**Software Tools for Technology Transfer manuscript No.**
(will be inserted by the editor)

# CTL-Property transformations along an incremental design process

**Cécile Braunstein and Emmanuelle Encrenaz**

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

Received: date / Revised version: date

**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 behaviours to a simple device in order to build a more complex one. The new behaviours 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: for all CTL formula $f$ with atomic propositions related to step $i$, $f$ is satisfied on a design at step $i$, *iff* $F(f)$ is satisfied on the design extended at step *i+1*. This result extends the classical CTL property preservation results in a particular context. Moreover, it simplifies the writing of properties for a new device. This approach has been applied in the design of bus protocol converters and the transformations were useful to perform non-regression analysis. It could also be applied in order to simplify both system and formulae in particular cases.

**Key words:** 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 adopt an *incremental strategy*: after having defined the information flow of the design, the rough structure of the data paths 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, thus building a more and more complex device. This is particularly true for devices implementing a pipe-line flow: stages of the pipe-line can be roughly drawn and then the stalling actions are added. We believe this incremental approach provides a framework that simplifies the design process, by treating difficulties one by one instead of having to face them altogether.

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] emerge 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 it into a verification environment mimicking the bus, and then check that all properties of the specification are verified. In particular, verification of a single component in isolation by means of CTL or LTL specification is commonly used in the assume-guarantee verification process [16, 10].

In a general way, the incremental design approach does not preserve the set of properties from a simple component to a more complex one. Once a behaviour is added to a simple model, a global property, which was true in the simple model, may be wrong in the extended one. Consequently, local and global properties (about the component plugged into a complex environment consisting of other components) have to be re-adapted for each incremental step of the design. The method we propose overcomes this limitation : the properties satisfied in the simple model are transformed into others satisfied in the complex model. The limit of the model's complexity is related to the use of symbolic model-checking: each component has to be passed into the model-checker (this is sufficient to perform assume-guarantee verification) and if one needs to verify a global property among several components, the complete system has to fit into the model-checker. In practice, this corresponds to medium-sized systems (reducible to about 10K logical gates).

This incremental design process is complementary to those applying a *refining strategy* as in the B method [12]. 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 design method ensures 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 behaviours, whereas the incremental method we propose is built on *addition* of new behaviours. In our case, the price to pay is the lack of property-preservation. Larsen and Thomsen [13] proposed another refining method based on Modal Transition Systems (MTS) which allows partially definedi specifications. A MTS can be refined by preserving at least all the *must-transitions* and maybe adding some, while eliminating some *may-transitions*. But any behaviours allowed in the implementation has to be allowed in the specification.

The way several increments interfere with each other has been extensively studied in the context of feature inconsistencies detection in telecommunication or software plug-ins. For instance, Plath and Ryan have proposed in [17,4] a feature integration automating tool coupled with model-checkers ([17] for SMV [15]). The inconsistencies between several added features are detected by LTL or CTL property violation. More recently in [4], Ryan, Cassez and Schobbens have stated a ATL-property (Alternating-time Temporal Logic[1]) preservation for a restricted type of feature. This is the transposition of the classical results of CTL-property preservation [9] into the context of ATL formulae, well-adapted to systems described in the game theory. They also integrate this feature inconsistency detection with ATL in MOCHA [1]. Others, as Cansell and Mery in [3] have proposed a method to compose features integrated into the Atelier B tool [6]: 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 [3,17,4] since our increment definition is much simpler than the feature integration they propose. Indeed, our increment *is* monotonic: there is no overriding of behaviours, all behaviours that were in the simple component are preserved in the more complex one and new behaviours are tagged with a particular value, thus one can recognize them. Hence property-preservation results we propose are stronger.

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 behaviours that were checked on the initial one. Conversely, from some property satisfied on the complex model, we can derive a simpler property to verify on the simpler one. 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 behaviours).

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 traversal to a sub-graph isomorphic to the one derived from the initial model. This guarantees that, if the extended model respects the extension rules defined in section 2, 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.

We show how these transformations can be used to perform non-regression analysis. In a general way, when a designer modifies a component, he has to insure that the modification did not induce *regression*: the correction does not disturb other correct functionalities. This is generally done by simulation: one has to re-simulate all the test cases that were successful on the previous model on the corrected one. By applying our approach, if the correction is manually performed, we can automatically derive the set of CTL formulae that will represent the specification of the corrected component. These properties will have to be verified on the complex system. If the correction is automatically performed by applying a pre-defined increment, the non-regression is guarantied *by construction*.

The paper is organized as follows: In the second section we present how new behaviour is added to an existing model and the associated definitions. The third section presents the Kripke structures derived from an initial model and the extended model, and characterizes the main properties of the latter. The fourth 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 initial one it includes. Then we present, in the fifth section, the way these transformations were applied during the incremental design process of protocol converters (between VCI (Virtual Component Interface) [7] and PI-bus [11]). Finally in the sixth section we draw some conclusions.

[1] ATL is a branching logic used to model systems evolution controlled by a set of agents, that may affect the future by making *choices*.

## 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 modeled 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 1.** Each *signal* is defined by a variable name $s$ and an associated finite definition domain *Dom(s)* of possible value.

**Definition 2.** Let $E$ be a set of signals $E = \{s_1, \ldots, s_n\}$. A *configuration c(E)* is a conjunction of the associations: for each signal in $E$, one associates one value of its definition domain. The *set of all configurations c(E)*, named *C(E)* is $Dom(s_1) \times Dom(s_2) \times \ldots \times Dom(s_n)$.

