A Further Step in the Incremental Design Process: Incorporation of an Increment Specification

Cécile Braunstein[1] and Emmanuelle Encrenaz[2]

[1] University Paris 6 - LIP6 - CNRS – France
cecile.braunstein@lip6.fr
[2] Laboratoire Spécification et Vérification - ENS Cachan - CNRS – France
emmanuelle.encrenaz@lsv.ens-cachan.fr

Abstract. Writing relevant properties for a realistic component's specification is not easy. In [1], we formalized an incremental design process. A component is obtained from a simpler component and some new behaviors modeled by an increment. From a component at a step i of the design process, its specification can be derived into a part of the specification of the same component at a step $i+1$. The obtained specification is exactly the set of rules necessary to check the non-regression between two components. In the present paper, we extend the transformation rules, previously stated, in considering that the increment verifies a set of CTL formulas. This allows to build automatically a larger set of formulas that is the entire specification of the component at step $i+1$.

1 Introduction

In [1], we defined an incremental process to design hardware components. This approach is very close to the way some hardware designers proceed : after having sketched the rough structure of the data part and its synchronization in the simplest case, one takes into account new events and defines the new behaviours they induce. The new behaviours may not override previously existing ones, and there is no deletion of behaviours. The design of a component is performed step by step, the designer starts with the simplest component that does not take into account all the possible events its environment may produce. Then the component is complexified in a stepwise manner by considering one (or a small number of) new events at each step. For example, while designing a processor, one may start by assuming that the cache will always hit a request and in a second step, add the new event *cache miss* and the corresponding management. We propose a framework that helps designers to focus on one problem at a time. The main advantage is the *guarantied-by-construction* non-regression of the transformed model with respect to the previous one.

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 [4, 2] a feature integration automating tool coupled with model-checkers. Our purpose is slightly different and is grounded on a much simpler definition of the increment than the feature integration definition. 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 or transformation results we propose are stronger.

The incremental design process allows one to derive a specification for a simple component into a part of the specification of a more complex one. The purpose of this paper extends these transformations by assuming that the increment verifies a set of CTL formulas (we add a specification to the increment). This assumption induces a reacher set of transformations; this alleviates the task of re-writing the specification of the complex component from the previous one.

2 Increment definition

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 without overriding nor deleting previous behaviours. In a general way, hardware designers represent their architecture with Moore machines and the semantic of CTL is defined over Kripke structure. For sake of conciseness, we do not introduce the Moore machine and its link to Kripke structure but we consider that component's behaviours are represented as Kripke structure. The translation between Moore machine and Kripke structure is well-known and can be automatically computed ([3]).

Definition 1. *Each* signal *is defined by a variable name, s and an associated finite definition domain* $\mathcal{D}om(s)$.
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* $\mathcal{C}(E)$.

Definition 2. *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;* $AP = I \cup O$ *is a finite set of atomic propositions, I is a set of input signals and O the set of output signals;* $\mathcal{L} = \{l_0, \dots, l_{|AP|-1}\}$ *is a vector of* $\mid AP \mid$ *functions. Each function defines the value of exactly one atomic proposition; for all* $0 \leq i \leq |AP|$ *we have* $l_i : S \rightarrow \mathbb{B}$*; 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.*

We define an increment as *adjunction* of a new functionality (or a set of functionalities) to a initial Kripke structure. Intuitively, an increment represents the reaction of the system to a set of new event e. A new event is represented by a new[3] set of signals added on the input interface of the system. The event may be active or not. The occurrence of the new event induces new behaviours and a new set of output signal.

Definition 3. *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,* $I \cap I_+ = \emptyset$.

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

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

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

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

When a designer adds the management of the new event, he may want to specify this new behaviour only. In practice, it is easier to think about the increment in term of an independent functionality. Thus the increment is seen as a component[4] that will be *plugged* into the initial model to enrich the initial behaviours. This new functionality is triggered by an active configuration of the new event. We present a new definition of the increment, it is more precise than the one described in [1], that focused on the triggering of the new functionality.

