# Property driven verification framework: application to real time property for UML MARTE software design

Ning Ge


# THÈSE

**En vue de l'obtention du**

## DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

**Délivré par :**

Institut National Polytechnique de Toulouse (INP Toulouse)

**Discipline ou spécialité :**

Sureté de Logiciel et Calcul à Haute Performance

---

**Présentée et soutenue par :**

Mme NING GE

le mardi 13 mai 2014

**Titre :**

PROPERTY DRIVEN VERIFICATION FRAMEWORK : APPLICATION TO REAL TIME PROPERTY FOR UML MARTE SOFTWARE DESIGN.

---

**Ecole doctorale :**

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

**Unité de recherche :**

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

**Directeur(s) de Thèse :**

M. MARC PANTEL

M. YAMINE AIT AMEUR

**Rapporteurs :**

M. FRÉDÉRIC MALLET, UNIVERSITE DE NICE SOPHIA ANTIPOLIS

M. KAMEL BARKAOUI, CNAM PARIS

**Membre(s) du jury :**

M. FERHAT KHENDEK, UNIVERSITE CONCORDIA MONTREAL, Président

M. FREDERIC BONIOL, INP TOULOUSE, Membre

M. MARC PANTEL, INP TOULOUSE, Membre

M. PHILIPPE DHAUSSY, ENSTA BRETAGNE, Membre

M. SILVANO DAL ZILIO, LAAS TOULOUSE, Membre

M. YAMINE AIT AMEUR, INP TOULOUSE, Membre

# Acknowledgments

The work presented in this thesis has been completed in the ACADIE team in IRIT, with the financial support from the French Ministry of Industry through the ITEA OPEES project and the FUI P project. The experience as a Ph.D student is full of excitement and adventure. At the end moment, I wish to express my gratitude to my supervisors, colleagues and families.

First and foremost I offer my sincerest gratitude to my principal supervisor Marc Pantel for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study. I offer the same gratitude to Yamine Aït Ameur for accepting to be the director of my thesis, and to Xavier Crégut for accepting to be co-supervisor of my thesis. Their good advice and support have been invaluable, for which I am extremely grateful.

I express the deepest appreciation to Kamel Barkaoui, professor at Cedric Cnam Paris, and Frédéric Mallet, associate professor (HDR) at Université de Nice Sophia Antipolis for accepting to examine my Ph.D work. I express the same appreciation to Ferhat Khendek, professor at Concordia University, Philippe Dhaussy, associate professor (HDR) at ENSTA Bretagne, Frédéric Boniol, professor at Université INPT and ONERA Toulouse, and Silvano Dal Zilio, full time researcher at LAAS-CNRS for accepting to be the jury members. I extend my gratitude for their constructive comments and open discussions during my Ph.D work and on the defense of the thesis.

In this Ph.D study, I had the opportunity to discuss my work with Bernard Berthomieu, Silvano Dal Zilio and François Vernadat at LAAS. I have greatly benefited from their advice and unsurpassed knowledge of model checking and formal verification. I am deeply grateful to all of them.

I would like to offer my thank to Michaël Lauer and Frédéric Boniol for their permission to use the case

study they developed.

I would like to offer my special thank to Shin Nakajima, professor at National Institute of Informatics in Japan, with whom I have worked 3 months during the research internship, for his insightful comments and suggestions on my research directions. I would like to thank Aurélie Hurault for her support and encouragement in this internship.

I thank all the colleagues at IRIT, with special attention to Sylvie Eichen and Sylvie Armengaud-Metche. They are always available to help me manage the administrative problems. Thanks so much to both for their kindness and efficiency. I also thank a lot to all members of ACADIE team. I would like to express my gratitude to Philippe Quélinnec, Mamoun Filali, Jean-Paul Bodeveix, Philippe Mauran, Xavier Thirioux, Meriem Quederni and Mounira Kezadri for their advice during my Ph.D study and their kindness. I want to thank the colleges sharing the same office with me: Arnaud, Florent and Faiez. It was a pleasure to work with them.

I cannot finish without thanking with all my heart my parents, for their support, encouragement and love throughout my whole life.

At last, I reserve my final thanks to you, Hongyu, my best friend, soul-mate and dear husband. Thank you for your continued and unfailing love, support and understanding my persistence in the research career.

# Abstract

Automatic formal verification such as model checking faces the combinatorial explosion issue, and thus limits its application in industrial projects. This issue is caused by the explosion of the number of states during system's execution , as it may easily exceed the amount of available computing or storage resources.

This thesis designs and experiments a set of methods for the development of scalable verification tools based on the property-driven approach. We propose efficient approaches based on model checking to verify real-time requirements expressed in large scale UML-MARTE real-time system designs. We rely on the UML and its profile MARTE as the end-user modeling language, and on the Time Petri Net (TPN) as the verification language. The main contribution of this thesis is the design and implementation of a property-driven verification methodology dedicated to real-time properties verification for UML-MARTE real-time software designs. We validate this method using an avionic use case and its user requirements. This method was implemented as a prototype toolset that includes five contributions: definition of real-time property specific execution semantics for UML-MARTE architecture and behavior models; specification of real-time requirements relying on a set of verification dedicated atomic real-time property patterns; real-time property specific observer-based model checking approach in TPN; real-time property specific state space reduction approach for TPN; and fault localization approach in model checking.

# Résumé

Les techniques formelles de la famille « vérification de modèles » (« model checking ») se heurtent au problème de l'explosion combinatoire. Ceci limite les perspectives d'exploitation dans des projets industriels. Ce problème est provoqué par la combinatoire dans la construction de l'espace des états possibles durant l'exécution des systèmes modélisés. Le nombre d'états pour des modèles de systèmes industriels réalistes dépasse régulièrement les capacités des ressources disponibles en calcul et stockage.

Cette thèse défend l'idée qu'il est possible de réduire cette combinatoire en spécialisant les outils pour des familles de propriétés. Elle propose puis valide expérimentalement un ensemble de méthodes pour le développement de ce type d'outils en suivant une approche guidée par les propriétés appliquée au contexte temps réel. Il s'agit donc de construire des outils d'analyse performants pour des propriétés temps réel qui soient exploitables pour des modèles industriels de taille réaliste. Les langages considérés sont, d'une part UML étendu par le profil MARTE pour la modélisation par les utilisateurs, et d'autre part les réseaux de pétri temporisés comme support pour la vérification. Les propositions sont validées sur un cas d'étude industriel réaliste issu du monde avionique : l'étude de la latence et la fraicheur des données dans un système de gestion des alarmes exploitant les technologies d'Avionique Modulaire Intégrée. Ces propositions ont été mise en oeuvre comme une boite à outils qui intègre les cinq contributions suivantes: la définition de la sémantique d'exécution spécifiques aux propriétés temps réel pour les modèles d'architecture et de comportement spécifiés en UML/MARTE; la spécification des exigences temps réel en s'appuyant sur un ensemble de patrons de vérification atomiques dédiés aux propriété temps réel; une méthode itérative d'analyse à base d'observateurs pour des réseaux de Petri temporisés; des techniques de réduction de l'espace d'états spécifiques aux propriétés temps réel pour des Réseaux de Petri temporisés; une approche pour l'analyse des erreurs détectées par « vérification des modèles » en s'appuyant sur des idées inspirées de la « fouille de données » (« data mining »).

# Contents

# Listing of figures

# Part I

# Introduction

# 1 Introduction


## Résumé

Le premier chapitre introduit la contexte de recherche, les défis et les contributions de cette thèse. Les systèmes embarquées temps réels jouent un rôle clé dans de nombreuses facettes de la vie quotidienne. Certains sont des applications spécialisées de grande échelle dans les domaines critiques tels que l'avionique, l'aérospatial, la défense, le nucléaire, l'automobile, la santé et le matériel médical. Ils doivent donc satisfaire de fortes exigences concernant la sécurité et la fiabilité. Tout manquement à ces exigences peut entraîner des conséquences graves en terme de pertes matérielles et de sécurité des personnes. La sécurité et la fiabilité des systèmes temps réels dépendent fortement de la satisfaction des exigences temps réel, à la fois pour les aspects qualitatifs et quantitatifs. L'état de l'art actuel des connaissances propose que ces exigences soient vérifiées et validées en utilisant des méthodes formelles en combinaison avec l'ingénierie dirigée par les modèles. Les méthodes formelles sont des techniques issues des mathématiques pour la spécification, le conception, la programmation et la vérification des systèmes matériels et logiciels. L'utilisation

d'approches mathématiques permet d'obtenir une plus grande assurance en ce qui concerne la fiabilité et la robustesse d'un système.

Les techniques formelles de la famille « vérification de modèles » (« model checking ») sont bien adaptées à une exploitation industrielle car elles permettent une automatisation complète des activités de vérification et la synthèse de contre exemples en cas de non satisfaction des exigences. Mais elles se heurtent au problème de l'explosion combinatoire qui impose la construction de modèles dédiés à la vérification de chaque exigence et limite les perspectives d'exploitation dans des projets de grande taille. Ce problème est lié à la combinatoire dans la construction de l'espace des états possibles durant l'exécution des systèmes modélisés. Le nombre d'états pour des modèles de systèmes industriels réalistes dépasse régulièrement les ressources disponibles en calcul et stockage.

En s'appuyant sur la pratique actuelle de la « vérification de modèles » consistant à construire des modèles dédiés à chaque vérification, cette thèse défend l'idée qu'il est possible de réduire cette combinatoire en spécialisant les outils selon des familles de propriétés. Elle propose puis valide expérimentalement un ensemble de méthodes pour le développement de ce type d'outils en suivant une approche guidée par les propriétés appliquée pour le contexte temps réel. Il s'agit donc de construire des outils d'analyse performants pour des propriétés temps réel qui soient exploitables pour des modèles industriels de taille réaliste. Les langages considérés sont, d'une part UML étendu par le profil MARTE pour la modélisation par les utilisateurs, et d'autre part les réseaux de pétri temporisés comme support pour la vérification. Les propositions effectuées sont validées en exploitant un cas d'étude industriel réaliste issu du monde avionique : l'étude de la latence et la fraicheur des données dans un système de gestion des alarmes exploitant les technologies d'Avionique Modulaire Intégrée. Ces propositions ont été mise en oeuvre sous la forme d'une boite à outils qui intègre les cinq contributions suivantes: la définition de la sémantique d'exécution spécifique aux propriétés temps réel pour les modèles d'architecture et de comportement spécifiés en UML/MARTE; la spécification des exigences temps réel en s'appuyant sur une traduction vers un ensemble de patrons de vérification atomiques dédiés aux propriété temps réel; une méthode itérative d'analyse à base d'observateurs pour des réseaux de Petri temporisés; des techniques de réduction de l'espace d'états spécifiques aux propriétés temps réel pour des Réseaux de Petri temporisés; une approche pour l'analyse des erreurs détectées par « vérification des modèles » en s'appuyant sur des idées inspirées de la « fouille de données » (« data mining »).

This thesis designs and experiments a set of methods for the development of scalable verification tools based on a property-driven approach. It develops efficient approaches based on model checking to verify real-time requirements expressed in large scale UML-MARTE real-time system designs.

## 1.1 Safety Critical Real-Time System Development

Real-time embedded systems play a key role in many facets of daily life. Some are specialized and large scale applications in the critical domains such as avionics, aerospace, defense, nuclear power, motor vehicles, health and medical equipment and thus have strong requirements concerning system's safety and reliability. Any failure could cause serious consequences that may result in massive material losses or endanger human safety. [Neu95] listed a large amount of accidents and disasters caused by errors in real-time systems. If it is possible to avoid these failures, large efforts and costs would be saved. In June 1996, the first flight of Ariane 5 launcher ended in failure caused by an overflow error. About 37 seconds after ignition, the rocket broke and self destruction was initiated. This accident led to a 370 million dollars cost [Lio96]. In December 1999, the last telemetry from Mars Polar Lander was sent. Just prior to cruise stage separation and the subsequent atmospheric entry, no further signals were received from the spacecraft. The most likely cause of this mishap was different interpretations of floating point data, which was implicitly specified as meters by NASA and implemented as feet by Rockwell Collins. This accident led to a 165 million dollars loss [BCAA00]. For systems where failure is unacceptable, reliable software is mandatory. Thus safe and efficient techniques are required to detect errors and thus avoid the accidents in such systems. The research context and main motivation of this work is how to design and implement safe and reliable real-time systems.

## 1.2 Model Driven Engineering

Model-Driven Engineering (MDE) targets the improvement of the reliability and efficiency of the traditional software engineering by introducing models and early verification and validation ($V\&V$) including the use of formal methods. It has evolved over the last 20 years and achieved success in many domains. Models are reduced/abstract representations of real systems that selectively remove some semantics to highlight the remaining expected properties from a given point of view. In the context of safety critical systems, models can be used during the requirement engineering process to derive the requirements for a system, during the

design process to describe the intended system to the implementation engineers, to verify and validate the properties, to automatically generate software, and also to document the system's structure and behavior after implementation.

The V-model [FM95] is a software development process broadly adopted in the industry to illustrate the various activities involved in the development of software and their ideal sequencing. In this thesis, we rely on the multi V-model (see Fig. 1.2.1) proposed in the MeMVaTEx methodology [ABD+07, ABB+08] to illustrate the use of MDE for developing real-time systems. In order to generate reliable software, the *V & V* activities are performed at each phase of the system development lifecycle. The architecture design is the phase to define the hardware and software architectures which is referred to as high-level design. It should involve a brief and abstract functionality of each module, their interface relationships, dependability, architecture diagrams, etc. The detailed design model can also be called module or function design model, where the low-level design including detailed functional logic of the module can be specified.

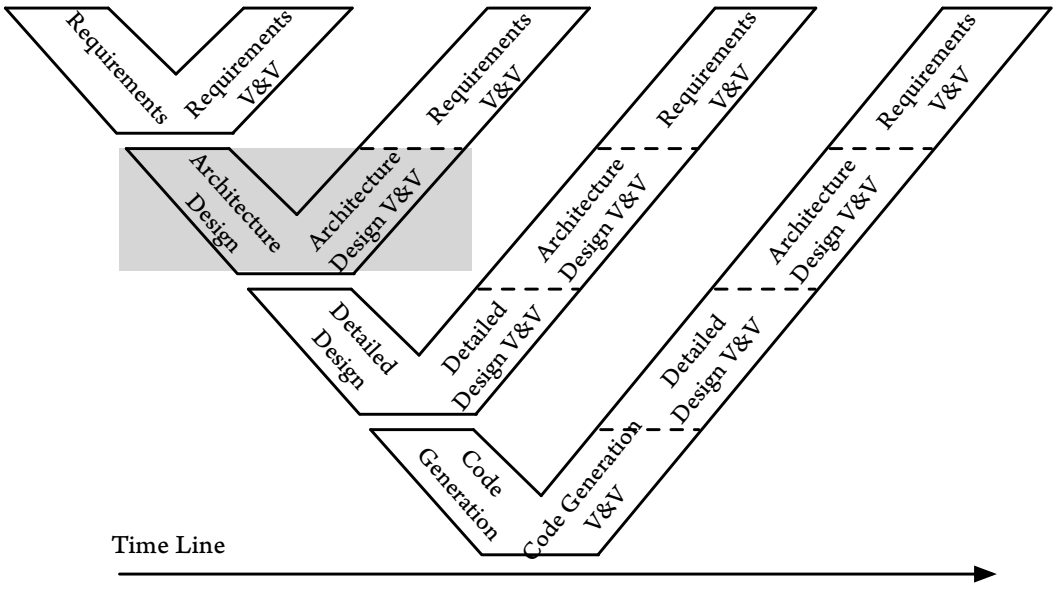



**Figure 1.2.1:** V-Model in Model-Driven Engineering

From the current practice, the architecture is usually modeled using Domain Specific Languages (DSL) such as AADL and EAST-ADL or specific diagrams in a General Purpose Language (GPL) such as UML Composite Structure Diagram, while the detailed design is usually modeled using DSL such as Simulink/ Stateflow

and SCADE or specific GPL diagrams such as UML Activity, State Machine diagrams, or ALF (Action Language for Foundational UML). The main purpose of our work is to propose efficient and effective formal verification tools to ease the use of MDE when developing large scale real-time systems. More precisely, this work targets the use of UML-MARTE in the early phases of MDE, that corresponds to the *Architecture Design* phase in the multi V-model (the grey box part in Fig. 1.2.1).

