Designing and Integrating Complex Systems: Be Agile Through Liveness Verification and Abstraction

Thomas Lambolais, Anne-Lise Courbis, Hong-Viet Luong, Thanh-Liem Phan

Designing and integrating complex systems: be agile through liveness verification and abstraction

Thomas Lambolais[1], Anne-Lise Courbis[1], Hong-Viet Luong[2], Thanh-Liem Phan[3]

1. LGI2P école des mines d'Alès, Site de Nîmes, Parc Scientifique Georges Besse, 30 035 Nîmes cedex 1, France.
2. M2M-NDT, 1 rue de Terre Neuve, Miniparc du Verger, btiment H, 91 940, Les Ulis, France.
3. LSEI, CEA INES, 50 avenue du lac Léman, BP 258, 73 375 Le Bourget du Lac, Cedex, France.

Abstract. Model Driven Architecture (MDA) is recognised as a strong way to develop high-quality systems, and specifically reactive systems. Within MDA, models are in the center of a stepwise development based on extensions, refinements and transformation. Systems Engineering addresses the problem of complex system development in a holistic way, however, there is a lack of tools to verify models from a behavioural point of view at the earlier stage of the development, taking into account that the specifications are evolving during the system development. We propose IDF, a framework for Incremental Development of Compliant Models, which is constituted with a set of relations based on the verification of liveness properties. It is computed on abstract models automatically set up from behavioural specifications of the system or its component. These relations detect non-conformance of models during their evolution (extension or refinement) such as the non-interoperability of sub-components belonging to an architecture.

1 Introduction

Model Driven Architecture (MDA) [22] is recognised as a strong way to develop high-quality systems, and specifically reactive systems which are event-driven systems that must continuously react to external stimuli. Such systems include for instance embedded controllers for automotives, avionics, train, telephony, but also communication network.

Within MDA, models are in the center of a stepwise development based on model extensions, refinements and transformations, from an abstract incomplete specification to a concrete complete model. By this way, models serve both as a description of the problem domain, i.e. a requirement, and a specification for the implementation, bridging the gap between problem and solution. Many methods and tools have been proposed to support model development based on standard modelling languages such as UML or SysML. Methodologies are also necessary

in order to deal with complex systems. Systems Engineering [1] addresses this challenge in a holistic way considering both business and technical aspects of a system design, integrating all stakeholders at the early stage of the development, starting from the user requirements and the definition of the environment of the system to be designed in order to produce high-quality systems. Many methodologies and many standards have been proposed to follow these recommendations as it is shown in the survey proposed in [8]. Our area of interest focuses on the definition and the analysis of the behavioural view of the system, expressed by a functional or organic architecture whose components are defined by a behavioural view or an architectural one. The target activities are therefore the functional analysis, the functional verification and the synthesis in the IEEE 1220 Process model [2]. Our experience in system modelling highlighted that architecture definition, behavioural abstraction and refinement are the core activities of system design. Designing a system consists not only in modelling its architecture, but also in evaluating its behavioural models and that of its components at the beginning of the modelling process, although the model is incomplete and non-deterministic. These features have to be considered as a support for designers and architects. It means that such verifications have not to be postponed at the end of the modelling process. They have to be integrated in the incremental development of the system and its components.

For this propose, we have defined IDF, an Incremental Development Framework. It is defined by a set of relations computed on an abstract formalism (LTS for Labelled Transition System), allowing models to be evaluated during their development. The environment of the system to be designed can be at its turn modelled taking into account its uncertain or non-deterministic behaviour. By this way, incompatibility or non-interoperability can be detected at early stages of the design process. The framework is supported by a tool, named IDCM (Incremental Development of Compliant Models). Experiments have been conducted on UML models. Our work is inspired by techniques of model checking [5]. Such verifications aims at:

- supporting the stepwise realisation of systems by applying refinement and extension operations
- analysing the interaction of the system with its environment, with respect to non-deterministic scenarios
- insuring the interoperability of the system components
- insuring the evolution of the system by substituting a component by a new one

This paper gives an overview of the concepts of IDF and tools we have developed to support IDF. The following section presents modelling concepts of architectures and behavioural components through an incremental development process in order to point out topics being addressed. Section 2 introduces definition of liveness and abstraction models allowing UML/SysML models to be analysed. Section 3 gives an overview of relations we have implemented to support IDF. Section 4 shows main functionalities of the tool IDCM for supporting IDF concepts. A presentation of our future work will close this article.

