Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 263–278
www.elsevier.com/locate/entcs

CTL-Property Transformations Along an Incremental Design Process

Cécile Braunstein[1] Emmanuelle Encrenaz[2]

LIP6
Université Pierre et Marie Curie, CNRS UMR 7606
Paris, France

Abstract

This paper formalizes an incremental approach to design flow-control oriented hardware devices described by Moore machines. The method is based on *successive additions* of new behaviors to a simple device in order to build a more complex one. The new behaviors added must not override the previous ones. A set of CTL formulae is assigned to each step of the design. The links between the formulae of two consecutive design steps are formalized as a set of formula-transformations F, stating that : a CTL formula f is satisfied on a design at step i, iff $F(f)$ is satisfied on the design extended at step $i+1$. This result has been applied during the design of bus protocol converters in the context on non-regression analysis. It could also be applied in order to simplify both system and formulae in particular cases.

Keywords: System Design and Verification, Simulation Relation, Computational Tree Logic.

1 Introduction

This paper stems from the observation of the way some hardware components may be designed. In some cases, hardware designers could adopt an *incremental strategy*: after having defined the information flow of the design, the rough structure of the data-path and the control part, they proceed to the implementation of the simplest cases up to the most complex ones. This is accomplished by *adding* new functionalities to already existing ones, building

[1] Email: cecile.braunstein@lip6.fr
[2] Email: emmanuelle.encrenaz@lip6.fr

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2005.04.016

a more and more complex device. This is particularly true for devices implementing a pipe-line flow: the stages of the pipe-line can be roughly drawn and then the stalling actions are added (e.g. stalling actions due to cache-miss, register dependencies between successive instructions, or exception handling).

Often, the verification of such devices is performed by simulation of test cases. Specification of components by means of a list of properties expressed as CTL formulae and verification by symbolic model-checking [2] emerges as a verification method complementary to simulation. For instance a bus protocol can be expressed by means of CTL formulae, and a new design conformable to the protocol, may be checked by plugging the design under test into a verification environment mimicking the bus, and then check that all properties of the specification are verified.

In general, the incremental design approach does not preserve the set of properties from a simple component to a more complex one. Once a behavior is added to an initial model, a global property, which was true in the initial model, may be wrong in the extended one. Consequently, local and global properties (about the component plugged in a complex environment made of other components) have to be re-adapted for each incremental step of the design.

This incremental design process is complementary to those applying a *refining strategy* as in [11]. In refining strategies, the global information flow is initially defined, and all cases, the simplest ones up to the most complex ones, are obtained by incremental refinements of the initial model. Each refinement is considered as a step towards a real implementation. The strength of these approaches resides in the preservation of global properties along the refinement process: if a property is true in a given model, then, if the refinement is well-defined, the refined model will preserve the initial property. This induces a design method ensuring that the implementation respects the properties of the specification. But this refining strategy excludes the addition of new functionalities during the design process: a refinement is a *specialization* of a pre-defined set of behaviors, whereas the incremental method is built on *addition* of new behaviors. In our case, the price to pay is the lack of property-preservation.

The way several increments interfere with each other has been extensively studied in the context of detecting features inconsistencies in telecommunication or software plug-ins. For instance, Plath and Ryan have proposed in [15,16,4] a feature integration automating tool coupled with model-checkers ([16] for Promela/SPIN [10], [15] for SMV [13] and [4] for MOCHA [1]). The inconsistencies between several added features are detected by LTL or CTL property violation. More recently in [4] they stated a ATL-property preserva-

tion for a restricted type of feature. Others, as Cansell and Mery in [3] have proposed a method to compose features integrated into the Atelier B tool [5]: here, the refining strategy is applied to guarantee the correctness of the implementation with respect to an abstract specification of the basic component plus the added services. The inconsistencies between services appear as non provable proof-obligations.

Our purpose differs from those described in [3,15,16,4] since our increment definition is much simpler than the feature integration they propose. As a consequence, our increment *is* monotonic: there is no overriding of behaviors, all behaviors that were in the simple component are preserved in the more complex one. But the CTL formulae that were true in the simple model may be falsified by the increment. Our goal is to build a set of CTL formulae that represents a specification for the complex component by re-using the specification of the simpler component (and adding new properties specific to the added behaviors).

We are interested in exploring the links between properties that are true in an initial model and those that are true in the extended one. This might be expressed as: "May we transform the CTL formulae that are true in the initial model into other CTL properties that are true in the extended one, capturing the way the extension was performed ?" If we can perform this, we can insure that the extended model preserves the behaviors that were checked on the initial one. Conversely, from some property satisfied on the complex model, can we derive a simpler property to verify on the simpler one?

