Formalizing the Incremental Design and Verification Process of a Pipelined Protocol Converter

Cécile Braunstein
Université Pierre et Marie Curie
CNRS UMR 7606 - LIP6 - ASIM
12, rue Cuvier
75252 PARIS cedex 5- France
cecile.braunstein@lip6.fr

Emmanuelle Encrenaz
Laboratoire Spécification et Vérification
CNRS UMR 8643 - ENS CACHAN
61, avenue du Prsident Wilson
94235 CACHAN Cedex - France
emmanuelle.encrenaz@lsv.ens-cachan.fr

Abstract

This work studies the relations between pipeline architectures and their specification expressed in CTL. We propose a method to build pipeline structures incrementally from a simple one (already verified) to a more complex one. Moreover, we show how each increment can be integrated in a CTL specification. We define increments to model treatment delay *and* treatment abortion *of a pipeline flow, and we formalize the composition of the different increments. In order to represent the increments added to an architecture, we derive a set of CTL formulae transformations. Finally we model a* control flow of a protocol converter *by composition of these increments. We show how CTL properties of the complex architecture are built by applying automatic transformations on the set of CTL properties of the simplest architecture.*

1. Introduction

This paper proposes a method to specify and design protocol converters. Oftentimes, protocol converter devices integrate pipeline functionality. This is because these converters are used to connect a component with communication devices like bus or network on chip which are pipelined. The difficulty is to design and check the flow control of various components with various pipeline flows. Our aim is to propose a method which helps designers to build efficiently a pipeline flow and provides a set of CTL properties that represents its specification.

The verification process is done by model checking ([6]). Although this latest is not adequate to verify complex systems, it has been successfully used for medium-sized systems. More precisely, model-checking techniques are well-suited for protocols verification :successful experiments are described in [15] and [4] where the specification is expressed in a temporal logic. More recently, the idea of abstracting a component by a subset of its specification appears as a new method to alleviate the state space explosion problem. Xie and Browne in [18] proposed a compositional model checking process integrating this idea. Each component is described by an automaton that represents its specification and it is packed with a set of LTL properties. A component abstraction is built from these properties and environment assumptions. Büttner [8] adopts a similar method with CTL properties in the context of synchronized module composition. Its abstract model of module is well suited to provide a cycle accurate abstraction to be used in micro-architecture verification.

In [5], we defined an incremental design process that is very close to the way hardware designers proceed: after having sketched the rough structure of the data part, and its synchronization in the simplest case, one takes into account new events , and defines the new behaviours they induce. The new behaviours may not override previously existing ones, and there is no deletion of behaviours. In the same paper, we also stated a first set of transformations of CTL properties, corresponding to the preservation of all the behaviours previously existing in the simple model into the augmented model. In the present paper, we formalize a particular class of increments related to pipeline flow. Then we state new transformations and preservations of CTL properties in this particular context. We present property transformation related to the interface of the pipeline but also property transformation related to the inner part of the pipeline and expressing isochronous treatments in different pipeline stages in a unified way.

The results are relevant in the protocol verification context, but they also apply to the microprocessor pipeline. However, verification of temporal logic properties is not the classical approach to insure the correctness of a pipelined complex processor. Various techniques have been proposed for the verification of pipeline microprocessor design (see [7, 12, 1, 16]). The main approach compares a specification representing the sequential machine defined by the instruction set architecture (ISA) to an implementation pipeline of the architecture. The proof states that the implementation conforms to the set of behaviours represented by the non-pipelined specification. One of the difficulties is to define observation points where the comparison is meaningful. The proof is performed with a proof assistant (PVS, ACL2, HOL,...) that requires an important manual effort. Alur and al. in [9] build their proof with a refinement checker included in MOCHA [2] but the designer has to provide a accurate abstraction and different witness modules. The main drawback of these methods is the strong human interaction required to build the proof.

In this paper we do not focus on a microprocessor pipeline because designing a pipeline micro-architecture is not the major difficulty of microprocessor design anymore. Nowadays, the difficulty comes from other mechanisms like reordering buffer or precise exception handling.

However, pipelining an architecture was not an easy task: researchers have proposed methods to help building such a processor pipeline. For instance, Huggins and Van Campenhout [10] simplify the design of a processor pipeline based on the decomposition in a series of steps. At each step the equivalence between models are manually stated. Kroening[11] has extended this idea to propose an automatic synthesis of the pipeline of a processor. Our work revisits the automatic design of pipelines in the context of protocol conversion, and provides new results in terms of temporal logic specifications, that was not covered in the context of microprocessor pipelines.