### *2.2 Definition of a component*

Our approach iteratively applies an increment to a component $W_0$ to build a more complex component. $W_i$ refers to the component resulting from $i$ successive increments.

**Definition 3.** A *component* $W_i = \langle S_i, I_i, O_i, T_i, L_i, s_{i_0} \rangle$ 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.
$O_i$: Finite set of output signals with their finite definition domain.
$T_i \subseteq S_i \times C(I_i) \times S_i$: Finite set of transitions, $\forall s \in S_i$ , $\forall c \in C(I_i)$, $\exists! \ s' \in S_i$ s.t. $(s, c, s') \in T_i$ ($\exists!$ means "there exists exactly one").
$L_i = \{l_0, \ldots, l_{|O_i|-1}\}$: Vector of generation functions, each function defining the value of exactly one output signal in each state; for all output signals $o_j$ $0 \leq j < |O_i|$ we have $l_j : S_i \rightarrow Dom(o_j)$.
$s_{i_0} \in S_i$: the initial state.

*Remark 1.* 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 occurrence 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. The new event implies new behaviours and a new set of output signals but it preserves all behaviours that already existed. Moreover, it may be active or not, and in the second case the new component $W_{i+1}$ behaves exactly like $W_i$.

The occurrence of the new event implies new behaviours and (possibly) new sets of states and output signals. The new event cannot remove or override behaviours that already existed, it only adds new ones. It can occur in different manners:

- Either the definition domain of one or more existing input signals is 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.
- One or more new input 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 modeled by the appearance of a new set of input signals (with their definition domain), $I_+$, extending $I_i$. In the first case we treat the extended signal as a new one. For example let $l$ be a signal with an initial domain $Dom(l) = \{a, b\}$. A more complex component has to take into account a new possible value, $c$, of signal $l$. This can be expressed by introducing a new signal $j$ such that $Dom(j) = \{0, 1\}$ and "$j$ takes value 0" means "$l$ takes a previously considered value ($a$ or $b$)" and "$j$ takes value 1" means $l$ takes a new value $c$ (that was not previously considered). The domain of $l$ is unchanged.

The set of all configurations corresponding to the new input signals is split in two disjoint sets representing that the new event is active or not.

**Definition 4.** The *event* $e$ is a triple $e = \langle I_+, C_{ACT}(I_+), C_{QT}(I_+) \rangle$ 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 $C_{ACT}(I_+) \cap C_{QT}(I_+) = \emptyset$.

In the following we denote $e\_qt$ a configuration of signals in $I_+$ belonging to $C_{QT}(I_+)$, respectively $e\_act$ represents a configuration of signals in $I_+$ belonging to $C_{ACT}(I_+)$. We have $\neg(e\_act) \in C_{QT}(I_+)$ and $\neg(e\_qt) \in C_{ACT}(I_+)$.

Figure 1 presents an admissible increment. On the left, the Moore machine describing a component $W_i$ contains three states and the input signal $k$ with $Dom(k) =$

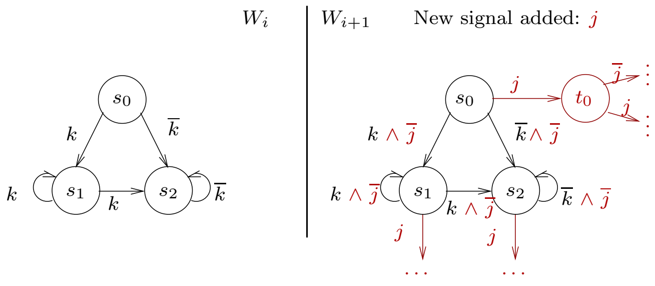


**Fig. 1.** Increment example

$\{0, 1\}$ drives the transitions. On the right, the component $W_{i+1}$ resulting from the increment of $W_i$ with an additional input signal $j$ with $Dom(j) = \{0, 1\}$. $j = 0$ belongs to the quiet configurations' set, and $j = 1$ to the active one. In $W_{i+1}$, $j = 0$ labels all transitions that were in $W_i$ ($j = 0$ is represented by the literal $\overline{j}$); new transitions leaving from states that existed in $W_i$ are labeled with the active configuration of $j$(that is, $j = 1$, represented by the literal $j$); Labels of transitions exiting new states in $W_{i+1}$ can either be $j$ or $\overline{j}$.

The occurrence of the new event induces new behaviors in the more complex component. It introduces new transitions, and possibly new states and actions: either one or more existing output signals may have their domain extended, or one or more new output signals may be created, *but* no one is deleted. All behaviours that previously exist in $W_i$ are present in $W_{i+1}$.

In the following definition, let $E_2 \subset E_1$ be sets of signals, let $c$ be a configuration in $C(E_1)$, we write $c' = proj(c, E_2)$ for the sub-configuration of $c$ restricted to the signals in $E_2$ ($c' \in C(E_2)$).

**Definition 5.** An *increment* from a component $W_i$ is a 4-tuple $INC = \langle e, \Sigma_+, T_+, O_+ \rangle$

$e$: the event described above.