Given an additive increment, the initial model and the extended one, we show that this CTL transformation is possible. The transformed CTL formulae, applied to the extended model, restrict the verification state-space traversals to a sub-graph isomorphic to the one derived from the initial model. This guarantees that, if the extended model respects the extension rules, then the verification results of the transformed CTL formulae, applied to the extended model, and the verification of the initial CTL formulae, applied to the initial model, are identical.

The paper is organized as follows: In a first section we present how a new behavior is added to an existing model. The second section presents the Kripke structures derived from an initial model and the extended model, and characterizes the main properties of the latter. From these considerations, the third section presents a set of transformations of CTL formulae, restricting the verification of CTL formulae in the Kripke structure of the extended model to the Kripke structure of the initial one it includes. Then we present, in the fourth section, the way these transformations were applied during the incremental design process of protocol converters — between VCI (Virtual

Component Interface) [6] and PI-bus [14] — and finally in the fifth section we draw some conclusions.

2 Increment formalization

In this section, we formalize the component being designed and the increment. Then we characterize the extended component.

A component is viewed as a control part driving a data-path. Its state-space is modelled by a complete and deterministic synchronous Moore machine. The component presents an interface made of directed typed signals.

2.1 Definitions of a signal and a configuration

Definition 2.1 Each signal is defined by a name s and a finite definition domain $Dom(s)$.

Definition 2.2 Let E be a set of signals. A configuration $c(E)$ is a vector that associates to each signal in E one value of its definition domain. The set of all configurations $c(E)$ is named $C(E)$.

2.2 Definition of a component

Our approach iteratively applies an increment to a component W to build a more complex component where W_i refers to the component resulting from i successive increments.

Definition 2.3 A component $W_i = < S_i, I_i, O_i, T_i, L_i, s_i >$ is described as a deterministic and complete Moore machine:

S_i: Finite set of states.

I_i: Finite set of input signals with their finite definition domain: $\{(sig_{in}, Dom_i(sig_{in}))\}$

O_i: Finite set of output signals with their finite definition domain: $\{(sig_{out}, Dom_i(sig_{out}))\}$

T_i: Finite set of transitions $\subseteq S_i \times C(I_i) \times S_i$ satisfying $\forall s \in S_i$, $\forall c \in C(I_i)$, $\exists! s' \in S_i$ s.t. $(s, c, s') \in T_i$ ($\exists!$ means "there exists exactly one").

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

$s_i \in S_i$: the initial state

Remark 2.4 Applying the vector of generation functions to a given state of S_i produces a configuration $c(O_i)$.

2.3 Increment

An increment is a set of modifications applied to a component's architecture in order to build a more complex component. It reflects the carrying out of a new event at the component's interface. The architecture of W_i does not consider the occurrence of this new event, while the architecture W_{i+1} does. An event is an input configuration which may produce a state change of the system. An increment is a new event (or a set of new events), introducing new behaviors. A new event can occur of different manners:

- Either the definition domain of one or more existing signals are extended. The interfaces of the component are fixed, but the incremental design process takes into account values of these interfaces that were not previously considered.
- Or one or more new signals are added (with a definition domain). This is the case of an increasing complexity of the data-path of the component.

In both cases, the new event is modelled by the appearance of a new symbol `e` in the set of input signals I_{i+1}, with a definition domain `Dom(e)`. This domain is split into two disjoint sets: the quiet values set (`V_QT(e)`) grouping the configurations of the extended or added signals meaning that the new event has not occurred, the active values set (`V_ACT(e)`) grouping the configurations of the extended or added signals, meaning that the new event occurred.
The denomination `e = val_qt` means that the extended signals present a configuration belonging to `V_QT(e)`, respectively `e = val_act` belonging to `V_ACT(e)`.

The new event introduces new transitions and states: the new behaviors are represented as new transitions and states in the previous Moore machine. Completeness and determinism of the Moore machine are preserved.

The new event drives new actions: either one or more existing output signals have their domain extended, or one or more new output signals are created.