The paper is organized as follows : section 2, introduces some definitions related to the incremental design process and describes the model of the pipeline we deal with; in section 3, the increments modeling the pipeline flow breakage are presented, and the structural properties between the initial model and the incremented ones are proven; consequences on CTL properties are defined in section 4. Section 5, firstly, shows how the defined increments can be composed to build the pipeline flow of a protocol converter between a VCI compliant component (Virtual Component Interface [13]) and a PI bus ([14]). Secondly, how some CTL formulae representing the converter specification evolve along the design process.

2. Preliminaries

2.1. Incremental Design Process

The incremental design process ([5]), starts from an initial step where the rough structure of the data-path and the control part is defined. Then the designer proceeds to the implementation of the simplest cases up to the most complex ones. This is accomplished by *adding* new functionalities without overriding nor deleting previous behaviours. Our models are represented by a Moore machine.

Definition 1 *Each* signal *is defined by a variable name, s and an associated finite definition domain Dom(s).*

Definition 2 *Let E be a set of signals. A* configuration *c(E) is the conjunction of the association : for each signal in E, one associates one value of its definition domain. The* set of all configurations *c(E) is named C(E).*

Definition 3 *A* Moore machine $M = \langle S, S_0, I, O, T, L\rangle$ *is such that S is a finite set of states ; $S_0 \subset S$ is a finite set of initial states ; I (resp. O) is a finite set of input signals (resp. output) with their definition domain ; $T \subseteq S \times C(I) \times S$ is a finite set of transitions such that $\forall s \in S$, $\forall c \in C(I)$, $\exists! s' \in S$ s.t. $(s, c, s') \in T$ ($\exists!$ means "there exists exactly one"); and $L\{l_0, \ldots, l_{|O|-1}\}$ is a vector of labeling function, each function defining the value of exactly one output signal in each state; for each output signal o_j we have $l_j : S \rightarrow Dom(o_j)$,*

In [5], we give a formal definition of an increment $INC = \langle e, \Sigma_+, T_+, O_+\rangle$. Intuitively, an increment represents the reaction of the system to a set of new event e, e. g. the set of new states, transitions and outputs signals. A new event is represented by a new[1] set of signals added on the input interface of the system. The event may be active or not. The occurrence of the new event implies new behaviours and a new set of output signal. This notion is formalized as follow.

Definition 4 *An* event $e = \langle I_+, C_{ACT}(I_+), C_{QT}(I_+)\rangle$ *is such that*

I_+ *The set of new input signals and their definition domain,* $I \cap I_+ = \emptyset$.

$C_{ACT}(I_+)$**:** *The set of configurations representing the occurrence of the new event. If one such configuration occurred the event would be said to be active. We denote c_qt a configuration belonging to C_{QT}.*

$C_{QT}(I_+)$**:** *The set of configurations representing the absence of the new event. If one such configuration occurred the event would be said to be quiet. We denote c_act a configuration belonging to C_{ACT}.*

We have $C_{ACT}(I_+) \cup C_{QT}(I_+) = C(I_+)$ and $C_{ACT}(I_+) \cap C_{QT}(I_+) = \emptyset$. We note $\neg c\_act \in C_{QT}$ and $\neg c\_qt \in C_{ACT}$.

In the incremented model, all transitions that were in the simplest model are labeled with a quiet value (c_qt). All transitions at the boundary of the simplest model and the incremented one, are labeled with an active value (c_act).

2.2. Pipeline representation

Figure 1 represents a typical pipeline flow of n stages. The control part contains a Moore Machine that produces the multiplexer command (x_i) driving the barrier register (R_i) at the input of each stage. Each state of the Moore machine represents a configuration of the pipeline stages where

[1]This can be extended to model the appearance of new value of existing signals (see [5])

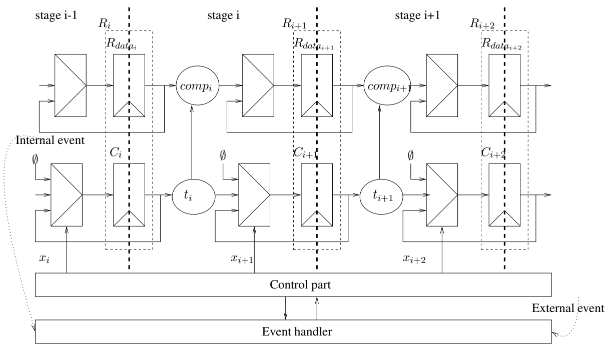

Figure 1. Pipeline flow architecture

the computation is valid (and then written in the barrier register at the beginning of the next stage) or not. Transitions represent how the pipeline fills. Two sets of registers compose the barrier : one containing command (C_i) and the other data (R_{data_i}) needed for the treatment into a stage. The event handler generates events stalling or breaking the pipeline flow from internal or external signals. From the control part point of view, there is no difference between external and internal events. Both comes from the event handler and both disturb the pipeline flow. Data treatment at each stage is represented by $comp_i$ and transitions by t_i. At each step the register of a stage may take a new data coming from the previous state, re-write its content or take an empty operation. An empty operation does not require any resource and do not disturb the state of the system.