## 1.3 Formal Methods

Formal Methods (FM) are mathematically based techniques for the specification, development and verification of software and hardware systems. The use of mathematical analysis can contribute to the reliability and robustness of a design [Hol97a]. Verification methodologies such as model checking, abstract interpretation, automated proof, etc. provide rules for inferring useful information from the specification. The conjunction of MDE and FM is a promising answer to the development of real-time systems, which makes it feasible to assess system's requirements since the early phases of system lifecycle and to iteratively improve the models according to the verification results.

However, automatic formal verification such as model checking faces the combinatorial explosion issue. This limits its application in industrial projects [CE82, HP94]. This issue is caused by the exponential number of generated states during system's execution that may easily exceed the amount of available computing or storage resources.

## 1.4 Methodology: Property Driven Approach

UML (Unified Modeling Language) [OMG11c] was developed to provide a common language for specification, modeling and documentation in the software development process in the 1990s. Today, UML is the industry standard for software modeling and specification. MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [OMG09] provides support for specification, design, and *V & V* for real-time and embedded system. We use the term UML-MARTE in the whole thesis to indicate the specification language.

As UML is a semi-formal language which exhibits ambiguous and imprecise (in terms of mathematical precision) semantics, most of the requirements expressed in UML models cannot be directly assessed using formal methods. Therefore, providing a formal executable semantics is now a common approach used to

assess the user requirements in UML models. There exists a number of formal languages dealing with real-time analysis issues, such as Timed Automata [AD94] and several extended Petri Nets such as Timed Petri Nets [RH80, Zub91], Stochastic Timed Petri Net [FFN91], Time Petri Net (TPN) [MF76], etc. Our work relies on TPN as the execution model, and uses the TINA toolset as the analysis toolbox.

From the viewpoint of methodology, our work is based on the pioneering work [CCG+07] by Combemale et al. Aimed to define all the steps from the property specification to effective verification, they introduced in [CCG+07] a generic approach to define the operational semantics (a semantics of observable events) built upon the properties expressed at the metamodel level. They illustrated this contribution through a simple process description language: SimplePDL on which a set of temporal properties were expressed. Property-driven means that the formal activities in the development process are based on the purpose of property-verification-ease. From a language point of view, a precise definition of model elements behavior allows the execution of behavioral models with respect to the intended requirements that must be assessed.

We follow the same methodology proposed by Combemale et al., and propose a property-driven framework dedicated to real-time property verification for UML-MARTE real-time designs. A key objective in our work is to assess this property driven approach on a large scale system relying on industrial modeling languages, requirements and use cases.

## 1.5 Real-Time Requirements

*A real-time system is a system whose correct operation depends on both the results produced by the system and the time at which these results are produced* [Som10]. The safety and reliability of real-time systems strongly depend on the satisfaction of its real-time requirements, in both qualitative and quantitative aspects. According to the survey collected from the industrial partners in several collaborative projects such as projects P [1], TOPCASED [2], OPEES[3], QUARTEFT [4], SPICES [5], SPACIFY [6], HiMoCo [7] and CESAR [8], we list some examples of in-

[1] http://www.open-do.org/projects/p/
[2] http://www.topcased.org/
[3] http://www.opees.org/
[4] http://projects.laas.fr/fiacre/
[5] http://www.spices-itea.org/public/news.php
[6] http://spacify.gforge.enseeiht.fr/
[7] http://www.systematic-paris-region.org/fr/projets/himoco
[8] http://www.cesarproject.eu/

dustrial real-time requirements in Table 1.5.1. To simplify the expression, we use $E_1$, $E_2$ and $E_3$ to denote events, and [a,b] to denote a time interval.

**Table 1.5.1:** Examples of Real-Time Requirements

| No. | Real-Time Requirements |
|---|---|
| 1 | $E_1$ must be sent after the reception of $E_2$. |
| 2 | A task cannot be executed after the emission of $E_1$. |
| 3 | The third occurrence of $E_1$ must be sent between the reception of $E_2$ and the emission of $E_3$. |
| 4 | A system state holds for at least $n$ time unit (*t.u.*) |
| 5 | A system state holds for at most $n$ *t.u.* after the emission of $E_1$. |
| 6 | If $E_1$ is sent, $E_2$ must be received after the emission of $E_1$ within [a,b]. |
| 7 | $E_1$ is received more than $n$ times after the reception of $E_2$ within [a,b]. |
| 8 | $E_1$ and $E_2$ must be sent simultaneously, within time tolerance $\delta$. |
| 9 | The execution of a task must start after the reception of $E_1$ within [a,b] in each period. |
| 10 | If $E_1$ has been received, $E_2$ must be sent before the reception within [a,b] in each period. |
| 11 | The worst case execution time of a task is $n$ *t.u.* in each periodic execution. |

In Table 1.5.1, the requirements 1 - 3 are related to the logical time, while 4 - 11 are related to the physical time. The requirements 9 -11 are applicable to systems with periodic execution. Regarding the logical time requirements, there exist many works to specify and assess real-time properties using logic formulae. In the context of this thesis, we focus on the physical time (quantitative) properties in finite state concurrent reactive systems. These real-time requirements are critical, and thus their correctness must be guaranteed at any cost. Appropriate development processes, methods and tools are expected to enable the efficient verification, and to help the users to improve their designs when the errors have been detected. However, in today's highly competitive industrial market, the scale and complexity of safety critical real-time system are rapidly increasing due to the growth of functional and non-functional requirements. For instance, since Airbus A300, the number of software control systems has been increased to add new functionality such as flight envelop protection, ground proximity warning and traffic collision avoidance for improved safety [ITI07]. Consequently, verification of the real-time requirements for real-time system development is becoming more and more difficult and expensive. Therefore, although many progresses in the last 20 years, how to design and implement highly safety critical real-time system and in the meanwhile control

the development cost is still an open problem in both industry and academia.

## 1.6 Challenges

The key obstacle that prevents a wide application of model checking in the industry is the scalability issue. The classic verification methodology usually encounters scalability issue very quickly along with the growth of system size. A complex system usually has thousands and even millions of states and transitions. Although a huge part of the impossible transition firing sequences is eliminated during the building of system's behavior, the probable permutation of all others is still a very large number that easily causes combinatorial state space explosion.

Although many formal verification languages such as Petri Net [Pet62] and Automata [Sal85] and their analysis tools are theoretically mature enough, the efficient application for real size systems is still an open question. As the scalability issues introduced by the combinatorial explosion problem is still one of the bottlenecks, the industrial partners would rather verify and validate the requirements using traditional final system tests. Another key issue is effective fault analysis for the verification failures. Once an error has occurred, effective debug information is expected to be derived from the verification results to help the designers improve their designs. The challenges in this work can be summarized as the following five aspects:

- **Challenge 1: Specification, implementation and validation of a real-time property specific execution semantics for UML-MARTE models that allows scalable verification.** As revealed by a number of surveys, even the most recent versions of the UML specification suffer from multiple ambiguities, inconsistency and incompleteness regarding the semantics of the language for the formal verification purpose. This is a major problem for MDE because the semantics contained in the user models will be directly propagated to the verification models. A formal execution semantics should thus be defined. The manner the execution semantics is defined is one of the important factors that impact the verification efficiency, especially for the large scale system development. The optimal execution semantics only preserves minimal property-relevant semantics. This may reduce the risk of combinatorial state explosion problem during model checking.

- **Challenge 2: Need for practical real-time requirement specification method for verification purpose.** Many studies have shown that most of the real-time requirements are composite proper-

ties based on a set of elementary patterns. Dwyer et al. defined the pioneer qualitative time patterns. Afterwards, Konrad et al. extended Dwyer's patterns by adding quantitative time extensions. These property patterns aimed to ease the expression of end-user time requirements, but usually they are not semantically atomic. These property specifications need to be decomposed into a set of atomic property elements to improve the verification efficiency. A property specification method that can ease the verification is needed to bridge this gap. CCSL, as a clock constraint specification language, can express event-based logical properties in the UML-MARTE models. A real-time property specification method is needed to map the requirements expressed using Dwyer's and Konrad's patterns or the CCSL to the verification-ease property patterns.

- **Challenge 3: Need for scalable model checking support for the verification of real-time properties in TPN model.** Despite the significant investment of research and development effort into state-of-the-art industrial MDE tools, model checking remains an expensive resource-consuming development method that requires special skills. The TINA model checking toolset supports logic formulae LTL and CTL for analyzing qualitative properties. To verify quantitative properties in large scale systems, an efficient real-time property analysis approach based on LTL, CTL or other logic formulae is required. The real-time property specific model checking approach should rely on the observer techniques, which transforms quantitative problems to reachability problems.

- **Challenge 4: Need for property-specific state space reduction method.** Combinatorial state space explosion issue in current TPN model checking approach limits its application. Many techniques have been studied to reduce the size of state space using different abstractions. These techniques usually provide generic abstraction methods to reduce the size of state space for all kinds of properties. In this work, a real-time property specific reduction technique is used to improve the scalability in TPN model checking.

- **Challenge 5: Need for failure analysis approach to locate the origin of fault.** The generation of counterexamples in case a formula is violated is a key service provided by model checkers. Counterexamples produced by model checkers often stand for error traces, which represent sequences of system states and transitions and are therefore usually lengthy and difficult to understand. The origin of error might be anywhere along these traces and even a combination of transitions that are not contiguous, thus it requires a lengthy analysis by designers. The automatic fault localization analysis

relying on the error traces in model checking is still an interesting challenge.

## 1.7 Contributions

In this Ph.D work, our objective is to propose a set of property-driven methods used to efficiently assess the real-time requirements in large scale concurrent reactive real-time systems. We rely on the UML and its profile MARTE as the end-user modeling language, and on the TPN as the verification language. The main contribution of this thesis is the design and implementation of a property-driven verification prototype toolset dedicated to real-time properties verification for UML-MARTE real-time software designs [GP12a, GPC14c]. We validate this toolset using an avionic use case and its user requirements. This research shows that the property-driven approach allows a better verification scalability. The architecture of the toolset is described in Fig. 1.7.1, which consists of five tools:

1. **System Model Mapping Tool: Definition of real-time property specific execution semantics for UML-MARTE architecture and behavior models [GPC12b].** With respect to the expected real-time requirement, we have defined the real-time property specific execution semantics for UML-MARTE architecture model (composite structure diagram) and behavior models (activity and state machine diagrams). The definition of execution semantics follows the property-driven approach. The execution semantics allows to map UML-MARTE entities to TPN models, which makes UML model executable and analyzable by the TINA toolset. This mapping conforms to the UML specification 2.4.1 [OMG11c]. It abstracts the system in order to provide more scalable verification.

2. **Property Specification Tool: Specification of real-time requirements relying on a set of real-time property patterns [GPC12a, GP12b].** From the viewpoint of requirement assessment, we advocate that the qualitative property patterns proposed by Dwyer and the quantitative property patterns proposed by Konrad are not semantically atomic. We have defined a set of real-time property patterns that contains the atomic property elements. These property patterns can be directly used to specify real-time requirements. The properties expressed using Dwyer/Konrad's patterns and CCSL languages can also be automatically translated to the verification targeted atomic property elements, which will then be assessed using the observer-based verification approach.

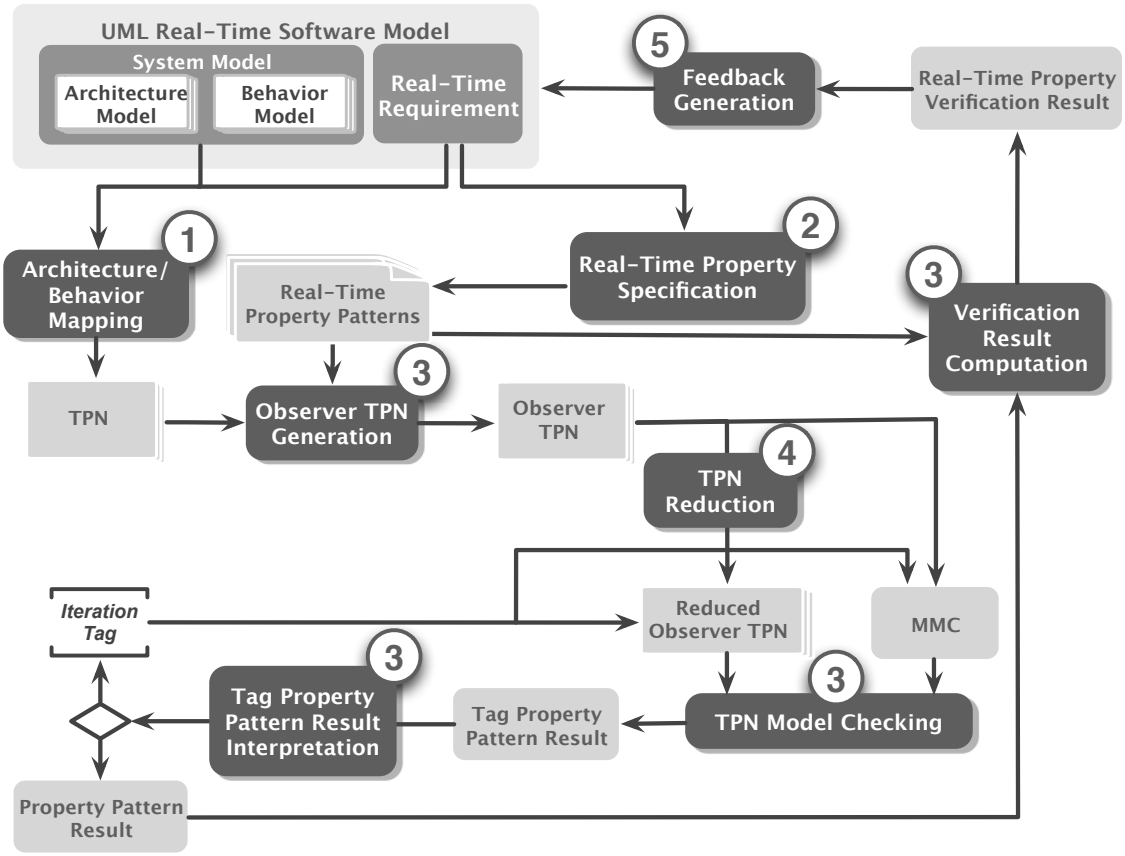



**Figure 1.7.1:** UML-MARTE Real-Time Properties Verification Architecture

3. **Property Verification Tool: Real-Time property specific observer-based model checking approach in TPN [GP12a].** The TINA model checking toolset that our work relies on can express qualitative properties on LTL and CTL logic formulae, but not the quantitative properties. To assess the real-time properties in an efficient manner, we define a set of event-based TPN observers and state-based tts observers, which will be associated to the TPN system under observation. These observers express the same semantics as the atomic elements defined in the real-time property patterns. The proposed observer-based approach allows to generate the high abstraction state class graph that only preserves marking information using the tina state space generation tool from the TINA toolset. It relies on the accessibility assertions in the modal $\mu$-calculus (MMC) and the muse model checker from the TINA toolset.

4. **Property Specific Reduction Tool: Real-Time property specific state space reduction approach for TPN [GP14].** We propose this property specific reduction tool to eliminate the property-irrelevant TPN structures and to build an equivalent of the property-relevant TPN structures in the system model. The reduction tool exploits the commutativity of TPN sub-nets which result in the same property-specific behavior before expanding the whole state class graph. The equivalent has less states and transitions, and thus directly reduces the scale of computation.

5. **Fault Localization Tool: Fault localization approach in model checking [GPC14a, GPC14b, GNP13b, GNP13a, GNP15].** We propose an automated fault localization approach based on model checking to ease and accelerate the debugging by locating and ranking the suspicious elements in a model when a safety property is unsatisfied. Inspired by the TF-IDF (term frequency-inverse document frequency) measure and the Kullback-Leibler Divergence theory, we propose a suspiciousness factor to rank the potentially faulty transitions. We apply this approach to property specific TPN model on which the observer-based verification approach is performed to obtain all the faulty execution traces and the violation states in the state class graph preserving markings. Based on the mapping semantics from UML to TPN, the faulty transitions is back-traced from TPN to UML.