Definition 4. Increment
Let K_i *be an initial structure, an increment applied to* K_i *is a 3-tuple* $INC = \langle K_{INC}, R_{i\to INC}, R_{INC\to i}\rangle$ *such that* K_{INC} *is defined as follows*

S_{INC} *: the set of new reachable states;*
$S0_{INC} \subseteq S_{INC}$ *: the set of initial states ;*
$AP_{INC} = AP_i \cup I_+ \cup O_+$*: the set of atomic propositions,* I_+ *and* O_+ *are the set of new input and output signals;*
L_{INC}*: Vector of* $\mid AP_{INC} \mid$ *labeling functions;*
$R_{INC} = S_{INC} \times S_{INC}$*: the set of new transitions.*

And the connection between K_i *and* K_{INC} *is such that:* $R_{i\to INC} \subseteq S_i \times S0_{INC}$ *such that* $((s_1,c_1),(s_2,c_2)) \in R_{i\to INC}$ *iff* $(s_1,c'_1) \in S_i$, $c_1 = c'_1 \wedge c\_act$; $R_{INC\to i} \subseteq S_{INC} \times S_i$

We define $SPEC_{INC}$ the set of CTL formulas verified by all initial states of K_{INC}. $SPEC_{INC}$ is the specification of the new functionality. We say that $K_{INC} \models \varphi$ iff $\forall s_0 \in S0_{INC}, K_{INC}, s_0 \models \varphi$.

The component at step $i+1$ of the design is represented as a Kripke structure K_{i+1} obtained by *adjunction* of the model K_i and the increment K_{INC} connected as defined by $R_{i\to INC}$ and $R_{INC\to i}$. Our design process preserves pre-existing behaviours, if the new event does not occur K_{i+1} behaves exactly as K_i. In the incremented model, all states that were in the simplest model are labeled with a quiet value (c_qt). All states at the boundary of the simplest model and the increment structure, are labeled with an active value (c_act) (see fig.1).

Definition 5. Incremented Kripke structure K_{i+1}
Let K_i *an initial model, let* $INC = \langle R_{i\to INC}, R_{INC\to i}, K_{INC}\rangle$ *be an increment, the incremented Kripke structure* K_{i+1} *is defined such that :*

[4] As a consequence the increments that do not induce new states can not be considered by this framework.

$S_{i+1} \subseteq S'_i \cup S''_i \cup S_{INC}$: *with* $\mathbf{S'_i}$ *the set of state coming from* S_i *with an extended label :* $a = (s,c) \in S'_i$ *iff* $(s,c') \in S_i$ *and* $c = c' \wedge c\_qt$; $\mathbf{S''_i}$ *the set of state coming from* S_i *with an extended label :* $a = (s,c) \in S''_i$ *iff* $(s,c') \in S_i$ *and* $c = c' \wedge c\_act$

$S_{0_{i+1}} = S_{0_i}$

$AP_{i+1} = AP_i \cup I_+ \cup O_+$: *the set of atomic propositions,* I_+ *and* O_+ *are the set of new input and output signals;*

$L_{i+1} = L_{INC} \bullet L'_i$: *Vector of* $\mid AP_{i+1} \mid$ *labeling functions,* $\bullet$ *is the concatenation function and* L'_i *is the vector of labeling function of* K_i *extends with the new atomic propositions ;*

$R_{INC} = R_{i \to INC} \cup R_{INC \to i} \cup R'_i \cup R_{INC}$: R'_i *is s.t.* $((s_1,c_1),(s_2,c_2)) \in R'_i$ *iff* (s_1,c'_1) *and* (s_2,c'_2) *are in* S_i *and* $c_1 = c'_1 \wedge e\_qt$, $c'_2 = \wedge e\_qt$ *or* $c'_2 = c_2 \wedge c\_act$.

3 Incorporation of the increment's specification

Return Connection between K_{INC} and K_i

An increment adds some new states and transitions useful for the new event management. The active configuration of the new event represent a border from K_i to K_{INC}. This border is well identified (first occurrence of c_act in K_{i+1}) and the set of CTL transformation defined in [1] takes advantage of this particularity. We would like to identify such border from K_{INC} to K_i to represent the *return* after the new functionality has been processed. If such border exists one can extend the set of CTL transformation of [1] to take into account the specification of the increment.