2 The architectural paradigms

In this section, we present main useful concepts to understand our proposal for incremental development of architectural models. We focus on the verification of behavioural specifications of a system all along its design life cycle. Figure 1 gives an overview of the useful operations for the development of a system based on a MDA approach. We suppose that the first step starts by defining a behavioural specification of the system (BEHAV1 in Figure 1) at a high abstraction level. Such a specification may evolve and be extended (BEHAV2 in Figure 1) until an agreement is reached between the various stakeholders of the system development (client, end-users, designers). This agreement may however evolve during the system design process and at every step, it will be necessary to be able to take into account new specifications.

When the system is complex, its design is structured into components that may represent functional components or physical components depending on the stage of the design process. Components defined according to a structural view are called architectures. For example, in Figure 1, the first architecture is named ARCHI1; it is extended into ARCHI2 whose components have to be refined. Architectures can be seen as a hierarchical tree whose leaves are behavioural components. Architectures may represent logical architectures or physical ones.

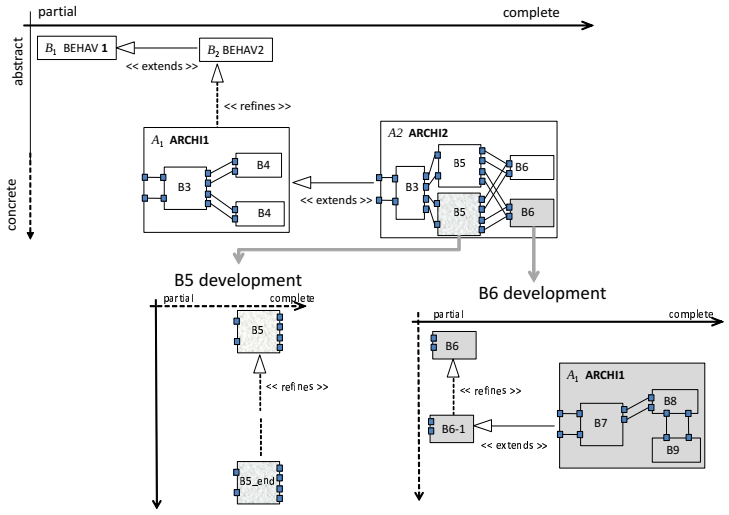

Fig. 1. Overview of an incremental development through refinement and extension operations.

Extensions means that new behaviours are introduced into the design, for whatever reasons: the system is too complex to be defined in one shot, the client

changes is mind, there is an already developed COTS whose specification is closed of the required one that could be integrated with lower cost, a product line has already be tested and its enhancement is expected by introducing new requirements, and so on.

Refinements aim at adding details and reducing non-determinism in order to get a concrete model closer to the final implantation of the system.

Developments of components may be processed by separate teams, by means of a collaborative platform, that increase the complexity of the process. One main concern of component designers is to develop components that meet their specification. Components are supposed to be defined for a given context, except that this context is evolving since it is itself under development. One goal of the architect is to verify the behavioural consistency of the models being developed. This task is critical since sub-systems have their own development life cycle. Nevertheless, the architect cannot wait until the final implantation model to check the consistency analysis of the system. He/she has to maintain the functional consistency of the system model under development whatever the abstractions of sub-system models. We characterize consistency by the following properties:

- conformance: the behavioural specification of the architecture that is deduced from the interaction of its components fulfils the mandatory parts of the specification [12].
- interoperability: the system is deadlock free; whatever point of interaction may be reached, communication will not be blocked and each part will reach one of its final states [4].

Architectures and behavioural components are defined from an external point of view, by a set of ports useful for establishing connections and a set of interfaces defining required and provided operations (or services). In order to illustrate concepts of architecture modelling, we will take as example the V76 case study proposed by [7], which is a simplified version of the protocol described in the ITU V.76 recommendation, based on LAPM (Link Access Procedure for Modems). Figure 2(a) represents an abstract external view of an architecture named V76-DL which represents the communication between two components that implement the protocol V76 and Figure 2(b) is a more detailed external view.

The internal view of an architecture is defined by its components and their interconnections. For example, Figure 3 illustrates the internal view of architecture V76-DL: it is constituted with two components of type V76 whose external view is given in Figure 4. The architecture allows two users to communicate through the ports U1 and U2.

