Development and Verification of UML Architectures by Refinement and Extension Techniques

Thomas Lambolais and Anne-Lise Courbis,
LGI2P, IMT Mines Ales, Univ Montpellier, Ales, France
thomas.lambolais@mines-ales.fr, anne-lise.courbis@mines-ales.fr

***Abstract*—We have developed an incremental development framework which supports progressive constructions of UML state machines. This framework includes behavioral refinement and extension verifications. In this paper, we extend this framework to UML composite components in order to guide architecture modeling and verification. It consists of a set of construction techniques which assist the designer in two complementary ways: concerning behavioral aspects, the techniques are based on formal relations which determine if the new architectures are true refinements and/or extensions of the former ones; concerning structural aspects, the techniques contribute to the software engineering design principles separation of concerns, information hiding and hierarchical modeling. The paper presents some of these techniques and illustrates them through a case study.**

I. Introduction

Our goal is to assist designers during modeling tasks of reactive systems. We consider two aspects: describing the history of an architectural model construction by a sequence of modeling steps; offering assistance during this history, by providing evaluation and construction techniques based on formal relations. In previous works, we have defined an incremental development framework (IDF) for UML state machines [21] and developed the IDCM tool (Incremental Development of Compliant Models) [16]. The main benefits of this framework are to early consider abstract models and to add formal refinement and extension relations into agile modeling processes [17]. Now, we consider UML architectures described by UML composite components.

In this paper, we propose a set of architectural construction techniques. On the one hand, these techniques have to contribute to well-known design principles in software engineering [31] (separation of concerns, hierarchy and information hiding), whose goal is to achieve architectural model qualities. On the other hand, these construction techniques have to include formal verifications for early detection of behavioral issues, i.e. safety and liveness problems.

The article is structured as follows. Section 2 presents the main incremental paradigms, safety and liveness concerns and incremental relations. This section also presents a semantics of UML architectures. Section 3 presents architectural modeling techniques and formally defines two of them. The use of these techniques is illustrated in a case study in section 4. A discussion about the validation of this work and related works is opened in section 5. We conclude in section 6.

II. Fundamentals of incremental modeling and UML composite component semantics

By *incremental* modeling, we mean that models are progressively developed. They may be refined or abstracted, and extended or restricted. At each step, the new model is compared to the previous one through a relation, which focuses on behavioral and temporal aspects. As shown by Alpern and Schneider [29], all temporal properties can be seen as a conjunction of safety and liveness properties. The relation used to compare models is chosen among a set of relations which can be interpreted with respect to the way they *preserve* safety and/or liveness properties. We say that a property ϕ is preserved by a relation $\mathcal{R}$ if, for any two models P and Q such that $P \mathcal{R} Q$, $P \models \phi \Rightarrow Q \models \phi$. An interesting advantage of the use of such relations is that properties are implicitly and not explicitly described by the designer. Let us first precise the way we understand safety and liveness properties.

A. Safety and liveness properties

We observe safety and liveness properties by means of the *interactions* of the system with its environment: accept an *event* (signal or operation reception), or perform an *action* requiring a signal send or operation call. A trace is a partial sequence of observable interactions starting from the initial state. The following informal definitions are in accordance with the more general definitions and topological characterization given in [2], [3].

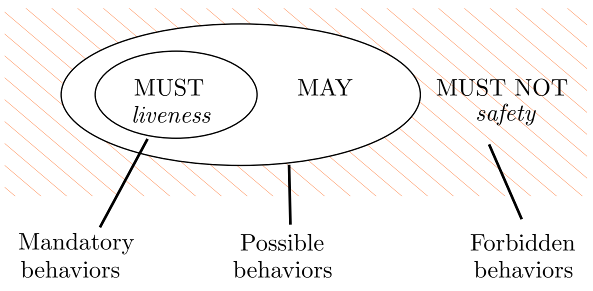


Fig. 1. Liveness and safety properties through ‘must’, ‘may’ and ‘must not’ behaviors.

a) Safety properties: A safety property states that some interactions are forbidden for the system after some given traces. This specifies that some traces must not be included in the set of system traces, e.g. “the automated light system must not automatically switch low beams off if ambient luminance is low". Safety properties are satisfied by systems whose behaviors are inside the '*may behaviors*' in Figure 1.

b) Liveness properties: A liveness property states that the system will eventually react as it should after some given traces. This specifies that some traces are included in the system trace set, *and* that after these traces, expected actions will eventually be offered, possibly after an unbound delay, e.g. "the AFLS will switch high beams on and off every time the driver uses the flash command". Liveness properties correspond to the *'must behavior'* set in Figure 1. We consider that deadlock freedom is a liveness property [26], [13] since a system is deadlocked when it rejects any input event.

The LTS (Labeled Transition Systems) semantics we give to UML primitive components behavior state machines is not recalled here, see [17], [21]. Let us briefly recall that an LTS P is a tuple $\langle \mathcal{P}, A, \rightarrow, P\rangle$ where $\mathcal{P}$ is a set of state names (also called processes and agents), A is a set of actions, $\rightarrow \subseteq \mathcal{P} \times A \times \mathcal{P}$ is a set of labeled transitions between states and P is the initial state. For instance, in Fig. 2, P and Q are two LTS described on the action set $A = \{a, b, c\}$.

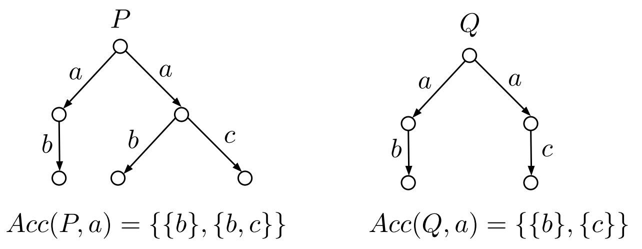


Fig. 2. Two LTS and their acceptance sets after action a: P and Q have the same safety properties but not the same liveness properties.