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

$T_+ \subseteq (S_i \times C(I_i \cup I_+) \times S_i) \cup (S_i \times C(I_i \cup I_+) \times \Sigma_+) \cup (\Sigma_+ \times C(I_i \cup I_+) \times \Sigma_+) \cup (\Sigma_+ \times C(I_i \cup I_+) \times S_i)$ : each transition $(s_1, c', s_2)$ in $T_+$ leaving a state that was in $S_i$ has an input configuration whose sub-configuration of the new input signals belongs to $C_{ACT}(I_+)$. We have $\forall t = (s_1, c', s_2) \in T_+ \cap (S_i \times C(I_i \cup I_+) \times \Sigma_+ \cup S_i \times C(I_i \cup I_+) \times S_i)$ , $c'$ is such that : $c' \in C(I_i) \cup C_{ACT}(I_+)$. In the following we will write $c' = c \wedge e\_act$, with $c \in C(I_i)$ and $e\_act \in C_{ACT}(I_+)$.

$O_+$ : 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 $O_+$ return a configuration in $C_{QT}(O_+)$ for all states that were in $S_i$.

We also need to extend the signature of the configuration of transitions that were in $T_i$. The function *Extend* fills the *old* configuration with a sub-configuration belonging to the quiet configuration set : let $t = (s_1, c, s_2) \in T_i$, $Extend(t, I_+) = t'$ such that $t' = (s_1, c', s_2)$ and $c' = c \wedge e\_qt$ (with $e\_qt$ in $C_{QT}(I_+)$) and $proj(c', I_i) = c$.

A component $W_{i+1}$ obtained by applying an increment to a component $W_i$ preserves all behaviours that were present in $W_i$, assuming that, in $W_{i+1}$, the new event stays to a quiet configuration. We have, $S_{i+1} = S_i \cup \Sigma_+$, $I_{i+1} = I_i \cup I_+$, $O_{i+1} = O_i \cup O_+$, $T_{i+1} = \{t' \mid t' = Extend(t, I_+), \forall t \in T_i\} \cup T_+$, $L_{i+1}$ conforms to the restriction imposed by $O_+$ and $s_{{i+1}_0} = s_{i_0}$.

We recall the definition of simulation relation as expressed by Grumberg and Long in [9].

**Definition 6.** (*simulation relation* [9]) Let $M$ and $M'$ be two Moore Machines with $I \subseteq I'$ and $O \subseteq O'$ and let $s_0$ (resp. $s'_0$) be states in $S$ (resp. $S'$). A relation $H \subseteq S \times S'$ is a *simulation relation* from $(M, s_0)$ to $(M', s'_0)$ iff the following conditions hold.

1. $H(s_0, s'_0)$.
2. for all $s$ and $s'$, $H(s, s')$ implies
   (a) The projection of $L'(s')$ onto $O'$ is equal to $L(s)$;
   (b) for every $p$ such that $(s, C(I), p) \in T$ there exists $p'$ $(s', C(I'), p') \in T'$ and $H(p, p')$.

**Proposition 1.** *$(W_{i+1}, s_{i+1})$ simulates $(W_i, s_i)$.*

*Proof.* We build $\rho_W$ a binary relation between the states of two consecutive components $W_i$ and $W_{i+1}$, such that $\rho_W \subseteq S_i \times S_{i+1}$ : $\forall (s, c, p) \in T_i$ and $(s', c', p') \in T_{i+1}$, we set $(s, s') \in \rho_W$ iff $s = s'$ and $c' = c \wedge e\_qt$. By construction $\rho_W$ 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 7.** A *Kripke structure* is a 5-tuple $K = \langle S, s_0, AP, \mathcal{L}, R \rangle$ 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.

$\mathcal{L} = \{l_0, \dots, l_{|AP|-1}\}$ is a vector of |AP| functions. Each function defines 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.

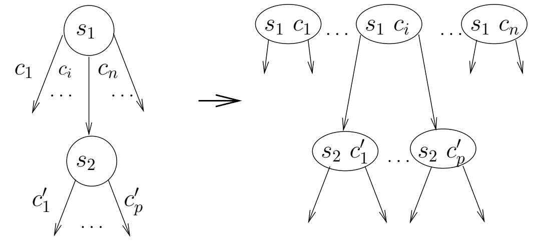


**Fig. 2.** Transformation of a Moore machine into a Kripke structure

**Definition 8.** Given a component $W_i$, we deduce the *Kripke structure* (adapted from [5]) $K(W_i) = \langle S_{K(W_i)}, s_{K(W_i),0}, AP_{K(W_i)}, \mathcal{L}_{K(W_i)}, R_{K(W_i)}\rangle$

$S_{K(W_i)} = S_i \times C(I_i)$.
$s_{K(W_i),0} = \{s_i\} \times C(I_i)$.
$AP_{K(W_i)} = I_i \cup O_i$.
$\mathcal{L}_{K(W_i)} = \{l_{I0}, \ldots, l_{I|I_i|-1}\} \cdot \{l_{O0}, \ldots, l_{O|O_i|-1}\}$ is a vector of $\mid AP_{K(W_i)} \mid$ functions.$\cdot$ means vector concatenation.
$R_{K(W_i)} \subseteq S_{K(W_i)} \times S_{K(W_i)}$ 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 $(s, c_i, s') \in T_i$.

Figure 2 shows the transformation of a Moore machine (on the left) into a Kripke structure (on the right). The state $s_1$ is transformed into $n$ states, each of which being labeled with $s_1$ and a different input configuration.

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 $K(W_i)$. One can derive a Kripke structure $K(W_{i+1})$ from the component $W_{i+1}$ by applying Definition 8. We are now interested in characterizing the properties of $K(W_{i+1})$ with respect to $K(W_i)$, i.e. to show that $K(W_i)$ is included into $K(W_{i+1})$, with all states that were present into $K(W_i)$ tagged with a quiet configuration of the new event.