## 1.8 The Structure of the Thesis

The thesis is structured into 4 parts containing 9 chapters (including this introduction), and 2 appendix complementing the main parts with additional information.

- **Part 1: Introduction**
  - Chapter 1 introduces an overview of the thesis.
  - Chapter 2 presents the state of the art of existing approaches.
- **Part 2: Contributions to property-driven approaches**
  - Chapter 3 introduces the definition of mapping semantics from UML-MARTE architectural and behavioral models to TPN models. (Contribution 1)
  - Chapter 4 presents a set of verification dedicated atomic real-time property specification patterns, and use it to translate the properties expressed using Dwyer/Konrad's patterns and the CCSL. (Contribution 2)

  - Chapter 5 proposes the observer-based model checking approach to verify the real-time property patterns in TPN. (Contribution 3)
  - Chapter 6 presents the property specific state space reduction approach for TPN models. (Contribution 4)

- **Part 3: Contribution to fault localization approach**

  - Chapter 7 proposes the automatic failure analysis approaches in model checking. (Contribution 5)

- **Part 4: Industrial application & Conclusion**

  - Chapter 8 uses an avionic case study, which is a part of the flight management system requiring latency and freshness real-time properties to test our toolset. The scalability test shows that the proposed approaches are capable to analyze large scale systems.
  - Chapter 9 concludes the main parts of the thesis and outlines future directions for research.
  - Appendix A gives the coverage library for mapping UML-MARTE to TPN model.
  - Appendix B contains the library for mapping real-time requirements expressed by Dwyer/Konrad's patterns to the proposed real-time property patterns.

# 2

# State of the Art

## Résumé

Le deuxième chapitre présente les informations sur l'état de l'art concernant les méthodes exploitées dans les chapitres suivants. Celui-ci comporte les éléments essentiels pour les travaux réalisés dans ce thèse :

- L'ingénierie dirigée par les modèles (IDM). Celle-ci vise à augmenter la fiabilité et l'efficacité de l'ingénierie traditionnelle du logiciel en exploitant des modèles exprimés dans des langages dédiés aux différents aspects d'un développement logiciel, des méthodes de validation et vérification des dits modèles, y compris l'exploitation de méthodes formelles, et des moyens de transformations automatiques de modèles.

- La modélisation de systèmes temps réels. Les travaux présentés dans cette thèse exploitent la notation UML (Unified Modeling Language) étendue par le profil MARTE (Modeling and Analysis of Real Time and Embedded systems) pour la modélisation au niveau utilisateur. Cette partie en présente

les aspects nécessaires à la lecture du manuscrit.

- La spécification formelle pour les systèmes temps réels. Les travaux réalisés s'appuient sur les méthodes formelles pour décrire un système, analyser son comportement et évaluer ses propriétés. Plusieurs formalismes traitent de l'analyse des propriétés temps réels. Nous présentons ici les automates temporisés et les réseaux de Petri temporisés et les outils correspondants.

- Les transformations de modèles. Celles-ci permettent de manipuler automatiquement les différents modèles intervenants dans le développement d'un système, et d'établir les liens entre les différents aspects et niveaux d'abstraction. Celles-ci peuvent être mis en oeuvre de différentes manière. Dans cette thèse, nous utilisons le langage de programmation Java et les outils « Eclipse Modeling Framework » pour transformer le modèle utilisateur (en UML/MARTE) en modèle de vérification (en réseau de Petri temporisé). Cela facilitera l'intégration des différents outils de la boîte à outils.

- La vérification formelle des systèmes temps réels. Cette tâche détermine si un système satisfait ses exigences lorsque ces deux éléments sont spécifiés formellement. Nous présentons les différentes approches pour la vérification temps réel et comparons les trois grandes classes de techniques appliquées dans la vérification formelle: l'analyse statique, la preuve de théorème et la vérification de modèles.

- La réduction de l'espace d'état dans la vérification de modèles. Ces techniques de vérification formelle souffrent du problème de l'explosion combinatoire de l'espace d'états. De nombreux travaux se sont consacrés à la recherche de solutions efficaces. Nous discutons dans cette partie des différentes stratégies de réduction couramment utilisées: analyse symbolique à base de diagramme de décision (BDD), réduction d'ordres partiels, raisonnement compositionnel, abstraction et symétrie.

- L'analyse des contre exemples de vérification des modèles. La génération de contre exemple lorsqu'une exigence n'est pas satisfaite est un service essentiel fourni par les vérificateurs de modèle. Un contre exemple est une trace d'exécution qui ne satisfait pas les exigences attendues. Nous présentons dans cette partie certains techniques d'assistance à la localisation des erreurs en fonction des contre exemples obtenus. Ces techniques visent à indiquer un ensemble d'éléments suspects dans le modèle sans les classer.

Cette état de l'art permet de conclure que, d'une part, nous pouvons proposer différentes méthodes pour réduire l'espace d'état lors de la vérification de modèles : définir une sémantique d'exécution du langage utilisateur UML/MARTE spécifiques à une famille de propriété; spécifier les exigences utilisateurs par traduction vers un ensemble de patrons de vérification atomiques; vérifier les propriétés en utilisant des observateurs; réduire l'espace d'état en s'appuyant sur les caractéristiques de la famille de propriétés considérée; et d'autre part, nous pouvons proposer un facteur de classement pour améliorer la précision de l'aide à la localisation des erreurs dans la vérification de modèles.

This chapter presents the background information and the state of the art related to the approaches detailed in the following chapters. The information contained in this chapter includes the introduction about model-driven engineering, modeling, specification, transformation and verification methods for real-time systems, information about the state space reduction techniques in model checking, details about the fault localization related feedback approaches in model checking. We conclude at last on the state of the art to explain why we choose to develop this work.

## 2.1 Model-Driven Engineering

Model-Driven Engineering (MDE) targets the improvement of the reliability and efficiency of the traditional software engineering by introducing models and early verification and validation ($V\&V$) including the use of formal methods. It has evolved over the last 20 years and achieved success in many domains. Models are reduced/abstract representations of real systems that selectively remove some semantics to highlight the remaining expected properties from a given point of view. In the context of safety critical systems, models can be used during the requirement engineering process to derive the requirements for a system, during the design process to describe the intended system to the implementation engineers, to verify and validate the properties, to automatically generate execution code, and also to document the system's structure and behavior after implementation.

Model Driven Architecture (MDA) [OMG01] is an important software design approach for MDE launched by the Object Management Group (OMG) in 2001. It is based on the standards UML, Meta Object Facility (MOF) [OMG11a], XML Metadata Interchange (XMI) [OMG11b], and the Common Warehouse Metamodel (CWM) [OMG03]. MDA provides a template for model-driven development processes and summarizes best practices and design patterns.

The work in this thesis is involved in the OPEES[1] and P[3] projects. Project P aims to support MDE of high-integrity embedded real-time system by providing an open code generation framework. It is able to verify the semantic consistency of systems described using safe subsets of heterogeneous modeling languages, ranging from behavioral to architectural languages and presenting a synchronous and asynchronous semantics (Simulink Stateflow/Matlab, Scicos, SysML, MARTE, UML); generate optimized source code for multiple programming (Ada, C/C++) and synthesis (VHDL, SystemC) languages; support a multi-domain

12

[3]http://www.open-do.org/projects/p/

(avionics, space, and automotive) certification process by providing open qualification material. Project OPEES aims to settle a community and build the necessary means and enablers to ensure long-term availability of innovative engineering technologies in the domain of dependable / critical software-intensive embedded systems. The goal is to build an ecosystem in the open source frame which provides a set of processes and guidelines for tools/components maturation, verification and qualification.

**Summary.** In the context of this thesis, we aim to design and experiment a set of methods for the development of scalable verification tools based on the property-driven approach. Our focus is on the analysis of the real-time systems in the architectural models specified by the modeling languages SysML, UML and MARTE.

## 2.2 Modeling of Real-Time Systems

*System modeling is the process of developing abstract models of a system, with each model providing a different view or perspective of that system. Building models which faithfully represent complex real-time system is a non trivial problem and a prerequisite to the application of formal analysis techniques. We may develop different models to represent the system from different perspectives, such as an external perspective for modeling the context or environment of the system, an interaction perspective for modeling the interactions between a system and its environment, a structural perspective for modeling the organization of a system or the structure of the data, and a behavior perspective for modeling the dynamic behavior of the system and how it responds to events.* [Som10]

Architecture Description Languages (ADL) have been used to model software system architecture since the 1990s. *An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization* [BRJ05]. A real-time software system is a system whose correct operation depends on both the results produced by the system and the time at which these results are produced. To deal with embedded real-time systems, some domain specific ADLs have been defined, such as AADL [FGH06] and EAST-ADL [DSLT+04]. UML is also a possible solution to address real-time embedded systems.

The Unified Modeling Language (UML) was developed and standardized by the OMG in 1997 to provide a common language for specification, modeling and documentation in the software development process.

In many senses, it was a success, because it established a standardized, graphical and easy-to-use notation modeling system which was comprehensive enough to capture all major aspects of software engineering. Today, UML is the industry standard for software modeling. As UML by itself was only a documentation and modeling standard, early UML tools were graphical editors used for communication rather than a central key technology of model-driven development. Although UML was not intended to be an ADL, the expressive capability of architecture by UML is more than any ADLs. UML provides large, useful and extensible set of predefined constructs, and meanwhile it has more potential for substantial formal analysis tool support. For this reason, UML can be used as an ADL.

UML has many kinds of diagrams and so supports the creation of different types of system model. However, a survey in 2007 [ES07] showed that most users of the UML thought that five diagram types could represent the essentials of a system: Activity diagrams, which define the activities involved in a process or in data processing; Class diagrams, which describe the static structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. State machine diagrams, which specify how the system reacts to the internal and external events; Sequence diagrams, which show interactions between actors and the system and between system components; Use case diagrams, which give the interactions between a system and its environment.

In the context of this thesis, since we are concerned with real-time property verification of concurrent reactive systems, we rely on the composite structure diagrams to specify the system architecture, and use the activity and state machine diagrams to specify the system behavior. The composite structure diagrams describe the internal structure of a class and the collaborations that this structure allows. Compared to the static-structured class diagram, composite structure diagram could be used to specify the behavior of collaborations.

Since the introduction of an extension language called UML Profile for Schedulability, Performance and Time (SPT) [OMG05a], UML enables the users to capture time and performance requirements, to assess those properties from early design models. However, practical experience with SPT revealed shortcomings within the profile in terms of expressive power and flexibility. MARTE (Modeling and Analysis of Real-Time and Embedded Systems) [OMG09] is intended to replace SPT to provide support for specification, design, verification/validation for real-time and embedded systems. It provides foundations for the model-based development. The architecture of the MARTE profile is shown in Fig. 2.2.1 [OMG09]. The shared package MARTE Foundation provides common concerns such as time and the use of concurrent resources; the package MARTE Design Model models the features of real-time embedded systems using the extensions GCM

(Generic Component Modeling), HLAM (High-Level Application Modeling), SRM (Software Resource Modeling), and HRM (Hardware Resource Modeling). The analysis features are supported by the package `MARTE Analysis Model`, which provides a generic package GQAM (Generic Quantitative Analysis Modeling) and two specific analysis domains SAM (Schedulability Analysis Modeling) and PAM (Performance Analysis Modeling). These first two specific analysis domains are entirely concerned with time, however the profile structure allows for adding additional analysis domains, such as power consumption, memory use or reliability.

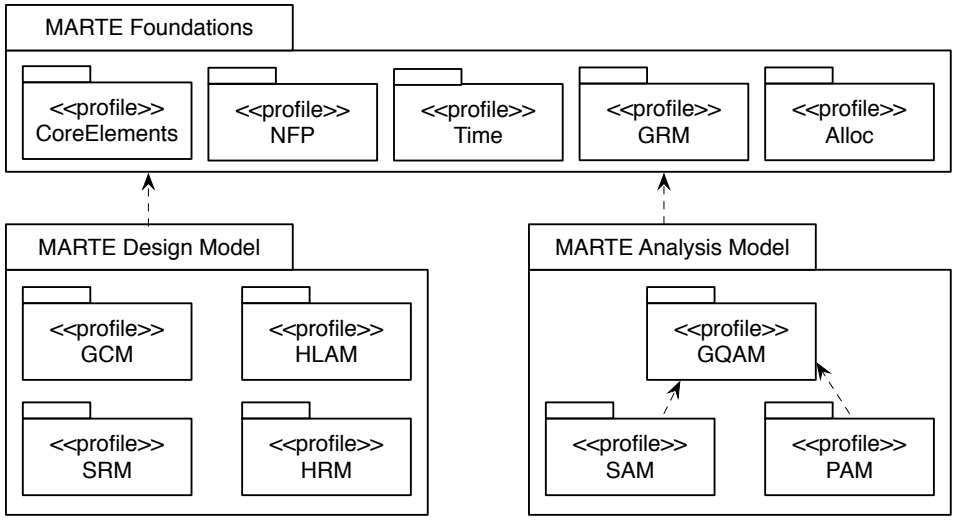



**Figure 2.2.1:** Architecture of `MARTE` Profile

**Summary.** In the context of the thesis, we use the term `UML-MARTE` to indicate the specification language.

## 2.3 Formal Specification of Real-Time Systems

As revealed by a number of surveys [Kob99], even the most recent versions of the `UML` specification suffer from multiple ambiguities and problems regarding the precise semantics of the language. This is a major problem for model-centric development which highly relies on precise modeling techniques, as the information contained in the user model will be directly propagated to the verification model.

Formal specifications are mathematical languages used to describe a system, analyze its behavior and help assess its properties. There exists a number of formal languages dealing with real-time analysis issues, such as Timed Automaton [AD94] and several extended Petri nets such as Timed Petri Nets [RH80, Zub91], Stochastic Timed Petri Net [FFN91], Time Petri Net [MF76], etc, among which Time Petri Net and Timed Automata are the most prominent.

### 2.3.1 Timed Automata

Timed automaton (TA) was introduced by Alur and Dill [AD94]. A timed automaton is a finite automaton extended with a set of dense time clocks, which are real-valued variables. A timed automaton evolves continuously and synchronously along with their physical clocks. In a timed automaton, each transition has a guard (a constraint over clock value or events) which indicates when such transition can be fired and a set of clocks to be reset when the transition is fired.

**Example 2.1 (Timed Automaton)** *The Timed Automaton in Fig. 2.3.1 models the processing of a task, where clk is a clock. After the reception of a signal proc, the automaton spends at least $t_{proc\_min}$ t.u. in the location Init, and then transits to the location Processing. Then, it emits the signal free_proc if the processing time does not exceed $t_{proc\_max}$, otherwise, it emits error_proc.*

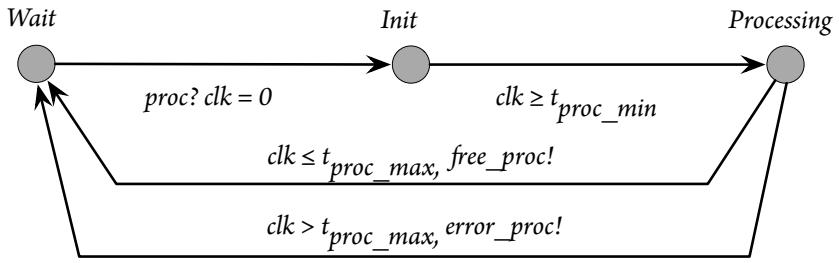



**Figure 2.3.1:** Timed Automata Example

Modeling and verification tools such as UPPAAL[4] and KRONOS[5] are based on timed automaton.

[4]http://www.uppaal.org/
[5]http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/index-english.html

### 2.3.2 Time Petri Net

Time Petri Nets [MF76] extends Petri Nets with timing constraints on the firing of transitions. Here we use the formal definition of Time Petri Net from [CR06] to explain its syntax and semantics.