Initially, P and Q must both accept action a and must refuse b and c actions. After a, P must accept b, may accept and may refuse c, and must refuse a, while Q after a may refuse and may accept b, may refuse and may accept c and must refuse a. P and Q have the same safety properties, but P has a liveness property that Q does not have: P must accept b after a. Even if P and Q are both non-deterministic processes, P is more deterministic than Q. This is formally written by acceptance sets which are the sets of action sets that may be accepted after given traces. The set of set inclusion relation $\subset\subset$ expresses the fact that a set is 'less non-deterministic' than another. For two acceptance sets $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{A}$ is less non-deterministic than $\mathcal{B}$ ($\mathcal{A} \subset\subset \mathcal{B}$) if for all $A \in \mathcal{A}$, there exists $B \in \mathcal{B}$ such that $B \subseteq A$. Again, for formal definitions, please refer to [21]. Here, $Acc(P, a) \subset\subset Acc(Q, a)$ and $Acc(Q, a) \not\subset\subset Acc(P, a)$.

Hence, in order to compare two processes P and Q, we compare their *acceptance graphs*. Acceptance graphs are deterministic graphs whose nodes are associated to acceptance sets. Here follows an intuitive presentation of incremental relations, and a proposal for the UML composite component semantics.

B. *Incremental relations between UML primitive components.*

Based on classical trace inclusion $\sqsubseteq_{\mathsf{MAY}}$ and conformance relation conf [22], we consider four *incremental* relations: *increments*, *refines*, *extends* and *safely substitutes* respectively written $\sqsubseteq_{\mathsf{INC}}$, $\sqsubseteq_{\mathsf{REF}}$, $\sqsubseteq_{\mathsf{EXT}}$ and $\sqsubseteq_{\mathsf{SUB}}$. The first three relations are formally defined in [17], [21]; $\sqsubseteq_{\mathsf{SUB}}$ is defined in [27]. We only give an intuitive presentation here. Given two models M_1 and M_2, where M_2 is 'supposed to be an improvement' of M_1:

$M_2 \sqsubseteq_{\mathsf{MAY}} M_1$ means that M_2 traces are included into M_1 traces. It ensures that M_2 satisfies any safety property of M_1: indeed, M_2 must refuse all what M_1 must refuse.

M_2 conf M_1, or M_2 *conforms to* M_1, if after any trace of M_1, M_2 must accept every action that M_1 must accept. It ensures that M_2 is more deterministic than M_1. This relation guarantees that any liveness property of M_1 is satisfied by M_2. The conformance relation is seen as an implementation relation. However, this relation is not transitive and cannot be used as such for incremental developments.

$M_1 \sqsubseteq_{\mathsf{INC}} M_2$, or M_2 *increments* M_1, if any model which conforms to M_2 also conforms to M_1. In particular, $M_1 \sqsubseteq_{\mathsf{INC}} M_2 \Rightarrow M_2$ conf M_1, and $\sqsubseteq_{\mathsf{INC}}$ is a transitive relation.

$M_1 \sqsubseteq_{\mathsf{REF}} M_2$, or M_2 *refines* M_1, if $M_1 \sqsubseteq_{\mathsf{INC}} M_2$ and $M_2 \sqsubseteq_{\mathsf{MAY}} M_1$. Hence, $M_1 \sqsubseteq_{\mathsf{REF}} M_2$ guarantees that liveness and safety properties of M_1 are also satisfied by M_2.

$M_1 \sqsubseteq_{\mathsf{EXT}} M_2$, or M_2 *extends* M_1, if $M_1 \sqsubseteq_{\mathsf{INC}} M_2$ and $M_1 \sqsubseteq_{\mathsf{MAY}} M_2$. It ensures that M_2 may do all what M_1 can do, and that that liveness properties of M_1 are also satisfied by M_2. The $\sqsubseteq_{\mathsf{EXT}}$ relation is useful for incremental constructions, in order to consider new behaviors in the new development model.

$M_1 \sqsubseteq_{\mathsf{SUB}} M_2$ means that M_2 can *safely substitute* M_1. $\sqsubseteq_{\mathsf{SUB}}$ is the largest congruence preorder stronger than $\sqsubseteq_{\mathsf{REF}}$ and $\sqsubseteq_{\mathsf{EXT}}$: the *architecture* into which M_2 replaces M_1 is a correct refinement and extension of the initial architecture containing M_1. Architectures of LTS are described by composition of processes, as described in the following.

The verification algorithms to check these relations have been implemented within the IDCM tool.

C. *UML composite components semantics.*

UML composite components are architectures of UML component instances, linked between themselves by assembly connectors and connected to the outside environment by delegation connectors. We give a semantics of UML composite component behaviors on parallel compositions of processes in the EXPOPEN process algebra [19]. EXPOPEN shares the same concepts as basic LOTOS process algebra [22]. Secondly, EXPOPEN models are translated into LTS by the CADP tool.

Fig. 3 presents a UML architecture (A_0) and its translation into process algebra. We only focus on the structure. A_0 details, given in section IV, are not required here. Detailed EXPOPEN code is not required either. In the architecture A_0, there are two component instances linked by the assembly connector `con1` on two ports which share the same UML interface. Outside ports are linked by delegation connectors on two other ports.

