Incremental Specification and Design of a Pipelined Protocol Converter

Cécile Braunstein and Emmanuelle Encrenaz
Pierre and Marie Curie University
Laboratoire d'Informatique de Paris 6 - ASIM - CNRS UMR 7606
12, rue Cuvier,75252 PARIS CEDEX 5- France
Email: {Cecile.Braunstein,Emmanuelle.Encrenaz}@lip6.fr

***Abstract*— The incremental design process we propose aim at helping the hardware designer during the design of a component. A complex component is built by *addition* of new behaviours to a simpler previously verified component. These new behaviours model the reactions to new events that were ignored in the simplest component. We investigate the way CTL properties that are verified on a simple model are preserved or may be transformed to be verified into the complex model, in the particular context of pipeline structures. This paper defines the increments modeling treatment delay or treatment abortion and their composition; then it derives a set of transformations of CTL formulae corresponding to these increments; finally it shows how these increments are composed to model the control of a protocol converter and how CTL properties are built.**

I. Introduction

This paper proposes a method to specify and design protocol converters. Oftentimes, hardware device verification is performed by simulation, which takes a lot of time and covers a very small amount of all possible behaviours. The specification of components by means of CTL formulae and verification by model-checking ([6]) emerged as an alternative verification method. Although this latest is not adequate to verify very complex systems, it has been successfully used for medium-sized systems. More precisely, model-checking techniques are well-suited for protocols verification. For instance, successful experiments are described in [16] and [4] where the specification is expressed in a temporal logic (CTL or LTL). 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 to help designers to build efficiently a pipeline flow and to provide a set of CTL properties that represents its specification. Moreover, our method guaranties, by construction, the correctness of the flow control.

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 (that were supposed not to occur in the previous steps of the design), 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 define a particular class of increments related to pipeline flow. Then we state new transformations and preservations of CTL properties in this particular context. The combination of increments models hardware devices whose treatments are pipelined. 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 [1], [7], [10], [11], [15], [17]). 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, PVS . . .) that requires an important manual effort. Alur and Henzinger 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 pipelining a microarchitecture is not the major difficulty of microprocessor design anymore. Nowadays, the difficulty comes from other mechanisms like reordering buffer, precise exception handling, or thread switching context in SMT.

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 [12] 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 [14] 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 results in terms of temporal logic specifications, that was not covered in the context of microprocessor pipelines.

The paper is organized as follows : section II recalls some definitions related to the incremental design process; section III describes the model of the pipeline we deal with; in section IV, 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 V. Section VI 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 [8]) and a PI bus ([13]), and how some CTL formulae evolve along the design process.

II. Preliminaries - Incremental Design Process

The incremental design process 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, thus building a more and more complex device. The new functionalities cannot override nor delete previous behaviours.

In this section we define an increment as a set of extensions applied on a model represented by a Moore machine, in order to build a more complex one.

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: Finite set of states.

S_0: Finite set of initial states.

I: Finite set of input signals with their definition domain.

O: Finite set of output signals with their definition domain.

T: Finite set of transitions $\subseteq S \times C(I) \times S$, $\forall s \in S$, $\forall c \in C(I)$, $\exists! s' \in S$ s.t. $(s, c, s') \in T$ ($\exists!$ means "there exists exactly one").

L: Vector of generation functions = $\{l_0, \ldots, l_{|O|-1}\}$ each function defining the value of exactly one output signal in each state; for each output signal o_j $0 \leq j < |O|$ we have $l_j : S \rightarrow Dom(o_j)$,

The increment represents the reaction of the system to a new event e. The new event is represented by a new 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.

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*.

$C_{QT}(I_+)$: 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 have $C_{ACT}(I_+) \cup C_{QT}(I_+) = C(I_+)$ and $C_{ACT}(I_+) \cap C_{QT}(I_+) = \emptyset$.

Definition 5: An *increment* is a 4-tuple where $INC = \langle e, \Sigma_+, T_+, O_+\rangle$

e : the event defined above.

Σ_+ : the set of new reachable states. $\Sigma_+ \cap S = \emptyset$

$T_+ \subseteq (S \times C(I \cup I_+) \times S) \cup (S \times C(I \cup I_+) \times \Sigma_+) \cup (\Sigma_+ \times C(I \cup I_+) \times \Sigma_+) \cup (\Sigma_+ \times C(I \cup I_+) \times S)$: The set of new transitions composed with the transitions present in M and the new ones introduced by the active configurations.