Formally, the states of the Moore machine of a n stages pipeline is a vector of x_i. We have $(x_0 \dots x_{n-1})$ such that $\forall j, x_j \in$ {0,1,R}. The meaning of these symbols is:

- $x_j = 0$ insertion of an empty operation in R_j.
- $x_j = 1$ insertion of the result of the computation of stage $j - 1$ in R_j.
- $x_j = R$ re-writing of the R_j's content in R_j.

We define the set of vectors $V_l^k = x_l, x_{l+1}, ..., x_{k-1}$ such that $\forall j, x_j \in$ {0,1,R}. This represents a contiguous subset of the pipeline stages ranking from stage l to stage k. Here are introduced functions representing the *prefix* or *suffix* of a state.

Definition 5 Prefix and Suffix functions.
The function pref*:* $\mathbb{N} \times S \rightarrow V_0^k$ *associates to each state s and stage number* $k \in \mathbb{N}$*, the prefix of the state ranking from* 0 *to* k*.*
The function suff*:* $\mathbb{N} \times S \rightarrow V_{k+1}^n$ *associates to each state s and stage number* $k \in \mathbb{N}$*, the suffix of the state ranking from* $k+1$ *to* $n-1$*.*
The notion of data progression inherent in pipeline flow is defined by the *progress* function, formalized as follow :
Definition 6 Progress function.
Function progress$_{k,l}$*:* $\{0,1\} \times V_k^l \rightarrow V_k^l$ *is the right shift of any element in* V_k^l *of* 1 *slot with either* 0 *or* 1 *injected in* x_k[2]*.*

[2]When there is no ambiguity, indexes k and l of *progress* will be removed.

3. Incremental design of Pipeline flow

In the following, we present the machine with regular flow. Then, we define the increments necessary to represent breakage and interrupts.

3.1. Optimal flow

The simplest architecture is modeled by a Moore machine $M_o = \langle S_o, S_{0_o}, I_o, O_o, T_o, L_o \rangle$. It is the implementation of an optimal flow (no event disturbs the flow). In this case we consider that no event stalling a stage or freezing the pipeline may occur : the pipeline flow is regular and by consequence all states are labeled with an unique succession of consecutive 1.

Let t be in T_o, t is the conjunction of elementary transitions t_i, each occurring at a given stage i of the pipeline, and potentially driving register R_i. $t \in T_o$ if and only if it is defined as definition 7. Let be $s = (x_j)_{j \in [0;n-1]}$, $s' = (x'_j)_{j \in [0;n-1]}$ and $s'' = (x''_j)_{j \in [0;n-1]}$ then we have the following rules :

Definition 7 Transition rules associated to an optimal flow*:*
R1 *After a* 0*, only* 0 *may enter the pipeline, except for the initial state : If* $x_0 = 0$ *and if* $s \notin S_{0_o}$ *then* $\exists t \in T_o$ *and* $\exists c \in C(I_o)$ *such that* $t = (s, c, s')$ *and* $s' = progress(0, s)$
R2 *Normal progression : there exists transitions with a new instruction or an empty operation entering the pipeline : If* $x_0 = 1$ *or s is the initial state then* $\exists t \in T_o$ *and* $\exists c \in C(I_o)$ *such that* $t = (s, c, s')$ *and* $s' = progress(0, s)$ *and* $\exists t' \in T_o$ $\exists c' \in C(I_o)$ *such that* $t' = (s, c', s'')$ *and* $s'' = progress(1, s)$*.*

3.2. Stall Increments applied to a pipeline flow

The possible increments for a pipeline flow can be of two types. The first type is an event, named `stall`, that introduces deceleration in the pipeline flow. This is the case when the pipeline waits for a condition like a cache miss or a ready acknowledgment. The second type, named `kill`, concerns the pipeline flow breaks or reset.