As previously done, these modifications are modelled by the appearance of a new output signal `o`, with a definition domain `Dom(o)`, split into `V_QT(o)` and `V_ACT(o)` disjoint sets. The new signal `o` is added into the states existing in the older Moore machine, driving one of its quiet values, and it appears in fresh states, driving either one of its quiet or active values. Figure 1 presents an admissible increment. On the left, the Moore machine describing a component W_i contai'= ns three states and the input signal `k` drives the transitions. On

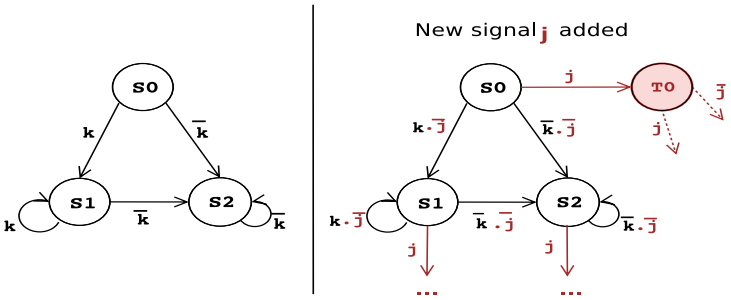

Fig. 1. Increment example

the right, the component W_{i+1} resulting from the increment of W_i with an additional input signal j and $Dom(j) = \{0, 1\}$; its quiet value is 0 and its active value is 1. In W_{i+1}, the quiet value of j labels all transitions that were in W_i; new transitions leaving from states that existed in W_i are labelled with the active value of j; new states are the source of transitions labelled with either the active or quiet value of j.

Definition 2.5 An increment from a component W_i is a 4-tuple $INC = < I_+, R_+, O_+, \Sigma_+ >$

Σ_+: The set of new reachable states where $\Sigma_+ \cap S_i = \emptyset$

R_+: The set of new transitions $\subseteq (S_i \times S_i) \cup (S_i \times \Sigma_+) \cup (\Sigma_+ \times \Sigma_+) \cup (\Sigma_+ \times S_i)$ where $R_+ \cap T_i = \emptyset$.

I_+: The set of new input signals and their definition domain = $\{(sig_{in_+}, Dom(sig_{in_+}))\}$. We name `e` the event associated to I_+, `e` = $\{sig_{in_+}\}$, and $C(e)$ = `V_QT(e)` $\cup$ `V_ACT(e)` such that:

- each transition (s_1, c, s_2) in T_i will have its input configuration extended with the sub-configuration concerning the new input signals belonging to the `V_QT(e)`, denoted by $c' = c$ where `e = val_qt`.
- each transition (s_1, c, s_2) in $R_+ \cap (S_i \times \Sigma_+ \cup S_i \times S_i)$ will have its input configuration extended with the sub-configuration concerning the new input signals belonging to `V_ACT (e)`, denoted by $c' = c$ where `e = val_act`.

O_+: The set of new output signals and their definition domain = (o,Dom(o)) with: Dom(o) = `V_QT(o)` $\cup$ `V_ACT(o)` such that the output function associated to `o` returns a value in `V_QT(o)` for all states that were in W_i.

A component $W_{i+1} = < S_{i+1}, I_{i+1}, O_{i+1}, T_{i+1}, L_{i+1}, s_i >$ obtained by applying an increment to a component W_i preserves all behaviors that were present in W_i, assuming that, in W_{i+1}, the new event is maintained to one of

its quiet values. We have $S_{i+1} = S_i \cup \Sigma_+$, $I_{i+1} = I_i \cup i_+$, $O_{i+1} = O_i \cup O_+$, $T_{i+1} = T_i \cup R_+$, and L_{i+1} conforms to the restriction imposed by O_+.

Proposition 2.6 *The initial state in W_{i+1} simulates the initial state in W_i.*

Proof (Sketch) : We build ρ_W a binary relation between the states of two consecutive components W_i and W_{i+1}, such that $\rho_W \subseteq S_i \times S_{i+1}$ and $\forall s \in S_i$, $(s, s') \in \rho_W$ if s' is the name of s in S_{i+1}. By construction, ρ_W is a simulation relation.

3 Translation of Moore machine into Kripke structure

The semantics of CTL formulae is defined on the Kripke structure derived from the initial Moore machine describing the component W_i. Informally, the input configurations that label the transitions in the Moore machine are incorporated into states in the Kripke structure. We formally define the Kripke structure $K(W_i)$ obtained from the component W_i.

Definition 3.1 A Kripke structure is a 5-tuple $\langle S, s_0, AP, \mathcal{L}, R\rangle$ where:

S is a finite set of states,

$s_0 \subseteq S$ is the set of initial states,

AP is a finite set of atomic propositions,

$\mathcal{L} = \{l_0, \ldots, l_{|AP|-1}\}$ is a vector of |AP| functions, each function defining the value of exactly one atomic proposition; for all $0 \leq i \leq$ |AP| we have $l_i : S \to \mathbb{B}$; for all $s \in S$, we have that $l_i(s)$ is true iff the atomic proposition associated to l_i is true in s,

$R \subseteq S \times S$ is the transition relation.