**Definition 2.1 (Time Petri Net)** *A Time Petri Net (TPN) $\mathcal{T}$ is a tuple $\langle P, T, {}^\bullet(.), (.)^\bullet, M_o, (a, \beta)\rangle$, where:*

- $P = \{p_1, p_2, \ldots, p_m\}$ *is a finite set of places;*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a finite set of transitions;*
- ${}^\bullet(.) \in (\mathbb{N}^P)^T$ *is the backward incidence mapping;*
- $(.)^\bullet \in (\mathbb{N}^P)^T$ *is the forward incidence mapping;*
- $M_o \in \mathbb{N}^P$ *is the initial marking;*
- $a \in (\mathbb{Q}_{\geq o})^T$ *and* $\beta \in (\mathbb{Q}_{\geq o} \cup \infty)^T$ *are respectively the earliest and latest firing time constraints for transitions.*

Following the definition of enabledness in [BD91], a transition $t_i$ is enabled in a marking $M$ iff $M \geq {}^\bullet(t_i)$ and $a(t_i) \leq v_i \leq \beta(t_i)$ ($v_i$ is the elapsed time since $t_i$ was last enabled). There exists a global synchronized clock in the whole TPN, and $a(t_i)$ and $\beta(t_i)$ correspond to the local clock of $t_i$. The local clock of each transition is reset to zero once the transition becomes enabled. The predicate $\uparrow Enabled(t_k, M, t_i)$ is satisfied if $t_k$ is enabled by the firing of transition $t_i$ from marking $M$, and false otherwise.

$$\uparrow Enabled(t_k, M, t_i) = (M - {}^\bullet(t_i) + (t_i)^\bullet \geq {}^\bullet(t_k)) \wedge ((M - {}^\bullet(t_i) < {}^\bullet(t_k)) \vee (t_k = t_i)) \qquad (2.1)$$

**Example 2.2 (Time Petri Net)** *An example of Time Petri Net (presented in Fig. 2.3.2) models concurrent execution of a process. Compared to Petri Nets, the transitions in Time Petri net are extended with a time constraint that controls their firing time. $P_{init}$ is the place holding an initial token. Through the fork transition $T_{fork}$, concurrent $task_1$ ($T_{exe1}$) and $task_2$ ($T_{exe2}$) start at the same time within respective execution time [11,15] t.u. and [19,27] t.u.. The time constraint uses a local clock which starts once a transition becomes enabled. Until meeting join state ($P_{join}$), the system will exit ($T_{exit}$) or restart ($T_{restart}$) the whole execution according to the running time.*

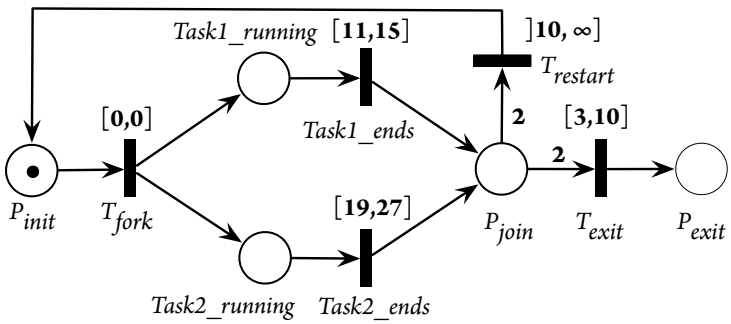



**Figure 2.3.2:** Time Petri Net Example

Time Petri Nets are widely used to formally capture the temporal behavior of concurrent real-time systems due to their easy-to-understand graphical notation and the available analysis tools, such as TINA[6] [BRV04], INA [7], Roméo [8], etc. Time Petri Nets are suitable for correctness, dependability, performance and timing analysis in early stages of design. Throughout the thesis, we use Time Petri Nets as the verification language for UML-MARTE models.

TINA allows data handling on TPN to perform classic imperative programming by adding common features like variable (of type integer and boolean) definition and arithmetic operation to each transition. The variable's value set extends the transitions and states in the reachability graph, which unifies the verification processes and makes it transparent to the TPN user while enlarging the modeling capability. An integer property from the state making can also be associated with the integer variables. These marking variables can only be read but not written. The formalism of TPN that is extended with arithmetic guards and actions that manipulate this set of variables is called Time Transition Systems (tts)[9]. Each transition in a tts has two associated functions:

- PRE represents an arithmetic guard: the transition will be enabled only when the TPN's marking and time preconditions and the guard are satisfied.

- ACT is the performed actions when the transition is fired. It can modify the variables that are used to compute the guards.

[6] http://projects.laas.fr/tina/
[7] http://www2.informatik.hu-berlin.de/ starke/ina.html
[8] http://romeo.rts-software.org/
[9] We use tts to distinguish from Timed Transition System (TTS)

An example of tts extends Ex. 2.2 by adding $\text{PRE}(T_{exe_1})$ ={$P_{task_2}$=0} and $\text{ACT}(T_{exe_1})$ = {X=10} on transition $T_{exe_1}$. When the number of token in the place $T_{task_2}$ is zero, $T_{exe_1}$ can be fired. Once $T_{exe_1}$ is fired, variable X is set to 10.

**Summary.** In the context of this thesis, we rely on the Time Petri Nets as the verification model. The end-user UML-MARTE model will be mapped to the Time Petri Net model to allow the formal analysis by model checking. The mapping work is realized using the model transformation techniques introduced in the next section.

## 2.4 Model Transformation

Model transformations are one of the central elements in MDE. They allow the automated processing and manipulation of multi-level models, and determine the propagation of information through various levels of abstraction and representation formats. They are used in MDA to translate system specification from abstract models to others, e.g. from platform-independent models into platform-specific ones or from models into an executable language.

Model transformation can be implemented in a number of ways [CH03]: by using a programming language such as Java or C++, by using a model transformation dedicated language and corresponding tools, or by a combination of native programs and hybrid transformations. We present some commonly used model transformation languages: QVT, ATL, and Kermeta.

**QVT** (Query/View/Transformation) [OMG08] is a model-to-model transformation standard adopted by OMG in 2007. It evaluates the expressions over a model to filter and select elements (Query), creates a new model from the original model (View), and finally expresses the transformation rules (Transformation) between both models. It uses the Object Constraint Language (OCL) [OMG05b] as mapping language and MOF as definition language. Model transformation engines that conforms to the QVT standard include for example SmartQVT [ABD+08], QVTo [QVT09], etc.

**ATL** (ATLAS Transformation Language) [JAB+06] is a model transformation language specified as both a metamodel and a textual concrete syntax. An ATL transformation program is composed of rules that describe how to create and initialize the elements of the target models. The transformation rules can be fully declarative, hybrid, or fully imperative. It allows expressing simple mappings between the source and target model elements. ATL was one of the experimental languages designed during the writing of the QVT

proposal.

**Kermeta** [FHN+06] is an executable metamodeling language which allows describing both the structure and the behavior of models. It supports EMF-based metamodeling, constraint checking, transformation and behavior support. The source models and metamodels are explicitly loaded and stored, and then the target elements are explicitly instantiated and added to the target model. Kermeta supports reflection, exception handling, object-orientation and aspect-orientation.

**Summary.** In the context of this thesis, we use the programming language JAVA to transform the end-user model to the verification model. This will ease the integration of different tools in the whole toolset.

## 2.5 Verification of Real-Time Systems

Verification of a system is the task that determines whether the system is built according to its explicit specification. Verification assesses the end products against its requirements and ensures that it will perform as specified. Model-based verification allows detecting errors earlier and preventing their propagation to later phases in the development. Since verification is conducted all along the development cycle, it provides manages with continuous and comprehensive information about the quality and progress of the development effort. The clients can also be given an incremental preview of system performances with the opportunity to make early adjustments to their requirements.

In practice, real-time property verification in MDE is implemented in 2 manners: simulation and formal verification. Simulation is relatively inexpensive in terms of execution time. But it only validates and verifies the behavior of concurrent systems for parts of possible computation paths. Several existing works have achieved success in the analysis of real-time systems. Contrasting to simulation, formal verification is a systematic process that uses mathematical reasoning to verify that the design intent is preserved in the implementation.

Cheddar [10] [SLNM04] is a real time scheduling tool designed for checking task temporal constraints of a real-time application/system described with AADL. It allows to specify systems composed of several processors which own tasks, shared resources, buffers that exchange messages. It provides a framework which implements most of the classic real time scheduling theory methods. The framework includes many feasibility tests and simulation tools. The tests can be applied to check that task response times are met

[10] http://beru.univ-brest.fr/ singhoff/cheddar/index.html

and that buffers have bounded size. SynDEx [11] is a system level computer-aided design tool intended to optimize the implementation, under real-time constraints, of embedded control applications onto multi-component architectures built from several processors and specific interconnected integrated circuits. It specifies and formally verifies software applications implemented on hardware. It analytically computes a schedule that matches the constraint (correct-by-construction) and generates optimized distributed real-time code thanks to formal verification and exploration of possible implementations manually, or automatically with optimization heuristics, based on multi-periodic distributed real-time scheduling analyses. UML-MAST [12] [MDH01] is a methodology and a set of tools for modeling and analyzing real-time systems expressed in UML. It provides an discrete-event simulator to assess the timing behavior of applications, including worst-case schedulability analysis for hard timing requirements and discrete-event simulation for soft timing requirements. MARTE2MAST [13] [MC11] is a tool that enables the extraction of schedulability analysis models and their direct analysis, which is similar to the methodology of UML-MAST but the modeling constructs are those defined in the MARTE standard. It supports analysis using simulation tools and static analysis.

There exists some other real-time property verification approaches such as the one based on Integer Linear Programming (ILP). Lauer et al. [LEBP11a] used a modeling approach for Integrated Modular Avionic (IMA) based on the tagged signal model [LSV97] and the abstraction of networks. The tag system was then transformed into an ILP problem. They proposed evaluation method for the end-to-end real-time properties based on ILP, and obtained optimal results.

Formal methods allow specifying a system's requirements, designing an implementation, and assessing its consistency, completeness, and correctness in a mathematical fashion. There are three main classes of techniques used in formal verification: static analysis, theorem proving and model checking.

### 2.5.1 Static Analysis

Static analysis [Kil73] is used to perform type checking and optimization in compilers, bug-finding in programs, and some formal verification on programs. When performing formal verification of property, it defines and proves a property of possible behaviors of a complex program without running the program. It's common to approximate or abstract information, e.g. instead of the natural numbers 0, 1, 2, ..., we could

[11] http://www.syndex.org/
[12] http://mast.unican.es/umlmast/
[13] http://mast.unican.es/umlmast/marte2mast/

use *zero*, *small*, *big*. Sound approximations include all the behaviors and reachable states of the real system, but are easier to compute.

Abstract interpretation [CC77] is a theory of sound approximation of the semantics of programs. It can be viewed as a partial execution of a program which collects information about its semantics (e.g. control-flow, data-flow) without performing all the computations. Static analysis is a main concrete application of abstract interpretation, which was first used in compilers for program optimization with FORTRAN [BBB+57] in 1954.

The advantages of static analysis include high efficiency for handling large systems, no need for the environment model (input/output, libraries, etc), and high degree of automation. The shortcomings include the production of false alarm caused by the imprecision, and the limitation for verifying dynamic variables because the analysis is not dynamic. Currently, there are approaches [Ern03] that compare static analysis and dynamic analysis to combine them in the verification.

The properties checked by static analysis are usually implicit, such as uninitialized variables, division by 0, index of array out of bounds, overflow/underflow, null pointer dereference, etc.

### 2.5.2 Theorem Proving

Theorem proving [Rus01] is a set of techniques to prove that an implementation satisfies its specification by mathematical deduction. The correctness claims are formulated as a mathematical theorems, which are then proved either manually or automatically with the help of a proof assistant such as `PVS` [ORR+96], `HOL4` [SN08], `Isabelle` [Pau00], `Coq` [DFH+91], etc. The automated theorem proving started in the 1960s.

The program and its execution context are first described using some appropriate language, which is then translated into logical formulae. The expected behavior is then itself described by a formula of the same language. The proof of correctness is then partly handwritten and synthesized. It is finally checked with the theorem prover by using a set of axioms and inference rules. Many different kinds of logic are used: propositional logic, first-order logic, and also non-classical logic and higher-order logic, etc.

Although theorem proving is sometimes able to prove the property fully automatically, it is more common that many human interventions are needed. Theorem-proving-based verification is thus very seldom fully automatic. In practice, theorem proving can only be used by experts. The construction of a proof may take lots of time, and might even be impossible as expressive logics are usually incomplete.

Another shortcoming with theorem proving is that it is not particularly good in providing debugging

information, that is, information as feedback to help locate errors in the system or expected behavior. Usually, when a proof fails, we often have to manually trace the invariant or variant that cannot be proved nor displayed, and analyze the reason of proof failure. Therefore, the debugging process is indirect.

Theorem proving allows the verification of the largest family of properties. It can cope with infinite state spaces of types. Theorem proving can be used in association with model checking by automatically generating a finite abstraction of the system to be verified. This method allows to decrease the complexity of the system and to resolve the undecidable boundedness problem [BCN98].

### 2.5.3 Model Checking

Model checking involves the design of a more abstract finite-state model $\mathcal{M}$, and the use of requirement $\phi$ expressed in temporal logics. A model checking problem requires to assess whether a given model satisfies a given property by searching state $s$ of $\mathcal{M}$:

$$\mathcal{M}, s \models \phi \tag{2.2}$$

Pioneer work in the model checking of temporal logic formulae was done by E. M. Clarke and E. A. Emerson [EC80, CE82, CES86] and by J. P. Queille and J. Sifakis [QS82]. Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their work on model checking.

In this part, as this thesis relies on model checking, we present its key principles (Section 2.5.3.1), the Kripke structure (Section 2.5.3.2), the temporal logic (Section 2.5.3.3), the model checking tools (Section 2.5.3.4) and discuss its strengths and weaknesses (Section 2.5.3.5).

#### 2.5.3.1 Model Checking Principles

Applying explicit-state model checking to a design relies on fours phases:

- **System formal specification** This phase builds a formal model from a design as input of model checker. This conversion can be a simple translation task or an abstraction work that eliminates irrelevant details. It can be partly or fully automated. It is common to use *Labeled Transition Systems* (LTS) as verification model that are equivalent to *Kripke structure*. [CGP99]

- **Property statement** This phase builds a formal model of the requirement (expected properties) in some logical formalism. It is common to use *temporal logic*, which can assert how the behavior of the

system evolves over symbolic time.

- **Model checking** This phase runs model checker to build all the execution paths and assess the satisfaction of expected properties by searching some desired (undesired) states. The verification by model checking is automatic and able to terminate if the state space is finite.

- **Feedback analysis** If the property is not satisfied in some executions, the model checker generates a counterexample. The counterexample is thus an error trace. It can be used to analyze the reason of error by simulation or other techniques, and furthermore to refine and adapt the design or the property.

#### 2.5.3.2 Kripke Structure

The state of a system is used to describe its status at a specific time instant. The behavior of a system can be seen as a finite or infinite set of transitions between the states. We use Kripke structures (`ktz`) to describe system's behavior. We recall the definition of Kripke structure from [CGP99, HC96].

**Definition 2.2 (Kripke Structure)** *A Kripke structure $M$ over a set of atomic propositions $AP$ is a 4 tuple $M = (S, S_0, R, L)$ where*

- *$S$ is a finite set of states.*
- *$S_0 \subseteq S$ is the set of initial states.*
- *$R \subseteq S \times S$ is a transition relation that must be left total, that is, for every state $s \subseteq S$ there is a state $s' \subseteq S$ such that $R(s, s')$.*
- *$L : S \to 2^{AP}$ is a function that labels each state with the AP that are true in that state.*

A path in a Kripke model $M$ can be an infinite sequence

$$\rho = s_0, s_1, s_2, \ldots \in S^*$$

such that $s_0 \in S_0$ and $(s_i, s_{i+1}) \in R$.

A state $s$ is reachable in $M$ if there is a path from one of the initial states to $s$. This path is then finite.

#### 2.5.3.3 Temporal Logic

Temporal logic was first introduced by Arthur Prior in [Pri57]. It describes the property of the ordering of events in time for LTS, and therefore can be used to specify the behavior of a reactive system and to specify the properties to be assessed for a given LTS.

**Linear Temporal Logic (LTL)** [Pnu77] is built from a finite set of atomic propositions, a set of logical operators negation ($\neg$), conjunction ($\wedge$) and union ($\vee$), constants true ($\top$) and false ($\bot$), and the temporal operators **G** for always ($\Box$), **F** for eventually ($\Diamond$), **X** for next ($\bigcirc$), **U** for until, **W** for weak until, and **R** for release.