The internal view of a behavioural component is defined by a behavioural specification defined according to its ports, the operations of its external view and private internal operations. Many formalisms may be used for behavioural specification depending on the system features and the progress of the development: sequence diagrams, state machines, functional flow block diagrams. For example, Figure 5(a) shows a simplified specification of the architecture V76-

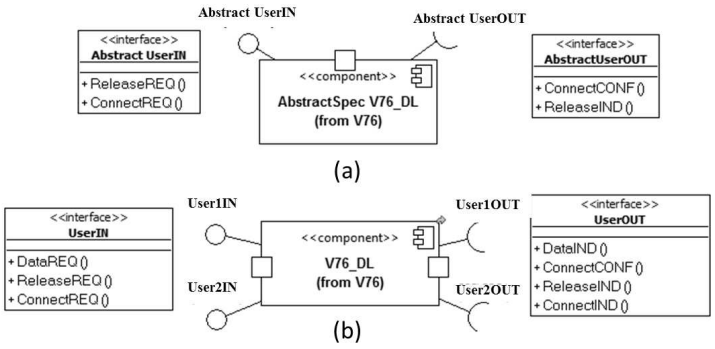

Fig. 2. External view of two points of view of architecture V76-DL.

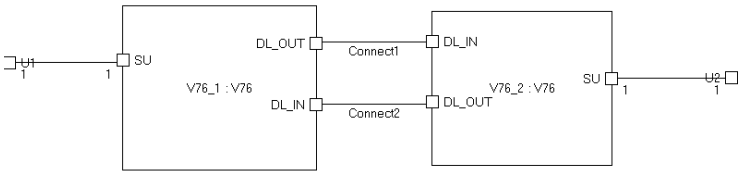

Fig. 3. Internal view of architecture V76-DL.

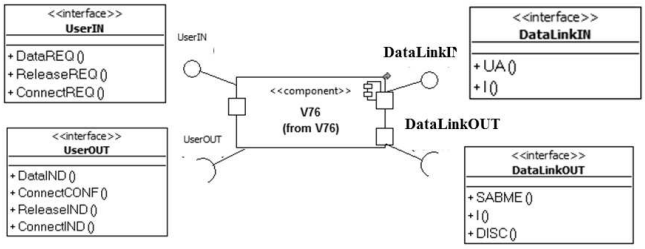

Fig. 4. External view of component V76.

DL from the transmitting user point of view and Figure 5(b) shows the state machine of component V76 belonging to architecture V76-DL.

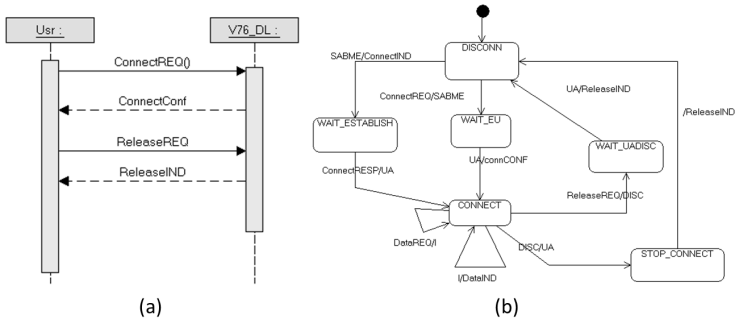

Fig. 5. Behavioural specifications: (a) sequence diagram associated with the abstract architecture V76-DL (b) state machine of component V76.

Analysing the consistency of an architecture during its development requires specific mechanisms and tools that are usually not proposed by CASE (Computer-Aided Software Engineering) tools. These mechanisms are divided into two groups:

- model verifications: adequate relations have to be defined to capture conformance, refinement, extension and interoperability
- model abstraction: adequate models have to be set up from the model under construction in order to capture behavioural specification from an external point of view and an appropriate abstraction in order to compare models defined at different abstraction levels.

These mechanisms are defined according to liveness properties that have to be preserved during development. This property is the liveness. Next section gives definition of liveness and motivates this choice.