Definition 3.2 Given a component $W_i = < S_i, I_i, O_i, T_i, L_i, s_i >$, we deduce the Kripke structure $K(W_i) = \langle S_{K(W_i)}, s_{K(W_i),0}, AP_{K(W_i)}, \mathcal{L}_{K(W_i)}, R_{K(W_i)}\rangle$ where:

$S_{K(W_i)} = S_i \times C(I_i)$,

$s_{K(W_i),0} = \{s_i\} \times C(I_i)$,

$AP_{K(W_i)} = I_i \cup O_i$,

$\mathcal{L}_{K(W_i)} = \{l_{O0}, \ldots, l_{O|O_i|-1}\} \cdot \{l_{I0}, \ldots, l_{I|I_i|-1}\}$;

$R_{K(W_i)} \subseteq S_{K(W_i)} \times S_{K(W_i)}$ and $\forall\, (s, c_i) \in S_{K(W_i)}, \forall\, (s', c_i') \in S_{K(W_i)}$, we have $((s, c_i), (s', c_i')) \in R_{K(W_i)}$ iff $(s, c_i, s') \in R$.

Applying an increment to a component W_i produces a component W_{i+1}. The CTL formulae associated to W_i are verified over the Kripke structure

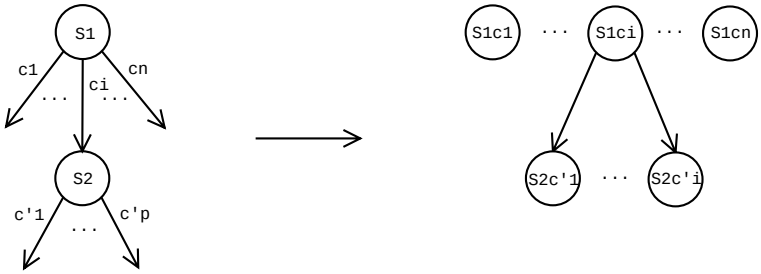

Fig. 2. Transformation of a Moore machine into a Kripke structure

$K(W_i)$. One can derive a Kripke structure $K(W_{i+1})$ from the component W_{i+1} by applying Definition 3.2. We are now interested in characterizing the properties of $K(W_{i+1})$ with respect to $K(W_i)$, namely to show that $K(W_i)$ is included into $K(W_{i+1})$, with all states that were present in $K(W_i)$ tagged with the quiet value of the new event.

3.1 Properties of $K(W_{i+1})$

By construction, the tree of behaviors of $K(W_i)$ is preserved in $K(W_{i+1})$, labelled with some quiet values of the new event. This preservation property can be expressed as the existence of a simulation relation between the states of the Kripke structures obtained from two consecutive components. More precisely, the *enrichment relation* captures the fact that the behaviors of the previous component are enclosed in the newer one, tagged with the event assigned to some of its quiet values.

Definition 3.3 Enrichment For all states $s_i = (s, c) \in K(W_i)$, let there be $s'_i = (s', c')$ and $s''_i = (s'', c'') \in K(W_{i+1})$ such that:
$s' = s, c' = c$ if `e = val_qt` and $s'' = s, c'' = c$ if `e = val_act`.
Then s'_i and $s"_i$ are said to *enrich* s_i (with (`e = val_qt`) in the first case).

Proposition 3.4 *Each initial state of $K(W_{i+1})$ that enriches the initial state of $K(W_i)$ with ($e =$ `val_qt`) simulates the latter.*

Proof (Sketch) : We define $\rho_{K_W} \subseteq S_{K(W_i)} \times S_{K(W_{i+1})}$, such that for all $s \in S_{K(W_i)}$, $c \in C(I_i)$, $s' \in S_{K(W_{i+1})}$, and $c' \in C(I_{i+1})$, we have $(s, s') \in \rho_{K_W}$ iff ($s = s'$ and $c' = c$) where `e = val_qt`. By construction, ρ_{K_W} is a simulation relation.

Remark 3.5 From the above, we obtain:
If s' enriches s with `e = val_qt`, then s' simulates s.
If s' enriches s with `e = val_act`, this does not imply that s' simulates s.
If s' simulates s, this does not imply that s' enriches s.

Corollaries

(i) If there exists some infinite path in $K(W_i)$, then there exists some infinite path in $K(W_{i+1})$ along which the event e has always one of its quiet values. If $\sigma = s_0 \dots s_n \dots$ is in K(W_i), then $\exists \sigma' = s'_0 \dots s'_n \dots$ in K(W_{i+1}) such that all s'_i enriches s_i with (`e = val_qt`).

