represent three arbitrary (but different) cells. Each cell consists of three data octet such that the *soc* bit of the first octet is set. Suppose *xsoc* represents an arbitrary octet with *soc* = 1. Let the

sequences <*xsoc*, 0, 0>, <*xsoc*, 1, 1> and <*xsoc*, 1, 0> represent respectively *cell0*, *cell1* and *cell2*. In the initial state, I_1 awaits *xsoc* from the FIFO *I*. The transition *s0 -> s1* of I_1 recognizes this *xsoc* under the following condition.

$$(socI \wedge oneCellfromI \wedge \sim fifoIempty \wedge deqI \wedge \sim qFempty \wedge deqF \wedge rok \wedge deqNeeded) \quad (5.1)$$

This means that it observes a start of cell, FIFO *I* contains 53 bytes, and the receiver is dequeuing (*xsoc*) from FIFO *I* and (an address) from queue *F*. Also the receiver received permission (*rok*) from the arbiter to read queue *F* from memory. The transitions *s0->s1->s2->s3* recognize *cell0*. After that, in state *s3*, I_1 awaits *rok* for *cell0* to be written in the memory. In state *s4*, the *address* of *cell0* is written in queue *R*, if queue *R* is not full and *rok* is received.

During the cell extraction from *I* in states *s1* and *s2*, if the *soc* bit is asserted then the start of another cell is detected. This means that the current cell is incomplete and the cell must be discarded. This is recognized by transition *t3*, i.e., before writing the cell in the memory. In that case, the address dequeued from *F* is retained for the next cell. The following condition will then hold at the beginning of a new cell.

$$(socI \wedge oneCellfromI \wedge \sim fifoIempty \wedge deqI \wedge \sim deqNeeded) \quad (5.2)$$

Similarly to *cell0*, transitions *s0->s1->s5->s6->s7* and *s8->s9->s13->s14->s15* recognize *cell1* and *cell2*, respectively. I_1 can thus recognize the language (*cell0*)*(*cell1*)(*cell0*)*(*cell2*)(*cell0*)* at the interface.

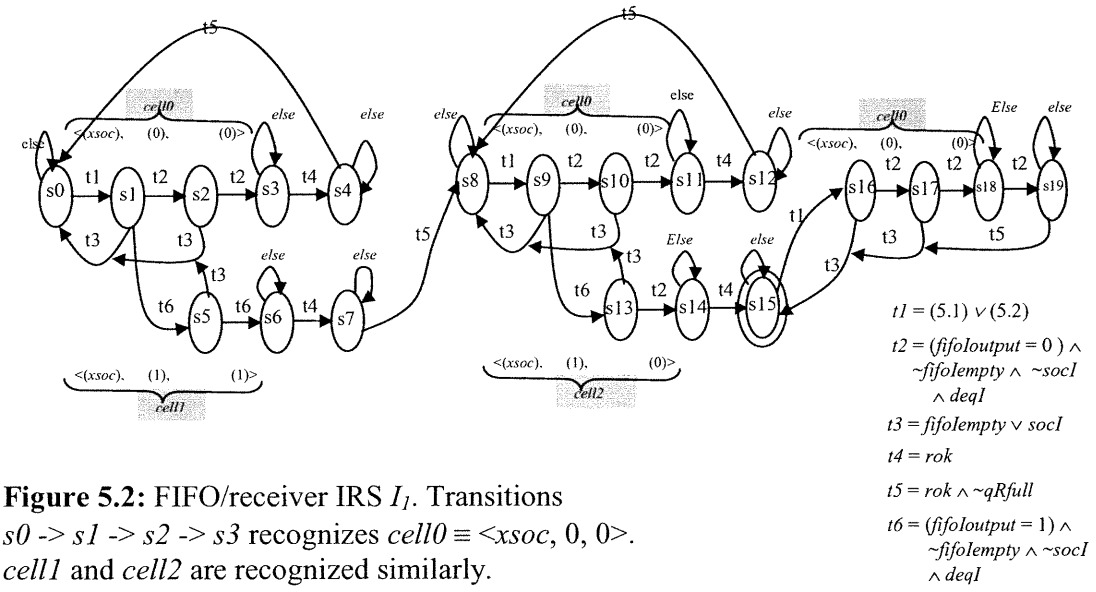


Figure 5.2: FIFO/receiver IRS I_1. Transitions *s0 -> s1 -> s2 -> s3* recognizes *cell0* ≡ <*xsoc*, 0, 0>. *cell1* and *cell2* are recognized similarly.

I_1 uses the following constraints C_{cell0} and $C_{uniCell1\_2}$ and C_{cell1} and C_{cell2} to impose that language on its inputs. When the constraints are activated on I_1, the inputs of the receiver will be restricted to follow that language.

$$C_{cell0} := [\ (\ (state = s2 \text{ or } state = s10) \text{ and } fifoIempty = 0 \text{ and } socI = 0\ ) \Rightarrow (fifoIoutput = 0)] \quad (5.3)$$

C_{cell0} corresponds to transition *t2*. For instance, it asserts that the output of FIFO *I* (*fifoIoutput*) should be 0 at states *s2*, and *s10* to supply the data value 0 of *cell0*. This restriction on the FIFO output should be supplied whenever the FIFO is not empty and the soc bit is not asserted. Remark that *IRS* I_1 allows the FIFO *I* to non-deterministically be empty or even soc to be set at those states. These cases represent incomplete cell reception (e.g., transition *t3* in *s1* and *s2*). If we removed the condition *socI* = 0 from C_{cell0}, the incomplete cell reception could be ignored in the verification. When C_{cell0} remains true and for instance, state *s3* is reached, *cell0* is

successfully extracted from the FIFO. Constraints C_{cell1}, C_{cell2} and $C_{uniCell1\_2}$ are defined similarly.

$$C_{cell1} := [\ (state = s5 \text{ and } fifoIempty = 0 \text{ and } socI = 0\ ) \Rightarrow (fifoIoutput = 1)] \quad (5.4)$$

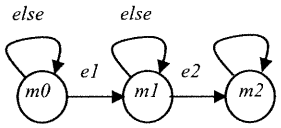
$$C_{cell2} := [\ (\ state = s13 \text{ and } fifoIempty = 0 \text{ and } socI = 0\ ) \Rightarrow (fifoIoutput = 0)] \quad (5.5)$$

$$C_{uniCell1\_2} := [\ (\ (state = s16 \text{ or } state = s17) \text{ and } fifoIempty = 0 \text{ and } socI = 0\ ) \Rightarrow (fifoIoutput = 0)] \quad (5.6)$$

By forcing (5.3), (5.4), (5.5), and (5.6) to be always true, I_1 supplies the appropriate input sequences to the receiver and allows us to verify cell extraction (*R1*) and cell-order preservation (*R2*) properties of the receiver.

5.3.2 Receiver/memory interface

An interface machine I_5 recognizes the writing of *cell1* and *cell2* in the memory at addresses *a1* and *a2*, respectively. These addresses were previously dequeued from *F* and a copy is kept in I_1. The receiver extracts cells from FIFO *I*, octet by octet to form words before writing them in the memory. *Cell1* and *cell2* are thus respectively written as 4 and 5. By the transition *m0->m1* in Figure 5.3, I_5 detects that *cell1* was written in memory. This happens when I_1 is in state *s6* (waiting for the receiver to obtain *rok* to write *cell1* in the memory) and the receiver asserted the memory write signal. The transition *m1 -> m2* similarly detects that *cell2* was written in the memory.



$e1 = I_1(state = s6) \wedge memInput = 4$
$\wedge (memAdrs = a1) \wedge write \wedge rok$

$e2 = I_1(state = s14) \wedge memInput = 5$
$\wedge (memAdrs = a2) \wedge write \wedge rok$

Figure 5.3: IRS I_5 for the receiver/memory interface

Consider the subsystem $I_1 \wedge$ receiver $\wedge I_5$ in Figure 5.4. Using the interface machines I_1 and I_5, we can verify that *cell1* and *cell2* are eventually written in the memory. We set the constraints C_{cell0}, C_{cell1}, C_{cell2}, and $C_{uniCell1\_2}$ of I_1 to be always true and verify that after *cell1* is extracted (i.e., state *s6* of I_1 is reached) and *rok* is asserted, then *cell1* is written in the memory (i.e., state *m1* of I_5 is eventually reached).

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models \text{AFTER: } rok \wedge I_1(state = s6)$$
$$\text{EVENTUALLY: } I_5(state = m1) \tag{5.7}$$

Similarly, we verify that after *s14* is reached, *cell2* is eventually written in the memory.

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models \text{AFTER: } rok \wedge I_1(state = s14)$$
$$\text{EVENTUALLY: } I_5(state = m2) \tag{5.8}$$

Following the data-independence assumption, we must show that *cell1* and *cell2* are uniquely recognized at I_5. We add a constraint $C_{mUnique}$ that asserts that after writing *cell1* and *cell2*, no *cell1* (i.e., *11*) or *cell2* (i.e., *10*) is written in the memory anymore (only 00 can be written then).

$$C_{mUnique} := [(state = m2 \text{ and } write = 1) \Rightarrow (memInput = 0)])$$

We then verify that $C_{mUnique}$ always holds *true*.

$$I_1(C_{cell0}, , C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models I_5(C_{mUnique}) \tag{5.9}$$

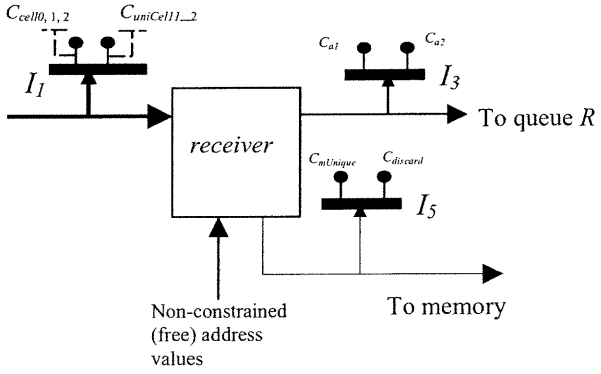


Figure 5.4: Receiver subsystem. Constraints C_{cell0}, C_{cell1}, C_{cell2} and $C_{uniCell1\_2}$ are activated and (property) constraints $C_{mUnique}$, $C_{discard}$, C_{a1}, and C_{a2}, are checked.

To verify that no incomplete cell is written in the memory, we consider the write signal and define a constraint $C_{discard}$ to assert that in states other than *s3*, *s6*, *s11*, *s14*, and *s18* the write signal is always deasserted. Therefore, only in states *s3*, *s6*, *s11*, *s14*, *s18* where a cell is completely received, the cell can be written in the memory.

$$C_{discard} := [not\ ((state = s3)\ \text{or}\ (state = s6)\ \text{or}\ (state = s11)\ \text{or}\ (state = s14)\ \text{or}\ (state = s18)) \Rightarrow (write = 0)]$$

We checked that $C_{discard}$ always holds *true*.

$$I_1(C_{cell0}, , C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models I_5(C_{discard}) \tag{5.10}$$

To verify the address path, we use an IRS I_3 to recognize that addresses *a1* and *a2* of *cell1* and *cell2* are successfully enqueued in queue *R*. We first verify that that the address *a1* is enqueued to *R*. This happens when FIFO *I/Receiver* IRS I_1 is in state *s7* and *rok* is asserted and queue *R* is not full.

Figure 5.5: IRS I_3 for the receiver/queue R interface

Transition $b1$ which is described by the following constraint detects that the receiver asserted *enqR* and inserted $a1$ to queue R.

$$C_{a1} := [(\ I_3(state = q0) \text{ and } I_1(state = s7) \text{ and } rok \text{ and } \sim qRfull)$$
$$\Rightarrow (enqR = 1 \text{ and } qRinput = a1)]$$

Using the following property, we verified that C_{a1} is always true:

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models I_3(C_{a1}) \tag{5.11}$$

Similarly, using a constraint C_{a2} on transition $b2$, we verified that $a2$ is enqueued in R. In summary, addresses $a1$ and $a2$ (of *cell1* and *cell2*) are correctly inserted in queue R.

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models I_3(C_{a1}, C_{a2}) \tag{5.12}$$

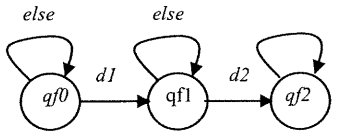
Remark that we have not yet proven that these addresses are uniquely inserted. For instance, $a1$ could have been written twice. That verification was not possible since we did not constrain the address value supplied by F. We employ the data-independence assumption and we verify that if a stream (0)*(1)(0)*(2)(0)* is supplied as address values at queue *F/receiver* interface, the same stream is inserted in queue R, thus, no duplication happens. This proves that, when the values 1 and 2 are

supplied once (and in order), they are recognized as such. Figure 5.6 shows the IRS I_2 used to impose the stream (0)*(1)(0)*(2)(0)* as the address values to the receiver. The constraints C_{uni1} and C_{uni2} restrict the inputs so that the values 1 and 2 are supplied once. For instance, C_{uni1} asserts that the data value 1 could only be supplied (non-deterministically) in *qf0*.

C_{uni1} := [(*state* = *qf1* or *state* = *qf2*) ⇒ (*qFoutput* /= 1)]

Similarly, C_{uni2} asserts that value 2 could only be supplied (non-deterministically) in *qf1*:

C_{uni2} := [(*state* = *qf0* or *state* = *qf2*) => (*qFoutput* /= 2)]

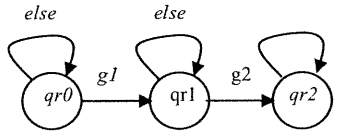


d1 = (*qFoutput* = 1) ∧ *deqF* ∧ *rok* ∧ (~*qFempty*)

d2 = (*qFoutput* = 2) ∧ *deqF* ∧ *rok* ∧ (~*qFempty*)

Figure 5.6: *IRS* I_2 for *queueF/receiver* interface

An IRS I'_3 with constraints C_{uni1o} and C_{uni2o} similar to C_{uni1} and C_{uni2} recognizes the language (0)*(1)(0)*(2)(0)* at the receiver output.



g1 = (*qRinput* = 1) ∧ *enqR* ∧ *rok* ∧ (~*qRfull*)

g2 = (*qRinput* = 2) ∧ *enqR* ∧ *rok* ∧ (~*qRfull*)

Figure 5.7: IRS I'_3 to verify the address path

C_{uni1o} := [(*state* = *qr1* or *state* = *qr2*) ⇒ (*qRinput* /= 1)

C_{uni2o} := [(state = *qr0* or *state* = *qr2*) $\Rightarrow$ (*qRinput* /= 2)]

We use an IRS I'_1 to supply an arbitrary stream of cells, e.g., (*cell0*)* at *FIFOI*/receiver interface. It allows us to carry on the address path verification. (I_1, for instance contains states *s0* to *s4* of I_1). In the subsystem $I'_1 \wedge$ receiver $\wedge I_2 \wedge I'_3$ (Figure 5.8), we verify that the address values 1 and 2 are uniquely enqueued in *R* so that no address is duplicated.

$$I_2(C_{uni1}, C_{uni2}) \wedge receiver \models I'_3(C_{uni1o}, C_{uni2o}) \tag{5.13}$$

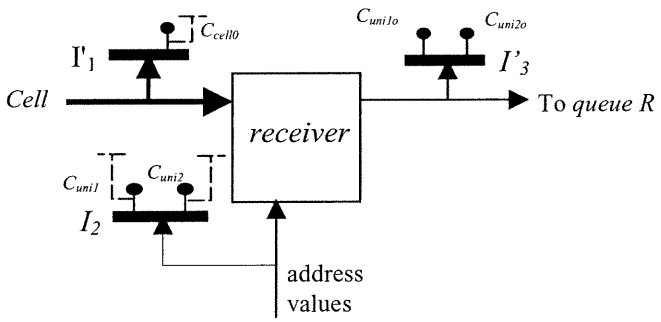


Figure 5.8: C_{cell0}, C_{uni1}, and C_{uni2}, are activated and C_{uni1o} and C_{uni2o} are checked to verify address path of the receiver.

Constraints C_{uni1o} and C_{uni2o} assure that address values 1 and 2 are recognized whenever states *qr1* and *qr2* are reached. However, the verification of (5.13) does ascertain that those states are ever visited. We still have to ensure that *qr1* and *qr2* are eventually reached. For instance, in the subsystem $I'_1 \wedge$ *receiver* $\wedge$ I$_2 \wedge I'_3$ (Figure 5.8), we verify that after the state *qf1* of I_2 is reached (i.e., 1 is dequeued as an address value by the receiver), address value 1 is eventually enqueued in *R* (i.e., the state *qr1* of I'_3 is reached).

$$I_2(C_{uni1}, C_{uni2}) \wedge receiver \models \text{AFTER: } I_2(state = qf1) \wedge I'_1(state = s4) \wedge rok \wedge \sim qRfull$$
$$\text{EVENTUALLY: } I'_3(state = qr1) \tag{5.14}$$

Next, we verify that the address value of 2 is eventually enqueued in R.

$$I_2(C_{uni1}, C_{uni2}) \wedge receiver \models \text{AFTER}:I_2(state = qf2) \wedge I'_1(state = s4) \wedge rok \wedge \sim qRfull$$
$$\text{EVENTUALLY}: I_3(state = qr2) \qquad (5.15)$$

By considering all the properties, we can conclude that the receiver discards incomplete cells, extracts complete cells from FIFO I, writes the cells in order and without any duplication in the cell memory, and finally it correctly writes addresses of cells in the queue R. In the next section, we verify that the queue R correctly forwards these addresses to the dispatcher.

5.4 Queue R interface

The queue size is a generic value. We used CBL SMV [24]to show that the generic queue (implementation) delivers data from its input to its output. We first proved that when the condition ($enqR \wedge \sim qRfull$) holds, an arbitrary input enters the queue. Then, using the SMV induction rule, we verified that the data in any position in the queue eventually reaches the top position, assuming that the dequeue is asserted infinitely often. This proves that the generic queue correctly delivers the data it receives. Similarly, we proved that this queue preserves data order. (We included the queue model and the verification results using SMV in Appendix 3).

Q1 (*Order preservation*) If a stream (0)*(1)(0)*(2)(0)* is supplied (by I_3) to the queue input, the same sequence is recognized (by I_4) at the queue output.

$$I_3(C_{uni1}, C_{uni2}) \wedge queue \models I_4(C_{uni1o}, C_{uni2o}) \qquad (5.16)$$

Q2 (*Data delivery*) After a value 1 is enqueued in R, it eventually reaches the queue output, if $deqR$ is repeatedly asserted.

$I_3(C_{uni1}, C_{uni2}) \wedge queue \wedge$ A2 |= AFTER: $I_3(state = qr1)$

EVENTUALLY: $I_4(state = s1)$ (5.17)

$A2$:= AFTER: $\sim qRempty$ EVENTUALLY: $deqR = 1$ (5.18)

Similarly assuming $A2$, after a value 2 enters the queue, it eventually reaches the queue output.

$I_3(C_{uni1}, C_{uni2}) \wedge queue \wedge A2$ |= AFTER: $I_3(state = qr2)$

EVENTUALLY: $I_4(state = s2)$. (5.19)

5.5 Combining local properties of the receiver and the queue R

We have separately verified the receiver and the queue. Before describing other components of the in port controller, we show how to prove the composed receiver $\wedge$ queue subsystem properties from the queue and the receiver properties. We will apply this methodology to deduce end-to-end properties of the controller from properties of its components. We prove the following properties for receiver $\wedge$ queue.

RQ1 (*Extraction*) receiver $\wedge$ queue discards incomplete cells and deposits complete ones in a cell memory.

Proof: By (5.10) and (5.9), the receiver discards incomplete cells, extracts the correct ones and writes them once and in order in the memory.

$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver$ |= $I_5(C_{discard}, C_{mUnique})$

The queue does not change header or body of cells in the memory, and consequently, it does not affect extraction property *R1* of the receiver.

$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \wedge queueR \models I_5(C_{discard}, C_{mUnique})$

Nevertheless, we have to show that the queue, the receiver, and $I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2})$ meet well foundedness/compatibility conditions (*W1-W5*) of the composition.

W1 (*No conflicting outputs in the implementation*) We verified that outputs of the receiver and *queueR* are disjoint. This way, only one module can constrain a signal of the interface.

W2 (*No cycle-of-gates in the implementation*) We should check that no cycle is formed in *receiver* $\wedge$ *queueR*. The receiver and the queue do not contain any cycle-of-gates, consequently, we only check that no cycle is formed in their composition. Queue *R* inputs {*dataIn*, *enqR*, *deqR*} and outputs {*dataOut*, *qRempty*, *qRfull*} are at least one cycle apart. Therefore, no cycle is formed when it is composed with the receiver.

W3 (*No conditional cycle-of-gates in IRS*) We verify that I_1 does not contain any conditional cycle-of-gates. Consider the constraint C_{cell1} := [(*state* = *s5* and *fifoIempty* = 0 and *socI* = 0) $\Rightarrow$ (*fifoIoutput* = 1)] of I_1. It has a predicate over *fifoIempty*, *socI*, and *fifoIoutput*, and thus introduces conditional zero-delay paths $socI \xrightarrow{0}_{I1} fifoIoutput$ and $fifoIempty \xrightarrow{0}_{I1} fifoIoutput$. There is no constraint in I_1 that introduces a zero-delay path in the inverse direction from *fifoIoutput* to *socI* or *fifoIempty*, so I_1 contains no conditional cycle-of-gates.

W4 (*No output-constraining*) $I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2})$ should not constrain any output of the receiver or the queue. I_1 constrains fifoIoutput which is the receiver input. Therefore, it does not constrain the receiver or the queue output.

W5 (*No cycle-of-gates in the subsystems*) The composition $I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \wedge queueR$ should contain no cycle-of-gates. Given the zero-delay paths $socI \xrightarrow{0}_{I1} fifoIoutput$ and $fifoIempty \xrightarrow{0}_{I1} fifoIoutput$ by I_1, no

zero-delay path should exist in *receiver* $\wedge$ *queue* from *fifoIoutput* to either *socI* or *fifoIempty*. There are no such paths in the receiver (and in fact *fifoIoutput*, *socI*, and *fifoIempty* are all primary inputs of the receiver).

By *W1-W5*, I_1 $\wedge$ *receiver* $\wedge$ *queue* satisfies the well-foundedness/compatibility conditions of the composition theorem. Similarly, other subsystems involved in the compositional reasoning have been checked for well-foundedness/compatibility.

After *RQ1*, we continue proving the other properties of the combined subsystem *receiver* $\wedge$ *queue*.

RQ2 (*Address order preservation*) Address order is preserved in *receiver* $\wedge$ *queue* subsystem. I.e., if a *cell1* is deposited in the memory before a *cell2* is deposited, then its address is dequeued from the queue (output) before *cell2* address is dequeued.

Proof: In Section 5.3.2, we showed that the receiver correctly (i.e., once and in order) writes the cells into the memory. It then correctly inserts their addresses to queue *R*. Suppose that *cell0*, *cell1* and *cell2* are written at addresses 0, 1 and 2 respectively. By (5.12), we have then

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \models I_3(C_{uni1}, C_{uni2}) \tag{5.20}$$

By (5.16), queue *R* preserves the order of data.

$$I_3(C_{uni1}, C_{uni2}) \wedge queueR \models I_4(C_{uni1}, C_{uni2}) \tag{5.21}$$

Remark that IRS I_3 acts as a recognizer in (5.20) and as a (constraint) supplier in (5.21). As mentioned in Chapter 2, this property of IRS allows us to implement a *transitivity rule* in our reasoning. For instance, from (5.20) and (5.21), it follows that:

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge receiver \wedge queueR \models I_4(C_{uni1}, C_{uni2})$$

This proves the order preservation from the receiver input to the queue R output.