- each transition (s_1, c, s_2) in T will have its input configuration extended with a sub-configuration of the new input signals belonging to $C_{QT}(I_+)$: there exists (s_1, c', s_2) s.t. $c' \in C(I) \times C_{QT}(I_+)$ and the projection of c' on I equals c. In the following we will write $c' = c \wedge c\_qt$, $c\_qt \in C_{QT}(I_+)$.
- each transition (s_1, c, s_2) in $T_+ \cap (S \times C(I \cup I_+) \times \Sigma_+ \cup S \times C(I \cup I_+) \times S)$ will have its input configuration extended with a sub-configuration of the new input signals belonging to $C_{ACT}(I_+)$: there exists (s_1, c', s_2) s.t. $c' \in C(I) \times C_{ACT}(I_+)$. In the following we will write $c' = c \wedge c\_act$, $c\_act \in C_{ACT}(I_+)$.

O_+ : the set of new output signals and their definition domain, with :

- $C_{ACT}(O_+)$: The set of configurations representing the activation of the output.
- $C_{QT}(O_+)$: The set of configurations representing the non-activation of the output.

The output functions associated to O_+ returns a configuration in $C_{QT}(O_+)$ for all states that were in S.

Remark 1: We have $\neg c\_act \in C_{QT}$ and $\neg c\_qt \in C_{ACT}$.

III. Pipeline Representation - Basic case

A pipeline is composed by a number n of stages, each stage i is separated with stages $i-1$ and $i+1$ by a register barrier. In a stage i , the input register barrier R_i, driven by x_i is read and the treatment is executed; then the resulting information is recorded into the next register barrier R_{i+1}, driven by x_{i+1}. We represent the pipeline flow control like a complete and deterministic Moore machine $M_o = \langle S_o, S_{0_o}, I_o, O_o, T_o, L_o\rangle$. Each state represents a configuration of the pipeline stages where 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. Figure 1 represents a typical pipeline flow. 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. 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. 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.

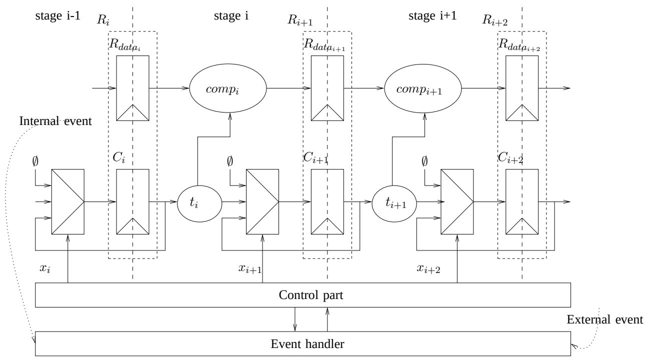

Fig. 1. Pipeline fbw design

The basic case we describe now is the optimal pipeline flow. The environment behaves ideally (no cache miss, no delaying actions).

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.

The states of a pipeline of n stages are in V_0^n. We have $(x_0 \ldots x_{n-1})$ 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.

Remark 2: In the basic case we consider that no event stalling a stage or freezing the pipeline may occur. In this case, the pipeline flow is regular and by consequence all states are labeled with an unique succession of consecutive 1.

Definition 6: Progress function.
The 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.

When there is no ambiguity the indexes k and l of *progress* will be removed.

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:
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 :

(R1) If $x_0 = 0$ and if s is not the initial state then $\exists t \in T_o$ and $\exists c \in C(I_o)$ such that $t = (s, c, s')$ and $s' = progress(0, s)$

(R2) 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 $s'' = progress(1, s)$.

IV. 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.

A. Single Stall

An event can stall a stage and all the stages upstream, the stages downstream progress and the stalled stages restart 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 rewrites 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. We firstly introduce functions representing the prefix or suffix of a state.

Definition 7: 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 $n-1$.

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

(R3) For all states s' in S_o,s.t. $\exists t = (s, c, s') \in T_o$, then $\exists t' \in T_d$ s.t. $t' = (s, c \wedge stall_k\_qt, s')$

(R4) $\exists s'' \notin S_o$, 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) $\exists s'$ s.t. $(s, stall_k\_qt, s')$ and s' is obtained by Rule R2.

(R6) $\exists s''$ 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 for the suffix and prefix functions.
Notation : $x \rightarrow x'$ means $\exists c \in C(I)$ and $(x, c, x') \in T$.
$\sigma = y \ldots y'$ is the path from y to y' such that $y \rightarrow y_0$, $y_0 \rightarrow y_1$, $\ldots$, $y_k \rightarrow y'$.