(ii) $K(W_i)$ is the maximal sub-graph in K(W_{i+1}), reachable from s'_0, that enriches s_0 (in $K(W_i)$) with `(e = val_qt)` when e remains in one of its quiet values.

(iii) The states in $K(W_{i+1})$ obtained by the expansion of a state in Σ_+ are only reachable from the initial state $s'_{K(W_{i+1}),0}$ that enriches $s_{K(W_i),0}$ by a path along which *at least* one state is labelled by (`e = val_act`).

(iv) If $s' \in K(W_{i+1})$ enriches $s \in K(W_i)$ with (`e = val_qt`), then for all $t' \in K(W_{i+1})$ such that $s' \rightarrow t'$, there exists $t \in K(W_i)$ such that t' is produced by the expansion of t due to the increment, and $s \rightarrow t$.

Proof.

(i) By induction on the length of σ'.

(ii) By construction of K(W_{i+1}), we have s'_0 enriches s_0, hence s'_0 simulates s_0. Due to corollary (i) there exists a path $\sigma' = s'_0 \dots s'_n \dots$ such that all states enriches a state in K(W_i). For each state $r' \in \sigma'$, r' enriches a state r in K(W_i) with (`e = val_qt`), let t' such that $r' \rightarrow t'$ and t' satisfies (`e = val_act`), then t' does not belong to $K(W_i)$.

(iii) From corollary (ii).

(iv) Let $s' \in K(W_{i+1})$ enriche $s \in K(W_i)$ with (`e = val_qt`) and $s' \rightarrow t'$. Assume t' is *not* obtained by expansion due to the increment of a state t in $K(W_i)$. By construction, t' is produced by the expansion of a state r in W_{i+1} reached by a transition labelled by (`e = val_act`). Contradiction.

Hence, $K(W_{i+1})$ includes $K(W_i)$ and $K(W_i)$ can be detected in $K(W_{i+1})$ since it is the maximal connected sub-graph tagged with (`e = val_qt`). This is captured by the enrichment relation that is included in a simulation. We now use this particularity to establish links between CTL formulae verified on $K(W_i)$ and some others verified on $K(W_{i+1})$.

4 CTL-formulae transformations

[12] and [8] have stated some CTL formulae-preservation results between two Kripke structures ordered by any simulation relation. We recall their results in our particular context.

In [12] the authors state the preservation of ECTL [3] formulae from K(W_i) to K(W_{i+1}), while in [8] the authors state the preservation of ACTL [4] formulae from K(W_{i+1}) to K(W_i)

The results we present are not based on the preservation of a fragment of CTL between a component and another one that includes it, but rather transform the whole CTL operators and provide a bi-implication between the initial formula and the transformed one.

Given a CTL formula Φ, we are going to set out the rules to transform Φ that is true in $s_{K(W_i),0}$ (named in short s_0) into Φ' that is true in $s'_{K(W_{i+1}),0}$ (shortly named s'_0) when s'_0 enriches s_0 with (`e = val_qt`).

Theorem 4.1 *Let be $s \in S_{K(W_i)}$ and s' $\in S_{K(W_{i+1})}$ such that s' enriches s with* (`e = val_qt`)*, for any atomic proposition $p \in AP_{K(W_i)}$, for any CTL formulas Φ, χ and Ψ (with all their atomic propositions in $AP_{K(W_i)}$), $s \models \Phi \Leftrightarrow s' \models \Phi'$, where Φ' is the formula obtained by recursively applying the following transformations:*

$$
\begin{array}{ll}
\Phi = p & \Leftrightarrow \Phi' = p. \\
\Phi = not\Psi & \Leftrightarrow \Phi' = not\Psi'. \\
\Phi = EX\Psi & \Leftrightarrow \Phi' = (e = val\_qt) \Rightarrow EX\Psi'. \\
\Phi = EF\Psi & \Leftrightarrow \Phi' = E((e = val\_qt)U\Psi'). \\
\Phi = EG\Psi & \Leftrightarrow \Phi' = EG((e = val\_qt) \wedge \Psi'). \\
\Phi = E\Psi U\chi & \Leftrightarrow \Phi' = E(((e = val\_qt) \wedge \Psi')U\chi'). \\
\Phi = AX\Psi & \Leftrightarrow \Phi' = (e = val\_qt) \Rightarrow AX\Psi'. \\
\Phi = AF\Psi & \Leftrightarrow \Phi' = AF((e \neq val\_qt) \vee \Psi'). \\
\Phi = A\Psi U\chi & \Leftrightarrow \Phi' = A(((e = val\_qt) \wedge \Psi')U((e \neq val\_qt) \vee \chi')). \\
\Phi = AG\Psi & \Leftrightarrow \Phi' = A(((e = val\_qt) \wedge \Psi')W(e \neq val\_qt)). \\
\Phi = A\Psi W\chi & \Leftrightarrow \Phi' = A(\Psi' W(\chi' \vee (e \neq val\_qt))).
\end{array}
$$

W stands for the "Weak until" operator.