Until now, we have proven extraction and order preservation properties of *receiver* $\wedge$ *queue* from the receiver and the queue local properties. These properties which are proven using IRS constraints are safety properties asserting that if the system states are visited, the corresponding constraints hold true. However, if the states are not visited, the constraints also remain true, by definition. We need then to add liveness properties to assure that the states are eventually reached.

RQ3 (*Liveness*) After extracting a cell, its address is eventually enqueud in queue R. Then, this address eventually arrives to the top of the queue, if dequeue is asserted infinitely often.

Proof: By (5.14), we verified that after address value 1 is dequeued for depositing *cell1*, that address is eventually enqueued in R.

$$I_2(C_{uni1}, C_{uni2}) \wedge receiver \models \text{AFTER: } I_2(state = qf1) \wedge I_1(state = s4) \wedge rok \wedge \sim qRfull$$
$$\text{EVENTUALLY: } I_3(\text{state} = qr1) \qquad (5.22)$$

Verifications of queue in Section 5.4 concluded that the addresses that were enqueued in R are eventually transferred to top of R when dequeue is asserted infinitely often.

$$I_3(C_{uni1}, C_{uni2}) \wedge queue \wedge A2 \models \text{AFTER: } I_3(state = qr1)$$
$$\text{EVENTUALLY: } I_4(state = s1), \qquad (5.23)$$

$A2$:= AFTER: $\sim qRempty$ EVENTUALLY: $deqR = 1$

The EVENTUAL event in (5.22) matches AFTER event in (5.23). By a simple transitivity rule of temporal logic, from (5.22) and (5.23), it follows that after *cell1* is extracted by the receiver (i.e., the state *s4* of I_1 is reached) and an address is obtained

for it (i.e., state *qf1* of I_2 is reached), the address is eventually forwarded to the queue output.

$I_2(C_{uni1}, C_{uni2}) \wedge receiver \wedge queue \wedge A2 \models$
AFTER: $I_2(state = qf1) \wedge I_1(state = s4) \wedge rok \wedge \sim qRfull)$
EVENTUALLY: $I_4(state = qr1)$ (5.24)

Similarly, it is verified that *cell2* address eventually arrives to the top of *R*.

Assuming that dequeue happens repeatedly, queue *R* will supply the language (0)*(1)(0)*(2)(0)* to the dispatcher. Before considering the dispatcher verification, we summarize the compositional reasoning we have followed.

1 We used IRS to specify and verify local (safety) properties of the components.

2 Using local properties, we proved the global ones. IRS transitivity rule allowed us to chain the local safety properties in order to prove end-to-end ones. A theorem prover could also be used during these verifications.

3 We checked conditions *W1-W5* to conclude that the components are actually compatible for the composition and the set of constraints are well-foundeded.

4 Liveness properties were added to ensure that the composed system is live. They proved that the states that were used for constraint definitions are reachable.

5.6 Dispatcher and scheduler subsystems

We should verify that

D1: The dispatcher updates cell headers according to a routing table, and

D2: Addresses of the cells are correctly inserted into queue *P1* or *P2*, depending on cell priority.

For these verifications, we reduce the cell size to 10 bytes. Each cell comprises 2 bytes of routing tags *h1* and *h2*, 4 bytes of cell header and 4 bytes of cell body. Figure 5.9 illustrates a cell $<h1, h2, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8>$ deposited in the memory. Each cell occupies three locations. The first location contains the routing tags *h1* and *h2*. The second location contains the header bytes, and the third one contains (the reduced) cell body.

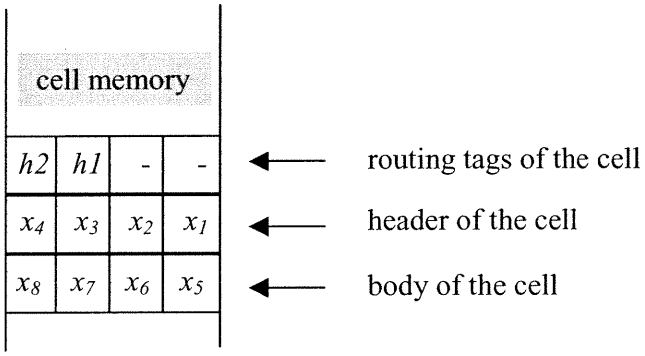


Figure 5.9:Placement of a cell $<h1, h2, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8>$ in cell memory

An IRS I_4 supplies the stream (0)*(1)(0)*(2)(0)* to the dispatcher, as address values at the queue *R/dispatcher* interface (Figure 5.10). These addresses are respectively pointers to *cell0*, *cell1*, and *cell2* in the memory. We initialize these cells in the memory and develop an IRS I'_5 (similar to memory/receiver IRS I_5) to monitor that:

1: Dispatcher reads headers of *cell1* and *cell2* from the cell memory (i.e., the second word of each cell in memory, figure 5.9) and based on them, adds two routing tags *h1* and *h2* to (the first word of) each cell in the memory.

2: According to the given routing table, the dispatcher updates (correctly and in order) the headers of *cell1* and *cell2*. These headers are in the second word of each cell in the memory.

Next, we used an IRS I_6 to recognize sequences of addresses (e.g., (0)*(1)(0)*(2)(0)*) written to queues *P1* and *P2*. Using the subsystem $I_4 \wedge dispatcher \wedge I'_5 \wedge I_6$, (Figure 5.10), we thus verified that the dispatcher satisfies properties *D1* and *D2*.

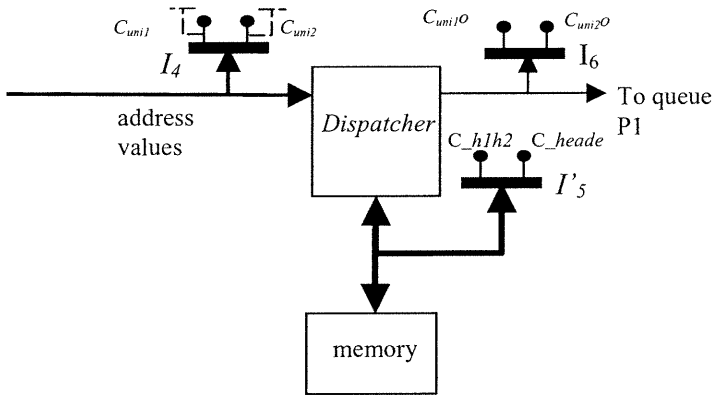


Figure 5.10: Dispatcher verification. We verify that the headers of cells are correctly updated in the memory.

Similarly, using interface machines, we verified that:

S1. The scheduler orderly and uniquely transfers addresses from queue *P1* to queue *T*, and

S2. Assuming that there are no high priority cells in queue *P1*, the dispatcher orderly and uniquely transfers addresses from queue *P2* to queue *T*.

5.7 Transmitter subsystem

We have shown that the receiver correctly extracts cells from input FIFOs and deposits them in the cell memory. The dispatcher converts cell headers and then appends routing tags to them. The scheduler transfers the addresses of cells to the transmit queue *T* based on the cell priority. Now, we show that the transmitter correctly extracts cells from the memory and forwards them to the fabric.

Consider the transmitter subsystem, composed of queue *T*, the transmitter, and the cell memory (Figure 5.1). Using our IRS approach, we show that the following properties holds true in the transmitter subsystem.

T1 (*Cell delivery*) When queue *T* supplies an address *a* to the transmitter, the transmitter sends out the corresponding cell, if it receives positive acknowledgement from the fabric during this transmission. Otherwise (no ack from the fabric), the transmitter drops them after several attempts.

T2 (*Serialization*) Given that the transmitter reads the cells word by word from the memory, and sends them byte by byte to the fabric, the transmitter should correctly convert memory words to cell bytes prior to the transmission.

T3 (*Order preservation*) The transmitter should preserve the order of cells. That is, if an address *a1* is supplied by queue *T* before an address *a2*, then the cell at *a1* should be sent out before the cell at address *a2* be sent out.

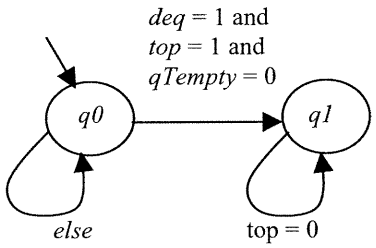
The transmitter subsystem is different than the receiver subsystem (or other subsystems of the in port controller) in that the later communicates through a queue, while the transmitter communicates with the fabric without using any queue. There is therefore a tighter (synchronization) relation between the transmitter and the fabric. We will show how IRS can model such (possibly reciprocal) communications.

Similar to other subsystems, we reduce transmitter verification by an appropriate data abstraction. Following the data independence assumption, we reduce the queue data set to two values {0, 1}. These values are respectively pointers to cell number 0 and 1 in the memory. The memory can be reduced to contain only *cell0* and *cell1*. Moreover, we downsize the cells to 10 bytes. This is possible since the transmitter has a generic architecture, parameterized on the cell length. (However, the

architecture is such that the cell size cannot be reduced to less than 10 bytes[15].) With these reductions, we initialize *cell0* in the memory to <1, 1, 0, 0, 0, 0, 0, 0, 0, 0> where the first two 1's represent routing tags of the cell (which were added by the dispatcher). Similarly, *Cell1*, the cell that we want to track is initialized to <1, 1, 1, 2, 3, 4, 5, 6, 7, 8>. This sequence of *cell1* helps us to verify the correct serialization property (*T2*) as well as the cell delivery property (*T1*) of the transmitter.

5.7.1 Interface recognizers in the transmitter subsystem

We first develop a recognizer I_9 at the queue output. I_9 reads three signals of the queue interface; *top* which shows the head of queue, *qTempty* which indicates whether the queue is empty, and *deq* that dequeues an element from the queue. When the condition ((*top* = 1) and (*deq* = 1) and (*qTempty* = 0)) becomes true, I_9 makes a transition (to state *q1*) to detect that a 1 was dequeued (Figure 5.11). Assuming that there is only one data value 1 in the data set, the head element should never get another 1. We thus obtain the constraint C_{uni1} shown in Figure 5.11:



C_{uni1} := [(*state* = *q1*) => (*top* = 0)]

Figure 5.11: An IRS for a data set reduced queue.

[15] Transmitter reads $4k + 2$ bytes from the memory, where k is the number of words in an ATM cell and the additional 2 bytes are routing tags. Each cell thus requires $(1 + k)$ memory accesses. A loop in the transmitter is repeated k times to transmit 4k bytes, but the first iteration is special since the acknowledgement from the fabric cannot be checked before transmitting the first byte of the first iteration (Figure 4.3). After this point, the acknowledgement (for the transmitted bytes) is checked before transmitting any new byte. The first iteration of the loop is therefore unfolded, while the normal loop is executed $(k - 1)$ times. The transmitter can send cells of $(2 + 4 + 4\ (k - 1))$ bytes, where $k \geq 2$. For $(k = 2)$, the cell size is reduced to a minimum of 10 bytes. More formally, one should use a theorem prover to prove that the transmitter works properly for any $k \geq 2$.

Using the timing diagram of the *transmitter*/fabric interface, an IRS I_{10} is designed to recognize the cells at the *transmitter*/fabric interface. The timing diagram and IRS I_{10} are shown in Figures 5.12 and 5.13. In the initial state, I_{10} awaits a routing *tag h1* from the transmitter. This happens when the fabric environment machine I_{env} that monitors the fabric inputs is in state *p9*. (States of I_{env} are shown in the timing diagram). By a transition *s0* -> *s1*, I_{10} detects the state *p9* of I_{env} and the routing *tag h1* = 1. The transitions *s0* -> *s1* -> *s2* -> *s0* trace the non-representative cell *cell0*, i.e., <1, 1, 0, 0, 0, 0, 0, 0, 0, 0>. The transitions *s0* -> *s1*-> ... -> *s10* recognize the cell *cell1*, i.e., <1, 1, 1, 2, 3, 4, 5, 6, 7, 8>. After that in state *s10*, only the values 0 or 1 are accepted, indicating that the sequence of *cell1* has occurred once, as required by the data independence assumption.

The ack signal has been taken into account in IRS I_{10} only after recognizing *h1* and *h2* and the first byte of the cell body in *s3*. This is derived from *ackOut0* in the timing diagram specification in Figure 5.13. In the case of a negative ack in states *s3* to *s9*, I_{10} returns to the initial state *s0*, since the transmitter stops and it will retransmit in the next frame. Remark that, following the timing diagram, after the last byte of the cell "8" is sent in *s9*, no ack is expected for it in *s10*. Due to the pipeline delay in the fabric, it is supposed that the last byte of the cell will be unconditionally accepted by the fabric, given that the preceding bytes of the cell were accepted [27].

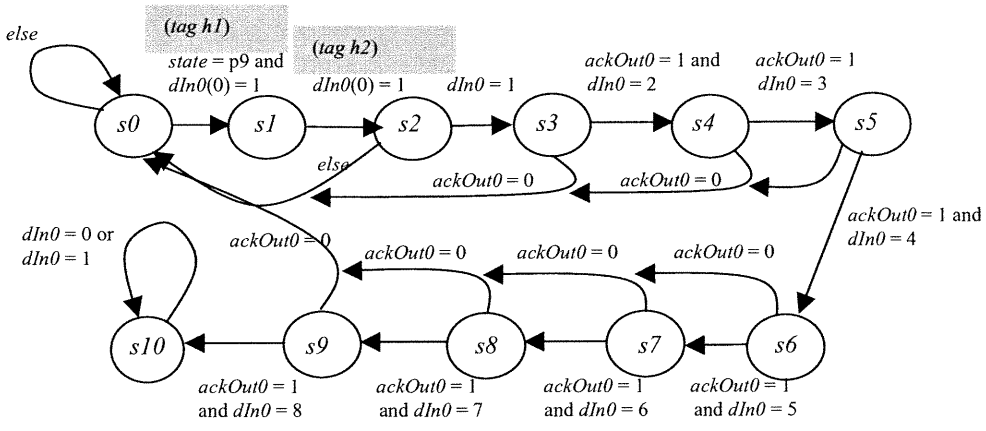


Figure 5.12: IRS I_{10} at the *transmitter/fabric* interface

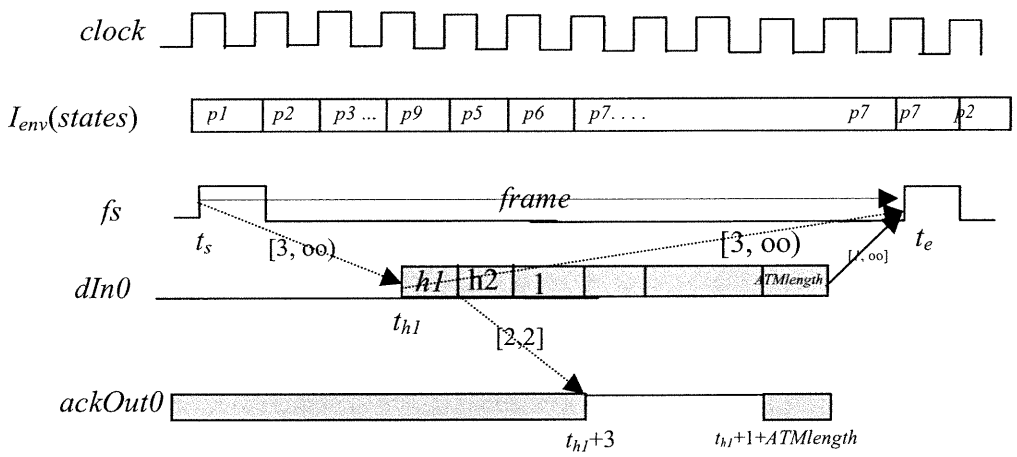


Figure 5.13: Timing diagram specification of the *transmitter/fabric* interface (Figure 5.1)

The recognition conditions are encoded by the following constraints on I_{10}:

1. (*routing tag h2*) C_{h2} := [(*state* = *s1*) ⇒ (*dIn0* = 1)], where *dIn0* designates the transmitter output.

2. (*data value 2*) C_2 := [(*state* = *s3* and *ackOut0* = 1) ⇒ (*dIn0* = 2)]

3. (*data value 0 or 1)* C_{01} := [(*state* = *s10*) ⇒ (*dIn0* = 0 or *dIn0* = 1)]

Constraint C_2 asserts that data value 2 is recognized in state *s3*, if ack is asserted in that state. Similar constraints are stated for data values 3, 4, .., 8. C_{01} asserts that a value 0 or 1 is recognized from state *s5* on.

We verified that the system composed of the transmitter, the memory, and the interface machines I_{fs}, I_9, I_{10} satisfies all mentioned constraints. For instance, by activating C_{fs} (on the frame start pulse generator IRS), we verify that
(ALWAYS: $C_{h2} = true$). The entire verification can be denoted by

$$I_{fs}(C_{fs}) \wedge I_9(C_{uni1}) \wedge transmitter \wedge memory \models I_{10}(C_{h2}, C_2, ..., C_8, C_{01}).$$

Now, we consider the order preservation property (*T3*). Similar to the receiver subsystem, we verify that the transmitter transfers the cells in order. Suppose that the address input of the transmitter is constrained to a sequence (0)*(1)(0)*(2)(0)* by an IRS I'_9. Let *cell0*, *cell1*, and *cell2* denote three arbitrary but different cells in memory addresses 0, 1, and 2 respectively. We use an IRS I'_{10} to recognize the sequence (*cell0*)*(*cell1*)(*cell0*)*(*cell2*)(*cell0*)* at the transmitter output. This verification is represented by the following formula:

$$I_{fs}(C_{fs}) \wedge I'_9(C_{uni1}, C_{uni2}) \wedge transmitter \wedge memory \models I'_{10}(C'_{cell1}, C'_{cell2}, C'_{uniCell1\_2}), \quad (5.25)$$

where C'_{cell1}, C'_{cell2}, and $C'_{uniCell1\_2}$ denote the constraints that recognize the sequence of *cell1* and *cell2* at I'_{10}.

5.7.2 Transmitter liveness

The properties checked by IRS constraints are safety properties asserting that if a particular state is reached, a corresponding property holds. These constraints do not assert, for instance, that the states that satisfy the properties are actually ever reached. Liveness properties have to be added to guarantee that these states are eventually visited. For instance, we verify that *cell1* is eventually sent to the *fabric/out* port interface I_{11}.

By the definition of I_{10} in Figure 5.12, *ackOut0* has to be asserted from state *s3* up to state *s9*.

$C_{ackOut0}$:= [((*state* = *s3*) or (*state* = *s4*) or (*state* = *s5*) or (*state* = *s6*) or (*state* = *s7*) or (*state* = *s8*) or (*state* = *s9*)) $\Rightarrow$ (*ackOut0* = 1)] (5.26)

Now, we activate (5.26) at I_{10} to prove that the transmitter eventually sends the unique cell to the fabric. As usual, we use the AFTER-EVENTUALLY construct of Formal Check [FC99] to express the following liveness property.

$I_{fs}(C_{fs}) \wedge I_9(C_{uni1}) \wedge$ *transmitter* $\wedge$ *memory* $\wedge$ $I_{10}(C_{ackOut0})$ |=
AFTER: I_9(*state* = *q1*) $\wedge$ I_{fs}(*state* = *f2*) EVENTUALLY: I_{10}(*state* = *s10*) (5.27)

(5.27) asserts that the model composed of I_{fs} with constraint C_{fs} activated, queue IRS I_9 with C_{uni1} activated, the transmitter, the memory, and I_{10} with $C_{ackOut0}$ activated as an *assumption* has the property that after I_9 reaches state q1 and I_{fs} reaches state *f2*, the transmitter eventually sends *cell1* and thus drives I_1 to state *s10*. Note that when I_9 reaches state *q1* and I_{fs} reaches state *f2*, the top element of the queue becomes 1, i.e., the pointer to the *cell1* comes out to the transmitter and a frame start pulse is generated by I_{fs}. Then, the unique cell is recognized at the *transmitter/fabric* interface, along the sequence prescribed by I_{10}.

5.8 Composing local properties of the in port controller components

We have separately verified the in port controller components. Now, before connecting them together, we have to verify that they are compatible, i.e., each subsystem involved in the reasoning satisfies no-output-constraining (*W1*, *W4*) and no-cycle-of-gates properties (*W2*, *W3*, *W5*).

1 (*Wellfoundedness/compatibility*) Similar to $I_1 \wedge$ *receiver* $\wedge$ *queueR* $\wedge$ I_4 subsystem in Section 5.5, we verify that $I_1 \wedge$ *receiver* $\wedge$ *queueR* $\wedge$ *dipatcher* $\wedge$ *queueP1* $\wedge$ *queueP2* $\wedge$ *scheduler* $\wedge$ *queueT* $\wedge$ *transmitter* $\wedge$ I_{10} satisfy well-foundedness

conditions. Having verified well foundedness, local safety properties can then be combined together to prove global ones.

2 (*Safety*) We have verified that the in port controller components individually preserve data order. By a reasoning similar to the one that was used in composing of the receiver and the queue R in Section 5.5, we use IRS transitivity rule to conclude that the in port controller preserves cells order.

$$I_1(C_{cell0}, C_{cell1}, C_{cell2}, C_{uniCell1\_2}) \wedge inPortController \models I'_{10}(C'_{cell1}, C'_{cell2}, C'_{uniCell1\_2}) \quad (5.28)$$

Constraints C'_{cell1} and C'_{cell2} in (5.28) recognize the sequence of *cell1* and *cell2* at I'_{10}. They assert that the headers of *cell1* and *cell2* are converted according to the routing table and then the routing tags *h1* and *h2* are prepended to them. We used cells of the same priority, since otherwise the order may not be preserved. In this way, (5.28) infers *cell extraction*, *header transformation, no duplication, and order preservation* properties of the in port controller that are defined in Chapter 4.

Verification of (5.28) proves safety properties such that, for instance, the in port controller cannot duplicate cells or change their order. It does not assert, however, that any cell is eventually transferred. This is considered in liveness verification.

3 (*Liveness*) We have to prove that the composed system is live. Suppose a stream (*cell0*)*(*cell1*)(*cell0*)* is "supplied" by I_1 at the in port controller input. From the liveness property of the components, we should prove that after a *cell1* (of high priority) arrives to the input, it eventually reaches I_{10} at the output of the in port controller. The following subgoal express this global liveness property. Fairness and other assumptions needed to prove the liveness property are stated as well.

$I_{fs}(C_{fs}) \wedge I_1(C_{cell0}, C_{cell1}, C_{uniCell1}) \wedge inPortController \wedge A0 \wedge A1 \wedge A2 \wedge A3 \wedge A4 \wedge A5$
$\wedge A6 \wedge A7 \models AFTER: I_1(state = s7) \quad EVENTUALLY: I_{10}(state = s10)$ (5.29)

where *A0-A7* are defined as follows: *A0* states that *fs* pulses are repeatedly generated; *A1* to *A7* are collected from the component verifications. In the following definitions, *rok*, *dok*, and *sok* denote the arbiter permission for the receiver, the dispatcher, and the scheduler, respectively. Interfaces $I_1, \ldots, I_{10}$ are pictured in Figure 5.1.