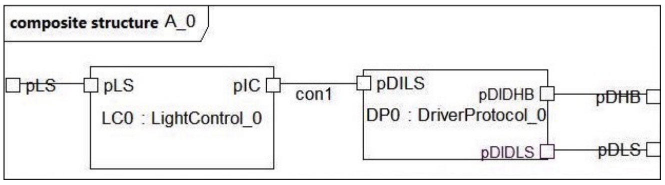
In the algebraic notation, this is represented by two parallel processes, synchronized on the action con1:

$$A_0 := \textbf{hide}\ con_1\ \textbf{in}\ \mathsf{LightControl}_0\ ||_{con_1}\ \mathsf{DriverProtocol}_0.$$

The external interfaces of A_0 correspond to actions exchanged on delegation connectors. All the synchronized actions (con_1) are hidden, which is done by the **hide ... in** operator. We will say that A_0 is described by a *configuration* f such that:

$$A_0 := f(\mathsf{LightControl}_0, \mathsf{DriverProtocol}_0).$$

Configurations are functions which take components as parameters and return a component.



Abbreviated algebraic notation:

$$A_0 := \textbf{hide}\ con_1\ \textbf{in}\ \mathsf{LightControl}_0\ ||_{con_1}\ \mathsf{DriverProtocol}_0$$

Fig. 3. A UML composite component (A_0) and its succinct algebraic notation

III. Refinement solving and component adding construction techniques

We classify incremental architectural construction techniques along two orthogonal axes: the vertical axis corresponds to the abstraction level, from abstract to concrete models; the horizontal axis corresponds to the level of requirement coverage, from partial specifications to complete behavior descriptions. Hence, vertical techniques include refinement and abstraction techniques while horizontal techniques include extension and restriction techniques. We have developed a comprehensive framework of incremental techniques which covers these different approaches (Fig. 4). The interest of this framework is to include most modelling approaches and to point out the formal relations that have to be verified between models. Note that some 'reverse engineering' or 'refactoring' techniques are also included under *abstraction* and *restriction* techniques, even if they are not detailed here. In the following, we present one refinement technique and one extension technique.

Refinement solving. *Let A be an architecture defined by a configuration f on components $C_1, ..., C_n$ such that $A := f(C_1, ..., C_i, ..., C_n)$. Given components $C_{i_1}, ..., C_{i_l}$ and a configuration g,* refinement solving *consists in finding a component X such that $C_i \sqsubseteq_{\mathsf{REF}} g(C_{i_1}, ..., C_{i_l}, X)$ and $A \sqsubseteq_{\mathsf{REF}} A'$, where $A' := f(C_1, ..., g(C_{i_1}, ..., C_{i_l}, X), ..., C_n)$.*

The idea of this technique is to be close to an equation solving: in A', everything is known apart X. In order to reduce the set of possible components X, one may want to use the equivalence relation $=_{\mathsf{REF}}$ instead of $\sqsubseteq_{\mathsf{REF}}$. For instance, if the architecture A is a single component, given a known component C_1, refinement solving consists in looking for a component X such that:

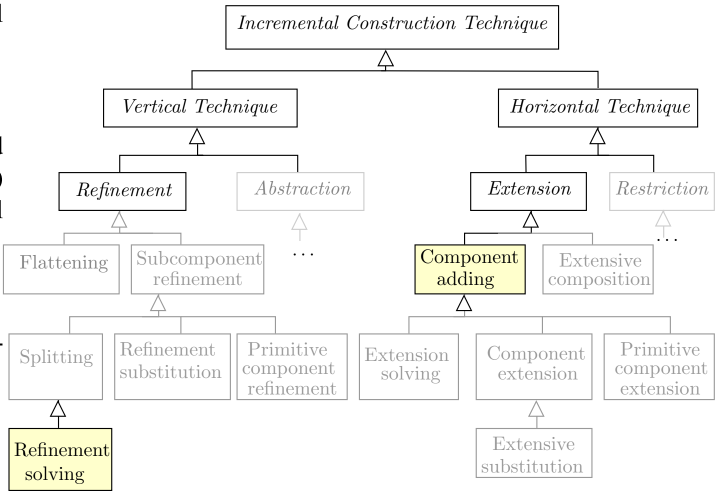


Fig. 4. Incremental construction techniques

$$A =_{\mathsf{REF}} \textbf{hide}\ con_1\ \textbf{in}\ X\ ||_{\mathsf{con}_1}\ C_1. \tag{1}$$

In most cases, such an equation has many different solutions: finding one of them automatically is outside of the scope of this article. Nevertheless, X interfaces are automatically determined by equation (1), and verdicts provided by the verification within the IDCM tool help us find an admissible solution.

This technique contributes to hierarchy, separation of concerns and information hiding.

Component adding. *Let A be an architecture defined by a configuration f, $A := f(C_1, ..., C_n)$. Given a component C, given a set T_s of some unwanted traces,* component adding *consists in finding a configuration g and a set of components $C'_1, ..., C'_n$ which adapt existing components such that $C_1 \sqsubseteq_{\mathsf{EXT}} C'_1, \ldots, C_n \sqsubseteq_{\mathsf{EXT}} C'_n$, $A \sqsubseteq_{\mathsf{EXT}} A'$ and $\forall t_s \in T_s\ .\ t_s \not\sqsubseteq_{\mathsf{MAY}} A'$, where $A' := g(C'_1, ..., C'_n, C)$.*

$A \sqsubseteq_{\mathsf{EXT}} A'$ entails that A' preserves liveness properties of A, but A' does not preserve safety properties of A. In order to verify some safety properties that A' should still satisfy, we use the set T_s of 'unwanted' traces.