3.2.1 Single Stall

An event can stall a stage and all the stages upstream, the stages downstream progress and the stalled stages re-start as soon as the stalling condition is not active anymore. The stalling condition is modeled by an event $\texttt{stall}_k = \langle stall_k, stall_k\_act, stall_k\_qt \rangle$.
When $\texttt{stall}_k$ occurs then the $(k+1)^{th}$ stage executes an empty operation; in all stages $l > k$, the flow progresses; in stages $l \leq k$, the flow does not progress : each register R_l re-writes the value it previously stored. When $\texttt{stall}_k$ becomes inactive then the normal progression takes place (as defined by Rule R2). These new behaviours are modeled in a new Moore Machine M_s obtained by applying the

incremental design process to M_o. Below we define the increment transforming M_o to M_s.

Definition 8 Transition rules associated to $\texttt{stall}_k$ in M_s: Let s be a state in S_o.

R3 *Existing transitions have their guards strengthened by $stall_k\_qt$: $\forall s' \in S_o$,s.t. $\exists t = (s, c, s') \in T_o$, then $\exists t' \in T_s$ s.t. $t' = (s, c \wedge stall_k\_qt, s')$*

R4 *The upstream of the pipe is frozen : $\exists s'' \notin S_o$, s.t. $\exists t = (s, stall_k\_act, s'')$ and*

(a) $\forall x''_j \in pref(k, s'')$: $x''_j = \begin{cases} R & \text{if } x_j = 1 \\ 0 & \text{if } x_j = 0 \end{cases}$

(b) $suff(k, s'') = progress(0, suff(k, s))$

Let be $s \in S_s \setminus S_o$.

R5 *After being unfrozen, progression is normal : $\exists s'$ s.t. $(s, stall_k\_qt, s')$ and s' is obtained by Rule R2.*

R6 *The downstream of the pipeline progresses : $\exists s''$ s.t. $(s, stall_k\_act, s'')$ and*
(a) $pref(k, s'') = pref(k, s)$,
(b) $suff(k, s'') = progress(0, suff(k, s))$

We state properties characterizing the flow of each stage between M_0 and M_s needed for the CTL properties transformations..

Notation : $x \rightarrow x'$ means $\exists c \in C(I)$ and $(x, c, x') \in T$.
$\sigma = y \dots y'$ is the path from y to y' such that $y \rightarrow y_0$, $y_0 \rightarrow y_1, \dots, y_k \rightarrow y'$.

Property 1 Suffix progression.
Let be a stall occurring at stage l or lower, inducing the machine M_s from M_o. Let R_l be a binary relation in $S_o \times S_s$ such that: $x\, R_l\, y$ iff $suff(l, x) = suff(l, y)$. $\forall x' \in S_o$ s.t. $x \rightarrow x'$, $\exists y' \in S_s$ s.t. $y \rightarrow y'$ and $x' R_{l+1} y'$.

PROOF: By construction of M_s

Unfortunately, R_{l+1} is not included into R_l, thus it is not a strong bisimulation [3]. Hence this property is local to the stall and expresses the progression of the suffix downstream, whenever the flow is broken upstream or not.

Property 2 Prefix weak bisimulation.
Let be a stall occurring at stage l or higher, inducing the machine M_s from M_o. Let R_l be a binary relation in $S_o \times S_s$ such that: $x R_l\, y$ iff $pref(l, x) = pref(l, y)$. R_l is a weak bisimulation [3].

PROOF: We have: $\forall x' \in S_o$ s.t. $x \rightarrow x'$, $\exists y' \in S_s$ s.t. $\sigma = y \dots y'$ and $x' R_{l+1} y'$. As $pref(l+1, x) = pref(l+1, y) \Rightarrow pref(l, x) = pref(l, y)$, R_{l+1} is included into R_l.
$\forall y' \in S_s$ s.t. $y \rightarrow y'$ s.t : $x \rightarrow x'$ and $x' = y'$ (when y is not stalled and reads $stall_l\_qt$), or (when y reads $stall_l\_act$) $x\, R_l\, y'$ and $y' \dots y''$ and $x'\, R_{l+1}\, y''$. R_{l+1} is included into R_l. This property formalizes that the prefix of the pipeline do not progress and is not destroyed while a stall is active.

Property 3 Stuttering progression.
In M_o: We have $\sigma = s_0 s_1 ... s_n$ such that in s_n: $V_l^{n+k} = progress^n(V_0^k)$.
In M_s: Let $\texttt{stall}_k$ be a stalling action occurring at stage k. Then $\exists \sigma' = s_0^* s_1 ... s_n$ such that s_n: $V_l^{n+k} = progress^n(V_0^k)$.