3 The use of liveness and abstraction as a design guideline

Liveness and safety properties allow systems to be analysed with respect to their behavioural specification as observed by their environment. This behaviour is observed by traces which are partial sequences of interactions (events or actions) starting from the initial state of the system. There are several ways to define safety and liveness, some of them being contradictory about the classification of deadlock property. We have selected definitions proposed by [14]: a safety

property asserts that the system always stays within some allowed set of finite behaviours, in which nothing "bad" happens. The violation of such properties occurs after a finite execution of the system. A liveness property asserts that the system eventually reaches a good set of states, that means it will eventually react as it should after some given traces. A liveness property represents what the system must do, while a safety represents what the system has not to do. When reasoning on models, liveness properties can only be established under some fairness assumption, stating that the system is not allowed to continuously favour certain choices at the expense of others [25]. The fairness assumption implies that the system will eventually accept an event occurring infinitely often. Lastly, we consider that deadlock freedom is a liveness property, as proposed in [23] since a deadlock means that the system refuses any input event.

Many formal methods addressing complex system development advocate refinement techniques [13, 27] such as B method [3] or Object-Z [26]. They focus on the preservation of safety properties all along the process of development. Such methods are adequate when the specification of the component or the complete system is definitive and not being defined or evolved. Another way to support designers during model development is to preserve the liveness properties as mentioned in [11]: liveness properties act as a design guideline for developing systems.

Liveness is crucial for reactive systems and is complementary to safety to support designers during an incremental development: observing liveness allows specification to be enriched, starting from a "draft" model that is completed by a stepwise approach in a non-regressive way.

It is therefore necessary to provide designers with tools to compare models according to their liveness properties, taking into account that they subcomponents can be defined at different abstraction levels. For example, how ensuring that architecture V76-DL fulfils the behavioural specification expressed by the sequence diagram? Are components of architecture V76-DL interoperable?

To answer these questions, we have defined two mechanisms: model abstraction and model analysis based on a liveness analysis.

Model Abstraction

With model abstraction, a simplified behaviour is extracted from models to be analysed. This extraction takes into account several criteria: the abstraction levels of models to be compared, the type of relation to be analysed (extension, refinement or interoperability), and of course, the goal of the analysis that is based on the analysis of the interaction of system (or one of its sub-system) and its environment. Abstract models are formalised by LTS (Labelled Transition System) [20]. Reasoning on such a formalism has many advantages: the system analysis is independent from the modelling formalism chosen by the designer; models can thus be compared even if their application domain is different, that is usual in System Engineering; existing relations already defined on LTS can be used for our purpose.

We do not formally introduce LTS and the process to abstract state machines into LTS. You can refer to [15] and [18] to get details about the transformation. Figure 6(a) illustrates the LTS generated from the state machine of component V76, and Figure 6(b) the LTS associated with the sequence diagram of the architecture V76-DL. The transformation does not handle data; it only focuses on provided and required events (or services) offered by the component under analysis. When the component is an architecture, we have defined a transformation [24] which computes all combinations of internal events between components and reduces the LTS to observable events by hiding internal synchronisations and internal operations. Hidden actions are noted i in the LTS. For example, the LTS associated with the architecture of Figure 3 handles operations defined on its interfaces given in Figure 2(b). Operations defined on interfaces of internal components, that is interfaces DataLinkIN and DataLinkOUT, are hidden. The LTS is built by synchronising the two LTS of Figure 6 on their internal connector. It contains 84 transitions and 54 states.

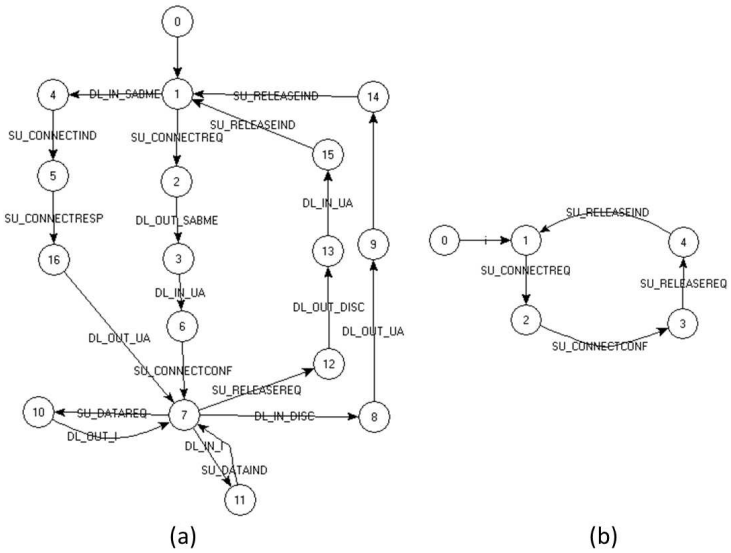

Fig. 6. (a) LTS associated with the state machine of component V76. (b) LTS associated with the sequence diagram of the simplified specification of architecture V76-DL.