IV. Illustration: Adaptive Front-lighting System

We consider a car Adaptive Front-lighting System (AFLS) implemented by several car manufacturers [30]. Among the five models (S_0, A_0, ..., A_3) incrementally defined, we present here S_0, A_0 and A_1 which fit the following requirements:

S_0, A_0: the front-lighting system comprises side lamps, low and high beams that the driver chooses according to a precise protocol. There are two driver commands: a manual lighting control position switch (Fig. 5.1) and a low and high beam lever (Fig. 5.3). The lighting control switch offers "off" (A), "side lights" (B) and "headlamps"

(C) positions. It is only when this switch is in the C position that the driver can change between the low and high beams with the lever. In any position, the low and high beam lever also offers a flash command.

A_1: an automatic mode is provided (Fig. 5.2, position D), which switches headlamps on and off, depending on the ambient light. High beams are still manually activated.

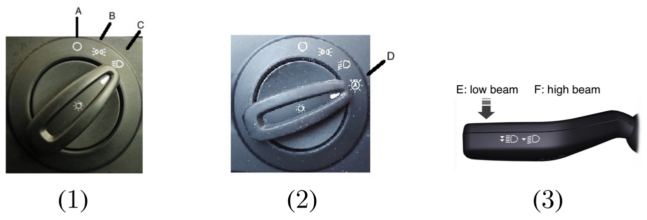


Fig. 5. Driver commands: (1) manual lighting control, (2) lighting control with auto mode, (3) low and high beam lever.

A. *Refinement solving: S_0 component and its refinement into A_0*

The first primitive component S_0 (Fig. 6) provides two interfaces (Driver Light Switch and Driver High Beam corresponding to Fig. 5.1 and Fig. 5.3) and commands the lighting device through a required interface.

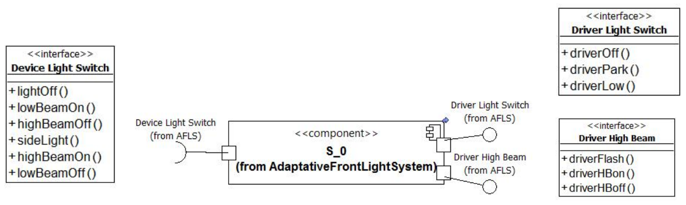


Fig. 6. Component S_0 and its interfaces

The behavioral specification of S_0 (Fig. 7) has two roles: (1) it defines when operations are provided to the driver: in particular, driverHBon and driverHBoff are only possible when the switch is in LowBeam mode; (2) it translates the driver commands into the lamp device operations: for instance driverLow switches low beams on, but keeps side lights on, driverPark switches side lights on or switches low beams off, and driverFlash effect is described by an activity of two sequenced operations: highBeamOn followed by highBeamOff.

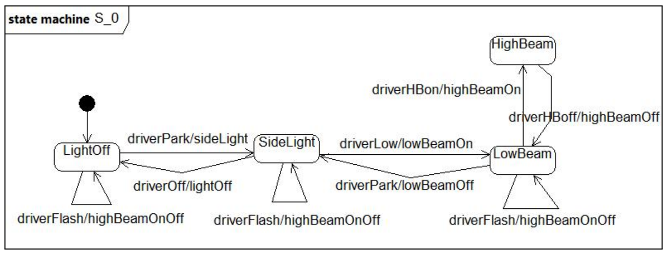


Fig. 7. State machine S_0

The goal is to define a first architecture A_0 which fulfills S_0 behavioral specification with respect to the refinement equivalence:

$$S_0 =_{\mathsf{REF}} A_0. \tag{2}$$

We apply the *refinement solving* technique. By separation of concerns, we extract the two roles of S_0 into two components: DriverProtocol$_0$ and LightControl$_0$. External interfaces of A_0 are the same as S_0, and a new interface named Light Controller is provided by DriverProtocol$_0$ and required by LightControl$_0$. The interface Light Controller is an exact relabeling of the six driver commands. All operations of Light Controller are driven by con_1 connector. By applying the refinement solving technique, the designer proposes the state machine DriverProtocol$_0$ and seeks for LightControl$_0$ such that:

$$A_0 := \textbf{hide}\, con_1 \textbf{ in } \mathsf{LightControl}_0 \;||_{con1}\; \mathsf{DriverProtocol}_0 \tag{3}$$

and A_0 satisfies equation (2).

UML A_0 definition is shown in Fig. 8. As shown in the refinement solving definition and equation (1), the only unknown component state machine of equation (3) is LightControl$_0$. The state machine DriverProtocol$_0$ (Fig. 9) is given by the designer: it is manually derived from S_0, where all transition triggers are accepted after the same sequences, but transition effects strictly correspond to call events. DriverProtocol$_0$ state names (A, B, CE, CF) correspond to driver command positions (Fig. 5).