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

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

**Definition 9.** *Enrichment Relation* For all states $t = (s, c) \in K(W_i)$, there exist $t' = (s', c')$ and $t'' = (s'', c'') \in K(W_{i+1})$ such that :
$s' = s$, $c' = c \wedge e\_qt$
$s'' = s$ , $c'' = c \wedge e\_act$.
$t'$ and $t''$ are said to *enrich* $t$ (with *e_qt* in the first case).

In the following we denote $s_{K(W_i),0}$ by $t_0$ and $s_{K(W_{i+1}),0}$ by $t'_0$.

In [9] a simulation relation between Kripke structures is also defined. We apply it here.

**Proposition 2.** *$(K(W_{i+1}), t'_0)$ simulates $(K(W_i), t_0)$ where $t'_0$ enriches the initial state $t_0$ with e_qt.*

*Proof.* We define $\rho_{K_W} \subseteq S_{K(W_i)} \times S_{K(W_{i+1})}$, such that $\forall t = (s, c) \in S_{K(W_i)}$, $s \in W_i$ and $c \in C(I_i)$, $\exists t' = (s', c') \in S_{K(W_{i+1})}$, with $s' \in W_{i+1}$ and $c' \in C(I_{i+1})$. For such $t$ and $t'$ we set $(t, t') \in \rho_{K_W}$ iff $s' = s$ and $c' = c \wedge e\_qt$. By construction, $\rho_{K_W}$ is a simulation relation.

*Remark 2.* From above,
$t'$ enriches $t$ with $e\_qt \Rightarrow t'$ simulates $t$.
$t'$ enriches $t$ with $e\_act \not\Rightarrow t'$ simulates $t$
$t'$ simulates $t \not\Rightarrow t'$ enriches $t$.
Indeed, nothing can be said once a state $t'$ enriching $t$ with *e_act* is encountered, since it represents the beginning of a new behaviour.

Figure 3 summarizes the incremental design process and the transformation of a Moore machine into a Kripke structure. The Moore machine $W_i$ (top left) is incremented by an event $e$. The result is a Moore machine $W_{i+1}$ (top right). $W_{i+1}$ has got a new state $r'$ that is not in $W_i$. The bottom of the figure shows the Kripke structures derived from the Moore machines. The state $r' \in W_{i+1}$ produces a *set* of states in $K(W_{i+1})$ labeled with $r'$ (and an input configuration of $I_{i+1}$), that may only be reached by a state $t''$ labeled with *e_act*.

**Corollary 1.** *1. 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 a quiet configuration. Let $\sigma = s_0 \ldots s_n \ldots$ be in $K(W_i)$. Then there is $\sigma' = s'_0 \ldots s'_n \ldots$ in $K(W_{i+1})$ such that all $s'_i$ enriches $s_i$ with e_qt.*

*2. $K(W_i)$ is the maximal sub-graph in $K(W_{i+1})$, reachable from $s'_0$,(that enriches $s_0$ with e_qt) when e remains in a quiet configuration.*

*3. The states in $K(W_{i+1})$ obtained by the expansion of a state in $\Sigma_+$ are only reachable from the initial state $s'_0$ that enriches $s_0$ with e_qt by a path along which* at least *one state is labeled by e_act. Moreover, along this path, the first state labeled by e_act enriches by e_act a state in $K(W_i)$.*

*4. Let $s' \in K(W_{i+1})$ enrich $s \in K(W_i)$ with e_qt , then for all $t' \in K(W_{i+1})$ such that $s' \to t'$, there exists $t \in K(W_i)$ such that $t'$ is produced by the expansion of t due to the increment, and $s \to t$.*

*Proof.* 1. By induction on the length of $\sigma$.

2. By construction of $K(W_{i+1})$, we have that $s'_0$ enriches $s_0$. By Corollary 1 item 1, all paths that where in $K(W_i)$ are present in $K(W_{i+1})$ with $e$ remaining to a quiet configuration. If there exists a state $t' \in K(W_{i+1})$ reachable from $s'_0$ along a path where $e$ is maintained to a quiet configuration and such that $t'$ satisfied *e_act* then $t'$ does not belong to $K(W_i)$.