Regarding the expressiveness of LTL, since LTL is propositional logic, i.e. it contains no quantification over variables, it can not specify quantitative properties, but only logical properties.

**Metric Temporal Logic (MTL)** [Koy90] is an extension of LTL over a discrete time line to support the specification of relative-time and real-time constraints. MTL contains time-constrained operators: always ($\Box$), eventually ($\Diamond$), next ($\bigcirc$), strong until (U), and weak until (W).

#### 2.5.3.4 Model Checking Tools

Many model checking tools have been developed to assess temporal logic formulae over labeled transition systems/Kripke structures. In this part, we present three widely used model checking tools: TINA, UPPAAL and SPIN.

**TINA** (TIme petri Net Analyzer) [BRV04] is a toolbox for the edition and analysis of Petri Nets, including inhibitor and read arcs, Time Petri Nets, including priorities and stopwatches, and an extension of Time Petri Nets with data handling called Time Transition Systems (tts) (that should not be mistaken with Henzinger's Timed Transition System (TTS)). TINA toolset includes the following tools: **nd** as editor and GUI for Petri nets, Time Petri nets and automata; tina for construction of reachability graphs; **sift** for construction and checking of reachability graph on-the-fly; **selt** as a State/Event LTL model checker; **muse** as a state-event modal $\mu$-calculus model checker; **plan** as an tool for computing the firing time of transitions or an example firing schedule (also called path); **play** as a simulator of the net described in any of formats .net, .ndr, .tpn, .pnml or .tts; and etc.

The tool tina provides various state space abstractions for Time Petri net (state class graph), following the techniques discussed in [BM83, BD91, Ber01, BV03]. Depending on the abstraction option selected, the construction preserves markings, states, LTL properties, or CTL * (a super set of computational tree logic (CTL) and LTL) properties of the concrete state space of the Time Petri Nets.

**UPPAAL** [14] [BDL04] is an integrated toolbox for editing and analyzing real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, user defined functions, etc.). UPPAAL consists of three main parts: a description language, a simulator and a model checker.

UPPAAL can check invariant and reachability properties by exploring the state space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints. UPPAAL uses a simplified version of CTL to specify property, where the query language consists of path formulas and state formulas. State formulas describe individual states, whereas path formulas quantify over paths or traces of the model. Path formula can be classified into reachability, safety and liveness. Each formula to be verified is transformed into a timed automata and composed with the system. If the property is not satisfied a counter-example is generated.

**SPIN** [15] [Hol97b] is the model checker for models of distributed software systems. It supports the analysis of model described in PROMELA language [Hol90]. SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in LTL, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL. Correctness properties can be specified as system or process invariants (using assertions), as LTL, as formal Büchi automata, or more broadly as general omega-regular properties in the syntax of SPIN.

#### 2.5.3.5 Strengths and Weaknesses of Model Checking

**Strengths.** Compared to theorem proving, model checking techniques are fully automatic, and do not require manual effort to construct the proofs. Another prominent advantage is that model checkers provide better feedback than other techniques. A counter example is provided when the property is not verified and this may help to find the origin of the error.

**Weaknesses.** Model checking can run into limitations due to the *combinatorial state explosion problem*. The number of states in the behavior of system model may easily exceed the amount of available resources. Several effective methods have been developed to handle this problem (see Section 2.6), however models of realistic systems may still be too large to fit limited resources. In practice, it is possible to use appropriate abstraction that preserves the information needed to assess a given property to reduce the state space. The user needs to build an appropriate finite-state model that fits the verification technology, which is also one

[14] http://www.uppaal.org/
[15] http://spinroot.com/spin/what.html/

of the objectives of this thesis.

**Summary.** In the context of this thesis, we rely on the TPN as the verification model and on the TINA toolset as the verification tools. One of the objectives is to provide efficient means to assess the real-time properties expressed with observers and accessibility assertions relying on high abstract state class graphs to reduce the cost of model checking.

## 2.6 State Space Reduction of Model Checking

Model checking techniques are user friendly as they provide better automation and error analysis than other techniques. Unfortunately, they suffer from state space explosion that can make it seem useless for large-scale systems. Indeed, in some systems, the size of a state space tends to grow exponentially in the number of its processes and variables, where the base of the exponentiation depends on the number of local states a process has and the number of values a variable may store, and on some kind of "tightness" of the connection between the components of the system i.e, the extent to which the local states of components are dependent of the local states of other components [Val98].

Many works were motivated to find effective solutions to state space explosion problem. There have been several major advances in addressing this problem. Most often, the advanced state space reduction takes advantage of details of the specific verification query. In this section, we discuss some commonly used state space reduction strategies: symbolic model checking with BDD (see Section 2.6.1), partial order reduction (see Section 2.6.2), compositional reasoning (see Section 2.6.3), abstraction (see 2.6.4) and symmetry (see Section 2.6.5.

### 2.6.1 Symbolic Model Checking with OBDD

In 1987, McMillan [BCM+92, McM93] acknowledged that the use of a symbolic representation for the state transition graphs allowed verifying much larger systems. Contrasting to the original implementation of model checking algorithm that represents explicitly the transition relations, symbolic model checking approach represents and manipulates a finite state transition system symbolically as a Boolean functions. The symbolic representation is based on Bryant's Ordered Binary Decision Diagram (OBDD) [Bry86].

Symbolic model checking can reduce the state space of explicit model checking, as it avoids explicitly

constructing the state graph of the system. By using OBDD, any finite-state system can be encoded using a set $\{b_1, b_2, \ldots b_n\}$ of binary variables. Sets of states, for example the set of initial states, can then be represented as propositional formulas over $\{b_1, b_2, \ldots b_n\}$, and sets of pairs of states, such as the pairs $(s, t)$ labeled with action $a$ can be represented as propositional formulas over $\{b_1, b_2, \ldots b_n, b'_1, b'_2, \ldots b'_n\}$. There exists many redundancies in the decision tree that can be removed by combining isomorphic sub-trees (producing a directed acyclic graph from the tree) and eliminating nodes with identical sub-trees [Mero1]. The model checking algorithm is based on computing fixpoints of predicate transformers that are obtained from the transition relation. The fixpoints are sets of states that represent various temporal properties of the system. In this way, the process of checking a propositional formula is to follow the path labeled with the boolean values for each of the inputs.

Most reduction methods were aimed to reduce the number of states. In symbolic model checking, the size of OBDD depends critically on the variable ordering, not on the total number of states. OBDD has achieved many successes especially in circuit design, as can be seen from the survey [MT98].

### 2.6.2 Partial Order Reduction

In asynchronous concurrent systems, most of the activities in different processes are performed independently, without a globally synchronization. The most successful techniques for dealing with this problem are based on the partial order reduction [CGP99, GvLH+96, GP93, Pel94, Val91].

The main idea of partial order reduction is to construct a reduced state graph by eliminating the unnecessary behaviors. This method is based on the dependency relations that exist between the transitions of a system. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. The reduction method then specifies the set of transitions that should be included in the reduced state graph. The reduced behavior is a subset of the behavior of the full state graph. Thus it does not add any information to the behavior of a system.

Many experiments showed that the state space for asynchronous concurrent systems can be significantly reduced. Different partial order reduction approaches works based on three different type of subsets of states: the stubborn sets [Val91], the persistent sets [GP93] and the ample sets [Pel94]. These techniques contain similar ideas, and only differ on details.

### 2.6.3 Compositional Reasoning

Compositional reasoning reduction techniques are considered effective in systems with modular structures when multiple processes are running in parallel. The main idea is a divide and conquer strategy that divides the whole specification into small parts and verify each part independently from the others.

The use of compositional reasoning follows three steps: first decompose the system specification into local properties that describe the behavior of small parts of the system; second, check each of the local properties using only the smallest part of system that it describes; finally, perform a conjunction of the local properties to derive the result for the full system.

Usually the system components exhibit dependencies to each other, which implies that the simple compositional reasoning is not feasible in such systems. Therefore, some assumptions about the behavior of other components are needed when verifying a property on one component. This strategy is called assume-guarantee reasoning [MC81, Jon83, Pnu85, GL94].

The main issues in compositional reasoning are how to devise proper assumptions and how to develop new proof rules. Some works were aimed to automatically generate the assumptions [CGP03, NMA08]. The assumption generation is based on machine leaning that uses the counterexamples generated by model checkers as the training data.

### 2.6.4 Abstraction

Abstraction reduction techniques [CGL94] are usually applied to systems that rely on data manipulations involved in the states. They attempts to reduce the state space by performing abstraction on the set of data and the operations applied to data. Two commonly used abstraction techniques are *cone of influence reduction* and *data abstraction*.

The technique of cone of influence reduction focuses on the total number of variables. It attempts to eliminate the variables that do not influence the verification of expected properties. As a consequence, a system description is simplified by referring only to the minimum set of variables, and the state space is reduced as states include these variables.

The technique of data abstraction focuses on the values of data. It attempts to find a mapping between the actual data values and a smaller set of abstract data values. By extending this mapping to states and transitions, it is possible to obtain a smaller abstract system that simulates the original system. Abstraction techniques are usually related to static analysis and abstract interpretation.

### 2.6.5 Symmetry

Symmetry reduction techniques [ID96, CEFJ96, ES96] are usually used in systems that exhibit topological symmetric identical components that are coupled to each other, e.g. redundant processes. The state space reduction approaches based on symmetry make use of the identical or isomorphic processes in a system to reduce the identical states in the state transition graph. Intuitively speaking, the behavior of a component can be replaced by the stored behavior of its identical component.

The construction of symmetry state space and its use in verification have been applied to Petri nets [Jen96], to CTL * model checking [CEFJ96, ES96], to Büchi automata [ES97, GS97], etc.

**Summary.** In the context of this thesis, instead of the generic reduction approaches presented in the above parts, we aim to provide property-specific reduction methods, which are dedicated to a new abstraction preserving all semantics related to real-time properties, while eliminating the others.

## 2.7 Model Checking Feedback

The generation of counterexamples in case a formula is violated is a key service provided by model checkers. A counterexample is a trace of execution that does not satisfy the expected properties. Several works investigated the algorithms for generating [CV03] and understanding [GKL04a, BBDC+09] counterexamples. Some work [ZCP13] builds failure scenarios for end user models using error traces. To diagnose a system design, generating a counterexample can be used to detect a fault, but the counterexamples produced by model checkers often stand for error traces in the verification model, which represent sequences of state changes and are therefore usually lengthy and difficult to understand, even worst with reduction and abstraction. More precisely, the origin of error might be anywhere along the error trace, thus requiring a lengthy analysis by designers. The ultimate goal is to trace the counterexample back to the designers' model in order to help fault detection, analysis and correction.

Fault localization is dedicated to monitoring a system, identifying when a fault has occurred, and pinpointing the location of the fault. One main fault localization approach derives the faults from some model, classified into the category Model-Based fault localization, which can be applied in MDE. In model-based fault localization, the system model may be mathematical, or knowledge based, including observer-based approach, parity-space approach, parameter identification based methods, etc [Din08]. The efficiency

and accuracy of model-based fault localization depends on the appropriate abstraction and reasonable assumptions. On the basis of a survey [Ali12], the major fault localization approaches are classified into 5 categories: program slicing [Wei81, AH90, GBF99], spectra-based fault localization [RBDL97, RtPR03, JHS02a], statistical inference [LYF+05], delta debugging [ZH02] and model checking. In this thesis, we discuss the techniques that use counterexamples that do not satisfy the expected behavior and try to locate the origin of faults in the model checking.

Automated fault localization in model checking intends to ease and accelerate debugging by indicating the suspicious components in the model. Current automated fault localization techniques usually either produce a set of suspicious statements without any particular ranking, or they use a suspiciousness factor and then rank all statements according it.

According to the survey [Ali12], we discuss the following important fault localization techniques.

- **Contrasting counterexamples with good traces using a single counterexample** [BNR03] proposed to analyze fault localization using one counterexample that violated the desired properties in a particular case. Whenever a counterexample is found, the approach compared the error trace derived from the counterexample to all the good traces that satisfied the property. On the observed error and good traces, the transitions that led to the deviation from good traces are marked as suspicious transitions. This technique has been implemented in the `SLAM` model checker [BR01].

- **Using multiple counterexamples** [GV03] introduced an approach that relies on multiple counterexamples. It defined the traces that started from initial states and ended with error states as negative traces, and the traces that did not take the error state as previous state as positive traces. The analysis approach denoted the transitions that existed in all positive traces, the transitions that appeared in all negative traces, the transitions that existed in one of positive traces but not in any negative traces, and the transitions that appeared in one of negative traces, but not in any positive traces. The algorithm then used the above marked transitions to define the cause of failure. This method has been implemented in `Java PathFinder` [HP00].

- **Distance metrics** [Gro04] proposed to define a distance between the error trace and the passing traces, which are traces in passing test cases that satisfy the requirements. The distance was then used to find the closest passing trace to the counterexample. The causes of error were then derived from the comparison results between the closest successful trace and the counterexample. This method was implemented in `explain` tool [GKL04b].

- **Abstract counterexamples** [CGS04] relied on a prediction that not all components of a model were involved in a specific property when performing model checking. It marked the states that actually affected the property. The algorithm was the same as `explain`, except that it used predicates when comparing error traces and passing traces. This technique has been implemented in the `MAGIC` model checker [CCG+04].

- **Reduction to Max-SAT** Max-SAT (Maximum Satisfaction) problem [Zim78] is the problem of determining the maximum number of satisfied clauses of a given boolean formula. [JM11b] proposed an approach that transformed fault localization problem to the Max-SAT problem. It used only one failing trace and the corresponding input to build the Max-SAT formulation. This method has been implemented in the `BugAssist` tool [JM11a].

All of the above 5 techniques were aimed to produce a set of suspicious statements without any particular ranking. The precision of fault localization can be improved by devising a suspiciousness factor and then ranking all suspicious statements according to it. There exist some works [JHS02b, AADW09] based on the suspiciousness factor.

To evaluate the success of a fault localization algorithm, some important criteria should be measured, such as effectiveness, precision, informativeness, efficiency, performance, scalability and information usage. Here we explain the most significant measurements: effectiveness and efficiency.

- **Effectiveness** An effective fault localization method should point out the origin of failure. The effectiveness can be evaluated by the precision. According to the survey [WD09], the effectiveness can be assessed by a score `EXAM` in terms of the percentage of statements that have to be examined until the first statement containing the fault is reached [EWDC10, WQ09, WSQG08, WWQZ08]. A similar score using the percentage of the program that need not be examined to find a faulty statement has been defined in [CZ05, JH05, RtPR03]. These two scores provide the same information, but the `EXAM` score is more direct and easier to understand. In this work, we use the `EXAM` score to assess the effectiveness of our approach, which is the percentage of transitions that have to be examined until the first faulty transition is found.

- **Efficiency** The fault localization techniques in model checking, like other techniques, should terminate in a timely manner, limited by some resource constraints. The efficiency can be assessed by the scalability and the performance.

**Summary.** In the context of this thesis, we aim to provide an automated fault localization approach based on model checking to ease and accelerate the debugging by locating and ranking the suspicious elements in a model when a safety property is unsatisfied.

## 2.8 Conclusion

This Ph.D work aims to design and implement a toolset used to verify real-time requirements in large scale UML-MARTE real-time designs. To deal with large scale systems, the main problem of verification techniques based on model checking is the combinatorial state explosion problem. It is interesting to adapt existing techniques or to construct new ones to prevent the combinatorial explosion in the process of real-time requirement verification. Based upon this purpose, the toolset should offer the following real-time property-specific tools, each of which can contribute to the prevention of the combinatorial state explosion:

- the tool for defining execution semantics to end user models (in our case UML-MARTE) and then mapping it to the execution model (in our case TPN)
- the tool for formally specify real-time requirements
- the tool for reduce the state space
- the tool for efficiently assessing the real-time properties using model checking
- the tool for feeding back the origin of errors in the model if a safety property is unsatisfied.