$A0 := AFTER: true$	$EVENTUALLY: I_{fs}(state = f2)$
$A1 := AFTER: I_1(state = s4)$	$EVENTUALLY: (rok \wedge \sim qRfull)$
$A2:= AFTER: \sim qRempty \wedge \sim qP1full$	$EVENTUALLY: deqR$
$A3 := AFTER: I_4(state = s1)$	$EVENTUALLY: (dok \wedge \sim qP1full)$
$A4:= AFTER: \sim qP1empty \wedge \sim qTfull$	$EVENTUALLY: deqP1$
$A5 := AFTER: I_7(state = s1)$	$EVENTUALLY: (sok \wedge \sim qTfull)$
$A6:= AFTER: \sim qTempty \wedge \sim qFfull$	$EVENTUALLY: deqT$
$A7 := ALWAYS: I_{10}(C_{ackOut0})$	

Primary assumptions *A0* and *A7* should be respectively satisfied by the *fs* pulse generator and the switch fabric which form the global environment of the in-port controller. However, *A1-A6* must be discharged on the components of the in-port controller. We have to prove *that rok* $\wedge$ *~qRfull, deqR , dok* $\wedge$ *~qP1full , deqP1, sok* $\wedge$ *~qTfull,* and *deqT* occur infinitely often.

Proof: We first show that *rok, dok,* and *sok* in *A1, A3*, and *A5* are infinitely often asserted by the arbiter. We use linear temporal logic formulas when reasoning about liveness properties. (For instance *GF p* expresses "infinitely often *p*", etc.) In Section 5.2, we mentioned that the arbiter can grant memory accesses to the receiver and the dispatcher if the transmitter releases the memory repeatedly. The transmitter, in turn releases the memory if it receives an acknowledgment from the fabric, during the cell transmission. However, if it does not receive any acknowledgment, it tries to retransmit the cell several times before dropping the cell. Whether the cell is

discarded or sent out, the transmitter will eventually release the cell memory (*GF ~transmitterMemReq*). Now, the arbiter can grant memory accesses to the components, when the transmitter is not accessing the memory.

$$
\begin{aligned}
&I_{fs}(C_{fs}) \wedge transmitter \wedge A0 \models GF\ \ (\sim transmitterMemReq) \\
&Arbiter \models G\ (\sim transmitterMemReq \Rightarrow\ \ rok \vee dok \vee sok) \\
&Arbiter \models G\ (\sim transmitterMemReq \wedge \sim dok \wedge \sim sok \Rightarrow rok) \\
&Arbiter \models G\ (\sim transmitterMemReq \wedge \sim rok \wedge \sim sok \Rightarrow dok) \\
&Arbiter \models G\ (\sim transmitterMemReq \wedge \sim rok \wedge \sim dok \Rightarrow sok)
\end{aligned}
\qquad (5.30)
$$

Next, we discharge assumptions *A2, A4,* and *A6* by proving that *deqR, deqP1*, and *deqT* occur infinitely often. The transmitter asserts *deqT* infinitely often to transmit new cells to the fabric.

$$I_{fs}(C_{fs}) \wedge transmitter \wedge A0 \models G\ (\sim qTempty \wedge \sim qFfull \Rightarrow F\ deqT) \qquad (5.31)$$

(5.31) discharges *A6*.

$$I_{fs}(C_{fs}) \wedge transmitter \wedge A0 \models A6 \qquad (5.32)$$

(5.32) is represented by an edge *A0 -> A6* in the "proof graph" shown in Figure 5.40. A proof graph [15] illustrates the dependencies of assumptions on other assumptions and properties. *A non-circular proof graph ensures that the overall liveness verification is well founded.*

By (5.31), *deqT* is asserted infinitely often as far as the queue *T* is not empty and *F* is not full. Queue *T* becomes infinitely often not full (i.e., *~qTfull*) letting the scheduler write new addresses to *T*, which are dequeued from *P1*.

$$\text{scheduler} \wedge GF\ \ (sok) \models G\ (\sim qTfull \wedge \sim qP1empty\ \Rightarrow\ F\ deqP1) \qquad (5.33)$$

(5.33) is represented by *GF (sok) -> A4* in Figure 5.14. By (5.33), *deqP1* is asserted infinitely often, making *P1* not full and letting the dispatcher write new addresses to *P1* that are dequeued from *R*.

$$dispatcher \wedge GF\ (dok) \models G\ (\sim qP1full \wedge \sim qRempty \Rightarrow F\ deqR)$$

Therefore, *deqR* is infinitely often asserted and the queue *R* becomes not full. This lets the receiver to read new cells from the input FIFOs and insert their addresses to *R*.

$$receiver \wedge GF\ (rok) \models G\ (\sim qRfull \wedge \sim FIFOempty \wedge \sim qFempty \Rightarrow F\ (deqF \wedge deqFIFO))$$

A8 in the proof graph is defined as follows:

A8 := AFTER: $\sim qRfull \wedge \sim FIFOempty \wedge \sim qFempty$
EVENTUALLY: $deqT \wedge deqFIFO$

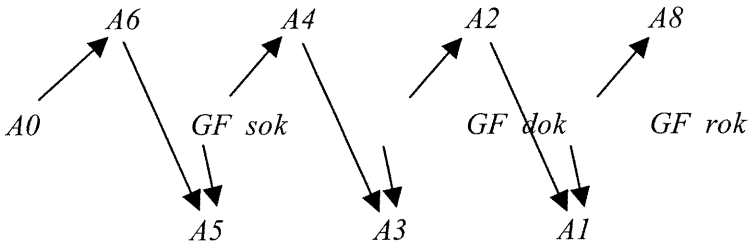


Figure 5.14: Proof graph to discharge assumptions *A1-A6*

The proof graph in Figure 5.14 indicates that *A1-A6* and *A8* hold true if *sok, dok*, and rok are infinitely often asserted and *A0* is respected by the pulse generator.

$$A0 \wedge GF\ sok \wedge GF\ dok \wedge GF\ rok \Rightarrow A1 \wedge A2 \wedge A3 \wedge A4 \wedge A5 \wedge A6 \qquad (5.34)$$

Using (5.34) and the arbiter verification (5.30), the liveness property (5.29) can be rewritten as follows:

$$I_{fs}(C_{fs}) \wedge I_1(C_{cell0}, C_{cell1}, C_{uniCell1}) \wedge inPortController \wedge A0 \wedge A7 \models$$
$$\text{AFTER: } I_1(state = s7) \qquad \text{EVENTUALLY: } I_{10}(state = s10) \qquad (5.35)$$

By (5.35), if the acknowledgement is asserted during cell transmission (i.e., *A7* is respected) and frame pulses are repeatedly generated (i.e., *A0* is met), then (high priority) cells will successfully be sent out to the fabric.

(5.35) proves the in port controller liveness property. Before presenting the switch fabric and its composition with the in port controller, we first consider a stronger liveness property, the proof of which requires circular reasoning.

5.8.1 Circular reasoning to prove liveness property

From the definition of assumptions *A2*, *A4*, and *A6* in Section 5.8, it follows that the in port controller is "active" (i.e., it reads the queues by asserting *deqR*, *deqP1*, and *deqT* repeatedly) as long as the corresponding queues are not empty. However, when the queues become empty, the controller eventually becomes inactive. We prove a liveness property that if complete ATM cells arrive infinitely often, then the queues *R*, *P1*, and *T* become repeatedly non-empty.

Proof: A proof graph for the goal that the queues become repeatedly not empty is illustrated in Figure 5.15. An edge $p \xrightarrow[M]{F} q$ indicates that p is to be assumed in module M when proving $F\,q$. Similarly $p \xrightarrow[M]{X} q$ indicates that p is assumed in the current state when proving q in the next state. As shown, the graph is circular. Intuitively, queue *T* becomes non-empty (i.e., *~qTempty*) if *enqT* is asserted. The scheduler can assert *enqT* if the queue *P1* is not empty, since it transfers addresses from *P1* to *T*. *P1* becomes non empty if *enqP1* is asserted. It is asserted if *R* is not empty. R becomes non empty if *enqR* is asserted and *enqR* is asserted if the input FIFO and the queue *F* are not empty. However, *F* to become non empty requires that *enqF* be asserted, and

it happens when T is not empty. This reasoning is apparently circular, since T to become non empty requires that T be non empty. As mentioned in Chapter 2, the circularity can be resolved by induction over time. This rule is shown in Figure 5.16. As shown, if the cycle is cut by at least one time unit delay (e.g., one X in the graph), then the circular reasoning is sound by induction. While we used the X operator to implement the proof graph, McMillan [15] uses UNTIL operator to implement it. In Appendix 2, we show that our approach is a conserative approximation to that in [15].

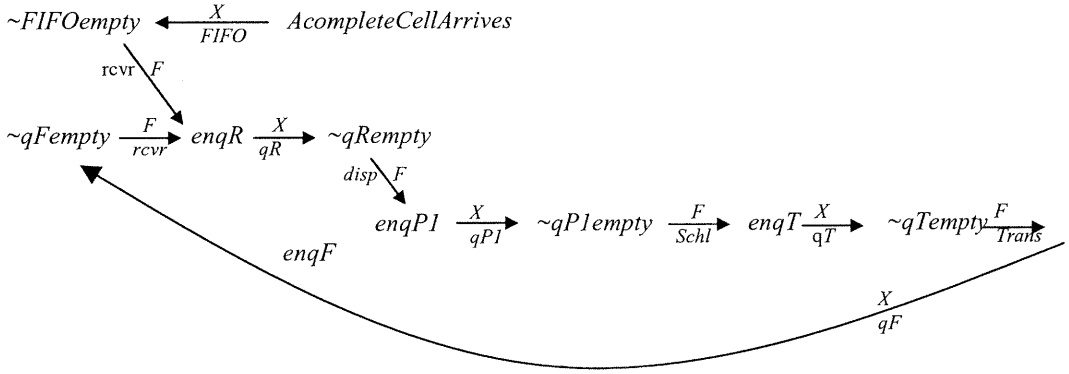


Figure 5.15: A proof graph for the in port controller global liveness

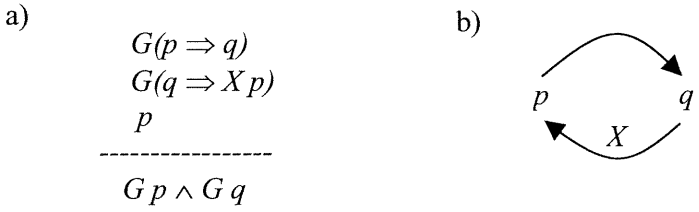


Figure 5.16: Circular reasoning by induction over time. a) The rule. b) The proof graph

Consider the graph in Figure 5.15. We assume that queue F is not empty (at the beginning) in the initial state, otherwise, there is no place available in the memory to deposit the very first cell. The cycle in the graph is cut by a delay of 4 time units, since there are 4 *Xs* on the graph edges. (For example, consider when *enqR* is asserted in the current state, R will become non-empty in the next state.)

Using relations

[G (p ⇒ F q)] ⇒ [G (F p ⇒ F q)] and [G (p ⇒ X q)] ⇒ [G (F p ⇒ XF q)], the proof graph can be represented by the formulas shown in (5.36). By an inductive reasoning similar to the basic case in Figure 5.16, it follows that all propositions on vertices of the cycle happen infinitely often. We have assumed that [*GF ~FIFOempty ∧ GF ~qFempty*] ⇒ [*GF (~FIFOempty ∧ ~qFempty)*], because (1) when *~FIFOempty* and *~qFempty* are true, they remain so until the queue or the FIFO is dequeued. (2) The receiver waits to observe that the queue F and the input FIFO both are not empty before asserting *deqF ∧ deqFIFO*. The conclusion is that the queue and the FIFO will be both not empty at the same time if they are so separately. This synchronization guarantees that *enqR* in the proof graph can be asserted repeatedly, if *~FIFOempty* and *~qFempty* become true (independently) infinitely often.

In summary, the queues R, *P1*, and *T* become repeatedly non empty and dequeues are repeatedly asserted given that complete ATM cells arrive repeatedly into input FIFOs.

$$G\,[F\ AcompleteCellArrives \Rightarrow XF\ {\sim}FIFOempty]$$

$$[GF\ {\sim}FIFOempty \wedge GF\ \mathbf{{\sim}qFempty}] \Rightarrow [GF\ ({\sim}FIFOempty \wedge {\sim}qFempty)]$$

$$G\,[F\ ({\sim}FIFOempty \wedge {\sim}qFempty)] \Rightarrow F\ enqR]$$

$$G\,[F\ enqR \Rightarrow XF\ {\sim}qRempty]$$

$$G\,[F\ {\sim}qRempty \Rightarrow F\ enqP1]$$

$$G\,[F\ enqP1 \Rightarrow XF\ {\sim}qP1empty]$$

$$G\,[F\ {\sim}qP1empty \Rightarrow F\ enqT]$$

$$G\,[F\ enqT \Rightarrow XF\ {\sim}qTempty]$$

$$G\,[F\ {\sim}qTempty \Rightarrow F\ enqF\ ]$$

$$G\,[F\ enqF\ \Rightarrow \mathbf{XF\ {\sim}qFempty}]$$

$$F\,[\mathbf{{\sim}qFempty} \wedge AcompleteCellArrives\ ]$$

---------- ----------(5.36)

$$GF\ (AcompleteCellArrives) \Rightarrow [GF\ ({\sim}FIFOempty) \wedge GF\ (enqR) \wedge GF({\sim}qRempty) \wedge GF\ (enqP1) \wedge GF({\sim}qP1empty) \wedge GF\ (enqT) \wedge GF({\sim}qTempty) \wedge GF\ (enqF\ ) \wedge GF\ ({\sim}qFempty)\ ]$$

Proving liveness properties is more difficult than proving safety properties. Soundness of the circular reasoning for safety properties is guaranteed by the no-cycle-of-gates conditions of the well-foundedness rules (*W1-W5*). Liveness properties usually require fairness assumptions. A proof strategy (or so called proof graph) must be planned to discharge these fairness assumptions. For a sound circular reasoning for liveness, every cycle in the proof graph must be cut by at least one time unit delay [15]. This is represented by an explicit *X* operator in our representation of the proof graph. This delay guarantees the result by induction over time.

After proving safety and liveness properties of the in port controller, we next consider the fabric properties. We verify that the fabric is timing-compatible with the in port controller. Finally, we will infer the global properties of the switch module from the in port controllers and the fabric properties.

5.9 Switch fabric verification

In this section, we specify the fabric properties and show how to compose them with the in port controllers properties.

F1 (*Cell transfer*) After dropping the routing *tag h1* from the cells, the fabric should forward them to the requested out ports.

F2 (*Ack transfer*) The fabric should forward the acknowledgement from the out ports to the successful in ports.

F3 (*Prioritization*) High priority cells should be given precedence over the low priority ones while arbitrating in the fabric.

F4 (*Order preservation*) The fabric should preserve the order of cells.

We use IRS to specify and verify these properties. In Section 5.7, we proved that the transmitter of the in port controller sends $(cell0)^*(cell1)(cell0)^*$ to the fabric on I_{10}.

$$I_{fs}(C_{fs}) \wedge I_9(C_{uni1}) \wedge transmitter \wedge memory \models I_{10}(C_{h2}, C_2, ..., C_8, C_{01})$$

Now, we prove that the fabric correctly transfers these cells to an out port. To verify *F1* and *F3* in a 4x4 switch fabric, we set a scenario in which in *port0* is requesting the out *port0*, with a high priority while other in ports are free to request either of out ports, however with a low priority. In *port0* should successfully be connected to out *port0*.

We explain how to set up the scenario[16]: Figure 5.18 illustrates that the inputs *dIn0* and *dIN1* of the fabric have been constrained by the constraints of I_{env}, I_{10} and I_{fs} and its outputs *dOut0* and *ackOut0* are observed by constraints of an IRS I_{11}. The

[16] We illustarte a 2x2 switch fabric, however, the verifications are carried on a 4x4 model.

environment IRS I_{env} shown in Figure 4.5 in Chapter 4 provides a correct sequence of inputs to the fabric. Suppose that I_{env} begins to send a non-zero $h1_{p0}$ (the routing tag on in port 0), and a non-zero $h1_{p1}$ (the routing tag on in port 1) in state *P9*. Both $h1_{\mathrm{p0}}$ and $h1_{\mathrm{p1}}$ should then be zero before *P9*. This (initial) zero segment of inputs can be enforced by $C_{h1\mathrm{p0}}$ and $C_{h1\mathrm{p1}}$ of $\mathrm{I_{env}}$. Let *x* represent a don't care value, i.e., a 0 or 1. Using the format shown in Figure 4.2 in Chapter 4, the scenario is initialized by activating a constraint C_{scen} that assigns "xxxx0011" to $h1_{\mathrm{p0}}$ and "xxxxxx01" to $h1_{\mathrm{p1}}$ in *P9*. Let C_{h1p0}, C_{h1p1}, and C_{scen} denote respectively the environment and the scenario constraints in Figure 5.18. Assuming that the fabric strips off *h1* before sending out the cell body, the following IRS I_{11} can recognize *cell1* at out port 0.

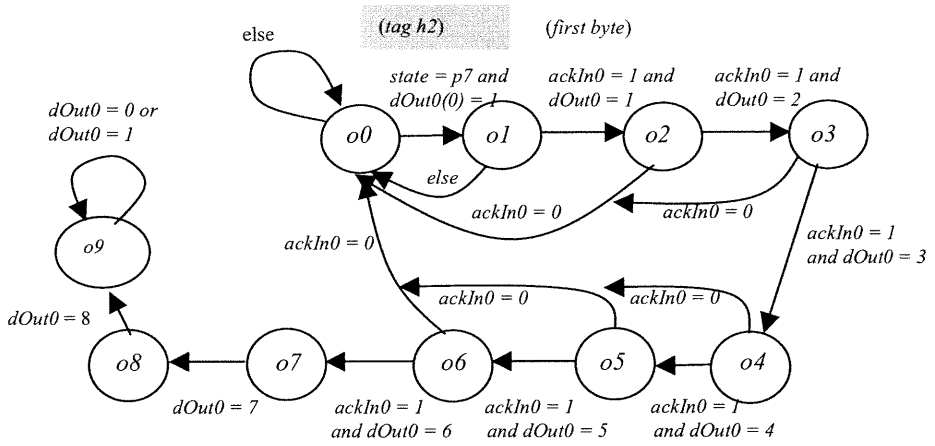


Figure 5.17: IRS I_{11} for the cells at the fabric/out port interface

The *tag h2* arrives at the fabric out port when IRS I_{env} reaches state *P7*. This state is used as a means of synchronization between the property IRS $\mathrm{I_{11}}$ and the environment IRS I_{env} which supplies the timing infra-structure for the verification. Constraints of I_{env} ensure that *h1* is kept zero before *P9* on all in ports of the fabric (therefore, the ports remain inactive). From *P9*, the constraints of I_{10} send *cell0* and *cell1* to the fabric on in port 0 (More precisely, $\mathrm{I_{env}}$ represents the environment assumptions, while I_{10} represents the preconditions of the cell delivery property). Finally, constraints of I_{11} recognize the cells at the fabric out port 0 (Figure 5.18).

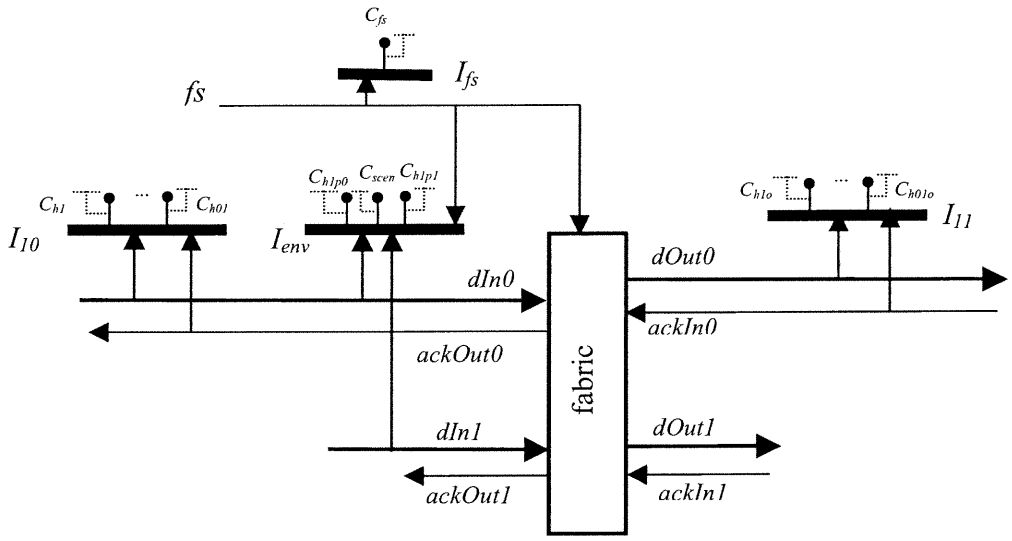


Figure 5.18: The fabric subsystem for cell delivery property

The first recognition constraint is written for data value 2.

1. (*data value 2*) C_{2o} := [(*state* = o2 and *ackIn0* = 1) ⇒ (*dOut0* = 2)]

2. (*data value 3*) C_{3o} := [(*state* = o3 and *ackIn0* = 1) ⇒ (*dOut0* = 3)]

3. (*data value 0 or 1*) C_{01} := [(*state* = o4) ⇒ (*dOut0* = 0 or *dOut0* = 1)]

Consider the subsystem $I_{10} \wedge I_{env} \wedge I_{fs} \wedge fabric \wedge I_{11}$ in Figure 5.18. We activate constraints (C_{h2}, C_2, ..., C_8, C_{01}) of I_{10} on the fabric inputs. We verify that the fabric respects constraints C_{2o}, C_{3o}, C_{01o} of I_{11}, indicating that the representative cell *cell1* is recognized at the fabric output. The verification can be denoted by the following formula.

$I_{fs}(C_{fs}) \wedge I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge I_{10}(C_{h2}, C_2, ..., C_8, C_{01}) \wedge fabric$

$$/= I_{11}(C_{2o}, ..., C_{8o}, C_{01o}) \quad (5.37)$$

The constraints of I_{11} (being true) assert a conditional property that *cell1* can be recognized on I_{11} if states of I_{11} are visited. We need a liveness property to prove that these states are eventually visited. For instance, we verify that after the transmitter sends *cell1*, i.e., I_{10} reaches its final state *s10*, the same cell is eventually recognized at the fabric output, i.e., the final state *o9* of I_{11} is eventually reached.

$$I_{fs}(C_{fs}) \wedge fabric \models \text{AFTER: } I_{10}(state = s10) \quad \text{EVENTUALLY: } I_{11}(state = o9)$$

Next we verify property *F2* by showing that the fabric correctly forwards the acknowledgement from the out ports to the successful in ports. First, we assume a condition on out *port0* that *ackIn0* is asserted during the cell transmission from state *o1* to state *o6* of I_{11} (Figure 5.17).

$$C_{ackIn0} := [\ (\ (state = o1\ ) \text{ or } (state = o2) \text{ or } (state = o3) \text{ or } (state = o4) \text{ or } (state = o5) \text{ or } (state = o6)\ )\ \Rightarrow\ (ackIn0 = \text{'1'})\ ] \quad (5.38)$$

Similarly, by the definition of I_{10} in Figure 5.12, *ackOut0* has to be asserted from state *s3* up to state *s9*.

$$C_{ackOut0} := [\ (\ (state = \text{s3}\ ) \text{ or } (state = s4\ ) \text{ or } (state = s5) \text{ or } (state = s6) \text{ or } (state = s7) \text{ or } (state = s8) \text{ or } (state = s9)\ )\ \Rightarrow\ (ackOut0 = 1)\ ] \quad (5.39)$$

Now, we can verify that the fabric correctly transfers ack from out *port0* to the in *port0* by checking that (5.38) discharges (5.39). Verification results show that constraint (5.38) is not strong enough to prove (5.39). The counter example indicates that a positive ack, sent in state *o1* by I_{11} could not arrive to I_{10} sooner than in *s4* . The

switch components can be made timing-compatible as follows. Currently, the transmitter sends routing *tags h1*, *h2*, and the first element of the cell without any ack (Figure 5.13). The solution consists of shifting the transmitter expectation one cycle away, e.g., it would expect the ack at *s4* , after sending *h1*, h2, byte number 1, and byte number 2 of the cell. Constraint (5.39) is modified as follows and the ack transfer can then be successfully verified:

$$C_{ackOut0} := [\ (\ (state = s4\ )\ \text{or}\ (state = s5)\ \text{or}\ (state = s6)\ \text{or}\ (state = s7)\ \text{or}\ (state = s8)\ \text{or}\ (state = s9)\ )\ \Rightarrow\ (ackOut0 = 1)\ ] \tag{5.40}$$

The ack transfer from I_{11} to I_{10} can be recapitulated as follows:

$$I_{fs}(C_{fs}) \wedge I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge fabric \wedge I_{11}(C_{ackIn0}) \models I_{10}(C_{ackOut0}) \tag{5.41}$$

Finally, we consider the last property of the fabric: Order preservation *F4*. We should verify that if a sequence (*cell0*)*(*cell1*)(*cell0*)*(*cell2*)(*cell0*)* is supplied by an IRS I'_{10} on the fabric input, the same sequence is recognized by I'_{11} at the fabric output. However, we can simplify the verification. Given that the fabric does not store cells and it only forwards them (with a fixed latency) to the successful out ports, we simply prove that after the arbitration finishes, the values on a destination out port of the fabric are equal to those that were on the (source) in port of the fabric. Consequently, the sequence of data on the out port will equal the sequence of data on the in port. The order will be preserved since fabric does not store any cell and cannot inverse their order.