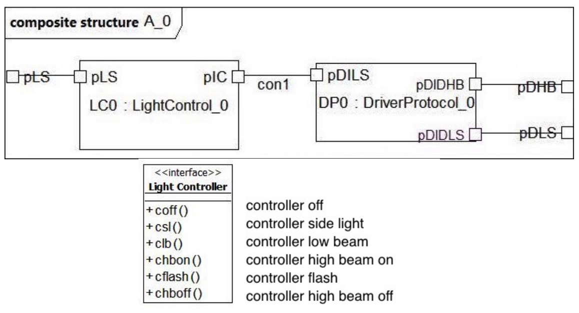


Fig. 8. Architecture A_0

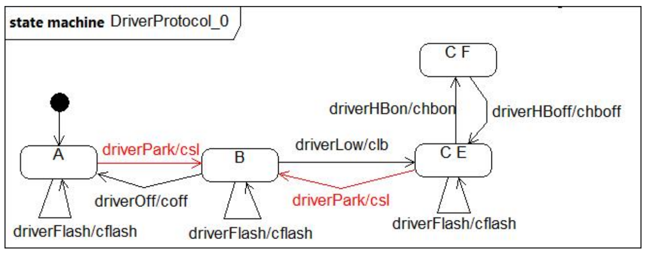


Fig. 9. State machine DriverProtocol$_0$

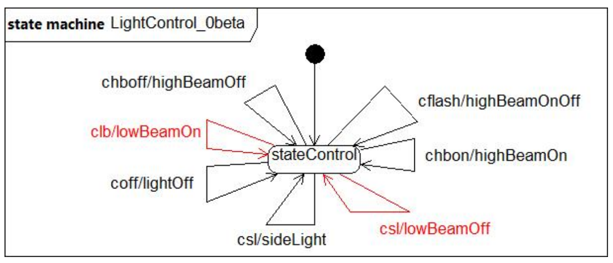


Fig. 10. State machine LightControl$_{0\mathrm{beta}}$

There is no full automatic procedure to find a component state machine $\mathsf{LightControl}_0$ which satisfies equation (2). However, the architecture A_0 determines $\mathsf{LightControl}_0$ interfaces, and the IDCM tool can automatically check whether the refinement relation ($=_{\mathsf{REF}}$) is verified or not. The designer first "tries" with the naive state machine $\mathsf{LightControl}_{0\mathsf{beta}}$ (Fig. 10) which corresponds to a simple translation table of driver commands into the device orders.

Verification within IDCM. Equation (2) is not satisfied. The tool exhibits: (i) a safety issue: A_0 may accept lowBeamOff after driverPark, which S_0 cannot accept; (ii) a liveness issue: A_0 may refuse sideLight after driverPark, which S_0 must accept.

More precisely, IDCM finds a trace, $\sigma = \mathsf{driverPark}$, after which two acceptance sets are not related by the desired set of set inclusion, although they should:

$$Acc(A_0, \sigma) \not\subset\subset Acc(S_0, \sigma),$$

where

$$Acc(A_0, \sigma) = \{\{\mathsf{sideLight}\}, \{\mathsf{lowBeamOff}\}, \{\mathsf{sideLight}, \mathsf{lowBeamOff}\}\}$$
$$Acc(S_0, \sigma) = \{\{\mathsf{sideLight}\}\}.$$

This shows that A_0 may accept lowBeamOff and may refuse sideLight after driverPark.

Hence, we modify $\mathsf{LightControl}_{0\mathsf{beta}}$ into $\mathsf{LightControl}_0$ (Fig. 11). Equation (2) is now satisfied. Note that this equation could have other solutions. In particular, $\mathsf{LightControl}_0$ has transitions (in blue) that will never be triggered by $\mathsf{DriverProtocol}_0$: these are the transitions on state Off triggered by clb, chbon, chboff and coff.

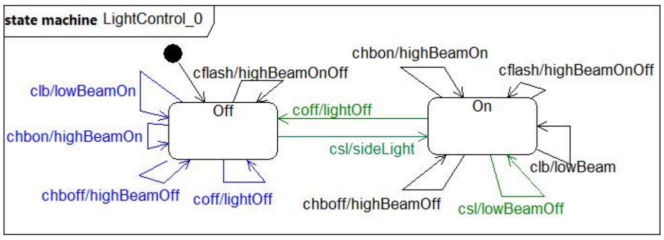


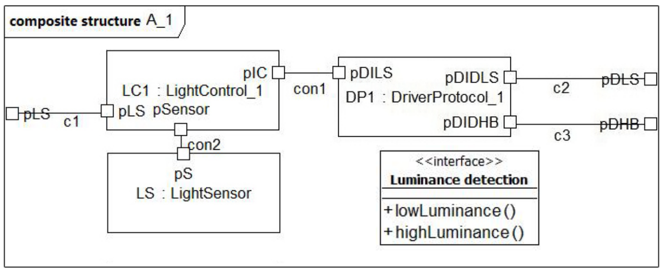
Fig. 11. State machine $\mathsf{LightControl}_0$, such that $S_0 =_{\mathsf{REF}} A_0$.

Here, the refinement solving technique remains at the same abstraction level as S_0, but it clarifies the two main roles of S_0. Compared to S_0, the two resulting components are slightly simpler: $\mathsf{DriverProtocol}_0$ only controls when commands are enabled or disabled to the driver, and $\mathsf{LightControl}_0$ translates driver commands into the proper light calls.

B. Component adding: construction of A_1

We want to include a known component LightSensor which evaluates the ambient light intensity. The *component adding* technique is used for this purpose. It consists here in proposing architecture A_1 (Fig. 12) including $\mathsf{DriverProtocol}_1$ and $\mathsf{LightControl}_1$ and identifying a set T_s of unwanted traces, such that:

$$\mathsf{DriverProtocol}_0 \sqsubseteq_{\mathsf{EXT}} \mathsf{DriverProtocol}_1, \quad (4)$$
$$\mathsf{LightControl}_0 \sqsubseteq_{\mathsf{EXT}} \mathsf{LightControl}_1, \quad (5)$$
$$A_0 \sqsubseteq_{\mathsf{EXT}} A_1 \quad (6)$$
$$\forall t_s \in T_s \ . \ t_s \not\sqsubseteq_{\mathsf{MAY}} A_1 \quad (7)$$



$$A_1 := \textbf{hide } con_1, con_2 \textbf{ in } (\mathsf{LightControl}_1 \ ||_{con_1,con_2} \ (\mathsf{LightSensor} \ ||| \ \mathsf{DriverProtocol}_1))$$

Fig. 12. Architecture A_1

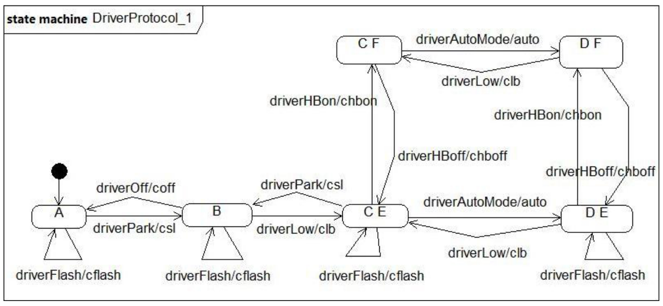


Fig. 13. State machine $\mathsf{DriverProtocol}_1$

The state machine $\mathsf{DriverProtocol}_1$ is first defined (Fig. 13), including a new operation: driverAutoMode. The state machine $\mathsf{LightControl}_{1\mathsf{beta}}$ is then defined (Fig. 14). It includes the new events cauto, highLuminance and lowLuminance.