Proof (Sketch) : The transformations are based on the reduction of the computational tree explored in K(W_{i+1}) to the sub-tree along which the active values of the event are not considered. By corollary (ii), this sub-graph represents K(W_i). The transformation is proven for each CTL operator applied to an atomic proposition by including the (`e = val_qt`) constraint in its definition. Then the proof proceeds by induction on the length of the formula Φ.

The transformations listed above do *not* modify the structure of the initial formula: the imbrication of temporal operators is preserved, hence the size of

[3] ECTL stands for CTL restricted to the Existential modalities.
[4] ACTL stands for CTL restricted to the Universal modalities.

the CTL formula (measured as the number of imbricated temporal operators) is unchanged. The transformation of an EF into an EU or an AG into an AW does not significantly change the complexity of the verification since they are based on the same fixpoint computation. The transformation is transferred into the propositional operations that are performed by classical BDD binary operations (`and, or, implies, ...`).

We implemented a tool that automates the transformation of the CTL formulae described in Theorem 4.1. This tool takes a file with a set of CTL formulae and a file containing the definition of an increment and returns the set of transformed CTL formulae.

5 Experiment

We experimented with the automatic construction of a part of the specification of a complex system by transforming the specification of a simpler one in the context of VCI-PI wrappers.

A wrapper is a device wrapping around an IP-core and implementing a given interface. In our context, the IP-core is supposed to be VCI compliant [6] 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.

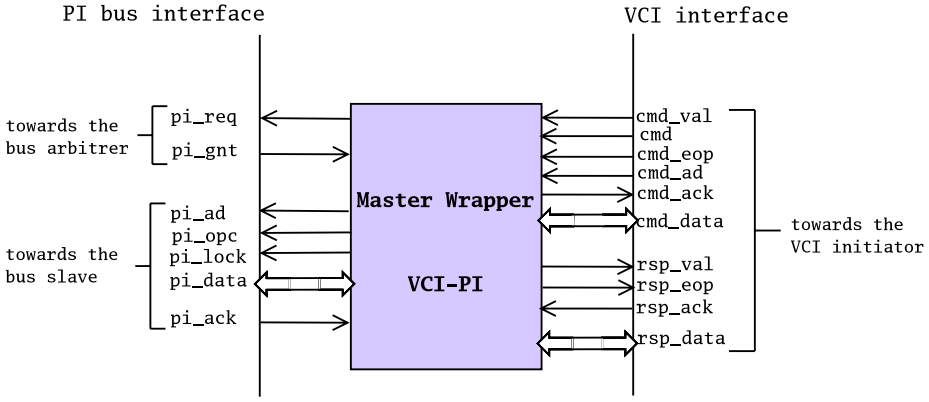

Fig. 3. Master Wrapper VCI and PI interfaces

The 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* functionality. Figure 3 illustrates the major signal interfaces a VCI-PI master wrapper has to deal with.

A VCI transfer is shown in Figure 4. 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.

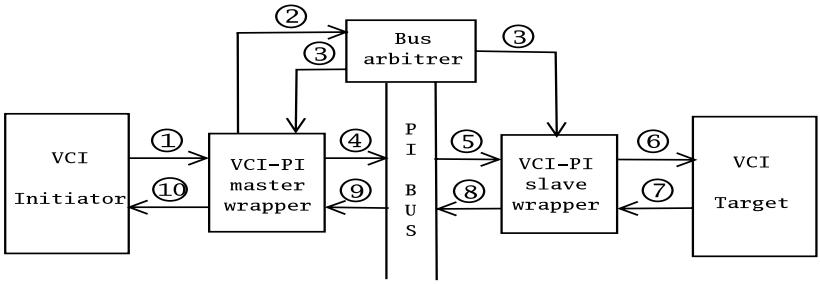

Fig. 4. Platform and VCI transfer

Using the incremental design process approach, we developed a set of six master VCI-PI wrappers, from a very simple one supposing that the VCI initiator and the PI target will always respond in one cycle, up to the most complex one supporting delays and retract events sent by the VCI initiator or the PI target. The hierarchy of the 6 master wrappers is shown in Figure 5.