# Part II

# Contribution to Property-Driven Approaches

# 3

# Semantic Mapping from UML-MARTE to Property-Specific TPN


## Résumé

UML, le langage de modélisation unifié, est un langage généraliste qui doit permettre de modéliser n'importe quel type de logiciel. Pour une activité particulière, il suffit en général d'un sous ensemble du langage adapté pour les experts qui l'exploiteront. De plus, pour exploiter des méthodes de vérification formelle, il faut choisir un sous-ensemble adapté à la traduction vers ce type de modèle. Nous avons sélectionné un sous-ensemble des diagrammes UML adapté à la modélisation de la structure et du comportement d'architectures logicielles temps réel. Ce chapitre présente la méthodologie de l'approche « dirigée par les propriétés », qui constitue la base de notre contribution en terme de sémantique de traduction. L'objectif est de traduire automatiquement le sous-ensemble retenu des modèles UML-MARTE dans le formalisme des réseau de Petri temporisés pour permettre une vérification efficace de propriétés temps réels. Il s'agit de

ne conserver de la sémantique d'UML-MARTE que le niveau de détail adéquat pour vérifier ces propriétés dans le but de réduire l'espace d'états lors de la vérification de modèles. Les diagrammes UML considéré sont « structure composite » (composite structure), « machine à états » (state machine) et « activité » (activity). La communication entre les éléments peut être synchrone ou asynchrone. Du point de vue des horloges, comme le modèle sémantique des réseaux de Petri temporisés repose sur une horloge globale, nous considérons en premier lieu des systèmes mono-horloge donc des systèmes synchrones au sens des horloges. Afin de modélisation des systèmes multi-horloges, nous introduisons une notion de dérive de l'horloge pour lier les différents horloges indépendantes à l'horloge de référence. Ceci permet de simuler plusieurs horloges asynchrones. Les exigences considérés étant les propriétés temps réels dans la conception de l'architecture, les valeurs des objets sont ignorées lors de la traduction des modèles. Seul le type et le nombre d'occurrences des objets sont considérés.

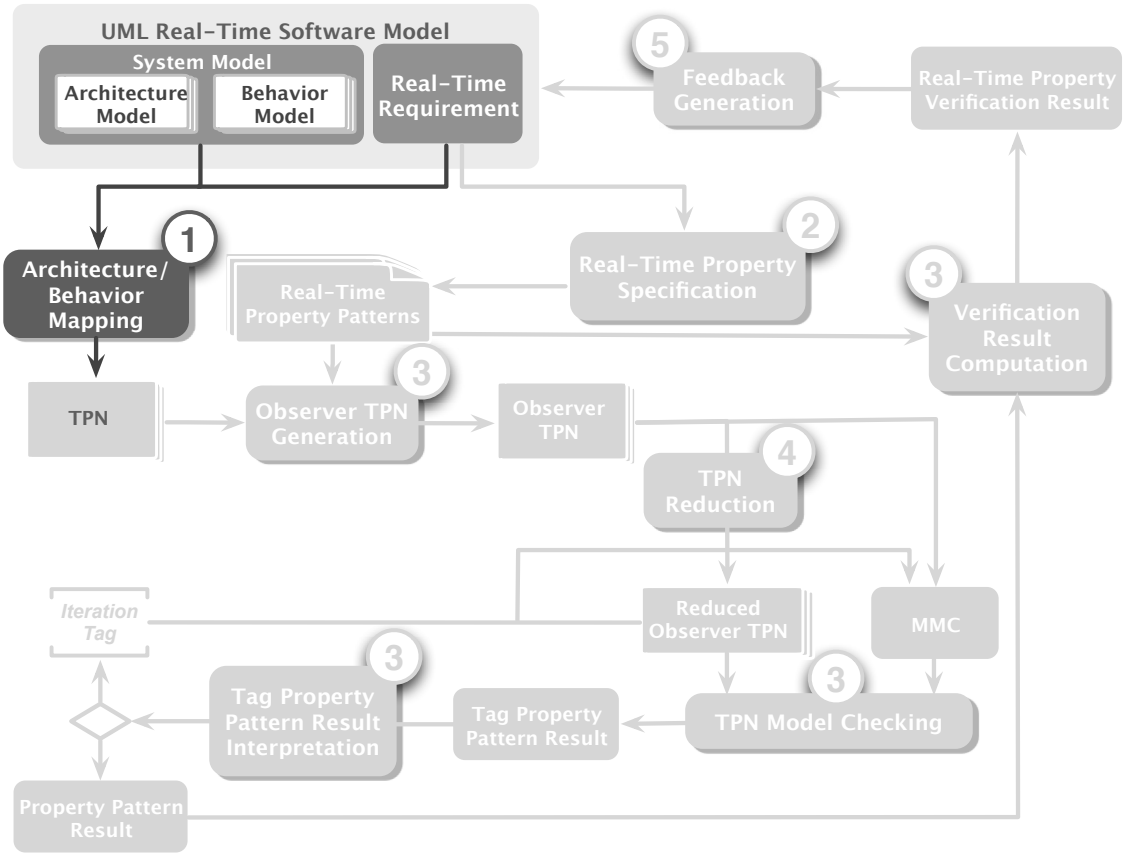
Les contributions principales de ce chapitre se résument en :

- Spécification de la sémantique d'exécution pour les digrammes de structure composite. Ce diagramme relie les comportements des sous-systèmes à travers les moyens de communication. La sémantique d'exécution traite donc les entités Part, Port et Connector.

- Spécification de la sémantique d'exécution pour les diagrammes d'activité. Le diagramme d'activité explicite le flot de contrôle (séquencement, coordination, ...) d'éléments de grain plus fin. La sémantique d'exécution prend en compte les noeuds de contrôle, les actions déclenchés par événement et par le temps, les objets et les connexions. Afin de normaliser la sémantique d'exécution pour le comportement asynchrone, nous étendons la sémantique d'origine des actions pour exprimer un comportement cyclique à l'aide du profil MARTE que nous traduisons en réseau de Petri temporisé. Il s'agit d'un modèle d'exécution classique pour les systèmes réactives asynchrones.

- Spécification de la sémantique d'exécution pour des diagrammes de machines d'état. Le diagramme de machine à état comportemental (behavioral state machine) est traité. La sémantique est explicitée en deux étapes. D'une part, les états hiérarchiques et les régions orthogonales n'apportent pas d'expressivité en terme de sémantique. Ils servent seulement à faciliter la modélisation de systèmes complexes. De telles machines peuvent être « aplaties » en exploitant des constructions des diagrammes d'activité. D'autre part, les machines à état « plates » comportant des état simples, des états finaux, des transitions et des pseudo non imbriquées sont traduits en réseaux de Petri temporisés.

- Spécification de la sémantique d'exécution pour les ordonnancements des ressources. Dans ce travail, nous ne cherchons pas à fournir la sémantique de toutes politiques d'ordonnancement de ressource mais d'assurer que l'ordonnancement est réalisable dans le pire cas. Nous proposons donc un algorithme d'ordonnancement générique avec possibilité de préemption. La spécification et la vérification en exploitant des politiques d'ordonnancement spécifiques pourront être traitées comme une extension de ces travaux.




*Progress Map 1:* Property-Driven Semantic Mapping

This chapter introduces the property-driven methodology, which provides the basis for the semantic mapping contributions in this thesis (Progress map 1). Property-driven mapping is aimed to map the end user source models (UML-MARTE in our case) to the target verification models (TPN in our case) on the basis of real-time property verification (Challenge 1 in page 22). The mapping consists in defining a dynamic semantics (a semantics of observable events) built upon the properties to be verified. To reduce the size of state space during the verification, the mapping eliminates the semantics irrelevant to the target property, while preserving a minimal set of property-relevant semantics. According to the target real-time property family, a denotational semantics is provided as a mapping from UML-MARTE architecture diagram (composite structure) and behavior diagrams (activity and state machine) to TPN. In addition, a generic scheduling algorithm with a preemption option is defined. (Contribution 1 in page 24)

## 3.1 Introduction

UML, by its nature, was intended to be a general purpose software modeling language, and as such, is not simple enough to be efficiently used by non-software experts. Many research works propose DSMLs (Domain Specific Modeling Languages) based on UML relying on profiles for different types of systems and different system engineering processes. For the purpose of this thesis, we have selected a large enough subset of UML diagrams and diagram elements for modeling real-time software architecture and behavior. We focus on the semantic mapping from the UML model to the verification model.

**Methodology.** From the viewpoint of methodology, our work is based on the pioneer work [CCG+07] by Combemale et al. Aimed to define all the steps from the property specification to effective verification, they introduced in [CCG+07] a generic approach to define the operational semantics (a semantics of observable events) built upon the properties expressed at the metamodel level. Their contribution was introduced through a simple process description language: SimplePDL on which a set of temporal properties, e.g a workdefinition must start after another workdefinition is finished, were expressed. Property-driven means that the formal activities in the development process are based on the purpose of property-verification-ease.

**Related work.** Currently there are many projects that have made great effort to define restrictions for UML. This is not the main concern of this thesis. We rely on the UML specification 2.4.1 [OMG11c] and the commonly accepted interpretation to define a formal semantics for the related UML subset.

Executable UML (fUML) [OMG13] aims to support a variety of different execution paradigms and environments. It is based on a very restricted subset of UML 2.4.1 that only handles parts of the activity diagram. fUML provides precise definition of the execution semantics at implementation level such as the type of variables, while in the work of this thesis, we focus on the operational semantics at verification level, i.e, implementation details should be eliminated in order to ensure the efficient formal verification.

Some works like [Cra05] studied the semantics of state machine, and provided the result of a comparative literature review on approaches to formally capture the semantics of UML state machines, and the underlying formalism of the approaches e.g., mathematical models (transition systems, abstract state machine, Petri Net, etc), rewriting systems and translation approaches.

Some existing works [LGMC04, TMH08, BBBB11, YYSQ10, AMCN09, CMC+08, MPFA06, MCBD02] have defined mapping semantics from UML diagrams to Petri Nets for verification purpose. The works

[LGMC04, TMH08, BBBB11] focused on the measurement of system's performance, or on the verification of correctness and inter-diagram consistency, or the cancellation and advanced synchronization patterns in untimed UML diagrams. The work [YYSQ10] mapped UML-MARTE activity diagram to Timed Color Petri Net with Inhibitor Arc to provide a possible foundation for analyzing time properties. This method focused on the mapping semantics of object flows. Colored Petri Nets (CPN) can ease the semantic mapping for data related elements, but to our knowledge there is no appropriate verification tools that handles the combinatorial state explosion problem. [MCBD02] presented a translation of untimed state machine diagram into Generalized Stochastic Petri Nets (GSPNs). This allows the qualitative and quantitative analysis of systems that are described using UML SMD by means of GSPN tools.

[AMCN09, CMC+08] relied on TPN to map SysML-MARTE activity and state machine diagrams to TPN with energy constraints to estimate the energy consumption and execution time of the system. This mapping is not generic, as it can only assess the execution time of the system. [MPFA06] defined a mapping for a restricted class of activity diagrams to Time Petri Nets to assess the quality of allocations of the system functionality. It provided a support for verifying deadline on activities.

Compared to the above related works, our semantic mapping targets the family of real-time properties, which is easier to encounter the combinatorial state explosion problem than the structural and temporal properties. Because the physical time behaviors will generate much more states than the logical behaviors in state class graphs. We do not target a new mapping completely different from the existing ones. On the other hand, it is impossible to define new mapping semantics for the UML elements like the initial, final nodes, as their semantics are very restricted. Our purpose is to define a mapping semantics by TPN for a relatively complete subset of UML-MARTE diagrams and diagram elements, and to apply this mapping semantics to a large scale systems to efficiently verify real-time properties. Therefore, we will adapt some of the execution semantics defined in the above related works, in order to improve the verification scalability.

**Modeling Context.** UML-MARTE can be used to model a wide range of real-time systems. It is not the main purpose of this thesis to cover all the modeling details of real-time systems. Therefore, before presenting our contributions, we need to clarify the modeling context in this work.

- **Synchronous & Asynchronous**: The systems that we are interested in are the concurrent real-time systems. From the viewpoint of message passing, we allow both synchronous and asynchronous communication modeling between different parts of the system. From the viewpoint of clock, as the TPN model provides a common synchronous clock, the modeling context allows synchronous

clock by default. To analyze quantitative properties, it usually needs for a reference clock, then the other clock are mapped to this reference. In order to enable the modeling of asynchronous clocks, we introduce the concepts of clock rate, drifts and offset to map multiple clocks to the reference clock, which is a simulation of multiple asynchronous clocks.

- **Cyclic execution**: Cyclic execution is a very common pattern for real-time systems that implement control & command to interact with the real world. An event-trigger cyclic execution can be simply specified using an action and a loop section. It is activated by the readiness of the data and the control flow. However, for the time-trigger execution, the activation of an action is controlled by the data, control flow and the rising edge of the cycle period. Although the semantics of offset and period can be modeled using MARTE profile, the whole behavior can be modeled in various ways. In order to ensure a standard mapping semantics, we decide to extend the original semantics for the action node defined in UML activity diagram. The original semantics of an action focuses on event-trigger behavior. We extend it by defining a time-trigger semantics using the MARTE profile, and then map it to the TPN model. This pattern is very general in the reactive asynchronous system, and thus can be reused in the modeling and verification. This issue is detailed in Section 3.4.3.2.

- **Object Value Issue**: As the object values do not affect the verification result of real-time properties in the architecture design of V-model (see page 18 in Chapter 1), they are ignored during the semantic mapping. Only type and the occurrence number of the objects are considered. This issue is detailed in Section 3.4.4.

- **Simplification on the use of MARTE**: In order to simplify the modeling and mapping works, we have used some simplifications on the use of MARTE profile. This simplification does not impact the execution semantics in TPN.

- **Resource scheduling**: In real-time systems, the behavior shares and consumes the resources such as the CPU, memory, bus. The scheduling policy applied by the scheduler will impact the real-time requirements. As modeling of scheduler policy is not the main concern of this thesis, we do not aim to provide mapping semantics for any specific policy. Instead, we propose a generic scheduling algorithm with preemption option, which is used to decide for the given time $T$, which resource instance(s) will be allocated to which requester(s).

**Contributions.** We aim to provide not only effective but also efficient property verification using the execution semantics derived from the mapping. Our contributions in this chapter are summarized as below:

- According to the expected real-time property family, we have defined the operational semantics for mapping UML-MARTE architecture diagram (composite structure) and behavior diagrams (activity and state machine) to the property-specific TPN model. The mapping library is provided in Appendix A. For some untimed UML elements not influencing real-time properties, such as Initial, Join, Fork and Merge nodes in the activity diagram, the target TPN semantics can be standardized and homogeneous for all families of the properties. For the timed UML elements, the mapping eliminates the semantics irrelevant to the target property, while preserves a minimal set of property-relevant semantics to reduce the size of state space in the verification.

- A generic scheduling algorithm including a preemption option is defined. This scheduling algorithm is used to decide for the given time $T$, which resource instance(s) will be allocated to which requester(s).

In this chapter, we give an overview of the property-driven approach in Section 3.2; then define the semantic mapping for composite structure diagram (Section 3.3), activity diagram (Section 3.4) and state machine diagram (Section 3.5); then we propose a mapping semantics for a generic scheduling algorithm including optional preemption (Section 3.6); we discuss the verification of model transformation and boundedness issues for the TPN with inhibitor arcs in Section 3.8; at last the time semantics in multi-clock modeling is discussed (Section 3.7).

## 3.2 Property-Driven Approach

### 3.2.1 Core Idea

Combemale et al. presented in [CCG+07] a property-driven approach for specifying and checking behavioral properties. The approach defined all the steps from the property specification to effective verification. Property-driven means that the formal activities in the development process are based on the purpose of property-verification-ease. Their contribution was introduced through a simple process description lan-

guage: SimplePDL on which a set of temporal properties were expressed. Combemale et al. defined the following steps to assess the properties relying on TPN and LTL:

1. The first step is to characterize the properties by the expert. The properties can be structural, temporal and quantitative ones according to SimplePDL. The *structural ones* are static construction rules that can be defined and checked by the use of OCL. The *temporal ones* are those properties that should be satisfied in every model execution. One example is a given process in SimplePDL should effectively terminate. The *quantitative ones* target the specification or synthesis of critical paths of executions in terms of minimal or maximal resource consumption such as the worst case execution time or resource use. [1]

2. The second step is to characterize a finite set of states for the metamodel entities from the property. For example, a *workdefinition* can either be not started, started or finished, and there is a notion of time and clock associated with each *workdefinition*.