PROOF: This is a direct consequence of rule R5 (assuming that the stalling action always terminates). This property formalizes that after being frozen, the prefix will progress(as it did in the previous model).

3.2.2 Composition of Stall Increments

It is possible to have a combination of events inducing stalls occurring at different stages. We define new transitions rules to model the dealing with multiple stalls. The transition rules are quite similar to the single stall increment we have seen before. But now, the increment that affects the highest stages has a greater impact on the pipeline flow, than the increment concerning lower stages.

Definition 9 Set of Stalls.
Let be F = $\{k \mid k \in [0, n-1]\}$ the set of stages where a stall currently occurs.

Let M'_s be the machine obtained by applying on the machine M_s that contains already some stalls (defined in F_s), a new stall at stage k s.t $k >$ max(F_s). F_s is increased with k: $F'_s = F_s \cup \{k\}$. M'_s is composed of states in $S'_s \supset S_s$

Definition 10 Transitions rules associated to M'_s.
Transitions in $T'_s \supset T_s$ are defined s.t.:

- *Let s be a state in $S_s \cap S'_s$. Its previously existing transitions are modified according to rule R3 with value $stall_k\_qt$.*
- *M'_s has got one new transition respecting rule R4 with value $stall_k\_act$.*
- *Let be $s \in S'_s \setminus S_s$,*
  1. *either s is the source state of the transition obtained by rule R6.*
  2. *or* **R5'** *After being unfrozen the entire pipeline progresses : $\exists s' \in S'_s$ s.t. $(s, c \wedge stall_k\_qt, s')$ with c equal the conjunction of all $stall_l\_qt$ $\forall l \in F \setminus \{k\}$ and s' is obtained by Rule R2 (either a 0 or a 1 is injected at stage 0).*
  3. *or* **R7** *The downstream of the pipeline defined by the active stall progresses : $\forall l \in F \setminus \{k\}$, $\exists s'' \in S'_s$ s.t. $(s, c \wedge stall_k\_qt, s'')$ with $c = \bigwedge_{\forall j \in [k;l[} stall_j\_qt \wedge stall_l\_act$ and with s'':*
     (a) $pref(l, s'') = pref(l, s)$
     (b) $suff(l, s'') = progress(0, suff(l, s))$.

When we introduce a new increment $\texttt{stall}_k$ occurring at a stage $k < max(F_s)$ the active configuration is now

$\forall l \in F_s \, and \, l > k$, $stall_l\_qt \wedge stall_k\_act$. This is because if a higher stall $\mathtt{stall}_l$ is active, no matter $\mathtt{stall}_k$ is also active, $\mathtt{stall}_l$ freezes $pref(l, s)$, that encompasses $pref(k, s)$.

Property 4 *(Extension of property 2 in case of multiple stalls). Let be a machine M'_s obtained by multiple stall increments from M_o, having a set of stalls F'_s. Let be $l \leq min(F'_s)$. Let R_l be the relation in $S_o \times S_s$: $x \; R_l \; y$ iff $pref(l, x) = pref(l, y)$. R_l is a weak bisimulation and $\forall$ $j > l$, R_j is not a weak bisimulation.*

PROOF: (sketch) The proof of the first statement proceeds as for the single stall increment case (property 2).
The idea of the proof of the second statement is the following: In case of a single increment at stage l, the stages ranking from 0 to $l - 1$ have the same progression: either they are fixed (while $stall_l\_act$), or they progress at the same speed (when $\mathtt{stall}_l$ is not active anymore). This is captured by the weak bisimulation of the prefix R$_l$ and the stuttering progression property. If $l > min(F_s)$, then there exists a stall , say $k < l$ splitting the interval $[0; l[$ of stages into $[0; k]$, where the behaviour is frozen until $\mathtt{stall}_k$ is removed, while the stages ranking from k to $l - 1$ may progress. Hence the similarity of behaviours of stages in $[0, l]$ are not captured in R$_l$ anymore but in R$_k$ (that is included in R$_l$), and the stuttering progression property.

3.3. Kill Increment

A kill action destroys the treatment at a given stage, but the pipeline flow is not disrupted. The kill action is the basic operation performed in case of retract, reset, exception or interrupt. We will show in section 5 how kills are used to manage these events. In our representation, a kill action consists in replacing the "1" corresponding to the progression of the treatment by an empty operation "0" that discards the result of the treatment.