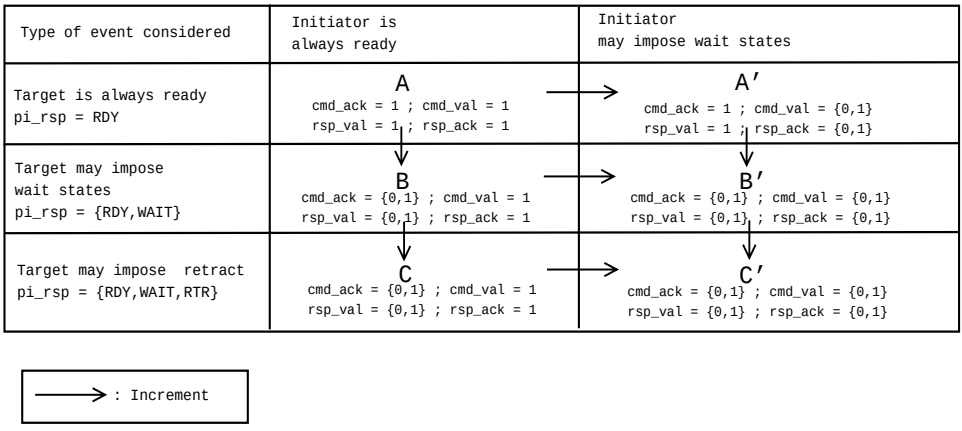

Fig. 5. Hierarchical VCI-PI wrapper. Each event added corresponds to an extension of the definition domain of one or more signals.

The behavior of the simplest wrapper (model A) is a 3-stages pipeline,

performing at the same time:

- 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.

The 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 freeze the pipeline.

The incremental data-path of the six master wrappers is presented in Figure 6, showing the behaviors successively added by increments ranking from A to C'.

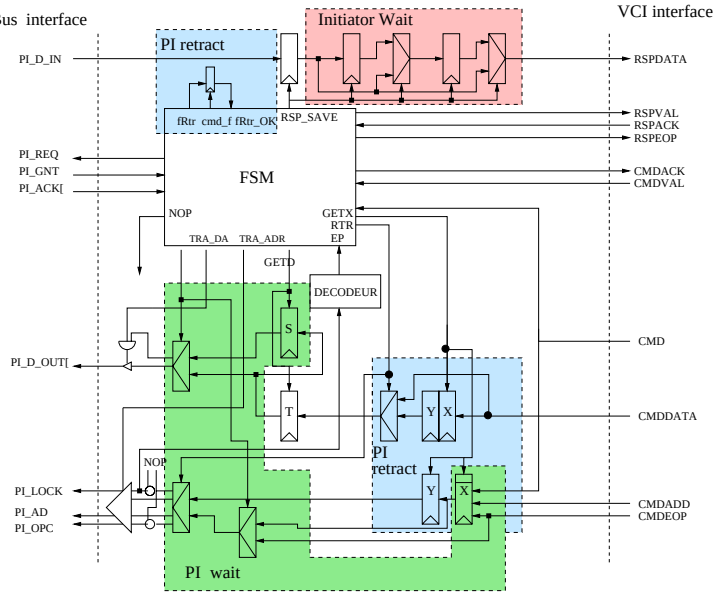

Fig. 6. Data-path of master wrapper C'. Each area corresponds to a different added increment.

We implemented in synchronous Verilog a platform as described in Figure 4. We verified this system with the VIS verification tool [7]. We checked about 80 CTL properties for the master wrapper B, the slave wrapper B and the complete system (when the VCI initiator and target may generate delay events).

Here are examples of CTL properties checked on the B platform:

```
# Check the interface between the PI bus arbiter and the master
wrapper. # property 1: # AG ( (wrap0.state = R_REQ) -> (A(
(m_pi_req = 1) U (m_pi_gnt = 1))));

# Check the behavior of the slave wrapper (its two automata # are
well synchronized). # property 2: #
!EF((wrap_cible.cmd_cible.state = CMD_IDLE) *
    !(wrap_cible.rsp_cible.state = RSP_IDLE));

# Check the behavior of the complete system: check that 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. # property 3: # AG( (m_cmd_plen[6:0] =
8 * m_cmd[0] = 1 ) ->
 A ( (A (
(A( (m_cmd_plen[6:0] = 8 * m_cmd[0] = 1 * m_cmd_eop = 0 *
m_cmd_val = 1)
 U  (m_cmd_ack = 1)))
            U    ( A( (m_cmd_eop = 1 * m_cmd_val=1)
                          U (m_cmd_ack = 1)))))
 U (m_cmd_val = 0) ));
```

We applied the transformations described in theorem 1 on the 80 CTL properties of the model B with the increment transforming B into B', and verified them on a system containing now B' VCI-PI master and slave. The verification results were successful to the expense of an increasing verification time mainly due to the increasing complexity of the system under verification. Of course, extra CTL formulae had to be added to the B'-platform in order to check the behaviors added by the increment.

