Property Expression and Verification in an Incremental Model Development Framework: a Case Study

Thomas Lambolais, Anne-Lise Courbis

Property Expression and Verification in an Incremental Model Development Framework: a Case Study

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

***Abstract*—The IDCM framework (Incremental Development of Conforming Models) supports incremental constructions and evaluations of UML behavioral models (architecture of components and state machines). This framework evaluates models with respect to *implicit* temporal safety and liveness properties. The specifiers and designers only describe models, they don't need to write down explicit temporal logic properties. In this paper, we show how explicit safety and liveness properties can also be described using semi-formal boilerplates, translated into classical temporal logic formulas which are themselves translated into testing transition systems.**

Adding evaluation means to model-based development (relation checking) is a way to develop models incrementally, following a spiral-based development cycle.

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 [26], we proposed a set of architectural construction techniques which (i) contribute to architectural model qualities through well-known design principles in software engineering (separation of concerns, hierarchy and information hiding); (ii) include formal verifications for early detection of behavioral issues, i.e. safety and liveness problems.

In this paper, we extend this approach and show how typical informal temporal properties, described by boiler plates, can be translated into Labelled Transition Systems and verified by our framework.

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 the way we can define and verify explicit safety and liveness properties. A case study is given in Section 4, on which two properties are specified. Comparison with existing work is given 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 [26], [27]. At each step, the new model is compared to the previous one through a suitable relation, which focuses on behavioral and temporal aspects.

As shown by Alpern and Schneider [35], 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 $P\ \mathcal{R}\ Q$, $P \models \phi \Rightarrow Q \models \phi$.

We observe safety and liveness properties by means of the *interactions* of the system with its environment. These interactions consist in accepting an *event* (signal or operation reception), or performing an *action* requiring a signal send or operation call. A trace is a partial sequence of observable interactions starting from the initial state.

The LTS (Labeled Transition Systems) semantics we give to UML primitive components behavior state machines is not recalled here, see [16], [27]. Let us briefly recall that an LTS P is a tuple $\langle \mathcal{P}, Act, \rightarrow, P \rangle$ where:

- $\mathcal{P}$ is a set of state names,
- Act is a set of action names,
- $\rightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$ is a set of labeled transitions between states,
- P is the initial state.

The tool we have developed includes UML State Machine transformation into LTS [15]. Here follows an intuitive presentation of incremental relations, and a proposal for the UML composite component semantics.

A. Incremental relations between UML components.

Based on classical trace inclusion $\sqsubseteq_{\mathsf{MAY}}$ and conformance relation conf [28], the IDCM framework implements several *incremental* relations. We only give an intuitive presentation here of conf, $\sqsubseteq_{\mathsf{INC}}$, $\sqsubseteq_{\mathsf{REF}}$ and $=_{\mathsf{REF}}$, which will be used in sections III and IV. Refer to [16], [27], [33] to find formal definitions and an extensive presentation of other incremental relations. Given two models M_1 and M_2, $M_2 \sqsubseteq_{\mathsf{MAY}} M_1$ 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. $=_{\mathsf{REF}}$ is the equivalence relation associated to $\sqsubseteq_{\mathsf{REF}}$.

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

B. UML composite components semantics.

UML composite components describe architectural systems in terms 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 [21]. EXPOPEN shares the same concepts as basic LOTOS process algebra [28]. Secondly, EXPOPEN models are translated into LTS by the CADP tool [21].

For instance, fig. 1 presents a UML architecture (A_0) which models an automotive front-light system (see details in section IV). In the architecture A_0, there are three component instances linked by assembly connectors c3 and c4. There are three delegation connectors c1, c2 and c5 that link ports of the architecture to ports of its components. Ports linked by a connector share the same UML interface.

The external interfaces of A_0 correspond to actions exchanged on delegation connectors c1, c2 and c5. These actions must be observable to verify the properties of the system, while the set of synchronized actions on assembly connectors c3 and c4 are not on the focus of verification and will be hidden.