When models to be compared do not belong to the same abstraction level, their interfaces may be different. For example, there are more operations in interfaces of component V76-DL than those of the specification of V76 protocol given by the sequence diagram. Comparison needs to align the abstraction levels. For this purpose, we use a hiding mechanism and a renaming mechanism, when operations are refined. For example, to compare V76-DL and the sequence

diagram, internal operations of the architecture (ua, i, sabme, and disc) are hidden such as the operations belonging to the port u2, which correspond with the user receiving the data. By this mechanism, the LTS associated with V76-DL architecture will be comparable to the abstract specification.

The main feature of this abstract model is that it captures what the system must do and what the system may do. That is crucial for liveness properties as we point out below.

Liveness analysis

There exists a specific relation, which lonely goal is to preserve liveness. This relation is conformance relation conf [6, 17]. Conformance testing methodologies proposed by ISO and ETSI [12] are designed to compare an implementation model with a standard specification. Standard specifications or recommendations serve to define both the mandatory and optional parts. The main idea behind conformance is to verify agreement between an implementation and its specification on required parts; informally speaking, an implementation conforms to a standard if it has properly implemented all *mandatory parts* of the standard [21].

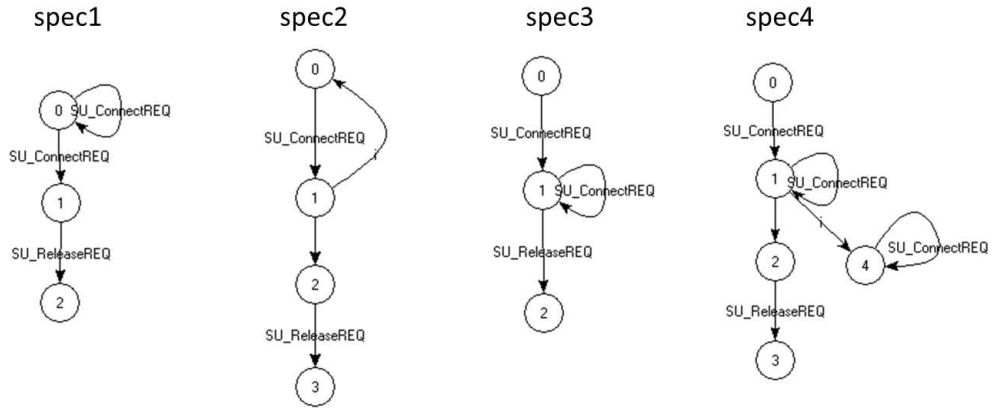

Fig. 7. Example of conf relation.

For instance, in Figure 7, we can deduce the following properties:

- *spec*1, *spec*2 and *spec*4 may accept releaseREQ or connectREQ after a sequence of connectREQ. As they may also refuse them, operations releaseREQ or connectREQ are optional.
- *spec*3 must accept releaseREQ after connectREQ. releaseREQ is thus mandatory after the trace connectREQ.

We can verify: *spec*1 conf *spec*2, *spec*2 conf *spec*1, *spec*1 conf *spec*4. However, *spec*1 $\not$conf *spec*3: from an observational standpoint, nothing distinguishes

*spec*1 from *spec*3 but conf relation detects non-determinism of *spec*3. In this example, *spec*1 may refuse releaseREQ after a non-empty unbounded occurrences of connectREQ, whereas *spec*3, which is deterministic, cannot. *spec*1 and *spec*3 are trace equivalent, yet not in conformance. Lastly, even if *spec*1 conf *spec*4*andspec*4 conf *spec*1, we can verify that *spec*4 cannot substitute *spec*1.

Even though the conformance relation has been defined by [17], we are still not aware of any published method to compute it. We have thus proposed an implantation of this relation and pointed out how extension and refinement relations can be defined from the conformance relation [18, 19]. In the same way, we have implemented the procedure allowing to check if a component can substitute another one, whatever its environment may be [24].

Next section gives an overview of the tool IDCM we have defined and implemented to provide designers with a tool box to analyse models.