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 an 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. $x \to x'$, $\exists y' \in S_s$ s.t. $y \to y'$ 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: $xR_l\ y$ iff $pref(l,x) = pref(l,y)$. R_l is a weak bisimulation [3].

Proof: We have: $\forall x' \in S_o$ s.t. $x \to x'$, $\exists y' \in S_s$ 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. $y \to y'$ s.t : $x \to x'$ 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. ■

Property 3: Stuttering progression.
In M_o: We have $\sigma = s_0 s_1 \dots s_n$ 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 $\exists \sigma' = s_0^* s_1 \dots s_n$ 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). ■

B. Composition of Stall Increments

In section IV-A we described the behaviours of the stepped up machine when taking into account the delays induced by a unique stall. However, it is possible to have 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 order in which increments are added is important. 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 augmented 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')** $\exists s' \in S'_s$ 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)** $\forall l \in F \setminus \{k\}$, $\exists s'' \in S'_s$ 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) $suff(l, s'') = progress(0, suff(l, s))$.

Remark 3: 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 $\texttt{stall}_l$ is active, no matter $\texttt{stall}_k$ is also active, $\texttt{stall}_l$ freezes $pref(l,s)$, that encompasses $pref(k,s)$.

Property 4: 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 $pref(l,x) = pref(l,y)$.

1) R_l is a weak bisimulation.
2) $\forall\ j > l$, 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 $\texttt{stall}_l$ 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 $\texttt{stall}_k$ 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. ■

C. Kill Increment

A kill action destroys the treatment performed 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 VI 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".

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' = $c \wedge$ $\texttt{kill}_k\texttt{\_qt}$.
2) $\forall s \in S_s \cap S_k$, $\exists s' \in S_k$ and (s,$\texttt{kill}_k\texttt{\_act}$,$s'$) $\in T_k$ and s' is defined s. t. :
   x'_0 = 0 or 1, x_k = 0 and $\forall i \neq k$, x'_i = x_{i-1}.

V. 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 stall increments or a kill. 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.

A. Properties related to the inner parts of the pipeline

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 there negation) related to a stage k (resp. l) in M_s.

From [5], the general transformations capturing the preservation of the behaviours of M_s in M'_s due to any increment hold.

From the previous paragraphs, the following properties hold:

Property 5: Let f and g be any formula built from the following rules:

- $p = \phi_k \mid \phi_k \vee \phi_l \mid TRUE \mid FALSE$
- $f_p = A\ p\ U f_p \mid A\ f_p\ Up \mid E\ p\ U f_p \mid E\ f_p\ Up \mid AGp \mid EGp \mid AGf_p \mid EGf_p$
- $f = A\ f_p\ Uf \mid A\ f\ Uf_p \mid E\ f_p\ Uf \mid E\ f\ Uf_p \mid A\ f\ Ug \mid E\ f\ Ug \mid AGf \mid EGf \mid f \vee g \mid f \wedge g$

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 formula built from the rules:

- $p = \phi_k \mid \phi_k \wedge \phi_l \mid TRUE \mid FALSE$
- $f = Ap\, Uf \mid Af\, Up \mid Af\, Ug \mid Ep\, Uf \mid Ef\, Up \mid Ef\, Ug \mid f \vee g \mid f \wedge g$

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 $\varphi =$ OP $(\phi_k \wedge \phi_l)$ then M_s,s $\models \varphi \Rightarrow M'_s$,s $\models \varphi'$ and $\varphi' =$ OP $(AF(\phi_l) \wedge \phi_k)$

Proof: Direct consequence of properties 3and 4. ■

B. Properties related to the outer parts of the pipeline

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 ([5]), it is composed of the set of VCI commands and of VCI responses. In case of a processor, the pipeline environment is composed of instructions on the software visible registers plus the program counter, instruction and exception registers, and the memory.

The environment is abstracted 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: $\phi_{n-1,k}$ = $(x_{n-1} = 1 \wedge C_{n-1} = Cmd_k)$

Example 1: A causality relation between Cmd_k and Res_k , expressed on the environment as $Cmd_k \Rightarrow$ AF Res_k is transposed as: $\phi_0 \Rightarrow AF(\phi_{n-1} \wedge AFRes_k)$.

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 command does not produce a result. In case of occurrence of a similar command not concerned with the kill event, 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 :

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

Line (1) expresses that there is 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.