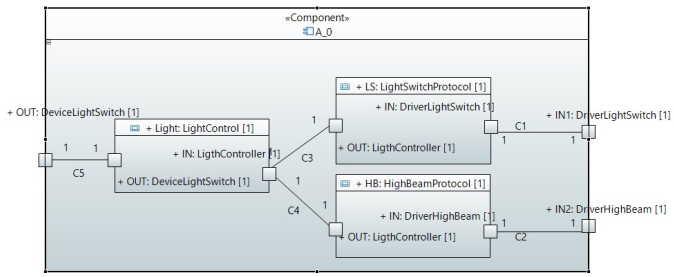


Fig. 1. A UML composite component (A_0)

III. PROPERTY DEFINITION AND VERIFICATION

Let a and b be two actions ($a, b \in Act$). We illustrate here two patterns of temporal properties to be satisfied by a model M, for safety (*S*) and liveness (*L*) properties:

(*S*) *"In any $\langle a \rangle$-circumstances, action b is never possible."*:

- an $\langle a \rangle$-circumstance means that M offers action a, what is formally written, in Hennessy Milner Logic (HML): $M \models \langle a \rangle tt$
- b is not possible is written, in HML: $M \models [b]ff$
- Hence, using usual logical implication $\Rightarrow$ and modal operator $\Box$ (always) for convenience, property (*S*) corresponds to:

$$M \models \Box(\langle a \rangle tt \Rightarrow [b]ff)$$

- This is a general expression of a safety property. Derived expressions of this kind are simply "*b is never offered*" or "*b is always possible*", since action b may also be a positive statement: $M \models \Box\langle b \rangle tt$
- For example, "*In any circumstances, the flash action is always offered*".

(*L*) *"In any circumstances, every a-action leads eventually to b."*:

- Using $\Box$ and $\Diamond$ (eventually) operators, such properties can be formally written:

$$M \models \Box([a]\Diamond(\langle b \rangle tt \wedge [Act - b]ff)$$

– A property $[a]F$ states in Hennessy-Milner logics that, for all processes after a, F is true. If there is no a-successor, this property is true whatever the value of F.

– $\langle b \rangle tt \wedge [Act - b]ff$ is true when there exists a b-successor, and no other successor.

- This is a general expression of liveness property: after any a-action, b will be done.
- For example, "*When the system is in headlamp mode (low or high beam), switching back to side lights switches off high beams and low beams*".

A. Verification of safety properties (S)

1) General case: Properties of kind (*S*) lead to:

$$M \models \Box([a]ff \vee [b]ff).$$

Such safety properties define unwanted action. We define the LTS T (using '+', '.' operators and recursion textual notations for convenience), which accepts at any time actions a or b:

$$T = a.T + b.T$$

Then, we observe the set of systems $ObsSet$ after any trace of M:

$$ObsSet = \{M' \mid M \stackrel{\sigma}{\Longrightarrow} M', \forall \sigma \in Tr(M)\}$$

Let us recall that $Tr(M)$, the set of traces of M, are all the sequences of observable actions starting from the initial state, and that $M \stackrel{a_1 \cdots a_n}{\Longrightarrow} M' = M \xrightarrow{\tau^*} \xrightarrow{a_1} \xrightarrow{\tau^*} \cdots \xrightarrow{\tau^*} \xrightarrow{a_n} \xrightarrow{\tau^*} M'$.

If the safety property (*S*) is satisfied by M, no set of $ObsSet$ should simulate T:

$$\forall M' \in ObsSet.\ T \not\sqsubseteq M'$$

The $\sqsubseteq$ relation is the preorder associated to Milner's congruence equivalence. This verification corresponds to a simulation, which is convenient to verify safety properties and the most efficient for our purposes.

2) Particular case for positive statements when there is no premise: We may choose a simpler verification means, when safety properties (*S*) are of the kind:

$$M \models \langle b \rangle tt \wedge \Box \bigcirc \langle b \rangle tt$$

which means that M must always accept b every two actions ($\bigcirc F$ means that F is true in the next state).