4 IDCM: Incremental Development of Compliant Models

IDCM is a tool box allowing models to be compared with respect to refinement, extension and substitution relations. It is based on concepts of IDF focusing on the analysis of liveness properties and abstraction of behavioural/functional models. It is developed in Java. Its first release is integrated into TopCased environment [9] and focus on UML state machines and composite component analysis. When a model is loaded for verification, the set of its components is proposed to be abstracted into LTS (see Figure 8).

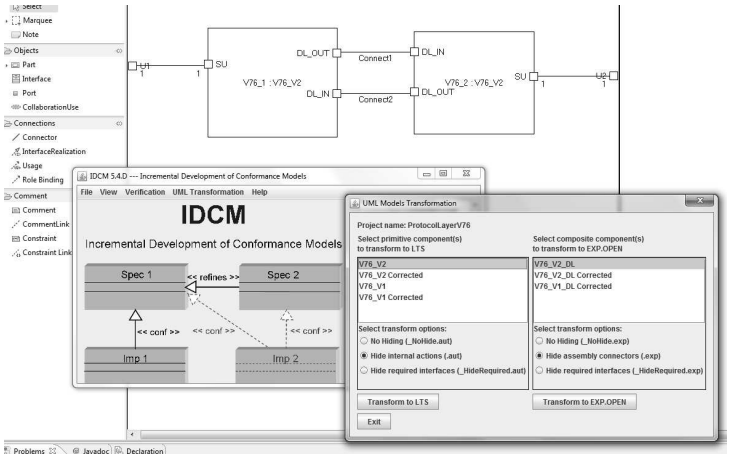

Fig. 8. Interface to transform behavioural and architectural components into LTS.

Behavioural component transformation is performed by an ad-hoc algorithm we have developed by parsing state machine XMI models. Composite components

transformation is done with two stages: the first one produces an intermediate file in EXP.OPEN format [16] that is obtained by parsing composite component xmi models; the second stage, consisting in transforming the intermediate file into LTS, is performed by the CADP toolbox [10]. LTS associated with state machines and composite components are generated into CADP textual and binary formats [10].

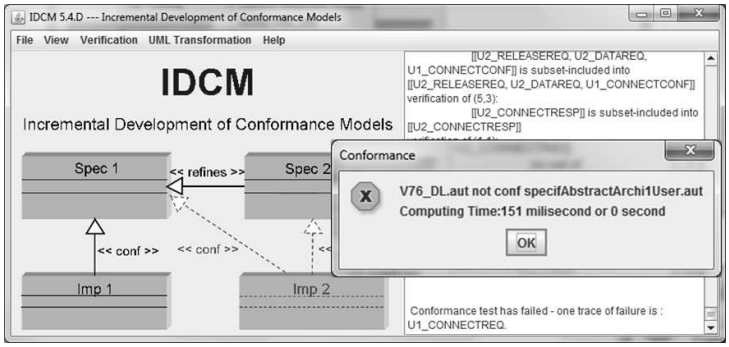

Fig. 9. Verdict of the conformance between the architecture V76-DL and its abstract specification.

IDCM proposes a set of relations for model comparison. They are classified in several families: relations for incremental development (extension or refinement), relation for liveness verification to check the conformance between an implantation and its specification, relations for assembling sub-components (compatibility) and lastly, relations to check if a component can substitute another one. When a relation between two models does not hold, a verdict is given as a sequence of observable events leading to a failure. Designers are in charge to analyse the trace, to execute it on the state machine, or in the architecture in order to find the mistake and correct it. For example, we have found a mistake (Figure 9) in the state machine of component V76 by comparing the architecture with its abstract specification. There exists a deadlock after the action connectREQ when the two users send together a connectREQ. We have corrected this mistake by adding a state and transitions between $\mathsf{wait-eu}$ and $\mathsf{wait-establish}$ states in the state machine of Figure 5(b).

5 Conclusion

Developing complex systems requires methodologies such as MDA and System Engineering. Nevertheless, there is an actual difficulty for designers and architects for evaluating the behaviour of a system being designed during its development. We have thus proposed a framework supported by a tool allowing models to be developed through a stepwise methodology using extensions, refinements

and substitutions. The development guarantees the liveness properties of the system. Our proposal is thus complementary to approaches of safety analysis that must also be performed during the development of critical systems.