We explain how we implemented this verification. Suppose that the fabric receives routing tags $h1_{p0}$, h_{2p0}, and a cell on its in *port0*. I_{env} observes the reception of $h1_{p0}$ and h_{2p0} while it is in states *P9* and p5, respectively (Figures 5.13 and 4.5). The fabric transfers cells with a latency of two cycles to the destination out ports. Suppose that it removes $h1_{p0}$ and sends h_{2p0} to the out port 0. Two cycles later, h_{2p0} is received on the

out *port0* (while I_{env} reaches state *P7*). Let *dIn0* _rr denote the values on *dIn0* that are delayed for two cycles. The following property asserts that in the current frame from state *p7* (i.e., two cycles after the reception of h_{2p0}) until state p3 (i.e., two cycles after the next frame start) the octets of the cell at the fabric out *port0* equal to those stored in *dIn0 _rr*.

$I_{fs}(C_{fs}) \wedge I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge fabric \models$
After: state = p7 *Always: dOut0 = dIn0 _rr* *Unless: state = p3*

Given that the frame is bigger than the cell size, the fabric forwards cells to the requested out ports in the order they are received from the in ports. Let us note that this implementation of the fabric does not contain any input or output latches, hence, the latency has been reduced from 4 cycles (reported in [32]) to 2 cycles.

5.10 Composing in port controllers and the fabric

We have separately verified the in port controller and the fabric. Before connecting them together, we have to verify that they are compatible. We consider compatibility of the transmitter and the fabric, since the fabric communicates only with the transmitter component of the in port controller.

1 (*Well foundedness/compatibility*) We have to verify that each subsystem involved in the compositional reasoning satisfies no-output-constraining and no cycle-of-gates conditions (*W1-W5*) (Chapter 3). For instance, we examine *W3-W5* for the subsystem $I_{10} \wedge I_{env} \wedge I_{fs} \wedge fabric \wedge I_{11}$ shown in Figure 5.19.

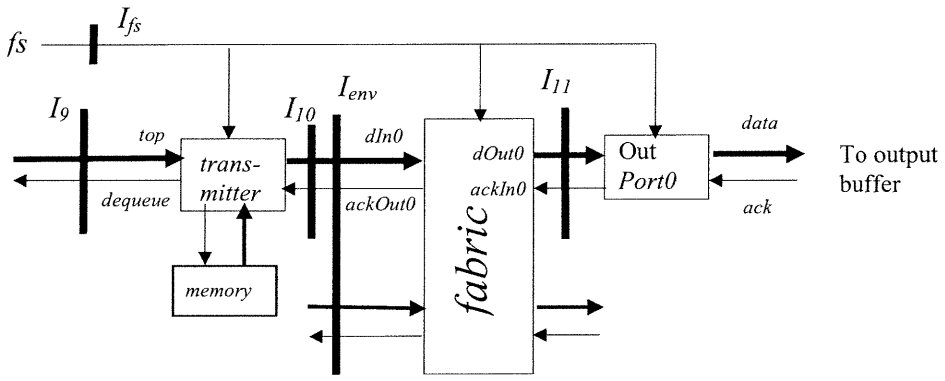


Figure 5.19: Well-foundedness check for the compositional verification of the transmitter and the switch fabric

I_{10} introduces $ackOut0 \xrightarrow{0}_{I10} dIn0$, however, there is no any zero-delay path from *dIn0* to *ackOut0* in $I_{env} \wedge I_{fs} \wedge fabric \wedge I_{11}$. I_{env} introduces $fs \xrightarrow{0}_{Ienv} dIn0$, and there is no path from *dIn0* to *fs in* $I_{10} \wedge I_{fs} \wedge fabric \wedge I_{11}$. So no (static or conditional) cycle can be formed in $I_{10} \wedge I_{env} \wedge I_{fs} \wedge fabric \wedge I_{11}$. IRSs I_{10}, I_{env}, and I_{fs} do not constrain outputs (*ackOut0* and *dOut0*) of the fabric. Therefore, $I_{10} \wedge I_{fs} \wedge I_{env} \wedge fabric \wedge I_{11}$ satisfies no-output-constraining and no-cycle-of-gates conditions. Similarly, we checked that $I_9 \wedge I_{fs} \wedge transmitter \wedge memory$ satisfies these conditions. Hence, the composition is sound and the properties proven using IRS constraints remain valid in the composed system.

2 (*Safety*) Safety properties of the *inPortController* $\wedge$ *fabric* can be deduced from the following verification: (1) In Section 5.8, we showed that the in port controller correctly extracts cells, deposits them in the cell memory, converts their headers, and appends two routing *tags h1* and *h2* to each cell in the memory. (2) The verification of the transmitter concluded that these cells are correctly forwarded from the memory

to the transmitter/fabric interface I_{10}. (3) The verification of the fabric proved that the same cells are correctly forwarded from I_{10} to the *fabric/outPort* interface I_{11}. For instance, they illustrated that the routing *tag h1* is removed and that the cells once and in order are routed to the out ports. From these verifications and the transitivity property of IRS I_{10}, it follows that the cells are correctly routed from end to the end, from the in port controller inputs to the switch fabric output.

3 (*Liveness*) *inPortController* $\wedge$ *fabric* should be live, i.e., it should be eventually possible to route a cell from input FIFOs to the out ports of the fabric. By (5.35), the in port controller forwards a cell to the fabric (i.e., to I_{10}) if it receives ack during the cell transmission.

$$I_{fs}(C_{fs}) \wedge I_1(C_{cell0}, C_{cell1}, C_{uniCell1}) \wedge inPortController \wedge A0 \wedge I_{10}(C_{ackOut0}) \models$$
$$AFTER:\ I_1(state = s7) \qquad EVENTUALLY:\ I_{10}(state = s10) \tag{5.42}$$

By (5.41), the fabric forwards ack from the out ports to the requested in ports.

$$I_{fs}(C_{fs}) \wedge I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge fabric \wedge I_{11}(C_{ackIn0}) \models I_{10}(C_{ackOut0}) \tag{5.43}$$

From (5.42) and (5.43) and the transitivity rule of IRS, it follows that:

$$I_{fs}(C_{fs}) \wedge I_1(C_{cell0}, C_{cell1}, C_{uniCell1}) \wedge inPortController \wedge A0 \wedge$$
$$I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge fabric \wedge I_{11}(C_{ackIn0}) \models$$
$$AFTER:\ I_1(state = s7) \qquad EVENTUALLY:\ I_{10}(state = s10) \tag{5.44}$$

Now, after receiving the cells, the fabric eventually forwards the successful ones to the requested out ports.

$$I_{fs}(C_{fs}) \wedge fabric \models AFTER:\ I_{10}(state = s10) \quad EVENTUALLY:\ I_{11}(state = o9)$$

From this verification, we conclude that *inPortController* $\wedge$ *fabric* can correctly route cells from input FIFOs (i.e., from I_1) to the fabric out ports (to I_{11}), given that it receives ack from the out ports as specified on I_{11}.

$I_{fs}(C_{fs}) \wedge I_1(C_{cell0}, C_{cell1}, C_{uniCell1}) \wedge inPortController \wedge A0 \wedge$

$I_{env}(C_{h1p0}, C_{h1p1}, C_{scen}) \wedge fabric \wedge I_{11}(C_{ackIn0}) \models$

AFTER: $I_1(state = s7)$ *EVENTUALLY:* $I_{11}(state = o9)$

5.11 Summary and Experimental Results

We used Interface Recognizers/Suppliers (IRS) to implement circular constraint model checking. IRS allows us to separately verify each component of the switch and then infer the end-to-end properties of the system. We applied the data independence assumption and cell size reduction techniques to further reduce the verification problem, since otherwise the verification was not possible. The IRS key functions provide a transitivity rule and a mechanism to prove end-to-end properties, i.e., supplying assumptions (as constraints on inputs) and verifying properties (using constraints). We specified both safety and liveness properties of the ATM switch, nevertheless, for the liveness properties, we used IRS and temporal operators of the model checker FormalCheck [3], since IRS alone has no mechanism to specify such properties.

Although the concept of the proof graph and inductive circular reasoning for liveness was recently proposed in [15], our implementation of the rule is slightly different. For instance, we explicitly introduce an initial requirement (as the base of the induction) and an induction step using next time (X) operator. In the original form, these steps are computed using the until operator (U). In Appendix 2, we compare these two implementations of the rule.

Using the ATM switch, we illustrated our approach including the specification, the compositional verification rule, and the appropriate reduction techniques tailored to

the verification of switch-type designs. This method can be used in the verification of similar network components that contain queues.

Table 1 illustartes experimental results of our case study. We spent 3 months on developing VHDL models of port controllers and 3 months out carrying on the verifications. As Table 1 shows each model checking run took under 2 minutes, however, developing and debugging IRSs and the switch components were the most time-consuming activity.

Table 1:Experimental results of the switch fabric verification

Module	Property	Reached states	Time (sec.)	Memory usage (M)
Receiver	(5.7)	5.09e+8	78	52.92
Receiver	(5.8)	1.10e+7	62	52.42
Receiver	(5.9)	5.48e+5	62	51.91
Receiver	(5.10)	6.38e+03	46	45.53
Receiver	(5.12)	1.06e+06	45	46.20
Receiver	(5.13)	1.29e+5	26	33.42
Receiver	(5.14)	1.67e+5	28	33.51
Receiver	(5.15)	1.41e+5	27	33.52
Dispatcher	(1)	4.08e+04	14	13.66
Dispatcher	(2)	4.08e+04	13	13.66
Scheduler	(S1)	5.21e+04	6	11.87
Scheduler	(5.33)	100	24	11.80
Transmitter	(5.25)	5.36e+04	61	21.65
Transmitter	(5.27)	1.02e+04	61	20.43
Arbiter	(5.30)	46	14	0.91
Fabric	(5.37)	3.37e+11	42	17.76
Fabric	(5.41)	3.32e+11	39	16.50
Fabric	(5.42)	3.37e+11	44	19.84

Chapter 6

Conclusions and future work

In this thesis, we proposed to use interface recognizers/suppliers (IRS) as a practical mechanism to specify environment assumptions of reusable components, e.g., IP cores. We have also used IRS to specify constraints and properties for model checking.

The main advantage of IRS is its ability to act (without any code modification) as a supplier of assumptions or verifier of a property. This key feature allows us to implement a compositional verification method. For instance, in one step of the reasoning, IRS verifies property on its inputs. Then, in the next step, it supplies that property as an assumption to the subsequent components in the system. This easy conversion of property/assumption provides a transitivity rule and allows us to implement end-to-end verification of modular systems.

IRS is different than a monitor in that it constrains (or shapes) its inputs. A monitor does not constraint its inputs; it only generates an output like other components in the system. This input-constraining property of IRS is instrumental for symmetrically being a supplier or a recognizer. On one hand, when IRS constraints remain true, they indicate that the property that is represented by the constraints is satisfied by the component. On the other hand, by forcing the constraint to be always true, IRS restricts its inputs to the sequence that satisfies the property. In this way, the property is supplied as an assumption to other components.

Although IRS constraints its inputs, we have to assure that the outputs of the components are not restricted when composed with IRS. We established this requirement by a set of well-foundedness/compatibility conditions that are adapted from reactive modules [36].

The original contributions of this thesis are as follows:

1 We reviewed recent developments in compositional and assume guarantee verification (Chapter 2). We discussed whether each method supports circular/non circular reasoning and whether it can be used when proving safety/liveness properties.

2 We formulated interface recognizers/supplies, which are recognizers augmented with Boolean constraints (Chapter 3). The constraints specify what values may occur on IRS inputs at each state. In other words, IRS can constrain its inputs.

3 We developed a composition theorem for circular reasoning using IRS. In this way, IRS framework extends non circular (or asymmetric) constraint model checking [25] to a circular (or symmetric) constraint model checking (Chapter 3).

4 We demonstrated an application of IRS in (1) specifying environment assumptions and in (2) modeling pre conditions / post conditions of properties of an ATM switch. We specified, implemented and verified the switch. [The 4x4 ATM switch is about 15000 lines of VHDL code and includes more than 1500 state variables.] Verification of such a complex system that handles cells of 53 bytes is far beyond the capacity of current industrial model checkers like Formal Check [3]. We proposed to use abstraction techniques, e.g., cell size reduction and data independence assumptions to reduce the complexity of the verification. Our approach can be applied to other systems that contain several queues or involve cell processing. Although the application of the data-independence assumption is not new, its application to an

ATM switch that processes bounded cells with header and routing tags (defining cell priorities) is new.

5 IRS can specify only safety properties. Nevertheless, we have shown how to use IRS and temporal operators of a model checker to specify and verify liveness properties.

6.1 Future work:

1. (*Automation*) We have constructed the environment IRS from a timing diagram specification of the fabric. It should be possible to automatically generate environment IRS from timing diagrams. For instance, Allara et al. [1] have developed a tool called STD to automatically translate timing diagrams into temporal logic formulas. Tableau algorithm [6] generates a finite state machine (FSM) for a temporal logic formula. Using STD and the tableau algorithm, one should be able to generate FSM from the timing diagram.

Property IRS is obtained from specifications other than the timing diagrams. We obtained the property IRS of the switch module from the switch specification. For instance, the order preservation property required that a *cell1* arrive before a *cell2* to the fabric. These specifications guided us to develop suitable IRS to model pre conditions/post conditions of the properties. Automatic generation of one or more IRS from such specifications should be investigated as well. Other issues concern the constraints. The constraints of IRS are obtained from the transition conditions. In our case study, we manually defined them. Automatically formulating constraints of the IRS is another step toward full automation. More experiments have to be carried out to fully understand problems related to the automatic construction of IRS.

2 (*Processor verification*) another possible direction is to study the application of IRS methodology to microprocessor verification. It should be investigated what kind of processor specifications or properties can be modeled by IRS. Switch is a data-independent system, i.e., there is less interdependency between cells. A general-

purpose microprocessor has different characteristics. The data independence assumption is not valid on them. In practice, there can be global dependencies between different IRSs in the system. For instance, when a particular state of one IRS is reached, it may prevent another IRS from receiving data (e.g., an operand of an instruction, etc). These interdependencies will require one or more additional IRS to coordinate the local IRSs.

3 (*Multiple layers of protocols*) IRS can possibly be used in multi-layer verification. Telecommunication protocols have multi-layer structures. The physical layers communicate data at bit level, data link layers at frame level, network layers at packet level, and transport layers at user defined message level [44]. Suppose that each layer is separately represented and verified by an IRS. Then, how could these IRS be related to each other? Consider a similar problem. If for a given interface several IRSs are defined, what relation must be established between these IRSs? There is some pioneering work in multiple layer verification [14]. It must be investigated how these multiple layer refinement maps could be specialized by IRS. And, finally, another question arises. Is it possible to replace one transition of the original IRS by a second IRS like in hierarchical state charts [7]?

References

[1]. A. Allara, M. Bombana, S. Comai, B. Josko, R. Schlor, D. Sciuto, "Specification of Embedded Monitors for Property Checking", *proceedings of Forum on Design Languages*, FDL'99, pages 117-126, 1999.

[2]. A. Dekdouk, O. Ait, M. S. Jahanpour, E Cerny, "On the Formal Verification of the IP-based designs", *International workshop on IP-Based synthesis and system designs (IWLAS'98)*, France, December 15-16 1998.

[3]. Affirma Formal Check, version 2.3, *Cadence Design Systems*, Inc. 1987-1999

[4]. A. Pnueli, In transition for global to modular temporal reasoning about programs", *In K. R. Apt, editor, Logics and models of concurrent systems*, volume 13 of NATO ASI series. Series F, Computer and system sciences, springer Verlog, 1984.

[5]. D. D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems", *PTR prentice Hall*, 1994.

[6]. D. E. Long, "Model Checking, Abstraction and Compositional Verification", *PhD Thesis, Dept. of CS, Carnegie Mellon University*, 1993.

[7]. D. Harel, "Statecharts : A visual Formalism for complex Systems", *Journal of Science of Computer Programming*, vol. 8, Number 3, pages 231-274, 1987.

[8]. Formal methods in system design, *http://kapis1.wkap.nl/kaphtml.htm/*

[9]. IEEE Std 1076-1993. "Institute of electrical Engineers", *Inc, New York*, USA, 1994.

[10]. International Technology Roadmap for Semiconductors, http://public.itrs.net/Files/2001WinterMeeting/Presentations/Design.pdf

[11]. J. E. Hopcroft, J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation", *Addison-Wesley publishing company*, Inc, 1979.

[12]. J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, "Automatic Vector Generation Using Constraints and Biasing", *Journal of Electronic Testing, Theory and Applications*, vol. 16, Numbers 1-2, pages 107-120, Feb/April 2000.

[13]. J. Lu, S. Tahar, "Practical approaches to the automatic verification of an ATM switch fabric using VIS", *Proceedings of the 8th Great Lakes Symposium on VLSI (Cat. No.98TB100222)*, pages 368-73, 1998.

[14]. K. L. McMillan, "A Compositional rule for hardware design refinement", *In CAV'97: Computer Aided Verification*, Lecture Notes in Computer Science, Springer Verlog, pages 24-35, 1997.

[15]. K. L. McMillan, "Circular Compositional Reasoning about Liveness Properties", *Cadence Berkley Labs*, http://www-cad.eecs.berkeley.Edu/~kenmcmil/papers/, 1999.

[16]. K. L. McMillan, "Symbolic Model Checking", *Kluwer*, 1993.

[17]. K. L. McMillan, "A methodology for hardware verification using compositional model checking", *Cadence Berkley Labs*, http://www-cad.eecs. berkeley.Edu /~kenmcmil /papers/, 1999.

[18]. K. L. McMillan, "Tutorial on model checking", *www-cad.eecs*.Berkely/Edu /kenmcmil, 1997

[19]. K. L. McMillan, "Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking" , *In A. Hu and M.Vardi, editors, CAV98: Computer Aided Verification*, Lecture Notes in Computer Science, Springer-Verlag, pages 100-121, 1998.

[20]. K. S. Namjoshi, R. J. Trefler, "On the Completeness of Compositional reasoning", *In CAV2000: Computer Aided Verification*, Lecture Notes in Computer Science, Springer Verlog, pages 139-153, 2000.

[21]. L. Lamport, "The Temporal Logic of Actions", *ACM Transactions on Programming Languages and Systems*, Vol. 16, number 3, pages 872-923, May 1994.

[22]. M. Abadi, L. Lamport, "Conjoining Specifications", *ACM Trans. on Prog. Lang. and Syst.*, Vol. 17, number 3, pages 507-534, May 1995.

[23]. M. Abadi, L. Lamport, "Composing Specifications", *ACM Trans. on Prog. Lang. and Syst.*, Vol. 15, number 1, pages 73-132, January 1993.

[24]. McMillan, "Getting started with SMV", *www-cad. eecs. berkely. edu/ kenmcmil*, 1999.

[25]. M. Kaufmann, A. Martin, and C. Pixley. "Design constraints in Symbolic Model Checking", *In A. Hu and M.Vardi, editors, CAV98: Computer Aided Verification, Lecture Notes in Computer Science, Springer-Verlag*, pages 477-487, 1998.

[26]. M. Kohler, "NP Complete", *Embedded Systems Programming*, Vol. 13, No.12, pages 45-60, November 2000.

[27]. M. Langevin, E. Cerny, "A Description of the Fairisle ATM Switch Controller", *www.iro.umontreal.ca*, labo LASSO, 1994.

[28]. M. S. Jahanpour, E. Cerny, "Compositional verification Using Interface Recognizers/Suppliers (IRS) ", in *Proc. Workshop on Advances in Verification (WAVe'2000),* June 2000. http://www.cs.utah.edu/wave/

[29]. M.S. Jahanpour, E. Cerny, "Compositional Verification of an ATM Switch Using Interface Recognizers/Suppliers (IRS) ", in *IEEE int'l High Level Design Validation and Test Workshop*, pp. 71-76, November 2000, Berkley, California

[30]. M. Sipser, "Introduction to the theory of computation", *preliminary edition, PWS publishing company, international Thompson Publishing Inc.*, 1996.

[31]. O. Grumberg, D. E. Long, "Model Checking and Modular Verification", *ACM Transaction on Programming Language and Systems*, Vol. 16, No. 3, Pages 843-871, May 1994.

[32]. P. Curzon, "The Formal Verification of the Fairisle ATM Switching Element", *Technical Report Number 329, Computer laboratory*, University of Cambridge, March 1994.

[33]. P. Kurshan, L. Lamport, "Verification of a multiplier: 64 bits and beyond", *In Computer-Aided Verification*, volume 697 of Lecture Notes in Computer Science, pages 166-179, Springer-Verlag, 1993.

[34]. P. Wolper. "Expressing interesting properties of programs in prepositional temporal logic", *In 13th ACM POPL*, pages 184-193, 1986.

[35]. R. Alur, T. A. Henzinger, "Reactive Modules", *proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS)*, pages 207-218, 1996.

[36]. R. Alur, T. A. Henzinger, "Reactive Modules", *www. gigascale. org/ mocha/ refs.shtml*, 1998.

[37]. R. Alur, T. Henzinger, O. Kuppfermann, "Compositionality: The significant Difference", in *W-P. de Rover, H. Langmeak, A. Pnueli (Eds) International Symposium COMPOS97*, B. Malentent, Germany, pages 23-60, September 1997.

[38]. R. Brayton et al., "VIS: A System for verification and synthesis", *Technical Report UCB/ERL M95, Electronics Research Laboratory*, University of California, Berkley, December 1995.

[39]. R. Milner. "A Calculus of Communicating Systems", *volume 92 of Lecture notes in Computer Science*. Springer-Verlog, 1980.

[40]. R. P. Kurshan, "Computer-Aided Verification of Coordinating processes, the automata theoretic approach", *Princeton University Press, Princeton*, New Jersey, 1994.

[41]. S. A. Kripke, "Outline of a theory of truth". *Journal of philosophy*, Vol. 72, pages 690-716, 1975.