3. Relying on these states, an observable abstraction of the generic operational semantics of the design model with respect to the properties is defined. This operational semantics makes the design model executable and thus analyzable by model checkers.

4. The fourth step expresses the property to be checked in the design model. For example, temporal properties can be expressed using TOCL (an extension of OCL with temporal operators) at the metamodel level. This has been implemented by Zalila in [ZCP12].

5. The fifth step formally expresses the operational semantics using the verification model TPN, and also formally expresses the property using LTL. The semantics is defined as a mapping from SimplePDL to TPN. The mapping can be implemented using model transformations, which are written in ATL in the work of Combemale and Zalila.

6. The final step performs the LTL properties checking on the TPN model using the TINA toolset. The feedback of properties results can be automatically computed using the transformation model defined during the translation SimplePDL2PetriNet.

[1] As the quantitative ones are more complicated, they illustrated their approaching using the structural and temporal properties.

In the context of this thesis, we follow the same methodology proposed by Combemale, and propose a property-driven mapping semantics to translate UML-MARTE models to the real-time property specific TPN models for the verification purpose. As the TPN models containing real-time semantics are easy to encounter the combinatorial state explosion problem, its mapping semantics needs to be property-specific, which preserves minimal property-relevant semantics.

Firstly, the real-time properties need to be characterized. For the second and third steps, as this work concerns the quantitative time properties, it is not as simple as the temporal or structural properties to characterize a finite set of states. As the quantitative time properties cannot be directly assessed using logic formulae such as LTL, CTL, the operational semantics built upon observable states/transitions needs to be defined using the standard observer techniques. In the fourth step, properties are expressed using a set of real-time property patterns defined in Chapter 4. Fifth, The operational semantics is formally expressed using the verification model TPN, associated observers and the logic formulae (Chapter 5). At last, the property expressed with logic formulae is checked using the model checker.

We do not follow exactly the same steps as the original work of Combemale, especially for the second, third, and fourth steps. Moreover, we add a new step for property specific state space reduction (Chapter 6) before the model checking, which aims to reduce the state space of model checking and thus improve the efficiency of verification.

### 3.2.2 Principles of Semantic Mapping

The semantic mapping approach for UML-MARTE is driven by the real-time properties that we plan to assess. The mapping should respect the following 5 principles:

1. The mapping rules for a UML entity may change according to the family of properties.

2. For some untimed UML elements not influencing real-time properties, the target TPN semantics is intuitive, and can be standardized and homogeneous for all families of the properties, and can be derived from the previously existing mappings.

3. The mapping rule should guarantee the consistency between high-level user models and lower-level verification models. However, a correct mapping does not imply a full semantics preservation, but rather to ensure as much as possible the scalability of verification while being correct according to the UML specification. For example, for the timed UML elements, the mapping eliminates the semantics

irrelevant to the target property, while preserving a minimal set of property-relevant semantics to reduce the size of state space in the verification.

4. The target TPN models should guarantee high performance verification, especially for large scale applications. Therefore, the mapping semantics should allow the TPN models to perform high-level abstraction to ease the generation of the state class graph during the model checking. In our case, we aim to generate the state class graph preserving only marking information (the minimal size state class graph), but not LTL information. This means the TPN model derived from the mapping must not possess the priority arcs. The priority arcs may ease the modeling work, but a TPN model with them is not allowed to generate state class graph preserving only marking information.

5. In order, first to be able to automate the model mapping process, and then to keep a simple mapping, a trade-off must be made to allow an easy assembly of the TPN mapping results for each UML entity. It seems that the verification efficiency would be decreased by this trade-off. But it in fact can be compensated later by a state space reduction phase that eliminates the elements irrelevant to the verification.

## 3.3 Composite Structure Diagram Mapping Semantics

The basic purpose of architecture model is to connect different sub-system behavior models and create a system-level model, by means of communication media. The objective of the mapping is to replace each architecture model's entities by its relevant behavior model while respecting a context-based naming convention and their structural relations.

*Composite Structure Diagram (CSD) is a kind of static structure diagram. It specifies the internal structure of a class, including its interaction points to other parts of the system, and the architecture of all parts managed by this class. CSD is used to explore run-time instances of interconnected instances collaborating over communications links.* (page 183 of UML Spec.)

The most significant entities in CSD are Part, Port and Connector. The others (Interface, Role) remain important, but either only disposing of static semantics for syntax consistency verification (e.g. Interface related nodes), or having ambiguities in common modeling work as its semantics differs according to the scenario (e.g. Role related nodes). In this section, we define the mapping semantics for the Part, Port and Connector and explain in detail why the others are not involved.

### 3.3.1 Part & Role

*A* `collaboration` *describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality.* (page 190 of UML Spec.) *A* `Part` *declares that an instance of this classifier may contain a set of instances by composition. All such instances are destroyed when the containing classifier instance is destroyed.* (page 206 of UML Spec.)

`Part` is an element that represents a set of one or more instances which are owned by a containing classifier instance. There is a tiny semantic difference between `Part` and `Role`. `Role` is a logical concept for a collection of functionality while `Part` is a physical instance that implements a collection of functionality. One `Part` can play different roles in the system simultaneously, and one `Role` could be played by only one `Part` at one time, but possible by different parts when time changes. For example, rear wheel and front wheel are two different roles in a car system, although they are designed to accept the same type of part (standard wheel) to implement. Therefore, we only consider the mapping semantics for `Part` in our work.

As a classifier can be either primitive or structural, the `Part` can also be primitive and structural (see Fig. 3.3.1). The `Part` itself is not mapped to any explicit `TPN` structure. Its semantics is mapped through the inner behaviors or structures. The mapping type provided by Fig. 3.3.1 is used to define the mapping semantics for ports and connectors in for following sections.

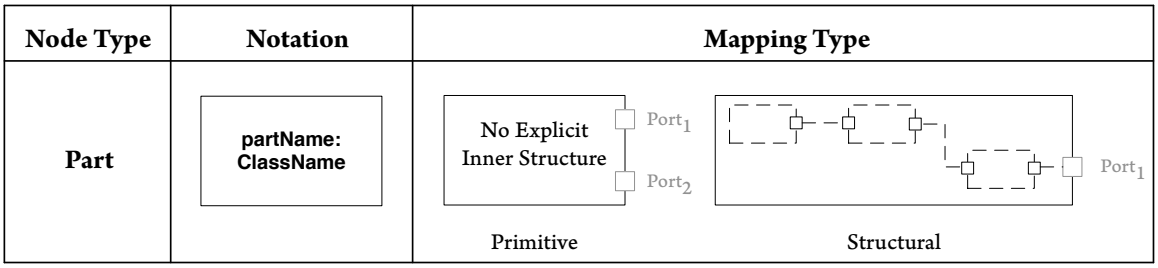



**Figure 3.3.1:** Mapping Semantics for Part

From the viewpoint of semantic mapping, a structured behavior is described by its inner structure, while a primitive behavior is described by its associated behavior model. In the structural `Part`, the architecture model could be considered as a tree-like structure, and the mapping process is based on a recursive tree-analysis approach. Each `Part` can be mapped to a `TPN` model, with inner behavior derived from the classifier or associated behavior model. These `TPN` models are then connected through the communication medias.

In our work, we map the whole system model to one TPN model containing all the parts and communication medias, in order to ease the analysis afterwards.

### 3.3.2 Port & Interface

*A* Port *represents an interaction point between a classifier instance and its environment or between a classifier instance and instances it may contain. Ports are connected to properties (parts) of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier.* (page 203 of UML Spec.) *The interfaces associated with a port specifies the nature of the interactions that may occur over a port.* (page 202 of UML Spec.)

A Port can appear on the boundary of a contained part, a class or a composite structure. A port may specify the services a classifier provides as well as the services it requires from its environment. An Interface is similar to a class with restrictions. All interface operations are public and abstract, and do not provide any default implementation. Both Port and Interface are often used to model interaction point. The logical view could be described by the interface, which specifies the provided service while its physical view is often modeled by port, which implements the specification. In our work, we only consider the mapping semantics for Port. A port is mapped to a place in the TPN models, as shown in Fig. 3.3.2.

| Node Type | Notation | Mapping Semantics |
|---|---|---|
| **Port** | **portName: ClassifierName** | ◯ |

**Figure 3.3.2:** Mapping Semantics for Port

**Allocation of ports.** To define the mapping semantics for the allocation of ports, we need to continue the discussion of primitive and structural parts mentioned in the previous section.

In the primitive part, the ports are in fact a semantic synonym for the Input/Output Pin of its inner behavior models; while in the structural part, they are a data buffer between the composites in CSD. In the former cases, an allocation semantics should be defined between the ports and the Input/Output Pin. Similarly, in the later case, the allocation semantics should also be defined, but between the ports on the container part and the ports on the inner parts. This allocation semantics is specified using the MARTE profile: MARTE::MARTE_Foundations::Alloc::Allocation. Fig. 3.3.3 provides an example of this allocation

in the primitive part. In its mapping semantics, the input pin is mapped to the same place of its associated port, and the output pin is mapped to the same TPN place as its associated port.

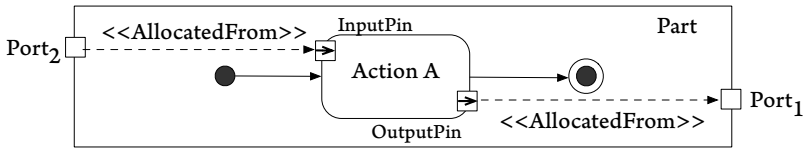



**Figure 3.3.3:** Example of Port Allocation Semantics

### 3.3.3 Connector

*Connectors specify links that enables communication between two or more instances. Each connector may be attached to two or more connectable elements, each representing a set of instances.* (page 197 of UML Spec.)

*If the type of the connector is omitted, the type is inferred based on the connector.* (page 197 of UML Spec.) In the context of this thesis, the type of the connector is always omitted during the modeling.

A connector is mapped to a transition with time constraint $[t_{min}, t_{max}]$ and relevant TPN arcs that connect the TPN places mapped from associated ports or Input/Output Pin (Fig. 3.3.4), where $[t_{min}, t_{max}]$ is the communication time specified by the following MARTE stereotype

```
MARTE::MARTE_Foundations::GRM::CommunicationMedia::Packet T.
```

| Path Type | Notation | Mapping TPN |
|---|---|---|
| Connector | ——— | $[t_{min},t_{max}]$ → |

**Figure 3.3.4:** Mapping Semantics for CSD Connector

**Allocation of connection.** *A connector consists of at least two connector ends, each representing the participation of instances of the classifiers typing the connectable elements attached to this end. The set of connector ends is ordered.* (page 197 of UML Spec.) Therefore, the connector end can be a part or a port. From a semantic

mapping point of view, there exist three kinds of connections: connection between two end ports, between two parts, and between one port and one part.

Similar to the allocation of ports, we recommend a modeling convention using the following MARTE stereotype to specify the allocation semantics between the connector ends:

```
MARTE::MARTE_Foundations::Alloc::Allocated
```

Fig. 3.3.5 provides the mapping semantics for these three kinds of connections respectively, where the ports and pins are mapped to TPN ports connected by the connector transitions.

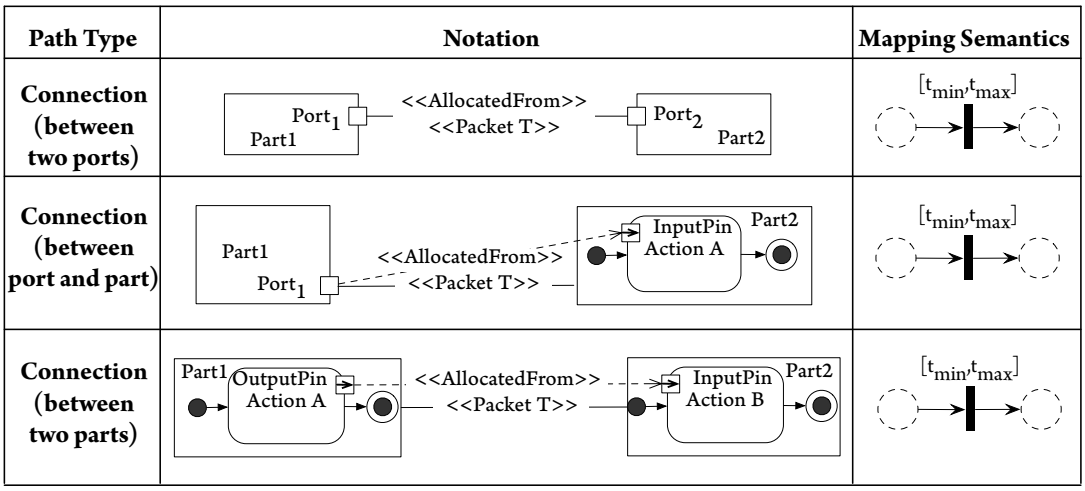



**Figure 3.3.5:** Mapping Semantics for CSD Connection

## 3.4 Activity Diagram Mapping Semantics

*Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow.* (page 319 of UML Spec.) The main elements in UML activity diagram (AD) behavior model are control nodes, actions, objects, and connection elements.

### 3.4.1 Semantic Mapping Pattern

In order to automate the assembly of the TPN elements mapped from UML-AD elements, a generic semantic pattern (Fig. 3.4.1) is defined. Only the TPN elements (transition, place, arc) in solid line belong to the mapping result of a given UML-AD element; those in dotted line are the mapping results of other connected AD elements. The TPN mapped from all the AD nodes except time-trigger actions, object nodes and connections. A target node could contain a set of transitions represented by T_IN at the beginning to connect with other predecessor structures. In the same manner, a set of places represented by P_OUT could exist in the end to connect with its successor structures.

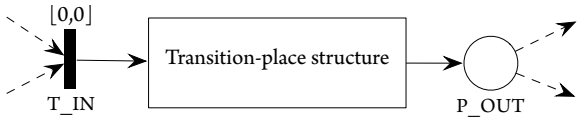



**Figure 3.4.1:** Generic Semantic Pattern of Activity Elements

### 3.4.2 Control Nodes

The mapping of some control nodes is intuitive, as Petri Net was the main inspiration for AD in the early versions of UML. Thus TPN possesses a similar semantics to the main control nodes (branch, concurrent, sequence, etc). For the pair of dual control nodes, the mapping results in TPN are also dual.

#### 3.4.2.1 Initial Node & Flow Final Node

Activity Initial node and FlowFinal node are dual nodes for control flow token. *An initial node is a control node at which flow starts when the activity is invoked.* (page 405 of UML Spec.) As the starting point of the diagram, Initial node emits the initial control flow token. An initial node does not have any predecessors.

*A flow final node is a final node that terminates a flow.* (page 402 of UML Spec.) FlowFinal node destroys the control flow token of one flow. A flow final node does not have any successors. The mapping semantics for this pair of nodes are shown in Fig. 3.4.2. Based on the mapping pattern introduced in the Section 3.4.1, initial node does not have any T_IN and flow final node does not have any P_OUT.

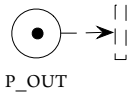
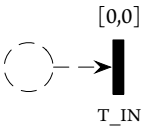
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **Initial Node** |  |   |
| **Flow Final** |  |   |

**Figure 3.4.2:** Initial Node & Flow Final Mapping Semantics

#### 3.4.2.2 Activity Final Node

*An* `ActivityFinal` *node is a final node that stops all flows in an activity.* (page 335 of `UML` Spec.) `ActivityFinal` node requires the immediate termination of all the activity flows and the destruction of all the control tokens. From the semantic mapping point of view, this means that all the activity flows should be terminated once the activity final node receives the control flow token.

In `TPN`, this "sudden exit" semantics is implemented using inhibitor arcs. An inhibitor arc enforces the precondition that the transition may only fire when the place is empty. Thus, when the activity node receives its token, all the transitions cannot be fired any more. The mapping semantics is thus defined as Fig. 3.4.3: each `TPN` transition in the activity is connected to the `ActivityFinal` node using an inhibitor arc. The decidability issue of the `TPN` with inhibitor arcs will be discussed in Section 3.8.