Our future work plans to extend the model transformation to other functional formalisms than state machines such as sequence diagrams and eFFBD (enhanced functional block diagram). We are also defining a UML profile for incremental development.

References

1. *Systems engineering handbook*. INCOSE, 2006.
2. IEEE 1220-2005. *Standard for Application and Management of the Systems Engineering Process*. IEEE Computer Society, 2005.
3. Jean-Raymond Abrial. *Modeling in Event-B — System and Software Engineering*. Cambridge University Press, 2010.
4. Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munidar P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. In Sierra Decker, Sichman and Castelfranchi, editors, *8th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, May 2009. Budapest, Hungary.
5. Edmund M. Clarke. The birth of model checking. *25 Years of Model Checking; Lecture Notes in Computer Science*, 5000:1–26, 2008.
6. Rance Cleaveland and Bernhard Steffen. A preorder for partial process specifications. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, pages 141–151, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
7. Laurent Doldi. *UML 2 Illustrated: Developing Real Time & Communication Systems*. TMSO, 2003.
8. Jeff A. Estefan. Survey of model-based systems engineering (mbse) methodologies. Technical Report INCOSE-TD-2007-003-01, INCOSE MBSE Focus Group, 2008.
9. P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. *Ingénieurs de l'Automobile*, 781:54–59, 2006.
10. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, Saarbrücken, 2011.
11. Simon Hudon and Thai Son Hoang. Systems Design Guided by Progress Concerns. In *Integrated Formal Methods*, pages 16–30. Springer Berlin Heidelberg, 2013.
12. ISO/IEC9646. Information technology – open systems interconnection – conformance testing methodology and framework – part 1: General concepts, 1991.
13. Amal Khalil and Juergen Dingel. Supporting the Evolution of UML Models in Model Driven Software Development : A Survey. Technical Report 602, School of computing, Queen's University, Ontario, Canada, 2013.
14. Orna Kupferman and MY Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
15. Thomas Lambolais, Anne-Lise Courbis, Hong-Viet Luong, and Thanh-Liem Phan. Interoperability analysis of systems. In *18th World Congress of the International Federation of Automatic Control (IFAC 2011)*, pages 7879–7884, 2011.

16. Frédéric Lang. Exp. open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In *Integrated Formal Methods*, pages 70–88. Springer, 2005.
17. Guy Leduc. A framework based on implementation relations for implementing LOTOS specifications. In *Computer Networks and ISDN Systems*, volume 25, pages 23–41, 1992.
18. Hong-Viet Luong. *Construction incrémentale de spécifications de systèmes critiques intégrant des procédures de vérification*. PhD thesis, Université Paul Sabatier Toulouse III, October 2010.
19. Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis. Implementation of the Conformance Relation for Incremental Development of Behavioural Models. In Krzysztof Czarnecki, editor, *Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *Lecture Notes in Computer Science*, pages 356–370. Springer-Verlag, 2008.
20. Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.
21. Scott Moseley, Steve Randall, and Anthony Wiles. In Pursuit of Interoperability. In Kai Jakobs, editor, *Advanced Topics in Information Technology Standards and Standardization Research*, chapter 17, pages 321–323. Idea Group Publishing, Hershey, 2006.
22. OMG MDA. *Model Driven Architecture Foundatation Model*. OMG ormsc/10-09-06, 2006.
23. Oracle Corp. The Java Tutorials — Trial Essential Classes: Concurrency. Liveness. http://docs.oracle.com/javase/tutorial/essential/concurrency/liveness.html/, 2015.
24. Thanh-Liem Phan. *Développement incrémental de spécifications d'architectures en UML intégrant des procédures de vérification*. PhD thesis, Université Montpellier II, 2013.
25. Antti Puhakka and Antti Valmari. Liveness and Fairness in Process-Algebraic Verification. In *Proceedings of the 12th International Conference on Concurrency Theory*, CONCUR '01, pages 202–217, London, UK, 2001. Springer-Verlag.
26. Graeme Smith. *The Object-Z Specification Language*, volume 1 of *Advances in Formal Methods*. Kluwer Academic Publishers, Boston, MA, 2000.
27. Muhammad Usman, Aamer Nadeem, Tai Hoon Kim, and Eun Suk Cho. A survey of consistency checking techniques for UML models. In *Proceedings of the 2008 Advanced Software Engineering and its Applications*, pages 57–62, 2008.