[42]. S. P. Rajan, M. Fujita, "Integration of High-Level Modeling, Formal Verification, and High-Level Synthesis in ATM Switch Design", *Proceedings of 11th International Conference on VLSI design: VLSI for Signal Processing*, 1997.

[43]. S. P. Rajan, M. Fujita, K. Yaun, M T-C. Lee, "High-Level Design and Validation of ATM Switch", *Proc. IEEE International High Level Design Validation and Test Workshop (HLDVT'97)*, Oakland, California, USA, November 1997.

[44]. S. Qadeer, "Algorithms and Methodology for scalable Model Checking", *PhD thesis, Dept. of EECS, University of California at Berkley*, Fall 1999.

[45]. S. S. Lam, A. U. Shankar, "A Theory of Interfaces and Modules–I Composition Theorem", *IEEE Transactions on Software Engineering*, Vol. 20, Number 1, pages 55-71, January 1994.

[46]. S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin, "Formal Verification of an ATM Switch Fabric using Multiway Decision Graphs", *IEEE trans. CADICS* Vol. 18, number 7, pages 956-972, July 1999.

[47]. T. A. Henzinger, S. Qadeer et al. "You assume we guarantee: Methodology and Case Studies*", In A. Hu and M. Vardi, editors, CAV'98: Computer Aided Verification*, Lecture Notes in Computer Science, Springer Verlag, pages 440-451 1998.

[48]. Verplex, *www.verplex.com*, 2000.

[49]. W-P. deRover, H. Langmaak, and A. Pnueli, editors. "Compositionality : The significant Difference", *volume 1536 of LNCS*. Springer-Verlag, 1997.

[50]. Y. Zorian, R. K. Gupta, "Design and Test of Core-Based systems on Chips", *IEEE DESIGN & TEST OF COMPUTERS*, page14, 1997.

[51]. Z. Manna, A. Pnueli, "The temporal logic of Reactive and Concurrent Systems", *Springer-Verlag*, 1992.

[52]. Z. Salcic, "VHDL and FPLDs in Digital Systems Design, Prototyping, and Customization", *Kluwer Academic Publishers*, 1998.

Appendix I

Assume guarantee in reactive modules [36]

A reactive module represents a system that interacts with an environment. A reactive module M (or module M, for short) has a finite set of variables, denoted V_M. A "state" of M is a valuation for V_M. V_M is partitioned into three sets; input variables I, output variables O, and privates variables P. While O and P are updated by M, I is updated by the environment. A module M consists of one or more atoms that control ($O \cup P$) variables of the module. Each atom controls one or more variable, however, every variable is controlled by one and only one atom. Let X_a be a finite set of variables of an atom a. X_a contains three sets of variables; a set of controlled variables $ctrX_a \subseteq X_a$, a set of read variables $readX_a \subseteq X_a$, and a set of awaited variables $waitX_a \subseteq \{X_a \setminus ctrX_a\}$. A controlled variable of an atom may depend sequentially on a read variable of the atom, much like a register output that depends on register input. A controlled variable y of an atom may depend combinationally on an awaited variable x of the atom. This is denoted $x <_a y$ to indicate that atom a can update y only after x has been updated.

We review the assume guarantee theorem in reactive modules. Reactive modules *M1* and *M2* are "compatible" if (1) their outputs are disjoint and (2) the transitive closure $(<_{M1} \cup <_{M2})+$ is asymmetric, i.e., they form no cycle-of-gates (Chapter 2). Let *M1* and *M2* be two compatible modules, and let *N1* and *N2* be two compatible modules such that every input of *N1* // *N2* is an input or an output of *M1* || *M2*. Let $\leq$ denote

the trace-containment relation. If $M1 \,//\, N2 \leq N1$ and $M2 \,//\, N1 \leq N2$, then $M1 \,//\, M2 \leq N1 \,//\, N2$.

Proof: Let $T_t = s_0 s_1 \ldots s_t$ represents a trace of length $(t+1)$ (for instance, T_0 is a trace of length 1). Let trace(M) represent the set of all traces of M. The relation $\leq$ represents the trace-containment (Chapter 2). Suppose a formula $M1 \leq (N1, t)$ denotes that all traces T_t of $M1$ of length (t+1) are traces of $N1$. Let $[T]_M$ represent the projection of a trace T over input/output variables of M (Chapter 2). The rule is proved by an induction on trace length t. Suppose that

1. Every trace T_{t-1} of $M1 \,//\, M2$ (of length t) is a trace of $N1 \,||\, N2$.

$T_{t-1} \in$ trace $(M1 \,||\, M2)$. $[T_{t-1}]_{N1||N2} \in$ trace $(N1 \,||\, N2)$ (A.1)

2. Every trace of $M1 \,||\, N2$ is a trace of $N1$.

$M1 \,||\, N2 \leq N1$ (A.2)

3. Every trace of $M2 \,||\, N1$ is a trace of $N2$.

$M2 \,||\, N1 \leq N2$ (A.3)

Consider a trace T_t of $M1 \,||\, M2$:

$T_t \in$ trace $(M1 \,||\, M2)$ (A.4)

We have to prove that $[T_t]_{N1 \,||\, N2} \in$ trace $(N1 \,||\, N2)$. From (A.4) and the definition of the composition, we have that the projection $[T_t]_{M1}$ is a trace of M1.

$[T_t]_{M1} \in$ trace (M1) (A.5)

Consider the trace T_{t-1} of $M1 \,||\, M2$. $[T_{t-1}]_{N1||N2}$ is a trace of $N1||\, N2$, by (A.1). Then, the projection of that trace over $N2$ is a trace of $N2$:

$[T_{t-1}]_{N2} \in$ trace $(N2)$ (A.6)

We show that $[T_t]_{M1||N2} \in$ trace $(M1||N2)$. From (A.5), we have $[T_{t-1}]_{M1} \in$ trace (M1). Then, by (A.6) we get $[T_{t-1}]_{M1||N2} \in$ trace $(M1 \,||\, N2)$. We know that $T_{t-1} \in$ trace $(M1 \,||\, M2)$. Let Compare $M1 \,//\, M2$ to $M1 \,||\, N2$ when $M1 \,//\, M2$ makes a transition from T_{t-1}

to T_t. We show that $O_{N2} \subseteq O_{M2}$. For any trace, we have $M2 \parallel N1 \leq N2$. Given that $O_{N2} \subseteq (O_{M2} \cup O_{N1})$ (By the definition of $M2 \parallel N1 \leq N2$), and $O_{N1} \cap O_{N2} = \varnothing$ (by the definition of $N1 \parallel N2$), then $O_{N2} \subseteq O_{M2}$. *M1* in $M1 // M2$ receives inputs from *M2* and also from an environment. *M1* in $M1 \parallel N2$ receives inputs from *N2* and from the environment. Since $[T_{t-1}]_{M1\parallel N2} \in$ trace (M1 $\parallel$ *N2*) and $O_{N2} \subseteq O_{M2}$, *M1* receives the same input (from *M2*) in $[T_{t-1}]_{M1\parallel M2}$ that it receives (them from *N2*) in $[T_{t-1}]_{M1\parallel N2}$ (given that the free inputs have been adjusted accordingly). It means that at the next time, *M1* can generate the same output in $M1\|N2$ as it generates in $M1 \parallel M2$. Hence, the outputs of *M1* in $M1 \parallel N2$ at time *T* are the same as its outputs in $M1 // M2$ at time t. The following denotes this.

$$\text{Val}(O_{M1}, M1 \parallel M2, t) = \text{Val}(O_{M1}, M1 \parallel N2, t), \quad (A.7)$$

where $\text{Val}(O_{M1}, M1 \parallel M2, t)$ represents the value of outputs of *M1* in $M1 \parallel M2$. Now, Similar to $O_{N2} \subseteq O_{M2}$, we have $O_{N1} \subseteq O_{M1}$. For any trace, we have $M1 \parallel N2 \leq N1$. By this trace containment, inputs/outputs of *N1* can be assigned values equal to inputs/outputs of *M1* in $M1 \parallel N2$.

$$\text{Val}([O_{M1}]_{N1}, M1 \parallel N2, t) = \text{Val}(O_{N1}, t) \quad (A.8)$$

(A.8) and the projection of (A.7) over *N1* give the following result.

$$\text{Val}([O_{M1}]_{N1}, M1 \parallel M2, t) = \text{Val}(O_{N1}, t) \quad (A.9)$$

Similarly, we can prove that

$$\text{Val}([O_{M2}]_{N2}, M1 \parallel M2, t) = \text{Val}(O_{N2}, t) \quad (A.10)$$

Consider the inputs of *N1* and *N2*. Inputs of *N1* in $N1\|$ *N2* come from the outputs of *N2* plus some free inputs. Given the equalities (A.9) and (A.10), and the fact that there is no cycle of gates in the designs and that the designs are non-blocking for their inputs, all inputs/outputs of *N1* in $N1 \parallel N2$ (including the free inputs and the inputs from *N2*) can be assigned values equal to those of *M2* in $M1 \parallel M2$. Therefore, the projection of $[T_t]_{M1}$ over *N1*, i.e., $[T_t]_{N1}$ is a trace of *N1*, $[T_t]_{N1} \in$ trace (*N1*). Similarly, inputs/outputs of *N2* can be assigned values equal to those of *M2* in $M1 \parallel M2$. So, $[T_t]_{N2} \in$ trace (*N2*). Putting both together, we get

$$[T_t]_{N1\|N2} \in \text{trace } (N1\|N2) \quad (A.13)$$

Consider the base case. An empty trace (i.e., a trace of length 0) of $M1 // M2$ is a trace of all systems, including $N1 \parallel N2$. Consider a trace T_0 (of length 1) of $M1 \parallel M2$. It

contains a valuation for the initial states of $M1 \parallel M2$. By definition, the projection of T_0 (of $M1 \parallel M2$) over $M1$, i.e., $[T_0]_{M1}$ is a trace of $M1$. Since, $M1$ and $N2$ are compatible, they contain disjoint set of outputs and no-cycle-of-gates. Therefore, $M1$ accepts any (initial) value for its inputs, coming from $N2$. $[T_0]_{M1}$ can thus be extended to contain any initial valuation for variables of $N2$. In particular, $[T_0]_{M1//N2}$ that contains initial assignments to $M1$ and $N2$ becomes a trace of $M1||N2$. By $M1 \parallel N2 \leq N1$, we have that $[T_0]_{N1} \in$ trace ($N1$). Similarly, from $[T_0]_{M2} \in$ trace(M2), it follows that $[T_0]_{M2//N1} \in$ trace ($M2||N1$), and $[T_0]_{N2} \in$ trace($N2$). By $[T_0]_{N1} \in$ trace ($N1$) and $[T_0]_{N2} \in$ trace($N2$), we get that $[T_0]_{N1||N2} \in$ trace ($N1||N2$). We conclude that any trace of $M1 // M2$ is a trace of $N1 \parallel N2$, by this induction on the trace length.

Appendix II

Technical details

In this appendix, we review data independence assumtion, simulation relation, and other technical details used in chapters 2, 5.

B1. Data abstraction by data independence assumption

Data abstraction is a technique that can be used in model checking of the systems that employ a large number of data values. In such systems, reducing the data set to fewer representative values may be sufficient to prove a property about the whole data set. An assumption known as data independence [34] allows one to implement such an abstraction. Data independent systems can be separated into two parts; a control and a data path such that the values of the data do not affect the control state. In a data independent system, if we change the input data, the behavior of the system will not change, except for the corresponding values of the data output.

When a system satisfies data independence property, its data set can be reduced to a smaller set while verifying the system properties. For instance, for the verification that a protocol delivers all data, only two values, say 0 and 1 may be enough to prove the data delivery property: One value for representing the data we are tracking for data delivery, and the other one for representing all the other values.

Example 1 (*Bounded buffer* [18]) Consider a bounded first-in-first-out (FIFO) buffer. Suppose that the data set is reduced to $\{0, 1\}$. Suppose we have verified that if data

value 1 enters the buffer exactly once, then it reaches out exactly once. For this verification, the model checker has used an arbitrary number of 0s and only one 1. (In automaton terminology, a stream described by (0)*(1)(0)* is provided to the buffer input, and the stream described by (0)*(1)(0)* is obtained on its output.) From this verification, it follows that the buffer can deliver any data without duplication. Let in(x) denote that a value x is enqueued in the buffer. Similarly, out(y) indicates that a value y is dequeued from the buffer. Suppose a data, for instance 2 is duplicated in the following sequence.

in(1); in(2); in(3); out(1); out(2) out(2); …

If the duplicated output is changed to 1, and all others to zero, then according to the data independence property of the buffer, we should get the following sequence.

in(0); in(1); in(0); out(0); out(1) out(1); …

This sequence clearly violates exactly_once_in(1)/ exactly_once_out(1) property of the buffer. Therefore, no data is duplicated by this data independent buffer, if it can not be done for the reduced set. Kurshan [40] states that to reduce the data set to two values during model checking, the following conditions should be met:

1. System be symmetric, with respect to permuting values,
2. The first value of data is generated precisely once, non-deterministically within a stream of data having the second value.

The order preservation property can similarly be verified using a reduced data set {0, 1, 2}. Suppose that, by supplying a stream (0)*(1)(0)*(2)(0)* to the buffer input, the same stream is recognized at its output. Notice that data values 1 and 2 are supplied once, non-deterministically among a stream of zero. Then, it follows that the buffer preserves the order among all the data it receives. The proof is similar to that of the

data-delivery property. Suppose that a data, e.g., 4, enters the buffer before a data 5, but comes out after 5 in the following sequence.

in(1); in(4); in(2); in(5); out(1); out(5); out(2); out(4) …

By replacing 4 by 1, 5 by 2, and all others with 0, we get the following sequence, which contradicts the recognized language (0)*(1)(0)*(2)(0)* at the buffer output. Therefore, the order will be preserved for any data set, if it is preserved for the reduced set {0, 1, 2}.

in(0); in(1); in(0); in(2); out(0); out(2); out(0); out(1) …

B2. Simulation relation

The simulation relation [39] is classically defined over "Kripke structures" [41]. Let $M = <S, Init, T, A, L>$ and $M' = <S', Init', T', A, L'>$ be two structures. A relation $H \subseteq S \times S'$ is a simulation relation over $M \times M'$ iff the following conditions hold:

1. $(Init, Init') \in H$.
2. For all $(s, s') \in H$, $L(s) = L'(s')$ and

$$\forall t\, [\, (s, t) \in T \;\Rightarrow\; \exists t'\, [\, (s', t') \in T' \wedge (t, t') \in H \,]\,].$$

Whenever there exists a simulation relation H over $M \times M'$, we write $M \leq M'$ to express that M is simulated by M'.

B3. Realtionship between upto- and at-inductions

We want to show that the at-induction

$$\forall t.\ [\, q(0) \wedge (p(t) \Rightarrow q(t+1))\,] \qquad \text{(B.1)}$$

implies the upto-induction:

$$\forall t\,.\;\; [\, p|^{(t-1)} \Rightarrow q(t)\,]. \qquad \text{(B.2)}$$

Proof: (B.2) at time 0 asserts $q(0)$, by definition,. (B.1) also asserts $q(0)$. Thus, (B.1) => (B.2) at time 0. Consider $T > 0$. Let $A = p|^{(t-1)}$, $B = p(t\text{-}1)$, and $C = q(t)$. We have to show that:

$$[\, B \Rightarrow C\,] \Rightarrow [\, A \wedge B \Rightarrow C\,] \qquad \text{(B.3)}$$

(B.3) can be rewritten as follows:

$$[\, (\neg B) \vee C\,] \Rightarrow [\neg(A \wedge B) \vee C\,] \qquad \text{(B.4)}$$

For any two propositions P and Q, we have that $[(P \wedge Q) \Rightarrow P\,]$. This is true whether $Q = true$ or Q = false. For A, and B this is written as follows:

$$[\, (A \wedge B) \Rightarrow B\,] \qquad \text{(B.5)}$$

Using the relation $[\, (P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)\,]$, (B.5) gives the following relation.

$$[\, (\neg B) \Rightarrow \neg\,(A \wedge B)\,] \qquad \text{(B.6)}$$

By conjuncting both sides of (B.6) with C, we get (B.4), i.e.,

$$[\, (\neg B) \vee C\,] \Rightarrow [\neg(A \wedge B) \vee C\,] \qquad \text{(B.7)}$$

Using this proof and the relation (2.16) in Chapter 2, we conclude that the at-induction is a conservative approximation to the upto-induction, i.e.,

$$[\, q(0) \wedge\; G\,(p \Rightarrow X\; q)\,] \Rightarrow [\neg(p\; U\; \neg q)\,]$$

It is easy to show that the upto-induction does not imply the at-induction. For instance, let $A = false$, $B = true$, $C = false$ in (B.3). Then, the upto-induction $(A \wedge B) \Rightarrow C$ holds true, while the at-induction $B \Rightarrow C$ does not hold true.

Appendix III

SMV [24] model of a queue

We used CBL SMV [24] to show that a generic queue delivers data from its input to its output. In this appendix, we first prove that when the condition (enqR $\wedge$ ~qRfull) holds, an arbitrary input enters the queue. Then, using the SMV induction rule, we verify that the data in any position in the queue eventually reaches the top position, assuming that the dequeue is asserted infinitely often. This proves that the generic queue correctly delivers the data it receives. Similarly, we prove that this queue preserves data order

```
/* the queue model */

scalarset DATA undefined;
ordset INDEX 0..;

module main(enq, deq, inp){

 input deq, enq : boolean;
 input inp : DATA;

 cells : array INDEX of DATA;
 count : INDEX;

 SIZE : INDEX;
 next(SIZE) := SIZE;


 if(enq = 1 & deq = 0){
     if(count < SIZE){   /* not full */
     forall(i in INDEX)
       next(cells[i]) := (i = count) ? inp
: cells[i];
     next(count) := count+1 ;
     }
  /* if full, the action is blocked */
     else {
        forall(i in INDEX)
       next(cells[i]) := cells[i];
     next(count) := SIZE ;
     }
 }
 else if (enq = 0 & deq = 1){
   if (count > 0){
    forall(i in INDEX)
    next(cells[i]) := cells[i+1] ;

     next(count) := count - 1 ;
   }

  }

 else if (enq = 1 & deq = 1){
  if (count > 0){
    forall(i in INDEX)
    next(cells[i]) := (i < (count - 1) ) ?
cells[i+1] :
```

```
                (i = count - 1)? inp :
cells[i];
    next(count) := count;
   }

  }

  else {  /* no enq no deq */
       forall (i in INDEX)
         next(cells[i]) := cells[i];
       next(count) := count;
  }

/* the property: any data can enter, if
Q Not full */

       forall(j in DATA)
    q[j] : assert G ( (inp = j & count <
SIZE & enq = 1 & deq = 0) -> X
(cells[count-1] = j) );


  forall(j in DATA) forall(s in
INDEX)forall(c in INDEX)
  subcase q[j][s][c] of q[j] for SIZE = s
& count = c;

  forall(j in DATA)forall(s in
INDEX)forall(c in INDEX)
  using INDEX -> {s-1, s} prove
q[j][s][c]; /* s-1 is needed for count - 1
in q[j] */


 /* the property: any data inside the
queue finally arrives to top of queue, if
deq asserted  inf. often */

  forall(i in INDEX) forall(j in DATA)
  P[i][j] : assert G ( (cells[i] = j & i <
count & count < SIZE) -> F (cells[0] =
j) );

  forall(s in INDEX) forall(i in
INDEX) forall(j in DATA)
  subcase P[i][j][s] of P[i][j] for SIZE
= s;

  fairDeq : assert G ( count > 0 -> F
(deq = 1) );
  assume fairDeq;

  /* the proof */

   forall(s in INDEX)forall(i in
INDEX) forall(j in DATA)
   using INDEX -> {s}, fairDeq prove
P[i][j][s];

}

/* Queue model for the order
preservation property */

/*   scalarset DATA undefined; */
  ordset INDEX 0..; /* index of cells */
  ordset SEQ 0.. ; /* sequence number
for inputs */


module main(enq, deq, inp, out){

  input deq, enq : boolean;

  input inp : struct{
    valid : boolean;
    seq_num : SEQ;
  }

  output out: struct{
    valid : boolean;
    seq_num : SEQ;
/*    data : DATA;*/
  }

  cells : array INDEX of struct{
    valid : boolean;
    seq_num : SEQ;
/*    data : DATA;*/
  }

  count : INDEX;
  cnt_i, cnt_o : SEQ;
```

```
/* generic queue size */


 SIZE : INDEX;
 next(SIZE) := SIZE;

 init(cnt_o) := 0; init(cnt_i) := 0;
 inp.valid := (enq = 1) & (count <
SIZE);

 if(enq = 1 & deq = 0){
    if(count < SIZE){   /* not full */
    forall(i in INDEX)
      next(cells[i]) := (i = count) ? inp
: cells[i];
    next(count) := count+1 ;

    next(cnt_i) := cnt_i + 1;

    }
  /* if full, the action is blocked */

    else {
       forall(i in INDEX)
       next(cells[i]) := cells[i];
    next(count) := SIZE ;

    next(cnt_i) := cnt_i;

    }
 }
 else if (enq = 0 & deq = 1){
   if (count > 0){
    forall(i in INDEX)
    next(cells[i]) := cells[i+1] ;

     next(count) := count - 1 ;
        next(cnt_o) := cnt_o + 1;
   }

  }

 else if (enq = 1 & deq = 1){
  if (count > 0){
    forall(i in INDEX)
    next(cells[i]) := (i < (count - 1) ) ?
cells[i+1] :
                    (i = count - 1)? inp :
cells[i];
    next(count) := count;

        next(cnt_i) := cnt_i + 1;
        next(cnt_o) := cnt_o + 1;
   }

 }

 else {  /* no enq no deq */
        forall (i in INDEX)
          next(cells[i]) := cells[i];
        next(count) := count;

        next(cnt_i) := cnt_i;
        next(cnt_o) := cnt_o;
 }



 /* the property */

        orderedInp : assert G ( ((enq =
1) & (count <= SIZE)) -> inp.seq_num
= cnt_i);

      ord : assert G ( ((deq = 1) &
(count>0) & (cells[0].valid = 1)&
(count<SIZE)) -> cells[0].seq_num =
cnt_o);


  forall(s in INDEX) forall(c in
INDEX) forall(i in SEQ)
  subcase ord_case[i][s][c] of ord for
cnt_o = i  & SIZE = s & count = c;


  forall(s in INDEX) forall(c in
INDEX) forall(i in SEQ)
   using
    INDEX -> {s} , ord_case[i-1]
     , orderedInp
   prove
    ord_case[i][s][c];
   assume orderedInp;
```

}

Appendix IV

VHDL models of the switch fabric and the port controller

In this appendix, we provide VHDL models of the ATM switch components, e.g., the receiver, the dipatcher, the scheduler, the transmitter, the arbiter, and the switch fabric.