VI. Incremental design of the VCI-PI wrapper

In [5] we show how CTL property could be automatically transformed from a simple component in order to derive a part of the specification of a more complex one. Now, we want to take advantage of the increment particularity induced by the pipeline structure of the wrapper VCI-PI. In this part, we briefly recall the wrapper structure and then show how the formulae are transformed or preserved from properties of section V.

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 [8] and the considered wrapper is an adapter between the VCI interface and the PI-bus protocol [13]; hence we are able to

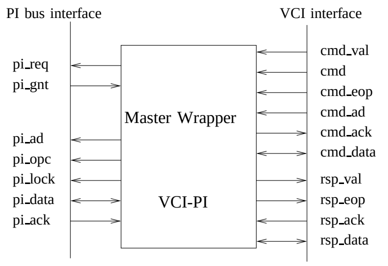

Fig. 2. VCI and PI interfaces of our set of master wrappers

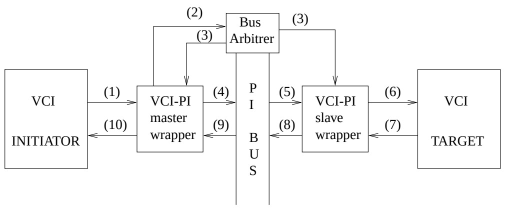

Fig. 3. The Platform performing the VCI-PI-VCI translation and illustration of a VCI transfer

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. Figure 2 illustrates the major signals handled by interfaces of a VCI-PI master wrapper.

A VCI transfer is shown in Figure 3. The VCI initiator sends a request to the VCI-PI-master-wrapper (1), that asks for the bus to the bus arbiter (2), and when the VCI-PI-master-wrapper owns the bus (3), it transfers each VCI request cell through the PI-bus to the VCI-PI-slave-wrapper (4,5). The VCI-PI-slave-wrapper translates the PI-cell into a VCI-cell to be given to the VCI target (6). The VCI-target transmits the VCI-response to the VCI-PI-slave-wrapper (7), which responds to the VCI-PI-master-wrapper through the PI bus (8,9). This latter translates the PI-response into a VCI-response and sends it to the VCI initiator (10). In some cases, the VCI-PI-slave-wrapper may implement a look-ahead mechanism in order to send the responses to the VCI-PI-master-wrapper in one cycle.

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 4.

The behavior of the simplest wrapper (model A) is a 3-stages pipeline, performing at the same time:

- accepting a VCI request k to be sent to PI from its VCI interface,
- sending the PI request corresponding to the $k-1^{th}$ VCI request on its PI interface,
- accepting the PI response to the $k-2^{th}$ VCI request on its PI interface.

Further models (B to C”) deal with external events disturbing the pipeline flow: either the k^{th} VCI request can not be given to the wrapper, or the $k-1^{th}$ response is delayed by the PI targets, or it says that a major problem occurred and the transaction has to be restarted later, or the $k-2^{th}$ response can not be returned to the VCI initiator; all these cases stall or break the pipeline flow. For instance, we build a model B from the model A that take into account delay impose by the target. The new behaviour added corresponds to a stall action. We obtained the model B by applying definition 8. The new event corresponding to this extension is $e = \langle (inc_b, \{0,1\}), \{0\}, \{1\}\rangle$ where $C_{QT} = \{0\}$ when $pi\_rsp = RDY$ and $C_{ACT} = \{1\}$ when $pi\_rsp = WAIT$.
Model B’ is obtained by applying two increments from model A in respect to definition 10 with the increment e defined above and the increment $e' = \langle (inc_{b'}, \{0,1\}), \{0\}, \{1\}\rangle$. The event e' represents the delay imposed by the master. The set of quiet configurations of e' is $C_{QT}(inc_{b'}) = \{0\}$. This configuration occurs when $(cmd\_val = 1) \wedge (rsp\_ack = 1)$. The set of active configurations is $C_{ACT}(inc_{b'}) = \{1\}$, that occurs when $(cmd\_val = 0) \vee (rsp\_ack = 0)$.