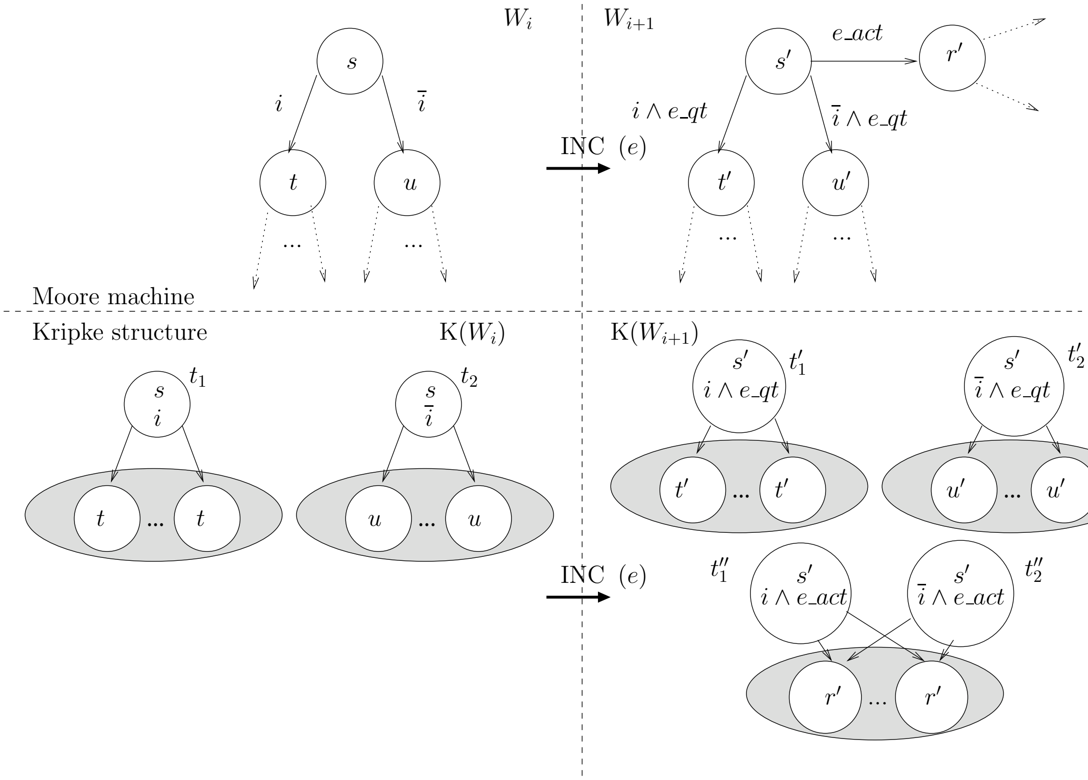


**Fig. 3.** Incremental rules and Kripke structure transformation

3. From Corollary 1 item 2.
4. Let $s' \in K(W_{i+1})$ enrich $s \in K(W_i)$ with *e_qt*, and $s' \to t'$. 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 $W_{i+1}$ (that is not in $W_i$) reached by a transition labeled by *e_act* (Corollary 1 item 3). Hence $t'$ can not be a successor of $s'$ (see Figure 3) (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_qt*, reachable from the initial state. This is captured by the enrichment relation. The enrichment relation with *e_qt* 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

[14] and [9] 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 [14] the authors state the preservation of ECTL[2] formulae from $K(W_i)$ to $K(W_{i+1})$, while in [9] the authors state the preservation of ACTL[3] formulae from $K(W_{i+1})$ to $K(W_i)$. These works only consider fragments of CTL and *do not* transform formulae. Moreover, the preservation is unidirectional : an ACTL formula that is verified in $K(W_{i+1})$ will hold in $K(W_i)$, but an ACTL formula that is verified in $K(W_i)$ may *not* hold in $K(W_{i+1})$.

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 CTL operators and provide a bi-implication between the initial formula and the transformed one.

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

**Theorem 1.** *Let $s \in S_{K(W_i)}$ and $s' \in S_{K(W_{i+1})}$ such that $s'$ enriches $s$ with e_qt.*
*We claim : for any atomic proposition $p \in AP_{K(W_i)}$ and for any CTL formula $\Phi$, $\chi$ and $\Psi$ (with all their atomic propositions in $AP_{K(W_i)}$),*

$$K(W_i), s \models \Phi \Leftrightarrow K(W_{i+1}), s' \models \Phi'$$

*where $\Phi'$ is the formula obtained by recursively applying the following transformations:*

[2] ECTL stands for positive CTL formulae restricted to the Existential modality

[3] ACTL stands for positive CTL formulae restricted to the Universal modality

*1* $\Phi = p \quad \Leftrightarrow \Phi' = p.$
*2* $\Phi = \neg\Psi \quad \Leftrightarrow \Phi' = \neg\Psi'.$
*3* $\Phi = EX\Psi \quad \Leftrightarrow \Phi' = e\_qt \wedge EX\Psi'.$
*4* $\Phi = EF\Psi \quad \Leftrightarrow \Phi' = E(e\_qtU\Psi').$
*5* $\Phi = EG\Psi \quad \Leftrightarrow \Phi' = EG(e\_qt \wedge \Psi').$
*6* $\Phi = E[\Psi U \chi] \quad \Leftrightarrow \Phi' = E[(e\_qt \wedge \Psi')U\chi'].$
*7* $\Phi = E[\Psi W \chi] \quad \Leftrightarrow \Phi' = E[(e\_qt \wedge \Psi')W\chi'].$
*8* $\Phi = AX\Psi \quad \Leftrightarrow \Phi' = e\_qt \Rightarrow AX\Psi'.$
*9* $\Phi = AF\Psi \quad \Leftrightarrow \Phi' = AF(e\_act \vee \Psi').$
*10* $\Phi = AG\Psi \quad \Leftrightarrow \Phi' = A[\Psi' W(e\_act \wedge \Psi')].$
*11* $\Phi = A[\Psi U \chi] \quad \Leftrightarrow \Phi' = A[\Psi' U((e\_act \wedge \Psi') \vee \chi')].$
*12* $\Phi = A[\Psi W \chi] \quad \Leftrightarrow \Phi' = A[\Psi' W((e\_act \wedge \Psi') \vee \chi')].$
*W stands for the "Weak until" operator.*

*Proof.* (Sketch): The transformations are based on the reduction of the computation tree explored in $K(W_{i+1})$ to the sub-tree along which the active configurations of the event are not considered. By Corollary 1 item 2, this sub-graph represents $K(W_i)$. The transformation is proven for each CTL operator applied to an atomic proposition : we include the *e_qt* constraint in its definition. Then the proof proceeds by structural induction on the formula $\Phi$.