Definition 11 *Let M_s be a machine, a kill event occurring at stage k induces the following machine M_k: $S_k \supset S_s$ and T_k is defined such that:*
1. *$\forall t \in T_k$, $t = (s, c, s')$, t is changed into (s, c', s') with c' = $c \wedge$ $\mathtt{kill_k\_qt}$.*
2. *$\forall s \in S_s \cap S_k$, $\exists s' \in S_k$ and (s,$\mathtt{kill}_k\mathtt{\_act}$,$s'$) $\in T_k$ and s' is defined s. t. :*
   x'_0 = 0 or 1, x_k = 0 and $\forall i \neq k$, x'_i = x_{i-1}.

4. Consequences on CTL formulae

This section gives results on CTL property preservation or transformation between a reference machine and the one obtained by a composition of increments. We show that global behaviours are preserved when stalling actions are added, e.g. when a command enters, a result will be produces later and it is guaranteed by construction. Moreover, specification related to inner part are preserved if the formulae concern a unique stage or a disjunction of stages. Nevertheless, adding stalling actions does not preserve the specification about conjunction of stages. But in this case, we state a new property transformation. The present section is organized as follow : in a first part, we consider properties with atomic propositions inside the pipeline. In a second part, we focus on properties concerning the macroscopic treatment performed by the pipeline.

4.1. Properties related to the inner parts

Let M_s be a machine obtained by composition of stall increments applied to M_o, and F_s be the set of associated stalls. Let M'_s be the machine obtained by composition of stall increments applied to M_s and $F'_s (\supset F_s)$ be the set of associated stalls. We name ϕ_k (resp. ϕ_l) an atomic proposition (or its negation) related to a stage k (resp. l) in M_s.

Property 5 *Let f and g be any positive CTL formula without any terms in the following form : $(\phi_l \wedge \phi_k)$ or $(\neg\phi_l \wedge \phi_k)$, $\forall l, k \in [0, n]$.*
Let M_s,$s \models f$, we have M'_s,$s \models f$.

PROOF: (Sketch) This is due to the weak prefix bisimulation and the stuttering progression: let ϕ_k (resp. ϕ_l) be a formula with atomic propositions related to stage k (resp. l), for any CTL\X operator OP, the formula of the form OP(ϕ_k)(resp. OPϕ_l) are preserved. Their disjunction is then preserved, and positive formulas built on their disjunction are also preserved. This is not true for the conjunction of atomic proposition concerning different stages (second item of property 4).

Property 6 *Let f and g be any positive CTL formula with conjunction of atomic propositions. We have the following properties for $k < l$ and a CTL\X operator OP:*
1. *if $\nexists i \in F'_s$ s.t. $i \geq l$, then M_s,$s \models f \Rightarrow M'_s$,$s \models f$.*
2. *if $\exists i \in F'_s$ s.t. $i < l$, and if φ = OP $(\phi_k \wedge \phi_l)$ then M_s,$s \models \varphi \Rightarrow M'_s$,$s \models \varphi'$ and φ' = OP $(AF(\phi_l) \wedge \phi_k)$*

PROOF: Direct consequence of properties 3 and 4.

4.2. Properties related to the outer parts

The environment of the pipeline is viewed as a set of actions composed of commands producing results. In case of a VCI-PI protocol converter, it is composed of the set of VCI commands and of VCI responses. In case of a processor, the environment is composed of instructions on the software visible registers plus the program counter, instruction and exception registers, and the memory.

We abstract the environment by a set E = $\{(Cmd_k , Res_k)\}$, where couples (Cmd_k ,Res_k) denotes the k^{th} command and its induced result. The causality between commands and results, and the interleaving of several actions are modeled by a set of CTL\X properties. A command Cmd_k entering the pipeline may be expressed as: $\phi_{0,k} = (x_0 = 1 \wedge C_i = Cmd_k)$. C_i denotes the contents of a register in stage i. The end of the computation induced by Cmd_k is expressed by: $\phi_{n-1,k} = (x_{n-1} = 1 \wedge C_{n-1} = Cmd_k)$

Property 7 *All positive CTL\X formulas with atomic propositions in E, that are true in M_o, are also verified in any machine obtained by composition of stall increments.*

PROOF: This is a direct consequence of property 5 that preserves positive CTL\X formulae when atomic propositions concern disjunction of stages (here concerned stages are 0 and $n-1$).