We define process

$$T = b.\sum_{a \in Act} a.T$$

T is a process that "always" does b, at least every two following actions, and possibly several times in sequence ($b \in Act$).

In order to check the safety property, we verify that the process T synchronized with M on every actions (operator '||)

$$T||M$$

is deadlock free. This means that M can always do T actions. Absence of deadlock can be verified in IDCM.

B. Verification of liveness properties (L)

We consider properties of the kind (*L*):

$$M \models \Box([a]\Diamond\langle b \rangle tt \wedge [Act - b]ff).$$

Let $success \notin Act$. We define LTS T and T_b as follows:

$$\begin{aligned} T &= \langle Act - a \rangle.success.T + a.T_b \\ T_b &= b.success.T + \langle Act - b \rangle.T_b \end{aligned}$$

Then, we observe the system M synchronized with T on every actions (operator '||'), where every actions except $success$ are hidden:

$$Obs = \textbf{hide} \;\; Act - success \;\textbf{in}\; (M||T)$$

If the liveness property (*L*) is satisfied by M, Obs should be refinement equivalent to a process $Ok = success.Ok$:

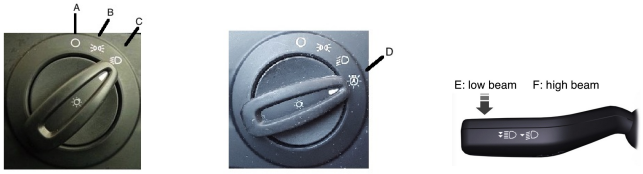
$$Obs =_{\mathsf{REF}} Ok$$

which means that, when all actions are hidden except success, Obs should perform $success$ infinitely often.

IV. Illustration: Adaptive Front-lighting System

We consider a car Adaptive Front-lighting System (AFLS) implemented by several car manufacturers [36]. Among the five models (S_0, A_0, ..., A_3) incrementally defined, we present here S_0 and A_0 which fit the following requirements:

(*User informal req.*): 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. 2.1) and a low and high beam lever (Fig. 2.3). 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.



(1) (2) (3)

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

A. S_0 and A_0 models

S_0 is intended to be a primitive component, representing the initial specification, whose behaviour is described by a single State Machine. A_0, representing a possible realization of S_0, is a composite component describing an architecture. Both S_0 and A_0 have the same outside provided and required interfaces (Fig. 3): Driver Light Switch and Driver High Beam correspond to Fig. 2.1 and Fig. 2.3, Device Light Switch is the required interface which commands the lighting device through Driver Light Switch and Driver High Beam interfaces correspond to driver commands of Fig.2.1 and Fig.2.3.

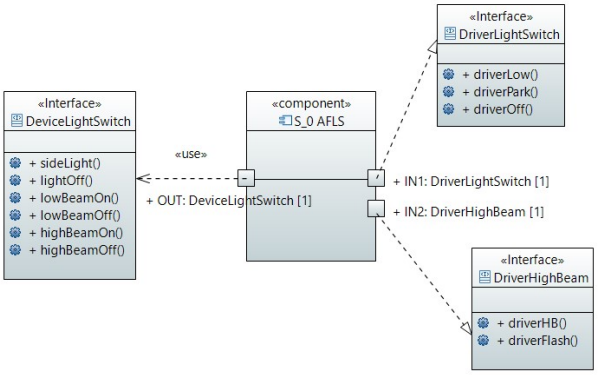


Fig. 3. UML provided and required interfaces for S_0 component

In complement to further models, we developed a Java prototype application, which simulates the AFLS behaviour (Fig. 4). The two driver commands are the two button lines at the bottom (the low and high beam lever is grey, while the beige one can only go from one position to its successive or preceding position).

The behavioral specification of S_0 (Fig. 5) has two roles: (i) it defines when operations are provided to the driver: in particular, driverHBon and driverHBoff are only possible when the switch is in LowBeam mode; (ii) 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. 4. Example of Graphical User Interface associated to a Java simulation of the AFLS.