C1. The receiver

```
-------------------------
-- receiver module
-------------------------

-- Written by M. Sadegh
Jahanpour.

-- parameters:
-- ATMlength is scaled down to
4, so, we set ATMlengthDiv4 =
1.
-- The first packet is written
at address 0, i.e.,
firstPacketAdrs = 0.
-- Queue element width is
scaled down to 2 bits.

------------------------
-- new modifications:

-- fifoIoutput is two bits
width and only the least
significant bit i.e,
fifoIoutput(0) is used as input
data. start-of-cell (SOC) bit
is additionaly provided along
each fifo element.
--------------------------


use work.atmDataTypes.all;

entity receiver is
  port( QFempty, QRfull,
fifoIempty, socI,
receiverGrant: in bit;
        fifoLempty, socL: in
bit;
        QFoutput: in
bit_vector(2 - 1 downto 0);
        fifoIoutput: in
bit_vector(2 - 1 downto 0);
        fifoLoutput : in
bit_vector(2 - 1 downto 0);
        reset, clock : in bit;
        onePacketFromI,
onePacketFromL : in bit;  --
added input ports
        consultFifoI,
consultQF, deqI, deqF: out bit;
        consultFifoL, deqL: out
bit;
        memInput : out
bit_vector( 4 - 1 downto 0) ;
        memAdrsRCVR : out
bit_vector(4 - 1 downto 0);
        enqR: out bit;
        receiverWillReq : out
bit;
```

```
        writeEnableRCVR : out
bit;
        QRinput : out
bit_vector(2 - 1 downto 0));

end receiver;

architecture stateMachine of
receiver is


  -- combinational circuit
signals
  type arbiterStates is (init,
i0, i1, i2, i3, i4, last, l0,
l1, l2, l3, l4) ;
  type count is range 0 to 2;
            -- ATMlengthDiv4

  signal state_c      :
arbiterStates ;  --
combinational
  signal state        :
arbiterStates;     -- registers
  signal wordCount    : count;
  signal wordCount_c : count;
  signal lb           : bit;  --
loop back register
  signal lb_c         : bit; --
and its wiring
  signal dn_c         : bit; --
internal signals for deqFneeded
  signal dn           : bit;
  signal r0_c         : bit;
  signal r1_c         : bit;
  signal r2_c         : bit;
  signal r3_c           : bit;
  signal r0           : bit;
  signal r1           : bit;
  signal r2           : bit;
  signal r3           : bit;
  signal e            :
bit_vector(2 - 1 downto 0); --
a register for queue outputs
  signal e_c          :
bit_vector(2 - 1 downto 0);  --

  signal adrs_c       :
bit_vector(4 - 1 downto 0);  --
register input
  signal address      :
bit_vector(4 - 1 downto 0);  --
the register output

  -- registered inputs

--  signal QFempty_r,
fifoIempty_r, socI_r,
receiverGrant_r: bit;  --
primary inputs
--  signal fifoLempty_r,
socL_r: bit;
--  signal QFoutput_r:
bit_vector(2 - 1 downto 0);
--  signal fifoIoutput_r:
bit_vector(2 - 1 downto 0);
--  signal fifoLoutput_r :
bit_vector(2 - 1 downto 0);
-- signal onePacketFromI_r,
onePacketFromL_r : bit;


begin  -- the state machine

  transitions: process (state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
fifoLempty, socL, wordCount,
fifoIoutput, address, r0, r1,
r2, r3, e, onePacketFromI,
onePacketFromL)
  begin
    case state is

      when init =>
        if (QFempty = '0' and
dn ='1' and lb = '0' and
            fifoIempty = '0'
and socI = '1' and
receiverGrant = '1' and
onePacketFromI = '1'
            -- new condition
            -- and
onePacketFromI = '1'
            ) then


          state_c <= i0;

        elsif (dn = '0' and lb
= '0' and fifoIempty = '0' and
socI = '1' and onePacketFromI =
'1'
               -- new condition
               -- and
onePacketFromI_r = '1'
               ) then

          state_c <= i0;

        elsif(dn = '1' and
QFempty = '1' and fifoIempty =
'1') then
```

```
                    state_c <=
init;

        elsif(dn = '0' and lb =
'0' and fifoIempty = '0' and
socI = '0') then

          state_c <= init;


        elsif(QFempty = '0' and
lb = '0' and fifoIempty = '0'
and socI = '0') then

          state_c <= init;

        elsif(dn = '0' and lb =
'0' and fifoIempty = '1') then

          state_c <= init;

        elsif(QFempty = '0' and
lb = '0' and fifoIempty = '1')
then

          state_c <= init;

        elsif(QFempty = '0' and
dn = '1' and lb = '0' and
              fifoIempty = '0'
and socI = '1' and
receiverGrant = '0') then
          state_c <= init;
          -- now, loob back
side for the init state
        elsif (QFempty = '0'
and dn ='1' and lb = '1' and
               fifoLempty = '0'
and socL = '1' and
receiverGrant = '1' and
onePacketFromL = '1'
               -- new condition
               -- and
onePacketFromL_r = '1'
               ) then

          state_c <= l0;

        elsif (dn = '0' and lb
= '1' and fifoLempty = '0' and
socL = '1' and onePacketFromL =
'1'
               -- new condition
               -- and
onePacketFromL_r = '1'
               ) then

          state_c <= l0;

        elsif(dn = '1' and
QFempty = '1' and fifoIempty =
'0') then  -- 1
          state_c <= init;

        elsif(dn = '0' and lb =
'1' and fifoLempty = '0' and
socL = '0') then  -- 2

          state_c <= init;


        elsif(QFempty = '0' and
lb = '1' and fifoLempty = '0'
and socL = '0') then -- 3

          state_c <= init;

        elsif(dn = '0' and lb =
'1' and fifoLempty = '1') then

          state_c <= init;

        elsif(QFempty = '0' and
lb = '1' and fifoLempty = '1')
then  -- 5

          state_c <= init;

        elsif(QFempty = '0' and
dn = '1' and lb = '1' and
fifoLempty = '0' and -- 6
              socL = '1' and
receiverGrant = '0') then
          state_c <= init;

        else      -- no change

          state_c <= state;

        end if;


      when i0 =>


        if (fifoIempty = '1')
then

          state_c <= init;
        elsif (fifoIempty = '0'
and socI = '1') then
          state_c <= init;
```

```
        elsif (fifoIempty = '0'
and socI = '0') then
          state_c <= i1;


        else      -- no change

          state_c <= state;

        end if;


      when i1 =>

        if ( fifoIempty = '1')
then
          state_c <= init;

        elsif (fifoIempty = '0'
and socI = '1') then
          state_c <= init;

        elsif (fifoIempty = '0'
and socI = '0') then
          state_c <= i2;


        else      -- no change

          state_c <= state;

        end if;


      when i2 =>

        if ( fifoIempty = '1')
then
          state_c <= init;

        elsif (fifoIempty = '0'
and socI = '1') then
          state_c <= init;

              elsif (fifoIempty
= '0' and socI = '0') then
                   state_c <=
i3;

              else      -- no
change

                   state_c <=
state;

              end if;


          when i3 =>

          if ( fifoIempty =
'1') then
            state_c <= init;

          elsif (fifoIempty =
'0' and socI = '1') then
            state_c <= init;

          elsif (fifoIempty =
'0' and socI = '0') then
            state_c <= i4;
          else      -- no
change

            state_c <= state;

          end if;

          when i4 =>

          if (receiverGrant =
'1' and wordCount > 0 and
fifoIempty = '1') then

            state_c <= init;

          elsif(receiverGrant =
'0') then
            state_c <= i4;


          elsif(receiverGrant =
'1' and wordCount > 0 and
fifoIempty = '0' and socI =
'1') then

            state_c <= init;

          elsif(receiverGrant =
'1' and wordCount > 0 and
fifoIempty = '0' and socI =
'0') then

            memInput(0) <= r0;
            memInput(1) <= r1;
            memInput(2) <= r2;
            memInput(3) <= r3;

            state_c <= i1;

          elsif(receiverGrant =
'1' and wordCount = 0) then

            memInput(0) <= r0;
```

```
            memInput(1) <= r1;
            memInput(2) <= r2;
            memInput(3) <= r3;

            state_c <= last;

          else      -- no
change

            state_c <= state;


          end if;

          when last =>

          if (receiverGrant =
'0' or QRfull = '1') then
state_c <= last;
          else
            state_c <= init;
QRinput <= e;
          end if;

          -- loop back side of
the state machine

          when l0 =>


          if (fifoLempty = '1')
then

            state_c <= init;
          elsif (fifoLempty =
'0' and socL = '1') then
                  state_c <=
init;

          elsif (fifoLempty =
'0' and socL = '0') then
            state_c <= l1;

          else      -- no
change

            state_c <= state;


          end if;


          when l1 =>

          if ( fifoLempty =
'1') then
            state_c <= init;
          elsif (fifoLempty =
'0' and socL = '1') then
            state_c <= init;

          elsif (fifoLempty =
'0' and socL = '0') then
            state_c <= l2;
          else      -- no
change

            state_c <= state;


          end if;
          when l2 =>

          if ( fifoLempty =
'1') then
            state_c <= init;

          elsif (fifoLempty =
'0' and socL = '1') then
            state_c <= init;

          elsif (fifoLempty =
'0' and socL = '0') then
            state_c <= l3;

          else      -- no
change

            state_c <= state;

          end if;


          when l3 =>

          if ( fifoLempty =
'1') then
            state_c <= init;

          elsif (fifoLempty =
'0' and socL = '1') then
            state_c <= init;

          elsif (fifoLempty =
'0' and socL = '0') then
            state_c <= l4;

          else      -- no
change

            state_c <= state;

          end if;
```

```
          when l4 =>

          if (receiverGrant =
'1' and wordCount > 0 and
fifoLempty = '1') then

            state_c <= init;

          elsif(receiverGrant =
'0') then
            state_c <= l4;


          elsif(receiverGrant =
'1' and wordCount > 0 and
fifoLempty = '0' and socL =
'1') then

            state_c <= init;

          elsif(receiverGrant =
'1' and wordCount > 0 and
fifoLempty = '0' and socL =
'0') then

            memInput(0) <= r0;
            memInput(1) <= r1;
            memInput(2) <= r2;
            memInput(3) <= r3;

            state_c <= l1;

          elsif(receiverGrant =
'1' and wordCount = 0) then

            memInput(0) <= r0;
            memInput(1) <= r1;
            memInput(2) <= r2;
            memInput(3) <= r3;

            state_c <= last;
          else      -- no
change

            state_c <= state;



          end if;
        end case;

        -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
fifoLempty,
          -- -- socL,
wordCount, fifoIoutput,
address, r0, r1, r2, r3, e ;

        end process
transitions;


          deqIConsultI: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, wordCount,
onePacketFromI)

          begin
            if ( (state = init
and QFempty = '0'
                  and dn ='1'
and lb = '0' and fifoIempty =
'0'
                  and socI =
'1' and receiverGrant = '1' and
onePacketFromI = '1')
                or (state =
init and QFempty = '0'
                    and dn
='1' and lb = '0' and
fifoIempty = '0'
                    and socI =
'1' and receiverGrant = '0')
                or (state =
init and dn = '1' and QFempty =
'1' and fifoIempty = '0')
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'1')
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'0')
                or (state =
init and QFempty = '0' and lb =
'0' and fifoIempty = '0' and
socI = '0')
                or (state = i0
and fifoIempty = '0' and socI =
'1')
                or (state = i0
and fifoIempty = '0' and socI =
'0')
                or (state = i1
and fifoIempty = '0' and socI =
'1')
                or (state = i1
and fifoIempty = '0' and socI =
'0')
```

```
                or (state = i2
and fifoIempty = '0' and socI =
'1')
                or (state = i2
and fifoIempty = '0' and socI =
'0')
                or (state = i3
and fifoIempty = '0' and socI =
'1')
                or (state = i3
and fifoIempty = '0' and socI =
'0')
                or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI = '1')
                or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI = '0')


                ) then

            consultFifoI <=
'1';



            else

              consultFifoI <=
'0';
            end if;

            -- deq signals

            if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '0'
and fifoIempty = '0' and socI =
'1' and receiverGrant = '1'
                 and
onePacketFromI = '1')
                or (state =
init and QFempty = '0'
                    and dn
='1' and lb = '0' and
fifoIempty = '0'
                    and socI =
'1' and receiverGrant = '0')
                or (state =
init and dn = '1' and QFempty =
'1' and fifoIempty = '0')
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'1')
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'0')
                or (state =
init and QFempty = '0' and lb =
'0' and fifoIempty = '0' and
socI = '0')
                or (state = i0
and fifoIempty = '0' and socI =
'1')
                or (state = i0
and fifoIempty = '0' and socI =
'0')
                or (state = i1
and fifoIempty = '0' and socI =
'1')
                or (state = i1
and fifoIempty = '0' and socI =
'0')
                or (state = i2
and fifoIempty = '0' and socI =
'1')
                or (state = i2
and fifoIempty = '0' and socI =
'0')
                or (state = i3
and fifoIempty = '0' and socI =
'1')
                or (state = i3
and fifoIempty = '0' and socI =
'0')
                or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI = '1')
                or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI = '0')


                ) then

            consultFifoI <=
'1';



            else

              consultFifoI <=
'0';
            end if;
```

```
            -- deq signals

            if ( (state = init
and QFempty = '0' and dn ='1'
                  and lb = '0'
and fifoIempty = '0' and socI =
'1' and receiverGrant = '1'
                  and
onePacketFromI = '1')
                 or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'1'
                   and
onePacketFromI = '1')
                 or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'0')
                 or (state =
init and QFempty = '0' and lb =
'0' and fifoIempty = '0' and
socI = '0')
                 or (state =
init and dn = '1' and QFempty =
'1' and fifoIempty = '0')

                 or (state = i0
and fifoIempty = '0' and socI =
'0')
                 or (state = i1
and fifoIempty = '0' and socI =
'0')
                 or (state = i2
and fifoIempty = '0' and socI =
'0')
                 or (state = i3
and fifoIempty = '0' and socI =
'0')
                 or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI = '0')

                 ) then

              deqI <= '1';

            else

              deqI <= '0';
            end if;

            -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant, wordCount;

        end process
deqIConsultI;

 ---------------------------
          deqLConsultL: process
(state, QFempty, dn, lb,
fifoLempty, socL,
receiverGrant, wordCount,
onePacketFromL)

          begin


            if ( (state = init
and QFempty = '0' and dn ='1'
                  and lb = '1'
and fifoLempty = '0'
                  and socL =
'1' and receiverGrant = '1' and
onePacketFromL = '1')
                 or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'1'
                    and
onePacketFromL = '1')
                 or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'0')  --2
                 or (state =
init and QFempty = '0' and lb =
'1' and fifoLempty = '0' and
socL = '0')

                 or (state = l0
and fifoLempty = '0' and socL =
'1')
                 or (state = l0
and fifoLempty = '0' and socL =
'0')
                 or (state = l1
and fifoLempty = '0' and socL =
'1')
                 or (state = l1
and fifoLempty = '0' and socL =
'0')
                 or (state = l2
and fifoLempty = '0' and socL =
'1')
                 or (state = l2
and fifoLempty = '0' and socL =
'0')
                 or (state = l3
and fifoLempty = '0' and socL =
'1')
```

```
                or (state = 13
and fifoLempty = '0' and socL =
'0')
                or (state = 14
and receiverGrant = '1' and
wordCount > 0 and fifoLempty =
'0' and socL = '1')
                or (state = 14
and receiverGrant = '1' and
wordCount > 0 and fifoLempty =
'0' and socL = '0')


                ) then

              consultFifoL <=
'1';



            else

              consultFifoL <=
'0';
            end if;

            -- deqL circuit


            if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '1'
and fifoLempty = '0' and socL =
'1' and receiverGrant = '1')
                or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'1')
                or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'0')  --2
                or (state =
init and QFempty = '0' and lb =
'1' and fifoLempty = '0' and
socL = '0')  -- 3

                or (state = 10
and fifoLempty = '0' and socL =
'0')
                or (state = 11
and fifoLempty = '0' and socL =
'0')
                or (state = 12
and fifoLempty = '0' and socL =
'0')
```

```
                or (state = 13
and fifoLempty = '0' and socL =
'0')
                or (state = 14
and receiverGrant = '1' and
wordCount > 0 and fifoLempty =
'0' and socL = '0')


                ) then

              deqL <= '1';

            else

              deqL <= '0';
            end if;

            -- wait on state,
QFempty, dn, lb, fifoLempty,
socL, receiverGrant, wordCount;

          end process
deqLConsultL;

          -- enqR circuit

        enqueueR: process
(state, receiverGrant)
        begin

            if (state = last
and receiverGrant = '1'and
QRfull = '0' ) then
              enqR <= '1';
            else
              enqR <= '0';
            end if;
            -- wait on state,
receiverGrant;
        end process enqueueR;

 --------------------------


        ConsultFdeqF: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant,
fifoLempty, socL)
          begin

            if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '0'
and fifoIempty = '0' and socI =
'1' and receiverGrant = '1')
```

```
                or (state =
init and QFempty = '0' and dn
='1'
                    and lb =
'1' and fifoLempty = '0' and
socL = '1' and receiverGrant =
'1')

                ) then
              consultQF <= '1';
deqF <= '1';

            else

              consultQF <= '0';
deqF <= '0';
            end if;
            -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
            --
fifoLempty, socL;

        end process
ConsultFdeqF;

-------------------------
  -- dequeue needed circuit

        deqNeed: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, fifoLempty,
socL,

onePacketFromI, onePacketFromL)

        begin

            if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '0'
and fifoIempty = '0' and socI =
'1' and receiverGrant = '1' and

onePacketFromI = '1')
                or (state =
init and QFempty = '0' and
                    dn ='1'
and lb = '1' and fifoLempty =
'0' and socL = '1' and
receiverGrant = '1' and

onePacketFromL = '1')


                ) then
              dn_c <= '0';

            elsif(state = last
and receiverGrant = '1' and
QRfull = '0') then
              dn_c <= '1';

            else

              dn_c <= dn;
            end if;
            -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
fifoLempty, socL;

        end process deqNeed;

---------------------------

          -- loop back circuit,
lb
          loopBack: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, fifoLempty,

socL, onePacketFromI,
onePacketFromL)

          begin

            if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '0'
and fifoIempty = '0' and socI =
'1' and receiverGrant = '1'
                and
onePacketFromI = '1')
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'1'
                    and
onePacketFromI = '1')
                or (state =
init and dn = '1' and QFempty =
'1' and fifoIempty = '1') --
dual
                or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'0')
                or (state =
init and QFempty = '0' and lb =
```

```
'0' and fifoIempty = '0' and
socI = '0')
                  or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '1')
                  or (state =
init and QFempty = '0' and lb =
'0' and fifoIempty = '1')

                  -- or (state =
init and dn = '1' and QFempty =
'1' and
                  --
fifoIempty = '0')

                  ) then

              lb_c <= '1';

            elsif( (state =
init and QFempty = '0' and dn
='1'
                   and lb =
'1' and fifoLempty = '0' and
socL = '1' and receiverGrant =
'1'
                   and
onePacketFromL = '1')  -- new
added, may 15 2000
                  or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'1'
                      and
onePacketFromL = '1')  -- neww
added may 15 2000


                  or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'0') --2
                  or (state =
init and QFempty = '0' and lb =
'1' and fifoLempty = '0' and
socL = '0') -- 3
                  or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '1')
                  or (state =
init and QFempty = '0' and lb =
'1' and fifoLempty = '1')
                  or (state =
init and dn = '1' and QFempty =
'1' and fifoIempty = '0')
                  --
correction carried out
                  ) then

              lb_c <= '0';

            else

              lb_c <= lb_c;
            end if;

            -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
fifoLempty, socL;

          end process loopBack;

-------------------------

          -- word counter
        wordCounter: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, fifoLempty,
socL, wordCount)

        begin


            if ( (state = init
and QFempty = '0' and dn ='1'
and lb = '0'
                  and
fifoIempty = '0' and socI = '1'
and receiverGrant = '1')
                 or (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI =
'1')

                  or (state =
init and QFempty = '0' and dn
='1' and
                      lb = '1'
and fifoLempty = '0' and socL =
'1' and receiverGrant = '1')
                or (state = init
and dn = '0' and lb = '1' and
fifoLempty = '0' and socL =
'1')

                  ) then

--            wordCount_c <= 2;
              wordCount_c <= 1;
      -- changed to 1
```

```
            elsif( (state = i3
and fifoIempty = '0' and socI=
'0')
                    or (state =
l3 and fifoLempty = '0' and
socL = '0')
                )        then

                wordCount_c <=
wordCount - 1;

            else   -- no change

                wordCount_c <=
wordCount;
            end if;

            -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant,
fifoLempty, socL, wordCount;

        end process
wordCounter;

---------------------------


        -- address circuit
        adrsCircuit: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, QFoutput,
fifoLempty, socL, address, e)

            variable temp :
natural;  -- temporary variable
            variable elem :
bit_vector(4 - 1 downto 0);
        begin


            if ( (state = init
and QFempty = '0' and dn ='1'
                    and lb = '0'
and fifoIempty = '0' and socI=
'1' and receiverGrant = '1')
                  or (state =
init and QFempty = '0' and dn
='1'
                      and lb =
'1' and fifoLempty = '0' and
socL = '1' and receiverGrant =
'1')

                  )then

--
adrs_c <= QFoutput;
                e_c <= QFoutput;
                -- typee
conversion for adrs_c

                temp :=
bits2natural(QFoutput);
                natural2bits
(temp, elem);
                adrs_c <= elem;

             elsif( (state =
init and dn = '0' and lb = '0'
and fifoIempty = '0' and socI=
'1')

                    or (state =
init and dn = '0' and lb = '1'
and fifoLempty = '0' and socL =
'1')
                    ) then

--              adrs_c <= e   ;

                -- assigning
adrs_c <= e

                temp :=
bits2natural(e);
                natural2bits
(temp, elem);
                adrs_c <= elem;


            elsif( (state = i2
and fifoIempty = '0' and socI=
'0')
                  or (state =
l2 and fifoLempty = '0' and
socL = '0')

                  ) then

             -- compute a
memory address from pointer e

                if (
bits2natural(address) =
bits2natural(e) ) then  --
first address
                  temp :=
bits2natural(e) * (1 + 1) + 0;
-- firstPacketAdrs : natural :=
0
```

```
                --
ATMlengthDiv4 = 2, and 1
location is preserved
                -- for fabric
and out controller headers

                temp := temp +
1;  -- increment address
                natural2bits
(temp, elem);
                adrs_c <= elem;

             else
      -- subsequent addresses

                adrs_c <=
increment(address);
             end if;


           else

             adrs_c <=
address;
           end if;

           -- wait on state,
QFempty, dn, lb, fifoIempty,
socI, receiverGrant, QFoutput,
fifoLempty, socL, address, e;

       end process
adrsCircuit;

-------------------------

       -- e register, i.e, a
temporary element
       tempRegister: process
(state, QFempty, dn, lb,
fifoIempty, socI,
receiverGrant, QFoutput, e)

       begin

           if ( (state = init
and QFempty = '0' and dn ='1'
                 and lb = '0'
and fifoIempty = '0' and socI=
'1' and receiverGrant = '1')
                or (state =
init and QFempty = '0' and dn
='1'
                    and lb =
'1' and fifoLempty = '0' and
socL = '1' and receiverGrant =
'1')
                    ) then
             e_c <= QFoutput;

           else

             e_c <= e;
           end if;

           -- wait on state,
QFempty, dn, lb,  fifoIempty,
socI, receiverGrant, QFoutput,
e;

       end process
tempRegister;

-------------------------
     -- register inputs

       reg: process (state,
fifoIempty, socI,
receiverGrant, wordCount,
fifoLempty, socL, r0, r1, r2,
r3, fifoIoutput, fifoLoutput)

       begin
           if( (state = i0 and
fifoIempty = '0' and socI= '0')
              or (state = i4
and receiverGrant = '1' and
wordCount > 0 and fifoIempty =
'0' and socI= '0')
              ) then

             r0_c <=
fifoIoutput(0);  -- forget the
soc bit

           elsif( (state = l0
and fifoLempty = '0' and socL =
'0')
                 or (state =
l4 and receiverGrant = '1' and
wordCount > 0 and fifoLempty =
'0' and socL = '0')

                 ) then

             r0_c <=
fifoLoutput(0);


           else r0_c <= r0;

           end if;
```

```
          if(state = i1 and
fifoIempty = '0' and socI= '0')
then

            r1_c <=
fifoIoutput(0);

          elsif(state = l1 and
fifoLempty = '0' and socL =
'0') then

            r1_c <=
fifoLoutput(0);

          else    r1_c <= r1;

          end if;



          if(state = i2 and
fifoIempty = '0' and socI= '0')
then
            r2_c <=
fifoIoutput(0);

          elsif (state = l2 and
fifoLempty = '0' and socL =
'0')
          then
            r2_c <=
fifoLoutput(0);


          else  r2_c <= r2;

          end if;

          if(state = i3 and
fifoIempty = '0' and socI= '0')
then
          r3_c <=
fifoIoutput(0);

          elsif(state = l3 and
fifoLempty = '0' and socL =
'0') then
            r3_c <=
fifoLoutput(0);



          else r3_c <= r3;

          end if;
```

```
          -- wait on state,
fifoIempty, socI,
receiverGrant, wordCount,
fifoLempty,
          --          socL, r0,
r1, r2, r3, fifoIoutput,
fifoLoutput;
        end process reg;

 -------------------------

        rw : process
(state,QFempty,dn,lb,
fifoIempty, socI,
                    fifoLempty,
socL, wordCount)

        begin

            if (
              (state = init and
QFempty = '0' and dn ='1'
               and lb = '0' and
fifoIempty = '0' and socI= '1')
              or (state = init
and QFempty = '0' and dn ='1'
                  and lb = '1'
and fifoLempty = '0' and socL =
'1')
              or (state = i4
and
                  (wordCount =
0 or
                   (fifoIempty
= '0' and socI= '0')))
              or (state = l4
and
                  (wordCount =
0 or
                   (fifoLempty
= '0' and socL = '0')))
              or (state = last)
) then


              receiverWillReq
<= '1';

            else

              receiverWillReq
<= '0';


            end if;
```

```
              -- wait on
state,QFempty,dn,lb,
fifoIempty, socI,
            --  fifoLempty,
socL,wordCount;

        end process rw;

-------------------------

        sequentials: process
        begin

            wait until clock =
'1';

            if (reset = '1')
then

              lb <= '0'; dn <=
'1'; state <= init; wordCount
<= 0 ;
--    socI <= '0';

            else

              state <= state_c;
              wordCount <=
wordCount_c;
              lb <= lb_c;
              dn <= dn_c;
              e <= e_c;
              address <=
adrs_c;
              r0 <= r0_c; r1 <=
r1_c; r2 <= r2_c; r3 <= r3_c;

            end if;

        end process
sequentials;

-----------------------------

        memAdrsRCVR <= adrs_c;

        writeEnableRCVR <= '1'
      when ( (state = i4 and
receiverGrant = '1'

and wordCount > 0

      and fifoIempty = '0' and
socI = '0')

or (state = i4 and
receiverGrant = '1' and
wordCount = 0)

or (state = l4 and
receiverGrant = '1'

         and wordCount > 0 and
fifoLempty = '0'

         and socL = '0')

or (state = l4 and
receiverGrant = '1' and
wordCount = 0))
                        else
'0';

        end stateMachine;

configuration config_receiver
of receiver is
  for stateMachine  -- the
architecture
  end for;
end config_receiver;
```