Verification within IDCM. Properties (4)–(6) (with the beta version) are satisfied. In unwanted traces T_s, we define the trace σ_1 which says that when autolamps has switched the low beams *off*, the driver switches the high beams on (the driver should *not* be able to switch the high beams on):

σ_1 := driverPark; sideLight; driverLow; lowBeamOn; driverAutoMode; ⟨internal sensor high luminance⟩; lowBeamOff; driverHBon; highBeamOn.

The LTS t_1 is built upon trace σ_1. We observe with IDCM that $t_1 \sqsubseteq_{\mathsf{MAY}} A_1$ (property (7) is not satisfied).

Under IDCM, this trace inclusion is verified by an observational simulation relation between t_1 and the acceptance graph obtained from A_1 (let us recall that an acceptance graph is a deterministic automaton where each state is associated with acceptance sets). Hence, IDCM detailed verdict is the set of tuples belonging to the simulation relation found:

A_1 `simulates` t_1.

If the designer wants to understand this simulation, the IDCM tool can give the state numbers that are related by the simulation relation:

`the simulation relation contains 16 pairs:` $\{(9,16),(68,15),(67,14),\ldots,(224,1)\}$.

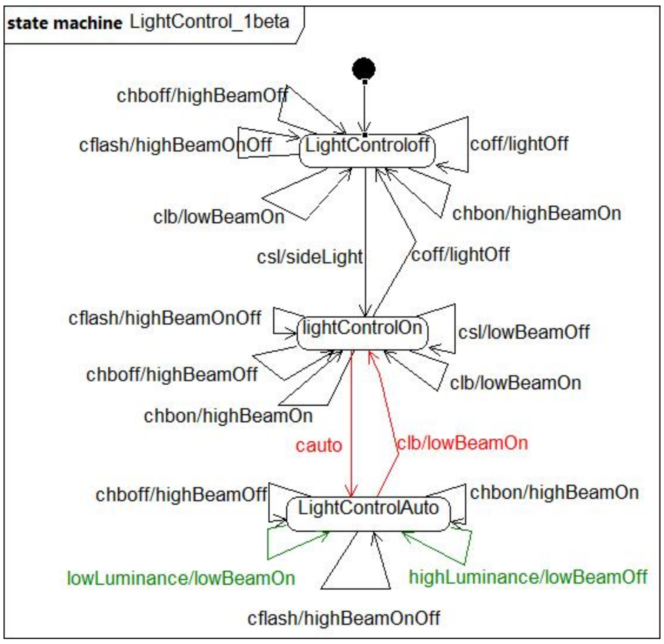


Fig. 14. State machine LightControl$_{1beta}$

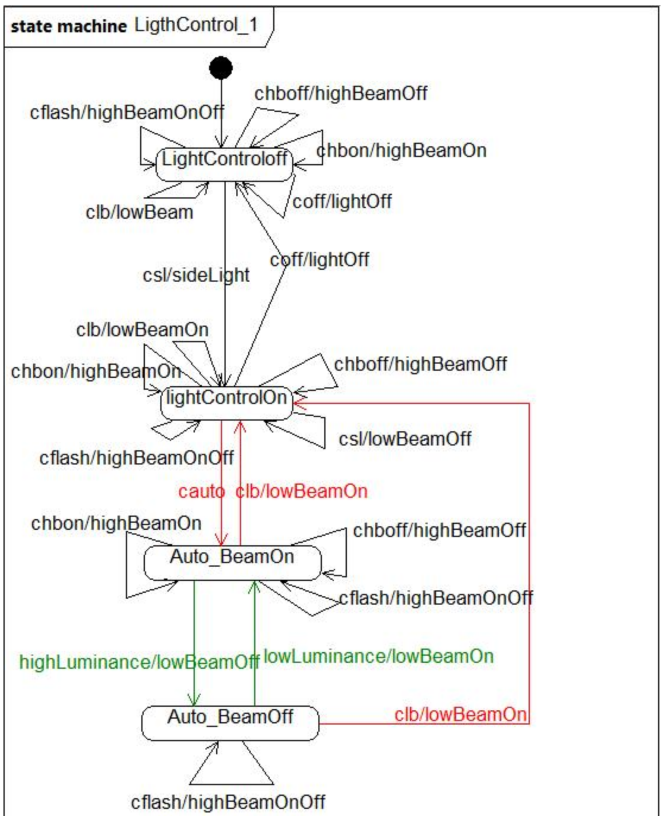


Fig. 15. State machine LightControl$_1$

The state machine LightControl$_1$ (Fig.15) fixes it: after highLuminance/lowBeamOff transition, state machine LightControl$_1$ now refuses the trigger event chbon.

Within IDCM, we observe that A_1 does not simulate t_1. In this case, IDCM recalls the trace σ_1, which is included in t_1 and not in A_1.

The state machine DriverProtocol$_1$ is unchanged. Of course, this verification process requires other safety traces not detailed here. We should define other traces, not detailed here, until we consider property (7) as satisfied.