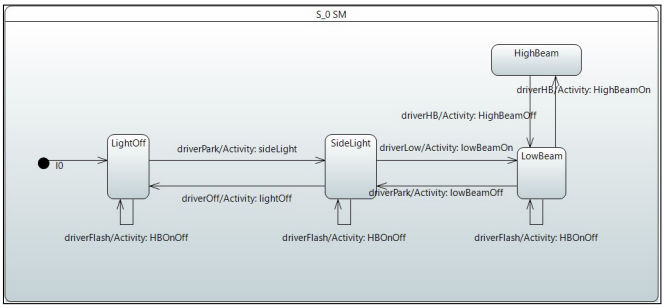


Fig. 5. S_0 state machine

In [26], we described a way to build an architecture A_0 which is a correct refinement of S_0:

$$S_0 =_{\mathsf{REF}} A_0. \tag{1}$$

It leads to A_0 (described in Fig. 1) which connects three primitive components. We give here the state machines of HighBeamProtocol (Fig. 6) and LightControl (Fig. 7).

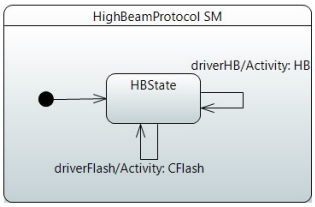


Fig. 6. HighBeamProtocol state machine

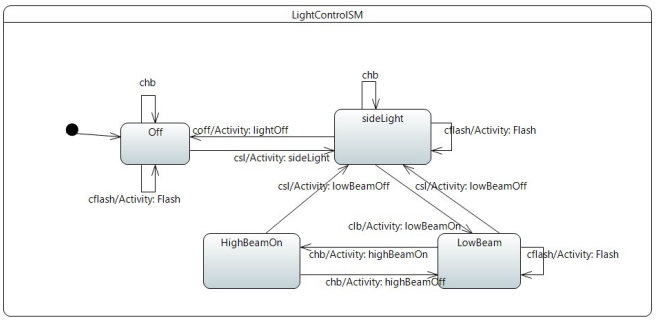


Fig. 7. LightControl state machine

B. Specification and verification of typical properties

While equation (1) guaranties that A_0 and S_0 share the same safety and liveness properties, obviously, it does not guarantee that S_0 satisfies the informal requirements (*User informal req.*).

1) Example of safety property: "*The system can always accept* driverFlash *commands every two steps, except in highbeam mode*." This is the following ϕ safety property:

$$\phi = \Box(\langle \mathsf{driverFlash} \rangle . \langle Act - \mathsf{driverHB} \rangle . \langle \mathsf{driverFlash} \rangle . tt)$$

In order to check this property, we build the LTS presented in Fig. 8, which first tests a driver flash action, and after any other action (driver low, driver park, driver off), tries to perform a driver flash action again.

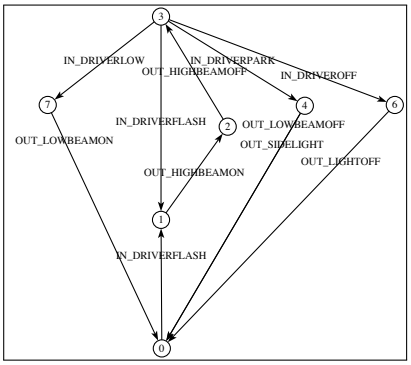


Fig. 8. LTS T to check if S_0 or A_0 can always accept the DRIVERFLASH action, except in high beam mode.

Using model transformation and IDCM, we can check that property ϕ is satisfied:

$$\textbf{hide } \mathsf{driverHB} \textbf{ in } (S_0 \,||\, T)$$

is deadlock-free.