C2. The dispatcher

```
use work.atmDataTypes.all;

entity dispPart is

  port(processorGrantD,
QRempty,QP1full, QP2full,
clock, reset : in bit;
       QRoutput_c : in
bit_vector(2 - 1 downto 0);
       memOutput : in
bit_vector(4 - 1 downto 0);
       QRoutputEnable, deqR,
enqP1, enqP2 : out bit;
       memAdrsDISP:  out
bit_vector(4 - 1 downto 0);
       memInput_c : out
bit_vector(4 - 1 downto 0);
       writeEnableDISP : out
bit;
       QP1input, QP2input : out
bit_vector(2 - 1 downto 0);
freePrio: in bit
       );
```

```
end dispPart;


architecture stateMachine of
dispPart is


  type dispStates is (d0, d1,
d2, d3, d4);

  -- combinational circuit
signals

  subtype wordType is
bit_vector(4 - 1 downto 0);

  -- a function that provides
new routing header

  function newHeader  (signal
oldHeader : wordType) return
wordType is
    variable newHead :
wordType;
  begin  -- a simple function
for now.

    newHead := not(oldHeader);

    return newHead;
  end newHeader;


  function fabricHeader (signal
oldFabHead : wordType) return
wordType is
    variable fabHead :
wordType;
  begin  -- a simple function
for now.

    -- fabHead := not
oldFabHead;
    fabHead := oldFabHead ;

    return fabHead;
  end fabricHeader;


  function headerPriority
(signal header : wordType;
signal freePrio : bit) return
bit is
    variable prio : bit;
  begin
-- A simple example. Don't
change the priority.
    prio := header(0);


    return prio;

  end headerPriority;


  subtype vector is
bit_vector(4 - 1 downto 0);

  signal state_c : dispStates ;
-- combinational
  signal state : dispStates;
-- regiters


  signal address, adrs_c :
bit_vector(4 - 1 downto 0);

  signal e, e_c  : bit_vector(2
- 1 downto 0);  -- queues
output

  signal header : wordType; --
header is embedded in the
memory word
  signal header_c : wordType;


-- signals for latchings



begin  -- the state machine


  combinationalDisp: process
(QRempty, processorGrantD,
header, state, e, QP1full,
QP2full)

  begin
    case state is

      when d0 =>
        -- if (processorGrantD
= '0' and QRempty = '0') then

        if (processorGrantD =
'0' or QRempty = '1') then
          state_c <= d0;

--     elsif (processorGrantD =
'1' and QRempty = '0' ) then
```

```
        else

          state_c <= d1;

          -- output signals are
given in another  processes,
below

        end if;


      when d1 =>
        if (processorGrantD =
'0') then
          state_c <= d1;

        else -- processorGrantD
= '1'
          state_c <= d2;

        end if;


      when d2 =>
        if (processorGrantD =
'0') then

          state_c <= d2;
        else -- processorGrantD
= '1'
          state_c <= d3;

                  -- write to
memory data bus
          memInput_c <=
newHeader(header);
          -- note: newheader ->
memInput_c -> memory_c ->
written with clock

        end if;

      when d3 =>
        if (processorGrantD =
'0') then
          state_c <= d3;

        else -- processorGrantD
= '1'
                  state_c <=
d4;

                  memInput_c <=
fabricHeader(header) ; -- write
the new fabric header
        end if;
```

```
      when d4 =>

        if(processorGrantD ='1'
and  headerPriority(header,
freePrio) = '1' and QP1full =
'0') then
          state_c <= d0;
          QP1input <= e;

        elsif(processorGrantD
='1' and headerPriority(header,
freePrio) = '0' and QP2full =
'0') then
          state_c <= d0;
          QP2input <= e;

        else

          state_c <= d4;
        end if;
    end case;

  end process
combinationalDisp;



  memoryAdrs: process (address,
state, processorGrantD)

  begin

    if ( (state = d1 or state =
d2) and processorGrantD = '1')
then
      memAdrsDISP <=
increment(address);
    else

      memAdrsDISP <= address;
    end if;

  end process memoryAdrs;


  -- values for address
(register) that goes through
queues


  Qelements: process(state,
processorGrantD, QRempty,
address, QRoutput_c, e)

    variable temp : natural;  -
- temporary variable
```

```
    variable elem :
bit_vector(4 - 1 downto 0);
-- memory address

  begin
    if (state = d0 and
processorGrantD = '1' and
QRempty = '0') then

      e_c <= QRoutput_c;       -
- a copy of the address

    temp :=
bits2natural(QRoutput_c) * (2 +
1) + 0;

      natural2bits (temp,
elem);
      adrs_c <= elem;

    else  -- no change

      adrs_c <= address;
      e_c <= e;

    end if;
  end process Qelements;



    -- header circuit:
combinational part

  headerCircuit: process
(state, processorGrantD,
header, memOutput)

  begin
    if (state = d1 and
processorGrantD = '1') then
      header_c <= memOutput;
    else
      header_c <= header;
    end if;

  end process headerCircuit;



  sequential : process


  begin

    wait until clock = '1' ;

    if (reset = '1') then


      state <= d0;

    else

      state <= state_c;

      address <= adrs_c;
      header <= header_c;
      e <= e_c;

    end if;


  end process;



  writeEnableDISP <= '1' when
((state = d2 or state = d3)

and processorGrantD = '1')
                    else ' 0';

  deqR <= '1' when (state = d0
and  processorGrantD = '1' and
QRempty = '0')
          else '0';


  QRoutputEnable <= '1'when
(state = d0 and
processorGrantD = '1' and
QRempty = '0')
                     else '0';


  enqP1 <= '1' when (state = d4
and processorGrantD ='1' and
headerPriority (header,
freePrio) = '1' and QP1full =
'0')

           else '0';


  enqP2 <= '1' when (state = d4
and processorGrantD ='1' and
headerPriority(header,
freePrio) = '0' and QP2full =
'0')
           else '0';
```

```
end stateMachine;


configuration config_dispPart
of dispPart is
  for stateMachine  -- the
architecture
  end for;
end config_dispPart;
```

C3. The scheduler

```
---------------------------
-- Scheduler module.
---------------------------
written by: M.Sadegh Jahanpour
-- Feb, 99

use work.atmDataTypes.all;

entity schlPart is

  port(processorGrantS,
QP1empty, QP2empty,
       QTfull, clock, reset :
in bit;
       QP1output_c, QP2output_c
: in bit_vector(2 - 1 downto
0);
       QP1outputEnable, deqP1,
QP2outputEnable,
       deqP2, enqT : out bit;
       QTinput : out
bit_vector(2 - 1 downto 0)
       );

end schlPart;


architecture stateMachine of
schlPart is



  type schlStates is (s0, s1);

  -- combinational circuit
signals


  signal stateSchl_c :
schlStates;


  signal stateSchl :
schlStates;


  signal e2, e2_c :
bit_vector(2 - 1 downto 0);  --
queues output


begin  -- the state machine


  combinationalSchl: process
(processorGrantS,QP1empty,QP2em
pty, stateSchl, e2, QTfull)

  begin
    case stateSchl is
      when s0 =>

        if(processorGrantS ='1'
and QP1empty = '0' and QTfull =
'0') then
          stateSchl_c <= s1;


        elsif(processorGrantS
='1' and QP1empty = '1' and
QP2empty = '0' and QTfull =
'0') then
          stateSchl_c <= s1;


        else

          stateSchl_c <= s0;
        end if;

      when s1 =>
        if (processorGrantS =
'0' or QTfull = '1') then
          stateSchl_c <= s1;

        else

          QTinput <= e2;
          stateSchl_c <= s0;

        end if;

    end case;


  end process
combinationalSchl;
```

```
  Qelements2:
process(stateSchl,
processorGrantS, QP1empty,
QP2empty,

QP1output_c, QP2output_c , e2,
QTfull)
  begin

    if (stateSchl = s0 and
processorGrantS ='1' and
QP1empty = '0' and QTfull =
'0') then


      e2_c <= QP1output_c ;

       elsif (stateSchl = s0
and processorGrantS ='1' and
QP1empty = '1'
                and QP2empty =
'0' and QTfull = '0') then

           e2_c <= QP2output_c;

       else

           e2_c <= e2;

       end if;

    end process Qelements2;



      seq2: process
      begin
      wait until clock = '1' ;

        if (reset = '1') then


          stateSchl <= s0;
          e2 <= "00";

        else

          stateSchl <=
stateSchl_c;
          e2 <= e2_c;
        end if;
```

```
      end process;


      deqP1 <= '1' when
(stateSchl = s0 and
processorGrantS ='1' and
QP1empty = '0' and QTfull =
'0')
               else '0';


      QP1outputEnable <= '1'
when (stateSchl = s0 and
processorGrantS ='1' and
QP1empty = '0' and QTfull =
'0')
                           else
'0';


      QP2outputEnable <= '1'
when ( stateSchl = s0 and
processorGrantS ='1' and
QP1empty = '1' and QP2empty =
'0' and QTfull = '0')

                           else
'0';

      deqP2 <= '1' when (
stateSchl = s0 and
processorGrantS ='1' and
QP1empty = '1' and QP2empty =
'0' and QTfull = '0')

               else '0';

      enqT <= '1' when (
stateSchl = s1 and
processorGrantS ='1' and QTfull
= '0')

              else '0';


    end stateMachine;



configuration config_schlPart
of schlPart is
  for stateMachine  -- the
architecture
  end for;
end config_schlPart;
```

C4. The transmitter

```
-----------------------
-- transmitter module.
-----------------------

-- written by: M.Sadegh
Jahanpour
-- Feb. 99


-- data path is 4 bits wide:
-- one bit for actbit, one bit
for hi/lo priority,
-- and two bits to select
either of 4 out ports.
-- QToutput is two bits wide.
-- memory address is 4 bits
wide.
-- memory word is 8 bits wide.
-- address pointer 0 generates
memory address 0,
-- address pointer 1 generates
memory address 3.
-- internal registers regT0,
regT1, regT2 and regT3 are two
bits wide.
-- Aug. 99


library IEEE;
use IEEE.std_logic_arith.all;
use work.atmDataTypes.all;

entity transmitter4 is

  port(frameStart, QTempty,
ackIn, clock, reset : in bit;
       QToutput_c: in
bit_vector(2 -1 downto 0);
       memOutput : in
bit_vector(16 -1 downto 0 );
       ConsultQT, deqT, enqF :
out bit;
       QFinput : out
bit_vector(2 -1 downto 0);
       address_c : out
bit_vector(4 -1 downto 0);
       transmitterWillReq : out
bit;
              -- dataOut : out
bit_vector(2 -1 downto 0)
       dataOut : out
bit_vector(4 -1 downto 0)
       );
end transmitter4;

architecture stateMachine of
transmitter4 is

  -- combinational circuit
signals
  type transmitterStates is
(t0, t1, t2, t3, t4, t5, t6,
t7, t8, t9, t10,
                          t11,
t12, t13, t14, t15, t16);
  subtype adrsType is
bit_vector(4 - 1 downto 0);  --
concrete memory addresses


  signal state_c :
transmitterStates ;  --
combinational
  signal state :
transmitterStates;    --
registers
  signal address : adrsType;
  signal adrs_c : adrsType; --
extra wiring for address
signals


  signal e, e_c : bit_vector(2
-1 downto 0);  -- Queues
outputs

  signal wordCount : natural;
  signal wordCount_c : natural;
  signal retransCount :
natural;  -- # of
retransmissions, tried so far
  signal retransCount_c :
natural;

   -- T registers
  signal regT0: bit_vector(4 -1
downto 0);
  signal regT1: bit_vector(4 -1
downto 0);
  signal regT2: bit_vector(4 -1
downto 0);
  signal regT3: bit_vector(4 -1
downto 0);


   -- inputs of registers
```

```
  signal regT0_c: bit_vector(4
-1 downto 0);
  signal regT1_c: bit_vector(4
-1 downto 0);
  signal regT2_c: bit_vector(4
-1 downto 0);
  signal regT3_c: bit_vector(4
-1 downto 0);


begin  -- the state machine

  combinational: process
(state, address, QToutput_c,
ackIn, frameStart, QTempty,
memOutput, regT0, regT1, regT2,
regT3, wordCount)

  begin
    dataOut <= "0000";

    case state is

      when t0 =>
        if (frameStart = '0' or
QTempty = '1') then

          state_c <= t0;

        elsif (frameStart = '1'
and QTempty = '0' ) then

          state_c <= t1;

        else  -- no change
          state_c <= t0;

        end if;

      when t1 =>

        state_c <= t2;

      when t2 =>
        regT0_c <= memOutput(3
downto 0);
        regT1_c <= memOutput(7
downto 4);
        regT2_c <= memOutput(11
downto 8);
        regT3_c <= memOutput(15
downto 12);

        state_c <= t3;


      when t3 =>

        state_c <= t4;

      when t4 =>

        state_c <= t5;

      when t5 =>

        dataOut <= regT2;
state_c <= t6;

      when t6 =>

        dataOut <= regT3;

        regT0_c <= memOutput(3
downto 0);
        regT1_c <= memOutput(7
downto 4);
        regT2_c <= memOutput(11
downto 8);
        regT3_c <= memOutput(15
downto 12);


        state_c <= t7;

      when t7 =>

        dataOut <= regT0 ;
        state_c <= t8;

      when t8 =>

        if (ackIn = '0') then
          dataOut <= "0000";

          state_cate_c <= t16;
        else
          dataOut <= regT1;
          state_c <= t9;
        end if;

      when t9 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        else

          dataOut <= regT2 ;
state_c <= t10;
        end if;
```

```
      when t10 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        else

          dataOut <=  regT3 ;

          regT0_c <=
memOutput(3 downto 0);
          regT1_c <=
memOutput(7 downto 4);
          regT2_c <=
memOutput(11 downto 8);
          regT3_c <=
memOutput(15 downto 12);


          state_c <= t11;
        end if;

      when t11 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        else
          dataOut <= regT0;
          state_c <= t12;
        end if;

      when t12 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        else

          dataOut <= regT1 ;
          state_c <= t13;
        end if;

      when t13 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        else
          dataOut <= regT2 ;
state_c <= t14;
        end if;

      when t14 =>

        if (ackIn = '0') then
          -- dataOut(0) <= '0';
dataOut(1) <= '0';
          dataOut <= "0000";
          state_c <= t16;
        elsif (wordCount > 0
and ackIn = '1' ) then

          dataOut <= regT3;

          regT0_c <=
memOutput(3 downto 0);
          regT1_c <=
memOutput(7 downto 4);
          regT2_c <=
memOutput(11 downto 8);
          regT3_c <=
memOutput(15 downto 12);


          state_c <= t11;

        elsif ( wordCount = 0
and ackIn = '1' ) then

          dataOut <= regT3 ;
          state_c <= t15;

        else

          state_c <= t14;  --
no change

        end if;


      when t15 =>

        -- dataOut(0) <= '0';
dataOut(1) <= '0';
        dataOut <= "0000";
        QFinput <= e;
        state_c <= t0;

      when t16 =>

        if (retransCount = 0)
then

          state_c <= t15;
```

```
          dataOut <= "0000";
        else
          dataOut <= "0000";
          state_c <= t0;

        end if;

    end case;

  end process combinational;


  adrsCircuit: process (state,
wordCount, ackIn, QToutput_c,
address, retransCount, e)
    variable temp : natural;  -
- temporary variable
    variable elem :
bit_vector(4 - 1 downto 0);  --
memory address

  begin

    if (state = t1 or (state =
t14 and wordCount = 0 and ackIn
= '1')
        or (state = t16 and
retransCount = 0 )   ) then

      e_cc <= QToutput_c;

      temp :=
bits2natural(QToutput_c) * (2 +
1) + 0;  -- firstPacketAdrs :
natural := 0
      -- ATMlengthDiv4 = 2, and
1 location is preserved
      -- for fabric and out
controller headers

      natural2bits (temp,
elem);
      adrs_c <= elem;



    elsif (state = t3 or state
= t7 or (state = t11 and ackIn
= '1'))
    then
      adrs_c <=
increment(address);

    else

      adrs_c <= address ;
      e_c <= e;

    end if;

  end process adrsCircuit;


  -- a process to count the
number of transmitted words

  wordCountCircuit : process
(state, ackIn, wordCount)

  begin

    if (state = t4    ) then
      wordCount_c <= 2;
      -- ATMlengthDiv4

    elsif((state = t8 and ackIn
= '1') or (state = t12 and
ackIn = '1') )then
      wordCount_c <= wordCount
- 1;

    else
      wordCount_c <= wordCount;
    end if;

  end process wordCountCircuit
;


  --  a process for the
retransmission circuit

  retransmissionCounter:
process (state, retransCount)

      begin

      if (state = t15) then
        retransCount_c <= 4 ;


      elsif(state = t16 and
retransCount > 0) then
        retransCount_c <=
retransCount - 1;

      else

        retransCount_c <=
retransCount;

      end if;
```

```
  end process
retransmissionCounter;


          -- to generate pulse
signals, i,e, to pull down
signals to zero
          -- after being set
to one, we specify an extra
process

  pulseGeneration: process
(state, wordCount, ackIn,
retransCount)

  begin
    if ( ( state = t14 and
wordCount = 0 and ackIn = '1' )
or (
      state = t16 and
retransCount = 0) ) then
      deqT <= '1';
    else

      deqT <= '0';
    end if;
    if (state = t15) then  enqF
<= '1';
    else  enqF <= '0';
    end if;

    if (state = t1 or (state =
t14 and wordCount = 0 and ackIn
= '1')
        or (state = t16 and
retransCount = 0 ) )then
      ConsultQT <= '1';

    else
      ConsultQT <= '0';
    end if;

  end process pulseGeneration;


  tw : process (state, ackIn,
retransCount)

  begin

    if ( (state = t2 or state =
t6 or state = t15)
         or
         ( (state = t10 or
state = t14) and ackIn = '1')
         or (state = t16 and
retransCount = 0) ) then
      transmitterWillReq <=
'1';
    else transmitterWillReq <=
'0';

    end if;

  end process tw;


  sequential: process

  begin

    wait until clock = '1' ;

    if (reset = '1') then
      state <= t0; wordCount <=
2 ; retransCount <= 4;

    else


      state <= state_c;

      regT0 <= regT0_c;
      regT1 <= regT1_c;
      regT2 <= regT2_c;
      regT3 <= regT3_c;

      wordCount <= wordCount_c;
      address <= adrs_c;
      retransCount <=
retransCount_c;
      e <= e_c;

    end if;


  end process sequential;

  address_c <= adrs_c;
end stateMachine;



configuration
config_transmitter4 of
transmitter4 is
  for stateMachine  -- the
architecture
  end for;
end config_transmitter4;
```

C5. The arbiter

```
---------------------
-- Arbiter module.
---------------------

-- written by: M.Sadegh
Jahanpour, Jan, 99.

-- Arbitrates the memory
accesses among the receiver,
the dispatcher, the scheduler,
and the transmitter.



entity arbiter is

  port(transmitterWillReq: in
bit;
       receiverWillReq : in
bit;
       reset : in bit;
       clock : in bit;
       receiverGrant: out bit;
       processorGrant: out
bit);

end arbiter;


architecture stateMachine of
arbiter is

  -- combinational circuit
signals
  type arbiterStates is (a0,
a1, a2, a3);
  signal state_c :
arbiterStates ;  --
combinational
  signal state : arbiterStates;
-- regiters



begin  -- the state machine



  cominational : process
(state, receiverWillReq,
transmitterWillReq)

  begin
    case state is

      when a0 =>
        if (transmitterWillReq
= '1') then

          state_c <= a0;


        elsif
(transmitterWillReq = '0' and
receiverWillReq = '0') then


          state_c <= a0;

        elsif
(transmitterWillReq = '0' and
receiverWillReq = '1') then


          state_c <= a1;

        else  -- no chang, ie,
self loop

          state_c <= state ;

        end if;


      when a1 =>

        state_c <= a2;


      when a2 =>

        state_c <= a3;


      when a3 =>

        state_c <= a0;

    end case;

  end process;


  outputs: process (state,
transmitterWillReq,
receiverWillReq)

  begin
    if (state = a0 and
transmitterWillReq = '0' and
receiverWillReq = '1') then
```

```
      receiverGrant <= '1';

    else
      receiverGrant <= '0';
    end if;

    if ( (state = a0 and
transmitterWillReq = '0' and
receiverWillReq = '0')
         or ( (state = a1 or
state = a2 or state = a3) and
transmitterWillReq = '0')
         )

    then
      processorGrant <= '1';

    else
      processorGrant <= '0';
    end if;

  end process outputs;

  sequential : process


  begin

    -- output signals are
registered
    wait until clock = '1' ;

    if (reset = '1') then

      state <= a0;

    else


      -- state variables

      state <= state_c ;

    end if;

  end process;

end stateMachine;


configuration config_arbiter of
arbiter is
  for stateMachine  -- the
architecture
  end for;
end config_arbiter;
```