In case of a kill increment in a stage i, the killed command does not produce a result. In case of occurrence of a similar command not concerned with the kill event (in a different stage), a result similar to the one destroyed by the kill will be produced. A causality property expressed as $\Phi_k = \phi_{0,k} \Rightarrow AF(\phi_{n-1,k} \wedge AF\, Res_k)$ can be transformed in the following form :

$$\Phi'_k = \neg kill_i \wedge \phi_{0,k} \Rightarrow$$
$$A(\neg kill_i U (Res_k \vee \qquad (1)$$
$$(kill_i \bigwedge_{l \in [0;n-1]} (\neg \phi_{l,k}) \Rightarrow AF \neg Res_k) \vee \qquad (2)$$
$$(kill_i \bigvee_{l \in [0;n-1]} (\phi_{l,k}) \Rightarrow AF\, Res_k)) \qquad (3)$$

Line (1) expresses that there exists some path where $kill_i$ is never true due to the incremental design rules. Line (2) says that if a kill event occurred and no stage contains the command then the associated result is not produced. Line (3) corresponds to the occurrence of a similar command that produces a similar result.

5. Incremental design of the VCI-PI wrapper

In this part, we briefly recall the wrapper structure and then show how the formulae are transformed or preserved according to properties of section 4 along the incremental design of the pipelined protocol converter.
The conversion between PI-bus and VCI protocols is realized by a component named a VCI-PI wrapper. A wrapper is a core wrapping device implementing a given interface. In our context, the IP-core is supposed to be VCI compliant [13] and the considered wrapper is an adapter between the VCI interface and the PI-bus protocol [14]; hence we are able to connect various IP-cores through a PI-bus. PI protocol distinguishes the component initiating a bus transfer, named *master*, and the component responding to a transfer, named *slave*. An IP-core may have both *master* and *slave* functionalities.

Using the incremental design process approach, we developed a set of nine master VCI-PI wrappers, from a very simple one supposing that the VCI initiator and the PI target will always acknowledge in one cycle, up to the most complex one supporting delays, retract and reset events sent by the VCI initiator or the PI target. The hierarchy of the nine master wrappers is shown in Figure 2.

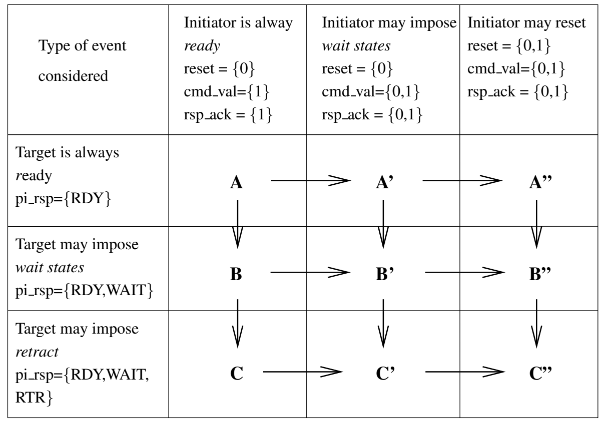

Figure 2. Hierarchy of VCI-PI wrappers ranking from **A** to **C"**. Each arrow corresponds to an increment whose associated event is an extension of the definition domain of one or more signals.

The behavior of the simplest wrapper (model A) is a 3-stages pipeline, performing at the same time:
(**stage 1**) accepting a VCI request k to be sent to PI from its VCI interface,
(**stage 2**) sending the PI request corresponding to the $k-1^{th}$ VCI request on its PI interface,
(**stage 3**) accepting the PI response to the $k-2^{th}$ VCI request on its PI interface.

In the following, we show step by step how we build a wrapper C" and a part of his specification from the wrapper B. The architecture is described in synchronous Verilog, and the specification is checked with the model checker VIS verification tool [17].

STEP 1 : (Wrapper B) We implemented a platform composed of a VCI initiator, a master wrapper B, a PI-bus, a slave wrapper B and a VCI target. We written and checked about 80 CTL formulae related to the master wrapper B, the slave wrapper B and the complete system (when the VCI initiator and target may generate delay events).

STEP 2 : (Wrapper B') We fit the platform in order to plug a wrapper B'. The wrapper B' can handle delays from the initiator. The increment applied is the composition of two stall increments. The first one stalling stage 1 and the other one stalling the stage 3. We reinforce our results by re-checking the set of all formulae written for the wrapper B. Of course, we transformed the formulae following the properties stated in section 4. In practice, it is not useful to re-check formulae, we can obtain the new set of formulae by applying the increment rules and the properties transformation or preservation.

STEP 3 : (Wrapper C') We incremented the wrapper

B' to wrapper C'. Wrapper C' can support retract from the target. It corresponds to a new behaviour that breaks the pipeline flow. This new event induces a kill increment to stage 1 and a stall increment to stage 2. We fit the platform and transform the formulae. The formulae with all atomic proposition corresponding to the suffix are transformed with properties 6 or 5. The others are transformed with the property stated in [5].