2) Example of liveness property: "*In low/highBeam mode,* driverPark *command always switches off low beams.*" This is the following ψ property:

$$\psi = \Box([\mathsf{driverLow}][\mathsf{driverPark}]\Diamond(\langle \mathsf{lowBeamOff} \rangle tt \wedge [Act - \mathsf{lowBeamOff}]ff))$$

Let $G = Act - \mathsf{driverLow} - \mathsf{driverPark}$. The testing LTS T is:

$$T = \langle G \rangle . success . T + \mathsf{driverLow.driverPark}.T_2$$
$$T_2 = \mathsf{lowBeamOff}.success.T + \langle G - \mathsf{lowBeamOff} \rangle . T_2$$

It appears that

$$\textbf{hide } Act - success \textbf{ in } (S_0 || T) \neq_{\mathsf{REF}} Ok.$$

with $Ok = success.Ok$

Indeed, when in HighBeam state, S_0 can not perform driverPark followed by lowBeamOff: a transition is missing from HighBeam state to SideLight state, triggered by driverPark.

V. Related Work

Most of the works dealing with verification (model-checking and theorem proving) take formal specifications as inputs. We used CADP toolbox [21], as well as CCS tools such as CAAL [1], [38]. Model checking tools use temporal logics (for instance Hennessy-Milner Logics in CAAL) do describe and verify properties. Compared to such tools, our objective

in the IDCM framework is threefold: (i) taking as input UML architectures and state machines, i.e. semi-formal descriptions (ii) describing properties within the same language (iii) providing incremental relations of refinement and extension. In this paper, safety and liveness properties are verified by LTS comparisons and deadlock detection, without having to use temporal logics.

Moreover, the increasing number of works dealing with formal model based analysis [9] do not ensure extension, refinement or substitutability of models [27]. 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 [23]	✓	✓			✓
UML/B [34]	∼	✓	✓		✓
SysML/Interface automata [14]	∼				
UML/omega2 [30]	∼	✓			
AADL/FIACRE [7]	✓	✓			
AADL/BIP[13]		✓			
Archware (LOTOS) [31]	✓	✓			
PADL-Æmilia [2]	✓	✓			
SafArchie [3]		✓	∼		
FIESTA [37]					∼

✓: supported; ∼: partially supported; ' ': not supported;

TABLE I
EVALUATION OF ARCHITECTURAL AND VERIFICATION TOOLS.

[23] 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 [34] focus on translating UML into B or Z. They include refinement techniques but do not address extension techniques. [14] 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. [30] has extended the analysis techniques proposed by [18] which defined OMEGA2, a UML profile. Architectures are translated into IF/IFx models [10], [11] in order to be analyzed by the CADP toolbox [21] for safety property analysis. However, model substitutability, extension and refinement are not supported.

[7] considers AADL descriptions and transforms them into FIACRE in order to apply the model checker TINA [8]. TINA analyzes safety, liveness and deadlocks under the fairness hypothesis, but it does not address extension, refinement and substitutability. [13] has a similar approach by translating AADL into the BIP language [4]. [17] transforms UML architecture into BIP. However, BIP focuses on safety properties and does not address liveness, extension, refinement, nor substitutability. Archware [32], [31] is a framework based on the LOTOS language allowing the use of the CADP model checker [21]. 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 [6], [2] are languages based on a stochastic process algebra. They are associated with the model checker TwoTowers [5]. Analyses can be conducted according to several bisimulation relations. It appears that these relations are too strong for incremental developments.

SafArchie and TranSAT framework [3] 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 [37] 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.

In [20], the authors propose a transformation of UML state charts and communication diagrams in LOTOS and use FOCOVE verification environment where properties expressed by CTL formulas are verified. [24] proposes UML statecharts and their synchronization transformation in LOTOS. No verification is proposed, nor extension and refinement.

[22] transforms UML protocol state machines into Alloy. No temporal properties are taken into account. Protocol state machines are convenient to express predicates on states, which depend on terms and values. We do not support such data verifications. On the opposite, standard Alloy models do not allow temporal logic verifications.

[25] presents a transformation of UML activity diagrams into Alloy. Such work has the same limits as [22] concerning temporal aspects, hence they do not verify liveness properties.