We implemented a platform as described in Figure 3 in synchronous Verilog. We checked about 80 CTL formulae for the master wrapper B, the slave wrapper B and the complete system (when the VCI initiator and target may generate delay events) with VIS verification tool [9] . Here are examples of CTL (untransformed) properties checked on the B platform. Formula 1 checks the interface between the VCI initiator and the wrapper: if 3 read cells are sent then the wrapper returns the response of the first cells and the acknowledgment of the third cells in the same cycle. Formula 2 checks the behavior of the complete system: the number of acknowledgment cells received by the VCI initiator is equal to the number of request cells it previously sent. Here, the initiator sends 2 requests.

```
# formula 1: #
AG( (cmd = READ_3_WORDS) ->
   AF( cmd_ack = 1 * rsp_val =1);
# formula 2: #
AG( (cmd = READ_2_WORDS) ->
 A ( (A (
(A((cmd = READ_2_WORDS * cmd_eop = 0 *
    cmd_val = 1)
 U  (cmd_ack = 1)))
      U ( A( (cmd_eop = 1 * cmd_val= 1)
  U (cmd_ack = 1)))))
 U (cmd_val = 0) ));
```

We fit the platform in order to plug a wrapper B’ obtained as described above. 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 V. 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.

Formula 1 contains a conjunction where atomics propositions concern stages 1 and 3. From B to B’ the increment may stall the stages 1 and 3, we apply Property 6. Formula 2 is

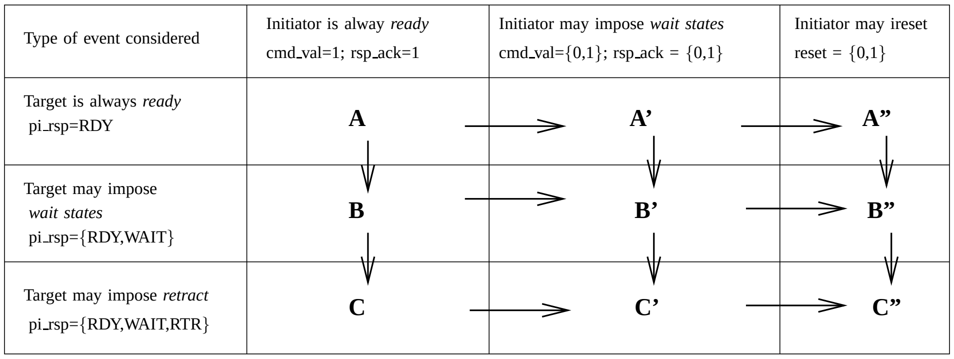

Fig. 4. Hierarchy of VCI-PI wrappers ranking from **A** to **C"**. Each arrow corresponds to an increment whose associated event is an extension of the definition domain of one or more signals.

a global property related to the outer part of the pipeline, by Property 7 it is preserved. The increment from B (resp. B') to C (resp C') corresponds to a new behaviour that breaks the pipeline flow. It's a Kill increment that kills stages 1 and 2. The Reset increment is also a kill increment but it kills all requests that were in the pipeline. In this case the formulae have to be transformed with the property stated in [5]. New 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.

VII. 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 state new CTL formulae transformations. 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.

This approach abstracts the control flow of a pipeline submitted to stall and kill actions. It will be interesting to use this abstraction instead of a real model in the development of complex synchronized pipeline flows such as superscalar architecture or complex protocol converter where several pipelines have to cooperate.

References

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

[2] R. Alur, T. A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. MOCHA: Modularity in Model Checking. In *CAV'98: Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525, London, UK, 1998. Springer-Verlag.

[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 - Proceedings of the 4th International Workshop on Automated Verification of Critical Systems*, volume 128 of *Electronic Note in Computer Science*, pages 263–278. Elsevier, 2004.

[6] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. 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.R. Burch and D.L. Dill. Automatic Verification of Pipelined Microprocessors Control. In D.L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV'94*, volume LNCS 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.

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

[9] 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.

[10] T.A. Henzinger, S. Qadeer, and S.K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451, London, UK, 1998. Springer-Verlag.

[11] R. Hosabettu, M. Srivas, and G. Gopalakrishnan. Decomposing the Proof of Correctness of Pipelined Microprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV'98*, volume 1427, pages 122–134, Vancouver, Canada, 1998. Springer-Verlag.

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

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

[14] D. Kroening and W.J. Paul. Automated Pipeline Design. In *DAC '01: Proceedings of the 38th conference on Design Automation*, pages 810–815. ACM Press, 2001.

[15] P. Manolios and S.K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB refinements. In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, pages 168–174, Washington, DC, USA, 2004. IEEE Computer Society.

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

[17] J. Sawada and W.A. Hunt. Trace Table Based Approach for Pipeline Microprocessor Verification. In *CAV'97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 364–375, London, UK, 1997. Springer-Verlag LNCS 1254.