STEP 4 : (Wrapper C") We added the new event reset, it kills all requests that were in the pipeline. We add 3 increments, one for each stage of the pipeline. In this case the formulae have to be transformed with the causality property stated in paragraph 4.2. Formulae can be automatically added to insure the preservation of non-reseted models into reseted one. These formulae state that after a reset occurrence, the converter returns into `idle` state and the pipeline is empty.

We have built a model which is guaranteed to behave according to pipeline and its specification as a set of 80 CTL formulae. One can pick some of them to build abstraction to alleviate the verification process of global properties as in [18].

6. Conclusion

On the one hand, we have formalized an incremental method that is very close to those used by the designers. Our approach decomposes the complexity of building a pipeline flow from scratch by adding the different increments one by one. The designer has got a framework to focus on one difficulty at a time. Moreover this technique is not regressive, all behaviours of the component are preserved when a new increment is added.

On the other hand we have shown that this method automatically derives the specification of a component from the specification of a simpler component. This specification is integrable into a general symbolic model checking process. By exploiting the behavioural characteristics that distinguish pipelines from other circuits we have particularized the pipeline increments and stated new CTL formulae transformation or preservation results. These transformations capture the behaviour that already existed and characterize the added behaviours.

The approach we propose can be viewed of two different ways. Either the component is built applying the increments, it is guaranteed to respect the new specification, and it can be plugged *as it is* in a more complex system, its specification being used for compositional verification (assume-guarantee). Or the design is manually augmented (step by step) and the new specification is the one that the system has to comply with.

The set of CTL properties automatically obtained with this incremented design process, exactly captures the increments successively added. It is the basis for an abstraction of each module by a subset of its formulae in order to alleviate the model checking verification process.

References

[1] M. Aagaard. A Hazards-Based Correctness Statement for Pipelined Circuits. In *CHARME'03*, volume 2860 of *LNCS*, pages 66–80. Springer-Verlag, 2003.

[2] R. Alur, T. A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in Model Checking. In *CAV'98*, volume 1427 of *LNCS*, pages 521–525. Springer-Verlag, 1998.

[3] A. Arnold. *Finite Transition Systems: Semantics of Communicating Systems*. Prentice Hall International Ltd., Hertfordshire,UK, 1994. Translator-John Plaice.

[4] M. Bozga, J.-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol Verification with the ALDÉBARAN Toolset. *STTT*, 1(1-2):166–184, 1997.

[5] C. Braunstein and E. Encrenaz. CTL-Property Transformations along an Incremental Design Process. In *AVOCS'04*, volume 128 of *ENCS*, pages 263–278. Elsevier, 2004. to appear in STTT.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992. Special issue for best papers from LICS'90.

[7] J. Burch and D. Dill. Automatic Verification of Pipelined Microprocessors Control. In D.L. Dill, editor, *CAV'94*, volume 818 of *LNCS*, pages 68–80. Springer-Verlag, 1994.

[8] W. Büttner. Is formal verification bound to remain a junior partner of simulation ? Invited talk, CHARME'05, 2005.

[9] T. Henzinger, S. Qadeer, and S. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *CAV'98*, volume 1427 of *LNCS*, pages 440–451. Springer-Verlag, 1998.

[10] J. Huggins and D. V. Campenhout. Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, 1998.

[11] D. Kroening and W. Paul. Automated Pipeline Design. In *DAC '01*, pages 810–815. ACM Press, 2001.

[12] P. Manolios and S. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB refinements. In *DATE '04*, pages 168–174. IEEE Computer Society, 2004.

[13] On-Chip Bus Development Working Group. *Virtual Component Interface Standard (VCI).* VSI Alliance, 2000.

[14] Open Microprocessors System Initiatives. *OMI324: PI-Bus Standard Specification*. Siemens, Munich, Germany, 1994.

[15] H. Peng, S. Tahar, and F. Khendek. Comparison of SPIN and VIS for Protocol Verification. *STTT*, 4(2):234–245, 2003.

[16] J. Sawada and W. Hunt. Trace Table Based Approach for Pipeline Microprocessor Verification. In *CAV'97*, volume 1254 of *LNCS*, pages 364–375. Springer-Verlag, 1997.

[17] The VIS group. VIS : A System for Verification and Synthesis. In *International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.

[18] F. Xie and J. C. Browne. Verified systems by composition from verified components. In *ESEC/FSE-11*, pages 277–286. ACM Press, 2003.