UML activity diagrams are also considered in [19], using model checkers such as: UPPAAL, SPIN, NuSMV and PES. Hence, safety and liveness properties are described in specific temporal logics. Nevertheless, the automated aspect of the Eclipse-plugin implementation of the tool allows users without a background in formal methods to verify the safety and liveness of a system.

[29] presents an interesting transformation of UML components diagram and state machines into timed automata that are checked with UPPAAL tool. [12] also proposes a transformation of UML state machines into timed automata. These work support timed properties, whereas we only consider temporal properties. However, user must provide explicit descriptions of properties using timed temporal logics.

VI. CONCLUSION

IDCM framework proposes architectural modeling techniques for reactive systems which cover refinement and extension approaches, as well as evaluation means, based on conformance and refinement relations. Such relations verify implicit safety and liveness properties. In this paper, we present patterns of explicit safety and liveness properties and a mechanism to check them on the desired models, using the refinement equivalence relation. This relation has the advantage of being weaker than the traditional observational Milner's relation.

Describing and verifying explicit properties is a complementary means to check: (i) first abstract models; (ii) extension points: in the incremental approach, we check that extension preserves liveness properties, but we were not able to check that a specific safety property is not violated by new behaviours.

This work has several limits. The designer does not need to express safety and liveness properties in a specific temporal logics, but he has to translate such properties into specific LTS. Even if we provide templates, this can be tricky task. Secondly, the UML State Machine translation into LTS does not consider data and timing aspect. We focus on 'pure' actions, without data parameters. Hence, guards, change event and time event in UML state machines are always translated by non deterministic LTS.

Further works consist in improving the IDCM tool on two points: offering a way to describe semi-formally such properties (formal translations being automatically generated); improving verdicts and counter-examples in case a relation or property is not satisfied.

REFERENCES

[1] L. Aceto, A. Ingólfsdóttir, K. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] S. Burmester, H. Giese, M. Hirsch, and D. Schilling. "incremental design and formal verification with uml/rt in the fujaba real-time tool suite". In *"International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language"*, 2004.

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

[14] 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.

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

[16] 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.

[17] A.-L. Courbis, T. Lambolais, and T.-H. Nguyen. Safe Incremental Design of UML Architectures. In *29th International Conference on Software Engineering and Knowledge Engineering*, 2017.

[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] Z. Daw, J. Mangino, and R. Cleaveland. Uml-vt: A formal verification environment for uml activity diagrams. In *P&D@ MoDELS*, pages 48–51, 2015.

[20] S. Djaaboub, E. Kerkouche, and A. Chaoui. Generating verifiable LOTOS specifications from UML models: A graph transformation-based approach. *International Journal of Embedded Systems*, 10(6):453–469, 2018.

[21] 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.

[22] A. Garis, A. C. Paiva, A. Cunha, and D. Riesco. Specifying UML protocol state machines in alloy. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7321 LNCS(June):312–326, 2012.

[23] 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.

[24] B. Hnatkowska and Z. Huzar. Transformation of Dynamic Aspects of Uml Models Into Lotos Behaviour Expressions. *International Journal of Applied Mathematics and Computer Science*, 11(2):537–556, 2001.

[25] M. Kherbouche and B. Molnár. Formal model checking and transformations of models represented in uml with alloy. In *International Workshop on Modelling to Program*, pages 127–136. Springer, 2020.

[26] T. Lambolais and A.-L. Courbis. Development and Verification of UML Architecture by Refinement and Extension Techniques. In *European Congress on Embedded Real Time Software and Systems (ERTS)*, 2018.

[27] 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.

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

[29] A. L. Muniz, A. M. Andrade, and G. Lima. Integrating uml and uppaal for designing, specifying and verifying component-based real-time systems. *Innovations in Systems and Software Engineering*, 6(1):29–37, 2010.

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

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

[32] 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.

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

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

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

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

[37] 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.

[38] J. K. Wortmann, S. R. Olesen, and S. Enevoldsen. Caal 2.0 recursive hml, distinguishing formulae, equivalence collapses and parallel fixed-point computations, 2015.