The intuitive meaning of these transformations is the following : Line 1 states that an atomic proposition labeling a state $s$ in $K(W_i)$ also labels the state $s'$ (that enriches $s$) in $K(W_{i+1})$. Line 3 states that if $s \models EX\Psi$, then there exists a successor $s''$ of $s'$ (that enriches s with *e_qt*) such that $s'' \models \Psi'$, where $\Psi'$ is obtained by recursively applying the transformation rules to $\Psi$. As $s'$ enriches $s$ with *e_qt*, $s''$ has a corresponding state in $K(W_i)$, and the bi-implication holds. Line 10 says that when all states along all infinite paths in $K(W_i)$ verify $\Psi$, the corresponding paths in $K(W_{i+1})$ are infinite paths tagged with *e_qt* and the transformation of $\Psi$, but also finite paths with a prefix tagged with *e_qt* and $\Psi'$, ending in a state where *e_act* holds. The presence of these infinite paths where $\Psi'$ and *e_qt* always hold explains the weak until operator W.

The transformations listed above *do not* modify the temporal structure of the initial formula in that the nesting of temporal operators is preserved, hence the size of the CTL formula (measured as the number of nested 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 fix-point computation. The transformation is transfered into the propositional operations that are performed by classical BDD binary operations (*and, or, implies, ...*).

We implemented a CTL formulae transformation automation tool that automates the rules described in Theorem 1. This tool takes a file with a set of CTL formulae and a file containing the definition of a new event and returns the set of transformed CTL formulae. The file defining the new event contains the set of new signals with their definition domain, and the quiet and active configuration sets.

## 5 Experiment

The task of writing pertinent CTL properties for a realistic component is not easy. The specification mixing natural language, partial chronogram, partial state-machine, describes very subtle behaviours that lead to complex CTL formulae. In practical use, the specification is composed of hundreds of formulae. They are written by a human being, who may make mistakes. The mistakes are detected, once the whole verification process is accomplished. The system does not satisfy the erroneous formulae and the designer has to state whether the mistakes are in the system or in the formulae.

Moreover, in the incremental design process, the CTL formulae for a component $W_i$ have to be transformed for a component $W_{i+1}$. A manual transformation may also introduce errors. The transformation rules we presented in Theorem 1 can be automated and used to produce a part of the specification of $W_{i+1}$, alleviating the burden of handwriting the whole specification. The part of the specification of $W_{i+1}$ automatically derived from $W_i$ is exactly the set of rules necessary to check the non-regression between the two components.

We experimented with this automatic production of CTL specifications in the context of a protocol conversion between PI-bus and VCI standard. We aim at analyzing the differential of verification complexity (in terms of memory and speed) of a realistic medium-sized system, obtained by our incremental design process, between two successive design steps. In this experiment, the increments are manually added (following the design process). Some properties we want to verify are local to a component, but others are global to a system composed of several components. We are interested in checking that the added behaviours do not introduce regression.

The conversion between PI-bus and VCI protocols is realized by a component named VCI-PI wrappers. A wrapper is a core wrapping device implementing a given interface. In our context, the IP-core is supposed to be VCI compliant [7] and the considered wrapper is an adapter between the VCI interface and the PI-bus protocol [11]; hence we are able to connect various IP-cores through a PI-bus.

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* functionalities. Figure 4 illustrates the major signals interfaces a VCI-PI master wrapper has to deal with.

A VCI transfer is shown in Figure 5. 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

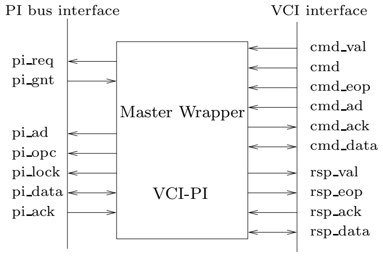


**Fig. 4.** VCI and PI interfaces of our set of master wrappers

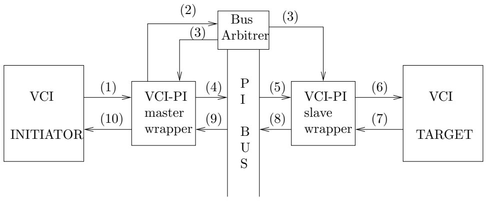


**Fig. 5.** The Platform performing the VCI-PI-VCI translation and illustration of a VCI transfer

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). The 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 six 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 and retract events sent by the VCI initiator or the PI target. The hierarchy of the six master wrappers is shown in Figure 6.

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.

The incremental architecture of the six master wrappers is presented in Figure 7, showing the behaviours successively added by increments ranking from A to C'. For each increment, the FSM part (Finite state machine) contains the FSM realization of the corresponding wrapper, and the data-path is augmented accordingly. Successive additions to the data-path are shown in different gray shades. The *PI retract* area is a piece of circuit induced by the increments to produce wrapper B into wrapper C. The same increment also transforms wrapper B' into C'. The *PI wait* area corresponds to the increment between wrappers A and B and between wrappers A' and B' also. The *Initiator wait* area represents the increments from A to A', from B to B' and from C to C'.

We implemented a platform as described in Figure 5 in synchronous Verilog. We verified this system with the VIS verification tool [8]. 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).