We identified three types of return from K_{INC} to K_i : (1) K_{INC} never returns to K_i ($R_{INC \to i} = \emptyset$) (figure 1(a)); (2) K_{INC} may return to K_i without any further assumption ($R_{INC \to i} \neq \emptyset$ (figure 1(b)); (3) K_{INC} may return to K_i and a border can be identified: $R_{INC \to i} \neq \emptyset$ is such that $\forall((s_1,c_1),(s_2,c_2)) \in R_{INC \to i}, c_1 \in C_{RTN}$ where C_{RTN} is a set of identified *return configuration*. Unlike the *active* value, the *return* value does not only depend on the input signals configuration. Indeed, the end can be stated by a final output signals configuration as well as the return to a *quiet* configuration of the input signals or even by a internal signal configurations. The *return* value is a set of configuration of all atomic propositions : $C_{RTN} \subset \mathcal{C}(AP_{INC})$. with $c\_rtn \in C_{RTN}(AP_{INC})$ and $\overline{c\_rtn} \notin C_{RTN}(AP_{INC})$.

Automatic construction of the specification of K_{i+1}

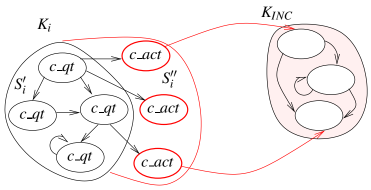
We stated in [1], a general CTL-property transformation : from the specification of K_i we directly obtain a part of the specification of K_{i+1}. In the present paper, we extend this result by giving the rules to incorporate the increment's specification into a new part of specification K_{i+1}. This new transformation depends on the return connection from K_{INC} to K_i.

Case 1 If there is no return from K_{INC}, the specification of K_{INC} holds in K_{i+1} as soon as the active value holds.

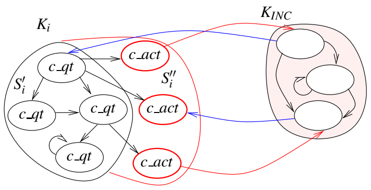
Theorem 1. *Let* K_{i+1} *be a Kripke structure obtained by applying the increment INC to* K_i *such that* $R_{INC \to i} = \emptyset$ *we have :*

$K_{INC} \models \varphi \Rightarrow \forall s_0 \in S_{0_{i+1}}, K_{i+1}, s_0 \models \mathbf{A}(e\_qt \mathbf{W}(e\_act \wedge \mathbf{AX}\varphi))$

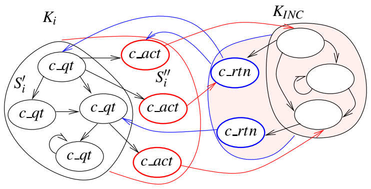
$\mathbf{W}$ *stands for the "weak until" operator.*



(a) K_{i+1} Without return



(b) K_{i+1} With return



(c) K_{i+1} With a return value

Fig. 1. The three increment structures

Proof. **Sketch.** Let K_i be a Kripke structure, *INC* an increment, such that $K_{INC} \models \varphi$ and K_{i+1} the incremented Kripke structure. In K_{i+1}, all the behaviours corresponding to K_i have all their states labeled with c_qt and all infinite path in K_i are also in K_{i+1}. Moreover all new behaviours added by the increment are only reachable through a state labeled with c_act. From an outgoing transition from a state in S_i, a state s_{inc} in $S_{0_{INC}}$ is reached and by hypothesis, this state models φ. Hence K_{i+1} has some infinite path along which c_qt holds and as soon as K_{INC} is reached φ holds, thus $K_{i+1} \models \mathbf{A}(e\_qt\mathbf{W}(e\_act \wedge \mathbf{AX}\varphi))$.

Case 2 If one cannot identify any border from K_{INC} to K_i, one cannot extends the existing transformation rules.
Case 3 The border between K_{INC} and K_i is identified by some states labeled by c_rtn. This case is the exact symmetric of the connection between K_i and K_{INC} delimited with a border tagged with the active value of the new event (c_act) and for which a set of CTL-transformation has been stated ([1]). By applying the same reasoning as in [1], we state: the initial states in K_{INC} in K_{i+1} do not satisfy $SPEC_{INC}$ anymore (due to the connection from K_{INC} to K_i), but rather satisfy a new set of specification $SPEC'_{INC}$. $SPEC'_{INC}$ is automatically obtained from $SPEC_{INC}$ by applying the recursive transformation rules defined below (theorem 2). This theorem is an adaptation of theorem 1 in [1] to fit with the particular context of its application.