Such verifications ensure liveness property preservation, which guarantee that the driver can still use the light commands as he/she was using them manually, and examine on particular traces safety properties.

V. Discussion: validation and related works

Although the informal AFLS specifications are simple, UML state machines quickly become intricate. Concurrent behaviors and communications are complex to model and to analyze. Proposed construction techniques provide some assistance: in A_0, the refinement solving technique shows that a component can be deduced from a first naive trial and in A_1 the component adding technique shows that an extended component also has to be checked on previous safety properties. We therefore consider the questions of incremental relation validity and construction technique validity.

A. Validation

Relation validation. As stated in [21], UML state machine transformation into LTS is sound: any linear temporal logic property satisfied by a generated LTS is also satisfied by the originating state machine. Nevertheless, if we compare two state machines by comparing their LTS, some state machine mistakes may by forgotten and some false alarms may appear. Again, refer to [21] for relation validation details.

Construction technique validation. Refinement and extension techniques preserve liveness properties. Refinement techniques also preserve safety properties. Extension techniques include a complementary verification of a set of safety properties. One of the main advantages of such incremental techniques is to offer some verification means without asking the designer to use other specification languages such as temporal logic. However, these techniques have some limits: for refinement and extension techniques, new desired liveness properties are not analyzed; for refinement techniques, new safety properties are not verified. The designer has to know the scope of such implicit verification techniques and has to use complementary means if required. For such purposes, refinement and extension techniques have to be completed by a separate verification of a set T_l of liveness properties for new detailed or extended behaviors.

Scalability. For a UML primitive component with n states, k transitions and m operation occurrences (number of operation calls which appear on transition effects), the generated LTS has at most n_{LTS} states and k_{LTS} transitions where $n_{LTS} = n + 2k + m$ and $k_{LTS} = 2(k + m)$.

The factor 2 is explained by the following transformation rule: for each transition triggered by a call event and each synchronous operation call, there are two LTS transitions representing operation call and reception, and consequently, new intermediate added states.

Parallel compositions of LTS generate interleaved sequential LTS, whose sizes are in worst case equal to the product of composed LTS sizes. Hence, for manually designed UML composite components, generated LTS sizes range from medium to large sizes. For instance, for a composition of five UML components, each of them described by a state machine of $n = 5$ states, $k = 10$ transitions, $m = 5$ operation occurrences (i.e. 30 LTS states), the generated LTS of the composition has in the worst case $30^5 \simeq 24.10^6$ states and around 200000 states if half of the transitions of the composite components are synchronized with one transition of another component. Such large LTS can still be handled by simulation relations. For conformance relations, LTS up to 1 million states can be supported. We have to keep in mind that proposed verification techniques are performed during the model construction, when model sizes are still reasonable.

B. Related Work

Within the increasing number of works dealing with architecture based analysis [10], they mainly address liveness analysis using bisimulation techniques or dead-lock detection and do not ensure extension, refinement or substitutability of models [21]. They only focus on safety analysis using explicit property checking. To the best of our knowledge, no work has defined relations for incremental development of architectural models, defined in UML. Table I gives the synthesis of the analyzed approaches along liveness, safety, substitution, extension and refinement aspects.

	Liv.	Saf.	Sub.	Ext.	Ref.
UML/Wright [20]	✓	✓			✓
UML/B [28]	∼	✓	✓		✓
SysML/Interface automata [15]	∼				
UML/omega2 [23]	∼	✓			
AADL/FIACRE [8]	✓	✓			
AADL/BIP[14]		✓			
Archware (LOTOS) [24]	✓	✓			
PADL-Æmilia [1]	✓	✓			
SafArchie [4]		✓	∼		
FIESTA [32]					∼

✓: supported; ∼: partially supported; ‘ ’: not supported;

TABLE I
EVALUATION OF ARCHITECTURAL AND VERIFICATION TOOLS.

[20] proposes a UML profile and translates UML models into Wright for using the model checker FDR. FDR focuses on safety and liveness analyses without fairness assumption. It does not analyze any extension nor substitution relation. Some work such as [28] focus on translating UML into B or Z. They include refinement techniques but do not address extension techniques. [15] considers SysML models in order to verify components assemblies. They perform behavioral compatibility verifications, but do not analyze any liveness property other than dead-lock detection and do not address extension and refinement problems. [23] has extended the analysis techniques proposed by [18] which defined OMEGA2, a UML profile. Architectures are translated into IF/IFx models [11], [12] in order to be analyzed by the CADP toolbox [19] for safety property analysis. However, model substitutability, extension and refinement are not supported.

[8] considers AADL descriptions and transforms them into FIACRE in order to apply the model checker TINA [9]. TINA analyzes safety, liveness and deadlocks under the fairness hypothesis, but it does not address extension, refinement and substitutability. [14] has a similar approach by translating AADL into the BIP language [5]. BIP focuses on safety properties and does not address liveness, extension, refinement, nor substitutability. Archware [25], [24] is a framework based on the LOTOS language allowing the use of the CADP model checker [19]. Safety and liveness properties are analyzed under fairness assumption. Compatibility between components is verified, but no extension nor substitution relations is considered. PADL and Æmilia [7], [1] are languages based on a stochastic process algebra. They are associated with the model checker TwoTowers [6]. Analyses can be conducted according to several bisimulation relations. It appears that these relations are too strong for incremental developments.

SafArchie and TranSAT framework [4] deal with the evolution of architectures using safe patterns. The compatibility between components is addressed from different points of view: structural, functional and behavioral. Substitutability of components is studied from a syntactical point of view by considering interfaces. This does not guarantee the behavioral conformance of the architecture in which the component is substituted. FIESTA [32] defines a generic framework where new components are introduced into architectural models. It is based on a pattern approach and focus on adding or modifying connections in order to ensure the compatibility between components. This work addresses a part of the incremental development in so far as the structural compatibility does not guarantee the behavioral one.