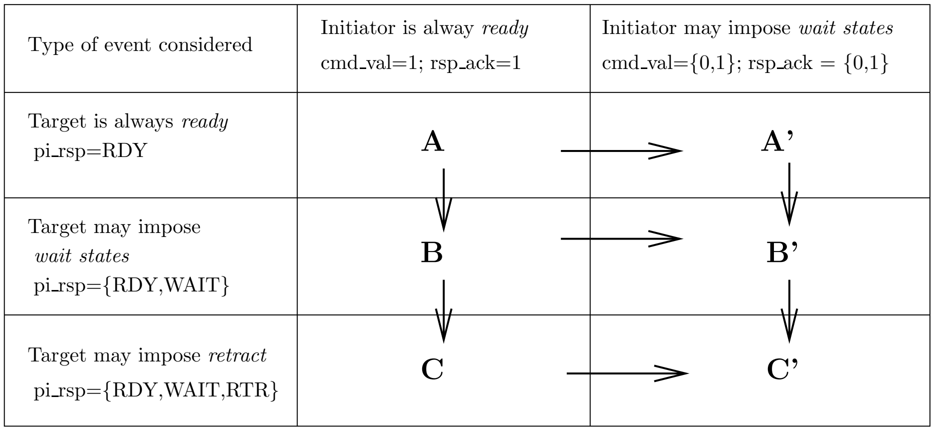
Here are examples of CTL (untransformed) properties checked on the platform containing a B wrapper:

```
#-----------------------#
# -> : implies operator #
# *  : and operator     #
# +  : or operator      #
#-----------------------#


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

# Check the behavior of the slave wrapper
# (its two automatas 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( (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) ));
```



| Type of event considered | Initiator is alway *ready*<br>cmd_val=1; rsp_ack=1 | Initiator may impose *wait states*<br>cmd_val={0,1}; rsp_ack = {0,1} |
|---|---|---|
| Target is always *ready*<br>pi_rsp=RDY | **A** → | **A'** |
| Target may impose *wait states*<br>pi_rsp={RDY,WAIT} | ↓<br>**B** → | ↓<br>**B'** |
| Target may impose *retract*<br>pi_rsp={RDY,WAIT,RTR} | ↓<br>**C** → | ↓<br>**C'** |

**Fig. 6.** 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 by one or more signals.

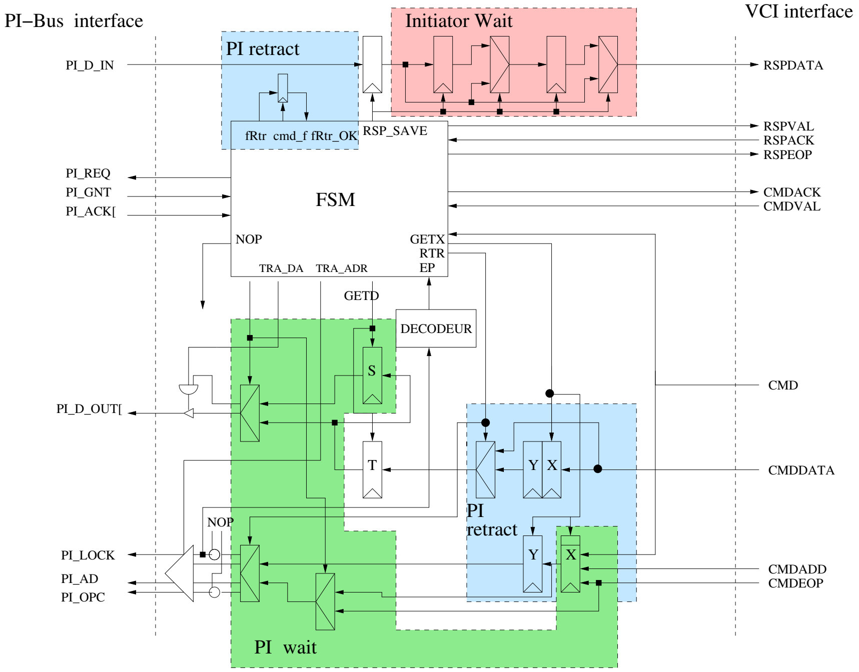


**Fig. 7.** Architecture of master wrapper C'.

The wrapper B supports delay from the PI bus, but it assumes the initiator is always ready to perform a writing or reading request and to acknowledge the response from the target. The wrapper B has been incremented to obtain the wrapper B'. The new event is an extended definition domain of the signals *cmd_val* and *rsp_ack*. Initially, in the wrapper B, these two signals always remain to have the value 1. Now both signals have their definition domain extended with the value 0, meaning that the initiator is not ready to perform a request, respectively not ready to acknowledge a response. We name the new event representing this extension $e = \langle(inc_{b'}, \{0,1\}), \{0\}, \{1\}\rangle$. The set of quiet configurations of the new event 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 $(cmd\_val = 0) \vee (rsp\_ack = 0)$.