C6. The fabric

```
---------------------
-- fabric module
---------------------

-- This model has four input
ports of 8 bits each.

use work.arbt;

entity fab4b4p is
  -- generic(PORTWIDTH :
positive := 4; PORTNUM :
positive := 4);
  port (dIn0, dIn1, dIn2, dIn3:
bit_vector(4 - 1 downto 0);  --
Data inputs for all ports
        ackIn0, ackIn1, ackIn2,
ackIn3 : in bit ; --
Acknowledge In signals

        frameStart: in bit;
        clock: in bit;
        reset : in bit;
        dOut0, dOut1, dOut2,
dOut3 : out bit_vector(4 - 1
downto 0);  --Data outputs for
all ports
        ackOut0, ackOut1,
ackOut2, ackOut3 : out bit
        );

end fab4b4p;



architecture mix of fab4b4p is

  signal dTerm0, dTerm1,
dTerm2, dTerm3 : bit_vector(4 -
1 downto 0); -- Intermediate
data outputs;

  signal dOut0_c,
dOut1_c,dOut2_c,dOut3_c :
bit_vector(4 - 1 downto 0);
  signal ackOut0_c,
ackOut1_c,ackOut2_c,ackOut3_c :
bit;

  signal co0 : bit;           -
- Control signals for output
  signal co1 : bit;           -
- port 0 to 3. If coi is '1',
```

```
  signal co2 : bit;           -
- it means that output port i
  signal co3 : bit;           -
- is enable, otherwise it
disable


  signal    co0_c : bit;
      -- Control signals for
output
  signal    co1_c : bit;
      -- port 0 to 3. If coi is
'1',
  signal    co2_c : bit;
      -- it means that output
port i
  signal    co3_c : bit;
      -- is enable, otherwise
it disable


  signal ip0,     -- The input
port which destinated
    ip1,          -- to output
port j (i=0~3)
    ip2,          -- If ipi=j,
it means that input port
    ip3: bit_vector(1 downto
0);         -- j will transfer
to the output i if

      -- coi = 1.


  signal    req_nopri00, --
Non-priority cell transfer
request
    req_nopri01,  -- from put k
to output j (req_noprikj)
    req_nopri02,
    req_nopri03,
    req_nopri10,
    req_nopri11,
    req_nopri12,
    req_nopri13,
    req_nopri20,
    req_nopri21,
    req_nopri22,
    req_nopri23,
    req_nopri30,
    req_nopri31,
    req_nopri32,
    req_nopri33 : bit;

  signal    req_pri00,    --
Priority cell transfer request
    req_pri01,      -- from put
k to output j (req_prikj)
    req_pri02,
    req_pri03,
    req_pri10,
    req_pri11,
    req_pri12,
    req_pri13,
    req_pri20,
    req_pri21,
    req_pri22,
    req_pri23,
    req_pri30,
    req_pri31,
    req_pri32,
    req_pri33:  bit;

  signal    one_pri_for0,
      -- At least one priority
request
    one_pri_for1,       -- for
output port i (one_pri_fori)
    one_pri_for2,
    one_pri_for3:  bit;

  signal    one_nopri_for0,
      -- At least one non-
priority request
    one_nopri_for1,           -
- for output port i
(one_nopri_fori)
    one_nopri_for2,
    one_nopri_for3: bit;


  signal     state:
bit_vector(1 downto 0);   --
Timing state, it could be
      -- 2'b00 (RUN), 2'b01
(WAIT)
  -- 2'b11 (ROUTE)
  signal     state_c:
bit_vector(1 downto 0);

  signal    anyActive: bit;
      -- Anyactive for all the
input ports



  -- component declaration

  component arbt
    port(one_pri_fori : in bit;
         one_nopri_fori : in
bit;
```

```
        state : in
bit_vector(1 downto 0);
        req_pri0i : in bit;
        req_pri1i: in bit;
        req_pri2i: in bit;
        req_pri3i:in bit;
        req_nopri0i :in bit;
        req_nopri1i : in bit;
        req_nopri2i : in bit;
        req_nopri3i : in bit;
        clock : in bit;
        reset : in bit;
-- this is new
        ipi : out bit_vector(1
downto 0)
        );

  end component;

begin  -- the architecture

  anyActive <= ( dIn0(0) or
dIn1(0) or dIn2(0) or dIn3(0)
);

-- Important note: req-nopri= 1
means either low or high prio
requests. However, req-pri
-- indicated only low prio
requests.



  req_nopri00 <= dTerm0(0) and
(not dTerm0(2)) and (not
dTerm0(3));
  req_nopri01 <= dTerm0(0) and
(not dTerm0(2)) and dTerm0(3);
  req_nopri02 <= dTerm0(0) and
dTerm0(2) and (not dTerm0(3));
  req_nopri03 <= dTerm0(0) and
dTerm0(2) and dTerm0(3);


  req_nopri10 <= dTerm1(0) and
(not dTerm1(2)) and (not
dTerm1(3));
  req_nopri11 <= dTerm1(0) and
(not dTerm1(2)) and dTerm1(3);
  req_nopri12 <= dTerm1(0) and
dTerm1(2) and (not dTerm1(3));
  req_nopri13 <= dTerm1(0) and
dTerm1(2) and dTerm1(3);

  req_nopri20 <= dTerm2(0) and
(not dTerm2(2)) and (not
dTerm2(3));
  req_nopri21 <= dTerm2(0) and
(not dTerm2(2)) and dTerm2(3);
  req_nopri22 <= dTerm2(0) and
dTerm2(2) and (not dTerm2(3));
  req_nopri23 <= dTerm2(0) and
dTerm2(2) and dTerm2(3);

  req_nopri30 <= dTerm3(0) and
(not dTerm3(2)) and (not
dTerm3(3));
  req_nopri31 <= dTerm3(0) and
(not dTerm3(2)) and dTerm3(3);
  req_nopri32 <= dTerm3(0) and
dTerm3(2) and (not dTerm3(3));
  req_nopri33 <= dTerm3(0) and
dTerm3(2) and dTerm3(3);

  req_pri00 <= dTerm0(0) and
dTerm0(1) and (not dTerm0(2))
and (not dTerm0(3));
  req_pri01 <= dTerm0(0) and
dTerm0(1) and (not dTerm0(2))
and dTerm0(3);
  req_pri02 <= dTerm0(0) and
dTerm0(1) and dTerm0(2) and
(not dTerm0(3));
  req_pri03 <= dTerm0(0) and
dTerm0(1) and dTerm0(2) and
dTerm0(3);


  req_pri10 <= dTerm1(0) and
dTerm1(1) and (not dTerm1(2))
and (not dTerm1(3));
  req_pri11 <= dTerm1(0) and
dTerm1(1) and (not dTerm1(2))
and dTerm1(3);
  req_pri12 <= dTerm1(0) and
dTerm1(1) and dTerm1(2) and
(not dTerm1(3));
  req_pri13 <= dTerm1(0) and
dTerm1(1) and dTerm1(2) and
dTerm1(3);


  req_pri20 <= dTerm2(0) and
dTerm2(1) and (not dTerm2(2))
and (not dTerm2(3));
  req_pri21 <= dTerm2(0) and
dTerm2(1) and (not dTerm2(2))
and dTerm2(3);
  req_pri22 <= dTerm2(0) and
dTerm2(1) and dTerm2(2) and
(not dTerm2(3));
  req_pri23 <= dTerm2(0) and
dTerm2(1) and dTerm2(2) and
dTerm2(3);
```

```
  req_pri30 <= dTerm3(0) and
dTerm3(1) and (not dTerm3(2))
and (not dTerm3(3));
  req_pri31 <= dTerm3(0) and
dTerm3(1) and (not dTerm3(2))
and dTerm3(3);
  req_pri32 <= dTerm3(0) and
dTerm3(1) and dTerm3(2) and
(not dTerm3(3));
  req_pri33 <= dTerm3(0) and
dTerm3(1) and dTerm3(2) and
dTerm3(3);

--  Finite inite state machine
to control the timing of the
fabric

  control: process (state,
frameStart, anyActive)
  begin

    case state is

      when "00" =>

        if (frameStart = '1')
then

          state_c <= "01";

        else

          state_c <= "00";

        end if;



      when "01" =>


        if (frameStart= '0' and
anyActive = '1' ) then

          state_c <= "11";

        else

          state_c <= "01";
        end if;


      when "11"  =>
```

```
      if (frameStart = '0')
then

        state_c <= "00";

      else

        state_c <= "01";

      end   if;

      when "10" => state_c <=
state;  -- I have added for
completeness



    end case;
  end process control;

  ackOut0 <= '1' when (
    (ackIn0 = '1' and (co0 =
'1') and (ip0 = "00")) or
    ( ackIn1 = '1' and co1 =
'1'  and (ip1 = "00")) or
    (ackIn2 = '1'  and co2 =
'1'  and (ip2 = "00")) or
    (ackIn3 = '1' and co3 = '1'
and (ip3 = "00")) )
              else '0' ;
      -- my modification


  ackOut1 <= '1' when(
    (ackIn0 = '1' and (co0 =
'1') and (ip0 = "01")) or
    (ackIn1 = '1' and co1 = '1'
and (ip1 = "01")) or
    (ackIn2 = '1'  and co2 =
'1'  and (ip2 = "01")) or
    (ackIn3 = '1' and co3 = '1'
and (ip3 = "01"))
    )
             else  '0';

  ackOut2 <= '1' when(
    (ackIn0 = '1' and (co0 =
'1') and (ip0 = "10")) or
    (ackIn1 = '1' and co1 = '1'
and (ip1 = "10")) or
    (ackIn2 = '1'  and co2 =
'1'  and (ip2 = "10")) or
    (ackIn3 = '1' and co3 = '1'
and (ip3 = "10"))
```

```
    )
              else '0';

  ackOut3 <= '1' when(
    (ackIn0 = '1' and (co0 =
'1') and (ip0 = "11")) or
    (ackIn1 = '1' and co1 = '1'
and (ip1 = "11")) or
    (ackIn2 = '1'  and co2 =
'1'  and (ip2 = "11")) or
    (ackIn3 = '1' and co3 = '1'
and (ip3 = "11"))
    )
              else  '0';


  OutputData : process( co0,
co1, co2, co3, ip0, ip1, ip2,
ip3, dTerm0, dTerm1, dTerm2,
dTerm3)

  begin


-- outputs of the out-port 0;

    if (( co0 = '1') and (ip0 =
"00")) then
      dOut0_c <= dTerm0;
    elsif (( co0 = '1') and
(ip0 = "01")) then
      dOut0_c <= dTerm1;
    elsif (( co0 = '1') and
(ip0 = "10")) then
      dOut0_c <= dTerm2;
    elsif (( co0 = '1') and
(ip0 = "11")) then
      dOut0_c <= dTerm3;
    else  dOut0_c <=  "0000";

    end if;



    -- out port 1

    if (( co1 = '1') and (ip1 =
"00")) then
      dOut1_c <= dTerm0;
    elsif (( co1 = '1') and
(ip1 = "01")) then
      dOut1_c <= dTerm1;
    elsif (( co1 = '1') and
(ip1 = "10")) then
      dOut1_c <= dTerm2;
    elsif (( co1 = '1') and
(ip1 = "11")) then
      dOut1_c <= dTerm3;
    else  dOut1_c <=  "0000";

    end if;


    if (( co2 = '1') and (ip2 =
"00")) then
      dOut2_c <= dTerm0;
    elsif (( co2 = '1') and
(ip2 = "01")) then
      dOut2_c <= dTerm1;
    elsif (( co2 = '1') and
(ip2 = "10")) then
      dTerm1Out2_c <= dTerm2;
    elsif (( co2 = '1') and
(ip2 = "11")) then
      dOut2_c <= dTerm3;
    else  dOut2_c <=  "0000";

    end if;


    if (( co3 = '1') and (ip3 =
"00")) then
      dOut3_c <= dTerm0;
    elsif (( co3 = '1') and
(ip3 = "01")) then
      dOut3_c <= dTerm1;
    elsif (( co3 = '1') and
(ip3 = "10")) then
      dOut3_c <= dTerm3;
    elsif (( co3 = '1') and
(ip3 = "11")) then
      dOut3_c <= dTerm3;
    else  dOut3_c <=  "0000";

    end if;


  end process OutputData ;


  ctrlOutSignals:
process(state, req_nopri00,
req_nopri10,
req_nopri20,req_nopri30,
frameStart,

req_nopri01, req_nopri11,
req_nopri21,req_nopri31,
```

```
req_nopri02, req_nopri12,
req_nopri22,req_nopri32,

req_nopri03, req_nopri13,
req_nopri23,req_nopri33, co0,
co1, co2, co3)

  begin

    if (frameStart = '1') then
      co0_c <= '0';
      co1_c <= '0';
      co2_c <= '0';
      co3_c <= '0';

    elsif (state= "11")  then
      -- note: req-nopri
includes both pri and nopri.
But, req-pri
      -- includes  only pri
requests.

      co0_c <= req_nopri00 or
req_nopri10 or req_nopri20 or
req_nopri30;
      co1_c <= req_nopri01 or
req_nopri11 or req_nopri21 or
req_nopri31;
      co2_c <= req_nopri02 or
req_nopri12 or req_nopri22 or
req_nopri32;
      co3_c <= req_nopri03 or
req_nopri13 or req_nopri23 or
req_nopri33;



    else


      co0_c <= co0;
      co1_c <= co1;
      co2_c <= co2;
      co3_c <= co3;
    end if;

  end process ctrlOutSignals;


  one_nopri_for0 <= req_nopri00
or req_nopri10 or req_nopri20
or req_nopri30;
  one_nopri_for1 <= req_nopri01
or req_nopri11 or req_nopri21
or req_nopri31;
  one_nopri_for2 <= req_nopri02
or req_nopri12 or req_nopri22
or req_nopri32;
  one_nopri_for3 <= req_nopri03
or req_nopri13 or req_nopri23
or req_nopri33;

  one_pri_for0 <= req_pri00 or
req_pri10 or req_pri20 or
req_pri30;
  one_pri_for1 <= req_pri01 or
req_pri11 or req_pri21 or
req_pri31;
  one_pri_for2 <= req_pri02 or
req_pri12 or req_pri22 or
req_pri32;
  one_pri_for3 <= req_pri03 or
req_pri13 or req_pri23 or
req_pri33;


  arbt0: arbt
    port map (
      one_pri_for0,
      one_nopri_for0,
      state,
      req_pri00,
      req_pri10,
      req_pri20,
      req_pri30,
      req_nopri00,
      req_nopri10,
      req_nopri20,
      req_nopri30,
      clock,
      reset,
      ip0
      );

  arbt1: arbt port map  (
    one_pri_for1,
    one_nopri_for1,
    state,
    req_pri01,
    req_pri11,
    req_pri21,
    req_pri31,
    req_nopri01,
    req_nopri11,
    req_nopri21,
    req_nopri31,
    clock,
    reset,
    ip1
    );

  arbt2 : arbt port map(
```

```
    one_pri_for2,
    one_nopri_for2,
    state,
    req_pri02,
    req_pri12,
    req_pri22,
    req_pri32,
    req_nopri02,
    req_nopri12,
    req_nopri22,
    req_nopri32,
    clock,
    reset,
    ip2
    );

  arbt3 :   arbt port map(
    one_pri_for3,
    one_nopri_for3,
    state,
    req_pri03,
    req_pri13,
    req_pri23,
    req_pri33,
    req_nopri03,
    req_nopri13,
    req_nopri23,
    req_nopri33,
    clock,
    reset,
    ip3
    );



  sequential : process
  begin
    wait until clock = '1';

    if (reset = '1') then

      state <= "00";
      dTerm0 <= (others =>
'0');
      dTerm1 <= (others =>
'0');
      dTerm2 <= (others =>
'0');
      dTerm3 <= (others =>
'0');
    else
      dTerm0 <= dIn0; dTerm1 <=
dIn1;
      dTerm2 <= dIn2; dTerm3 <=
dIn3;

      state <= state_c;

      dOut0<= dOut0_c; dOut1 <=
dOut1_c;
      dOut2<= dOut2_c; dOut3<=
dOut3_c;

      co0 <= co0_c;
      co1 <= co1_c;
      co2 <= co2_c;
      co3 <= co3_c;
    end if;

  end process sequential ;

end mix;


configuration fabric_config of
fab4b4p is
  for mix
  end for;
end fabric_config;
```

C7. Submodule arbt of the fabric

```
-----------------------------
--  Arbittation module inside
the switch fabric

--  When there are several
inputs for one same destination
output port, it performs a
round-robin algorithm between
them.


-- This vhdl program is a vhdl
translation of the switch
-- fabric developped by
jianping lou in concordia
university in verilog. For more
information see the technical
report No. 401, september 97


entity      arbt is
  port(one_pri_fori : in bit;
       one_nopri_fori : in bit;
       state : in bit_vector(1
downto 0);
       req_pri0i : in bit;
       req_pri1i: in bit;
       req_pri2i: in bit;
       req_pri3i:in bit;
       req_nopri0i :in bit;
       req_nopri1i : in bit;
       req_nopri2i : in bit;
       req_nopri3i : in bit;
```

```
       clock : in bit;
       reset : in bit;       --
this is new
       ipi : out bit_vector(1
downto 0)
       );
end arbt;


architecture stateMachine of
arbt is

    -- ipi_i and ipi are
registers output. ipi_c is
combinational
    -- circuit.

  signal ipi_c, ipi_i :
bit_vector(1 downto 0);  --
internal connections


begin

  combinational: process(ipi_i,
state, req_pri1i, req_pri2i,

req_pri3i, req_pri0i,
one_nopri_fori,

req_nopri0i,
req_nopri1i,req_nopri2i,

req_nopri3i)


  begin

    if (state= "11" and
one_pri_fori = '1') then

      case ipi_i is

        when "00" =>

          if ( req_pri1i = '1')
then

            ipi_c <= "01";

          elsif ( req_pri2i =
'1') then

            ipi_c <= "10";

          elsif ( req_pri3i =
'1')then

            ipi_c <= "11";

          else

            ipi_c <= "00";
          end if;

        when "01" =>

          if (req_pri2i = '1')
then

            ipi_c <= "10";

          elsif (req_pri3i =
'1') then

            ipi_c <= "11";

            elsiff (req_pri0i =
'1') then

              ipi_c <= "00";

            else

              ipi_c <= "01";
            end if;

            when "10" =>

            if ( req_pri3i =
'1') then

              ipi_c <= "11";

            elsif ( req_pri0i =
'1') then

              ipi_c <= "00";

            elsif ( req_pri1i =
'1') then

              ipi_c <= "01";

            else

              ipi_c <= "10";

            end if;

            when "11" =>
```

```
            if ( req_pri0i =
'1') then
              ipi_c <= "00";
            elsif ( req_pri1i =
'1') then
              ipi_c <= "01";
            elsif ( req_pri2i =
'1') then
              ipi_c <= "10";
            else
              ipi_c <= "11";
            end if;
          end case;


        elsif ( state ="11" and
one_nopri_fori = '1') then

            case ipi_i is
              when      "00" =>
                if (
req_nopri1i = '1') then
                  ipi_c <=
"01";
                elsif (
req_nopri2i= '1') then
                  ipi_c <=
"10";
                elsif (
req_nopri3i= '1') then
                  ipi_c <=
"11";
                else
                  ipi_c <=
"00";
                end if;
              when "01" =>
                if (
req_nopri2i = '1') then
                  ipi_c <=
"10";
                elsif (
req_nopri3i = '1') then
                  ipi_c <=
"11";
                elsif
(req_nopri0i = '1') then
                  ipi_c <=
"00";
                else
                  ipi_c <=
"01";
                end if;
              when "10" =>
                if (
req_nopri3i = '1') then
                  ipi_c <=
"11";
                elsif (
req_nopri0i = '1') then
                  ipi_c <=
"00";
                elsif (
req_nopri1i = '1') then
                  ipi_c <=
"01";
                else
                  ipi_c <=
"10";
                end if;
```

```
              when "11" =>

                if (
req_nopri0i = '1') then

                  ipi_c <=
"00";

                elsif (
req_nopri1i = '1') then

                  ipi_c <=
"01";

                elsif (
req_nopri2i = '1') then

                  ipi_c <=
"10";

                else

                  ipi_c <=
"11";

                end if;

            end case;
        else   -- no change

            ipi_c <= ipi_i;

        end if;

      end process
combinational;



      sequential: process

      begin

        wait until clock = '1';
        if (reset = '1') then

          ipi_i <= "00";

        else

          ipi_i <= ipi_c;

        end if;

      end process sequential;

          ipi <= ipi_i;

      end stateMachine;
```

C8. Data type ATMdataType used in VHDL models

```
package atmDataTypes is

  function bits2natural (signal
bits : in bit_vector)   return
natural;
  procedure natural2bits (nat :
in natural; bits: out
bit_vector);
  function increment(signal
bits : in bit_vector)   return
bit_vector;
  function decrement(signal
bits : in bit_vector)   return
bit_vector;
--    procedure random (seed:
inout real; output : out real);
  function bit2digit(b: in bit)
return natural;
  function digit2bit(nat: in
natural) return bit;

end atmDataTypes ;


package body atmDataTypes is


  function bit2digit (b: in
bit) return natural is
    variable result : natural;

  begin
    if(b = '0') then
      result := 0;
    else
      result := 1;
    end if;
    return result;
  end bit2digit;

  function digit2bit(nat: in
natural) return bit is
    variable result : bit;
  begin
    if(nat = 0) then
      result := '0';
    else
      result := '1';
    end if;
    return result;
  end digit2bit;
```

```
  function bits2natural (signal
bits : in bit_vector) return
natural is
    variable result : natural;

  begin

    result := 0;
    for index in bits'range
loop
      result := result * 2 +
bit2digit(bits(index));

      -- result := result * 2 +
bit'pos(bits(index));
    end loop;
    return result;
  end bits2natural;


  procedure natural2bits (nat :
in natural; bits: out
bit_vector) is
    variable temp: natural;
    variable result :
bit_vector(bits'range);

  begin
    temp := nat;
    for index in
bits'reverse_range loop
      result(index) :=
digit2bit(temp rem 2);
            -- result(index) :=
bit'val(temp rem 2);
      temp := temp / 2;
    end loop;
    bits := result;
  end natural2bits;


  function  increment(signal
bits : in bit_vector)   return
bit_vector is

    subtype QelementSizeInBit
is bit_vector(bits'range);
    variable result :
QelementSizeInBit;
    variable tempNatural :
natural;
  begin

    tempNatural :=
bits2natural(bits) + 1 ;
    natural2bits (tempNatural,
result);

    return result;
  end increment;


  function  decrement(signal
bits : in bit_vector)   return
bit_vector is

    subtype QelementSizeInBit
is bit_vector(bits'range);
    variable result :
QelementSizeInBit;
    variable tempNatural :
natural;
  begin

    tempNatural :=
bits2natural(bits) - 1 ;
    natural2bits (tempNatural,
result);

    return result;
  end decrement;


end atmDataTypes;
```