VI. Conclusion

We propose architectural modeling techniques for reactive systems, which combine two main points: assisting the design of composite components and state machines; early detecting and fixing safety and liveness issues. These techniques cover refinement and extension approaches. The proposed techniques assist the designer in setting up models which satisfy desired constraints, as well as finding and reusing formerly defined components, by verifying if a new architecture is actually an increment or not of the previous one. All the verification relations used are implemented in the IDCM tool.

Further work consists in enriching refinement and extension techniques by the verification of a set of *new* liveness properties. In contrast with safety properties, which are traces that can be modeled in UML by sequence diagrams, we plan to model liveness properties by separate UML state machines. A UML profile for incremental construction, taking into account component increments but also operation increments, is currently being developed.

References

[1] A. Aldini and M. Bernardo. On the usability of process algebra: An architectural view. *Theoretical Computer Science*, 335(2-3):281–329, May 2005.

[2] B. Alpern and F. Schneider. Defining liveness. *Information processing letters*, 21(October):181–185, 1985.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987.

[4] O. Barais, E. Cariou, L. Duchien, N. Pessemier, and L. Seinturier. Transat: A framework for the specification of software architecture evolution. *Issues on Coordination and Adaptation Techniques*, pages 31–38, 2004.

[5] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society Washington.

[6] M. Bernardo. TwoTowers 5.1 User Manual, 2006.

[7] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, Oct. 2002.

[8] B. Berthomieu and J.-P. Bodeveix. Formal Verification of AADL models with Fiacre and Tina. In *Embedded Real-Time Software and Systems (ERTS 2010)*, 2010.

[9] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA: Construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14):2741–2756, 2004.

[10] A. Bertolino, P. Inverardi, and H. Muccini. Software architecture-based analysis and testing: a look into achievements and future challenges. *Computing*, 95(8):633–648, 2013.

[11] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *International Conference on Computer Aided Verification*, pages 343–348. Springer, 2002.

[12] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 237–267. Springer Berlin Heidelberg, 2004.

[13] E. Brinksma and G. Scollo. Formal Notions of Implementation and Conformance in LOTOS. Technical report, Twente University of technology, Enschede, Dec. 1986.

[14] M. Y. Chkouri and M. Bozga. Prototyping of distributed embedded systems using AADL. *ACESMB 2009*, pages 65–79, 2009.

[15] S. Chouali and A. Hammad. Formal verification of components assembly based on SysML and interface automata. *Innovations in Systems and Software Engineering*, 7(4):265–274, Oct. 2011.

[16] A.-L. Courbis and T. Lambolais. IDCM. http://idcm.wp.mines-telecom.fr. Accessed: 2017-04-01.

[17] A.-L. Courbis, T. Lambolais, H.-V. Luong, T.-L. Phan, C. Urtado, and S. Vauttier. A formal support for incremental behavior specification in agile development. In *The 24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 694–699, 2012.

[18] A. Cuccuru. Meaningful composite structures. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *LNCS*, pages 828–842. Springer Berlin Heidelberg, 2008.

[19] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*, pages 372–387. Springer Berlin Heidelberg, Saarbrücken, 2011.

[20] M. Graiet, M. T. Bhiri, F. Dammak, and J.-P. Giraudin. Adaptation d'UML2.0 à l'ADL Wright. In *CAL*, pages 83–100, 2006.

[21] T. Lambolais, A.-L. Courbis, H.-V. Luong, and C. Percebois. IDF: A framework for the incremental development and conformance verification of UML active primitive components. *Journal of Systems and Software*, 113:275–295, 2016.

[22] G. Leduc. Conformance relation, associated equivalence, and minimum canonical tester in LOTOS. *PSTV XI. North-Holland*, pages 249–264, 1991.

[23] I. Ober and I. Dragomir. Unambiguous UML composite structures: the OMEGA2 experience. *SOFSEM 2011: Theory and Practice of Computer Science*, pages 418–430, 2011.

[24] F. Oquendo. π-Method: A Model-Driven Formal Method for Architecture-Centric Software Engineering. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–13, 2006.

[25] F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, and C. Occhipinti. ArchWare: Architecting Evolvable Software. In *Software Architectures*, volume 3047 of *LNCS*, pages 257–271. Springer Berlin Heidelberg, 2004.

[26] Oracle-Corporation. The Java Tutorials — Trial Essential Classes: Concurrency. Liveness, 2013.

[27] T.-L. Phan. *Développement Incrémental de Spécifications d'Architectures en UML Intégrant des Procédures de Vérification*. PhD thesis, Montpellier 2, France, 2013.

[28] M. Y. Said, M. Butler, and C. Snook. A method of refinement in UML-B. *Software & Systems Modeling*, 14(4):1557–1580, 2015.

[29] F. B. Schneider. Decomposing Properties into Safety and Liveness using Predicate Logic. Technical report, Cornell Univ. Ithaca, NY, Dept. of Computer Science, 1987.

[30] Texas-Instruments. Automotive Adaptive Front-lighting System Reference Design. Technical Report SPRUHP3, Texas Instruments, System Application Engineering, July 2013.

[31] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer. *Software Architecture: a Comprehensive Framework and Guide for Practitioners*. Springer Science & Business Media, 2011.

[32] G. Waignier, A.-F. Le Meur, and L. Duchien. FIESTA: A Generic Framework for Integrating New Functionalities into Software Architectures. In F. Oquendo, editor, *Software Architecture*, volume 4758 of *LNCS*, pages 76–91. Springer Berlin Heidelberg, 2007.