By applying Theorem 1 the first property shown above is transformed into the following one:

```
A( (incb' = 0 * (wrap0.state = R_REQ) ->
 A( incb' = 0 *(pi_req = 1) U
 (incb' = 1 + pi_gnt = 1))) W (incb' = 1))
```

This can be rewritten with the existing signals:

```
A( ((cmd_val = 1 * rsp_ack = 1) *
        (wrap0.state = R_REQ) ->
 A( (cmd_val = 1 * rsp_ack = 1) *
 (pi_req = 1) U (cmd_val = 0 + rsp_ack = 0)
  + (pi_gnt = 1))) W
     (cmd_val = 0 + rsp_ack = 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 the B' VCI-PI master and slave. The verification results were successful, meaning that the modifications from B to B' do not introduce regression. Of course, extra CTL formulae had to be added to the B'-platform in order to check the behaviours added by the increment.

We now present some quantitative information related to the verification. The verification is performed with VIS. A first step computes the set of reachable states of the system, and then a second step checks each property.

Table 1 presents the time and memory required for the verification of the three properties given above (and the corresponding transformed ones) on a platform composed of :

1. One master and one slave of type B with a VCI initiator, a VCI target and a PI bus (with a unique master)(1M1SB).
2. One master and one slave of type B' with a VCI initiator, a VCI target and a PI bus (with a unique master)(1M1SB').
3. Two masters and one slave of type B with two VCI initiators, a VCI target and a PI bus(2M1SB).
4. Two masters and one slave of type B' with two VCI initiators, a VCI target and a PI bus(2M1SB').

For a small-size system (platforms 1 and 2), the overall verification time is increased for the complex model but most of this time is consumed during the reachable state-space construction (11s vs 68s). The extra cost of property verification is of the same order of magnitude for both platforms (2s vs 9s). These results are confirmed for the medium size systems (3 and 4) 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 (34s vs 3h30). Once the reachable state-space is built, the verification of each property is performed in 2s for the B platform vs 5s up to 25s for the B' platform. This is not surprising since, for the property verification, the same piece of state-space is analyzed, but the BDD's representing the transition relation and the state-space of platform B' are much bigger than those of platform B.

The incremental design process is also interesting in case of simplification. If one knows that the platform to be built will contain slaves that always respond "ready", then it is worth using master B instead of B': the models are simpler, the properties are given, or can be derived from the ones in B, and the verification cost will be reduced.

## 6 Concluding Remarks

The transformation rules of CTL formulae we propose are the basis to an approach to automatically derive part of the specification of a component, from the specification of the simpler component it comes from. Moreover, this approach facilitates the handwriting of CTL specifications and is a step toward the automatic reuse of existing specifications of a simpler component.

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 nested CTL operators) as the initial CTL formulae. This is confirmed by experimental results showing the increased verification time of the complex system is mainly due to the reachability analysis instead of the CTL formula verification.

It is our intention to extend 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 adding 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 behaviours.

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 transform 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 to the whole set of behaviours of the complex system, but could give some information if the complex system is too big to be verified with classical model-checking tools.

Another perspective is the increment composition. The example of wrappers presents seven increments, different compositions of some of them converge to the same "most complex" component. We are interested in finding some rules to compose various increments, and to define increments satisfying some "minimality" properties that are the basis of complex increments.

We are also interested in studying the way this approach could be mixed with an Assume-Guarantee verification process generally applied in the refinement design process [10].

**Table 1.** Verification time and memory required for reachable analysis and property checking. The experiments were performed under a Pentium III 1.4GHz with 1GB of memory with a sift dynamic reordering.

| Platform<br>(name) | Number of<br>variables | Reachable<br>states | BDD size<br>(# of nodes) | Reachable State Space<br>analysis time | Property<br>(#) | Checking<br>time |
|---|---|---|---|---|---|---|
| 1 Master<br>1 Slave B<br>(1M1SB) | 305 | 115405 | 3227 | 11.25s | 1 | 2.64s |
| | | | | | 2 | 2.68s |
| | | | | | 3 | 2.97s |
| 1 Master<br>1 Slave B'<br>(1M1SB') | 333 | 1,4e+07 | 18391 | 68.52s | 1' | 8.73s |
| | | | | | 2' | 8.88s |
| | | | | | 3' | 8.95s |
| 2 Masters<br>1 Slave B<br>(2M1SB) | 476 | 2.87e+06 | 9141 | 34.62s | 1 | 1.94s |
| | | | | | 2 | 1.91s |
| | | | | | 3 | 2.16s |
| 2 Masters<br>1 Slave B'<br>(2M1SB') | 528 | 9.32e+11 | 107571 | 3h30 | 1' | 5.7s |
| | | | | | 2' | 5.12s |
| | | | | | 3' | 24.5s |

## References

1. 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.
2. 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.
3. D. Cansell and D. Méry. Abstraction and Refinement of Features. In S. Gilmore and M. Ryan, editors, *Language Constructs for Designing Features*, pages 65–84. Springer-Verlag, 2001.
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, 2001.
5. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
6. STERIA Technologie de l'information. *Atelier B, Manuel Utilisateur*. Aix-en-Provence, France, 1998.
7. On-Chip Bus Development Working Group. *Virtual Component Interface Standard (VCI)*. VSI Alliance, 2000.
8. 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.
9. 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.
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. Open Microprocessors System Initiatives. *OMI324: PI-Bus Standard Specification*. Siemens, Munich, Germany, 1994.
12. K. Lano. *The B Language and Method, A guide to Practical Formal Development*. Springer-Verlag, Secaucus, NJ, USA, 1996.
13. K. G. Larsen and B. Thomsen. A Modal Process Logic. In *Third Annual Symposium on Logic in Computer Science*, pages 203–210, Edinburgh, Scotland, UK, 1988. IEEE Computer Society.
14. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
15. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academics Publishers, 1993.
16. C.S. Pasareanu, M.B. Dwyer, and M.Huth. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 168–183, London, UK, 1999. Springer-Verlag.
17. M. Plath and M. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.