For a small-size system (platforms B and B' with one master wrapper and one slave wrapper), the overall verification time is increased for the complex model but most of this time is consumed during the reachable state-space construction (5s vs 40s). The property verification extra-cost is of the same order of magnitude for both platforms (3s vs 9 s). These results are confirmed for the medium size systems (platforms B and B' with two master wrappers and one slave wrapper) where the gap between the B and B' verification time is mostly due to the increasing complexity of the system, rather than the complexity of the formula, since most of the verification time is spent during the reachable state-space construction (25s vs 4h). Once the reachable state-space is built, the verification of each property is performed in 10s for the B platform vs 20 up to 50s for the B' platform.

6 Concluding Remarks

The transformation rules of CTL formulae we propose are the basis for an approach to automatically derive part of the specification of a component, from the specification of the simpler component is comes from.

We have shown this approach can be used during the design of a concrete component, assuming the increment respects the rules we formalized, as we take advantage of the existence of a particular value tagging the initial part of a model included in an extended model. The transformed CTL formulae have the same complexity (in terms of CTL imbricated operators) as the initial CTL formula. This is confirmed by experimental results showing that the increasing time of the verification of the complex system is mainly due to the reachability analysis instead of the CTL formula verification.

It is our intention to pursue this study towards the following directions:

- Up to now, we did not take into account all the particularities of the increment; we considered only the existence of a particular event splitting the set of states with the ones that appeared in the initial model and the new ones (this event may be due to the extension of existing signal domains and/or to the addition of new signals). We did not take advantage of the graph structure of the increment; most of the time, this increment consists of the adding of a new state (or set of states) characterizing the freezing of the data-path waiting for some `continue` signal to be set, allowing the data-path to pursue. In these cases, a new set of CTL transformations may be defined, capturing the added behaviors.
- The opposite analysis can also be of interest: given a formula to be verified on a complex model, can we find an increment (in the sense we defined in this paper) such that the complex model has been built from the application of this increment to a simpler model. If yes, can we transpose the formula of the complex model to a simpler one to be verified on the simpler model? The verification would be partial since it would not apply on the whole set of behaviors of the complex system, but could give some information if the complex system is too big to be verified with classical model-checking tools.
- We are also interested in studying the way this approach could be mixed with an Assume-Guarantee verification process [9].

References

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, Shaz Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525, 1998.

[2] J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2):153–181, 1992.

[3] D. Cansell and D. Méry. Abstraction and refinement of features. In S. Gilmore and M. Ryan, editors, *Language Constructs for Designing Features*. Springer, 2000.

[4] F. Cassez, M. Ryan, and P-Y. Schobbens. Proving feature non-interaction with alternating-time temporal logic. In S. Gilmore and M. Ryan, editors, *Language Constructs for Describing Features*, pages 85–104. Springer Verlag London Ltd, 2001.

[5] STERIA Technologie de l'information Aix-en Provence (F). In *Atelier B, Manuel utilisateur*, version 3.5, 1998.

[6] On-Chip Bus Development Working Group. VSI Alliance - Virtual Component Interface Standard (VCI). version 2, 2000.

[7] 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.

[8] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 250–263. Springer Verlag, 1991.

[9] 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. Springer-Verlag, 1998.

[10] G. J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[11] K. Lano. In *The B Language and Method, A guide to practical Formal Development*. Springer-Verlag, 1996.

[12] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. volume 6 of *Formal Methods in System Design*, pages 1–35. Kluwer, 1995.

[13] K. L. McMillan. In *Symbolic Model Checking*. Kluwer Academics Publishers, 1993.

[14] European project Open Microprocessor System Initiative (OMI). Pi-bus standard specification (OMI 324), 1994.

[15] M. C. Plath and M. D. Ryan. A feature construct for promela. In *SPIN'98*, 1998.

[16] M. C. Plath and M. D. Ryan. SFI: a feature integration tool. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 201–216. Springer, 1999.