Theorem 2. *Let $s \in K_{INC}$ and $s' \in K_{i+1}$ such that $s' = s$.*
We claim : for any atomic proposition $p \in AP_{INC}$ and for any CTL formula Φ, χ and Ψ (with all their atomic propositions in AP_{INC}), $K_{INC}, s \models \Phi \Leftrightarrow K_{i+1}, s' \models \Phi'$ where Φ' is the formula obtained by recursively applying the following transformations:

1 $\Phi = p \quad \Leftrightarrow \Phi' = p.$
2 $\Phi = \neg\Psi \quad \Leftrightarrow \Phi' = \neg\Psi'.$
3 $\Phi = \mathbf{EX}\Psi \quad \Leftrightarrow \Phi' = \overline{c\_rtn} \wedge \mathbf{EX}\Psi'.$
4 $\Phi = \mathbf{EF}\Psi \quad \Leftrightarrow \Phi' = \mathbf{E}(\overline{c\_rtn}\mathbf{U}\Psi').$
5 $\Phi = \mathbf{EG}\Psi \quad \Leftrightarrow \Phi' = \mathbf{EG}(\overline{c\_rtn} \wedge \Psi').$
6 $\Phi = \mathbf{E}[\Psi\mathbf{U}\chi] \quad \Leftrightarrow \Phi' = \mathbf{E}[(\overline{c\_rtn} \wedge \Psi')\mathbf{U}\chi'].$
7 $\Phi = \mathbf{E}[\Psi\mathbf{W}\chi] \quad \Leftrightarrow \Phi' = \mathbf{E}[(\overline{c\_rtn} \wedge \Psi')\mathbf{W}\chi'].$
8 $\Phi = \mathbf{AX}\Psi \quad \Leftrightarrow \Phi' = \overline{c\_rtn} \Rightarrow \mathbf{AX}\Psi'.$
9 $\Phi = \mathbf{AF}\Psi \quad \Leftrightarrow \Phi' = \mathbf{AF}(e\_rtn \vee \Psi').$
10 $\Phi = \mathbf{AG}\Psi \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}(e\_rtn \wedge \Psi')].$
11 $\Phi = \mathbf{A}[\Psi\mathbf{U}\chi] \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{U}((e\_rtn \wedge \Psi') \vee \chi')].$
12 $\Phi = \mathbf{A}[\Psi\mathbf{W}\chi] \quad \Leftrightarrow \Phi' = \mathbf{A}[\Psi'\mathbf{W}((e\_rtn \wedge \Psi') \vee \chi')].$

The proof of this theorem is done by induction of the length of the formula to be transformed. It is similar to the one of theorem 1 in [1].

Let us now state the transformation to be applied to the specification of K_i in order to obtain a part of the specification of K_{i+1} *taking into account* the specification $SPEC_{INC}$ of the increment K_{INC}.

Theorem 3. *Let K_{i+1} be a Kripke structure obtained by applying the increment INC to K_i such that $R_{INC\to i} \neq \emptyset$ and $\forall(s,s') \in R_{INC\to i}$, $s \models c\_rtn$. We have :*
$K_{INC} \models \varphi \Rightarrow K_{i+1} \models \mathbf{A}(e\_qt\mathbf{W}(e\_act \wedge \mathbf{AX}\varphi'))$, *$\varphi'$ is obtained by applying theorem 2*

Proof. **Sketch.** This is a direct consequence of theorems 1 and 2.

4 Conclusion

The paper proposes a extension of the incremental design process. We are now able to take into account the specification of the increment in the automatic construction of the entire specification of the incremented component. This extension enriches the previous definition, the incremental design process is really suitable for designing real component, and was successfully applied to the design and verification of protocol converters and processor pipelines.
Obviously, the incremental design process guaranties, by construction that the incremented model verifies its specification. There is no need to verify it by model checking. Rather this specification can be used as an abstraction of the incremented component. This is the subject of a forthcoming paper.

References

1. C. Braunstein and E. Encrenaz. CTL-property Transformations along an Incremental Design Process. *STTT*, 2006. proof are available at www-asim.lip6.fr/˜cecile.
2. 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.
3. E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *LICS*, pages 353–362. IEEE Computer Society, 1989.
4. M. Plath and M. Ryan. Feature Integration using a Feature Construct. *Science of Computer Programming*, 41(1):53–84, 2001.