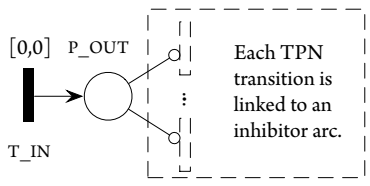
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **Activity Final** |  |   |

**Figure 3.4.3:** Activity Final Node Mapping Semantics

#### 3.4.2.3 Fork Node & Join Node

`Fork` and `Join` are dual nodes for concurrent control flow. *A `Fork` node is a control node that splits a flow into multiple concurrent flows. The edges coming into and out of a fork node must be either all object flows or all control flows.* (page 403 of `UML` Spec.) `Fork` node denotes the beginning of concurrent processing.

*A `Join` node is a control node that synchronizes multiple flows. A join node has one outgoing edge. If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow.* (page 409 of `UML` Spec.) `Join` node denotes the end of concurrent processing. All flows going into a `Join` node must provide a control token before processing may continue.

In the context of this thesis, we only focus on `Fork` and `Join` nodes for the control flows. The mapping semantics is defined in Fig. 3.4.4.

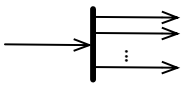
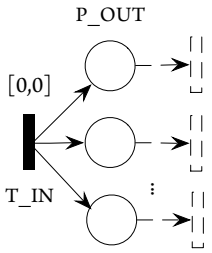
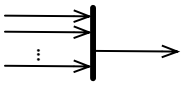
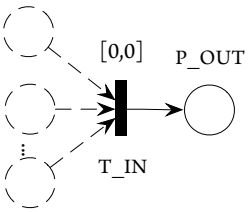
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **Fork Node** |  |   |
| **Join Node** |  |   |

**Figure 3.4.4:** Fork Node & Join Node Mapping Semantics

#### 3.4.2.4 Decision Node & Merge Node

`Decision` and `Merge` are dual nodes for branch control. The mapping semantics is defined in Fig. 3.4.5.

*A decision node accepts tokens on an incoming edge and presents them to multiple outgoing edges. Which of the edges is actually traversed depends on the evaluation of the guards on the outgoing edges.* (page 386 of `UML`

Spec.)

The outgoing control flows of a decision node usually include guard conditions which will allow the control of flow if the guard condition is met. The guard conditions are ignored in the mapping semantics. The reason is detailed in the following part (page 78).

*A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows. The edges coming into and out of a merge node must be either all object flows or all control flows. All tokens offered on incoming edges are offered to the outgoing edge. There is no synchronization of flows or joining of tokens.*(page 415 of UML Spec.)

Merge node brings together multiple alternate incoming flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows but only one token can be accepted. In the TPN of Merge node, if two incoming flows arrive at the same time, the place P_OUT will have two tokens. As a Merge node has a single outgoing edge, which maps to a single TPN arc, this enforces that the two tokens in P_OUT are consumed one by one. Thus this mapping semantics conforms the standard specification of UML.

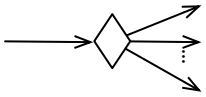
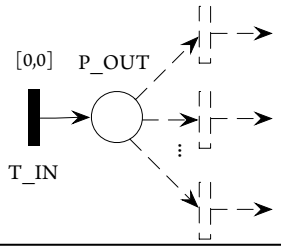
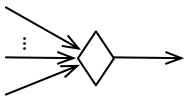
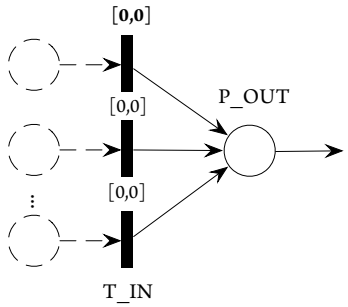
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **Decision Node** |  |  |
| **Merge Node** |  |  |

**Figure 3.4.5:** Decision Node & Merge Node Mapping Semantics

### 3.4.3 Action

*An* Action *represents a single step within an activity, that is, one that is not further decomposable within the activity. As a piece of structure within an activity model, it is a single discrete element; as a specification of behavior to be performed, it may invoke a referenced behavior that is arbitrarily complex.* (page 335 of UML Spec.)

UML-AD generalizes more than 50 types of concrete actions. Basic actions include those that perform operation calls, signal sends, and direct behavior invocations. In this thesis, we focus on the abstract action (page 259 of UML Spec.), all action executions will be executions of specific kinds of actions. When the action executes, and what its actual inputs are, is determined by the concrete action and the behaviors in which it is used.

The activity diagram can be used to model low-level behavior for both event-trigger and time-trigger requests. *Except where noted, an action can only begin execution when it has been offered control tokens on all incoming control flows and all its input pins have been offered object tokens sufficient for their multiplicity.* (page 336 of UML Spec.) This semantics is usually the case for event-trigger requests. For time-trigger requests, the action can begin execution when it has been offered control tokens on all incoming control flows, all its input pins have been offered object tokens sufficient for their multiplicity, and moreover it has been offered the rising edge of the periodic clock.

Cyclic execution is a very common pattern for real-time systems that implement control & command to interact with the real world. An event-trigger cyclic execution can be simply specified using an event-trigger action and a loop section. Although the semantics of offset and period can be modeled using MARTE profile, the whole time-trigger behavior can be modeled in various ways. In order to ensure a standard mapping semantics, we decide to extend the original semantics for the action, which focuses on event-trigger behavior. We extend it by defining a time-trigger semantics using the MARTE profile, and then map it to the TPN model. This time-trigger action is very general in reactive systems, and thus can be reused in the modeling and verification.

#### 3.4.3.1 Event-trigger Action Mapping Semantics

*An action begins execution by accepting all the offers of control and object tokens allowed by input pin multiplicity. When the execution of an action is complete, it offers control tokens on its outgoing control flows and object tokens from its output pins.* An event-trigger action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an event-trigger action may enable the execution of a set

of successor nodes and actions that take their inputs from the outputs of the action.

An event-trigger action may have sets of incoming and outgoing activity edges that specify control flows and data flows from and to other nodes. As the sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object tokens respectively, an action must contain inner behaviors for waiting/releasing control flow token and receiving/sending object. In addition, although in this thesis we focus on the verification of software system, it will be useful to keep an interface for the schedulability analysis on hardware systems in the future research, which means that some resource states are needed. Therefore, we add the extra behaviors of waiting/releasing resources in the mapping semantics for an action.

An event-trigger action can thus be defined as a 5-tuple $(I, C, T, R, D)$, in which:

- $I$ refers to identification, which is derived from its behavior semantics. Only two actions with exactly the same behavior can have the same identification.
- $C$ refers to behavioral context. If an action is reused in different activities, then they should be labeled with the same identification, and different context name.
- $T$ refers to time measure, e.g. the minimum and maximum execution time.
- $R$ refers to resource usage. The execution of an action will go on only when its required resources are ready and allocated to it. More precisely, the resource usage is a set of <$R$, $N$>, which indicates that for a given resource type $R$, the action requires $N$ of its available instances.
- $D$ refers to data section. It contains both inputs and outputs. The data are transferred by the Input Pins and Output Pins.

An atomic event-trigger action execution completes without interruption. However, it may stop to wait for the shared resources or transferred data. Therefore, an action is separated in 5 phases: *activity ready, resource ready, input ready, output released* and *resource released*, which are mapped to 5 TPN states. The mapping semantics is illustrated by Fig. 3.4.6.

The input resources are linked to transition Require_Res, and the output resources are linked to transition Release_Res. The resources are specified using MARTE profile. The details about the resource and the scheduling analysis are presented in the Section 3.6. The input data-related flows (Input Pin or Object Flow) are linked to transition Wait_Input, and the output data-related flows (Output Pin or Object

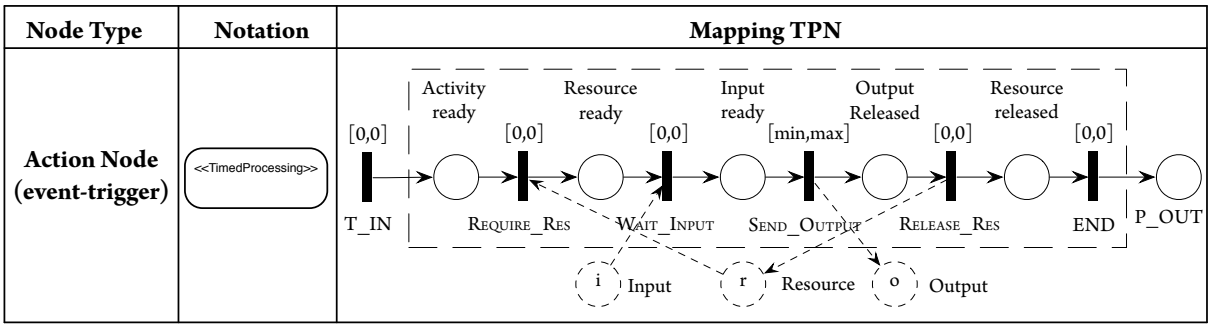



**Figure 3.4.6:** Event-trigger Action Mapping Semantics

Flow) are linked to transition SEND_OUTPUT. The transition SEND_OUTPUT also represents the execution of the action after the control flow, resource and input objects are all ready, on which the execution time is expressed by the time constraint `[min,max]`. This execution time constraint is specified using the `MARTE` profile: `MARTE::MARTE_Foundations::Time::TimedProcessing`

### 3.4.3.2 Time-trigger Action Mapping Semantics

For time-trigger requests, the sequencing of actions is controlled by both the control and object flows and the periodic clock. Before giving its mapping semantics, we first explain the commonly used time-trigger pattern in Fig. 3.4.7. Precisely, when a new cycle starts, if the input (e.g. Input A) is ready, the time-trigger action will start the execution and generate the output (output A) in the same way as an event-trigger action does. Nevertheless, if at this time the input is not yet available (e.g. input B), the time-trigger action will not wait but just ignore the current cycle, then retry the execution at the next cycle and produce the output (Output B).

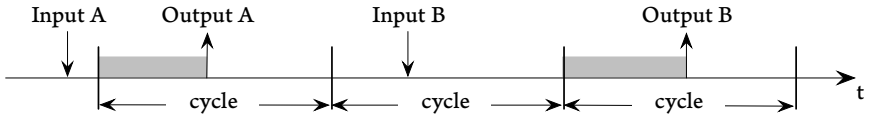



**Figure 3.4.7:** Time-trigger Action Pattern

A time-trigger action is defined by a 7-tuple $(I, C, T, R, D, Off, P)$, where:

- The first 5-tuples $(I, C, T, R, D)$ are the same as event-trigger action.

- *Off* refers to the offset. Before the first period starts, a time-trigger action can hold during the offset with respect to the local system's initial time.
- *P* refers to the period of the action. The action will be activated at the rising edge of the period, and then its input readiness is checked.

The mapping semantics is illustrated by Fig. 3.4.8. The `MARTE` profile allows to specify the real-time semantics relevant to time-trigger systems:

- **Offset**: `MARTE::MARTE_DesignModel::HLAM::RtSpecification::occKind::PeriodicPattern(Phase)`
- **Execution Time**: `MARTE::MARTE_Foundations::Time::TimedProcessing`
- **Period**: `MARTE::MARTE_DesignModel::HLAM::RtSpecification::occKind::PeriodicPattern(Period)`

Suppose that the given time-trigger action will handle *n* Input Pins and generate *m* Output Pins. In the mapping semantics defined by Fig. 3.4.8, in order to ease the explanation, we assume there are 2 input pins and 3 output pins.

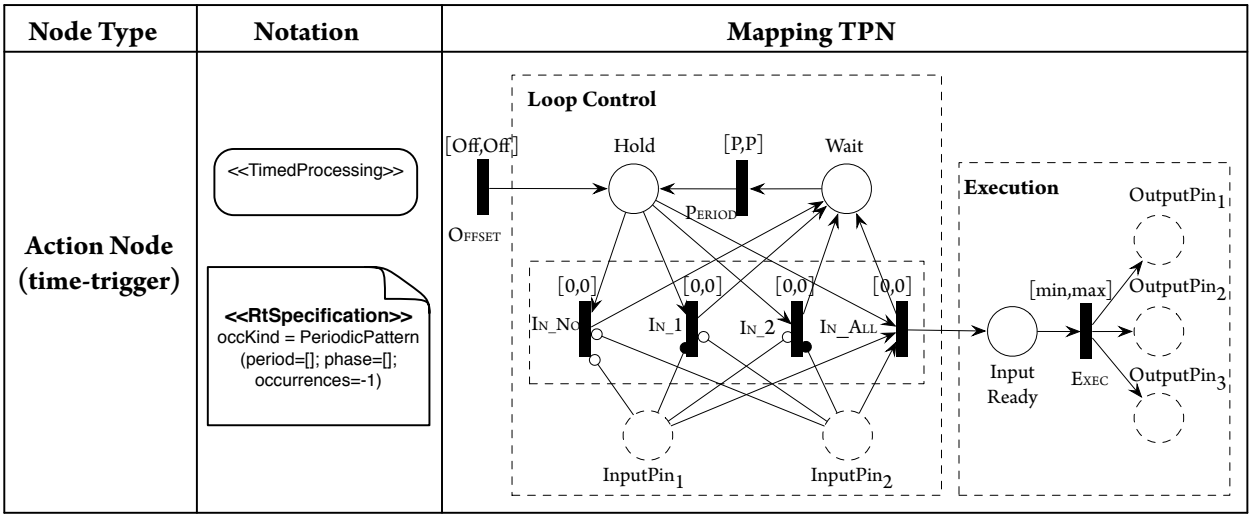



**Figure 3.4.8:** Time-trigger Action Mapping Semantics

The processing steps are as follows:

1. Once it enters into the **Loop Control** section after the activation of offset, the action enters the place *Hold* and will select its execution path when the rising edge of period arrives.

2. The selection is due to the input readiness at the moment: for the given $n$ input pins, there will be $2^n$ possibility of input readiness. As there are 2 input pins in Fig. 3.4.8, 4 possible paths are constructed: no input ready (transition IN_NO), only $input_1$ is ready (transition IN_1), only $input_2$ is ready (transition IN_2), both $input_1$ and $input_2$ are ready (transition IN_ALL). The mapping semantics for the data readiness is obvious. Take the transition IN_1 for example. When the place $InputPin_1$ has tokens while the place $InputPin_2$ has not, the transition is enabled by the *read arc* from the place $InputPin_1$ and the inhibitor arc from the place $InputPin_2$, and meanwhile a token is filled in the place *Wait*.

3. Once the period time *P t.u.* passes, the transition PERIOD is fired, and the asynchronous behavior comes back to the *Hold* state.

4. When both input data are ready, the transition IN_ALL is fired, and the place *Waint* and the place *Input Ready* in the **Execution** section are both filled with tokens. It means that the processing of data can start and the activity waits for the processing of next cycle.

5. The processing executes between *min* and *max t.u.* on the transition EXEC and produces the output data.

The scenario with resource usage for time-trigger actions relies on the same principle. Suppose the given time-trigger action have $n$ input pins while $r$ resources must be available to start the execution. There will be $2^{n+r}$ paths to choose: for each input readiness combination, there will always be a resource readiness combination to associate with. The *Execution* section will be connected to the transition standing for the path that both inputs and resources are available. This mapping semantics specifies exactly the same behaviors as described in the time-trigger action pattern (Fig. 3.4.7).

Time-Triggered systems are easy to encode and do not have state explosion problems. The cost do not depend on the number of inputs or resources. The semantic mapping itself does not introduce extra semantics. Suppose in a systems, a time-trigger action relies on 5 types of resource and 10 types of input data. In each period, if one of them is not ready, the action will not enter into the Input Ready state.

### 3.4.4 Object Nodes

*An object node is an abstract activity node that is part of defining object flow in an activity. An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to, as described in the semantics sub clause.* (page 421 of UML Spec.)

There are 5 types of object nodes in the activity diagram: pin nodes (`InputPin` and `OutputPin`), `CentralBuffer` node, `DataStore` node, `ActivityParameter` node, and `Expansion` node. We provide mapping semantics for the pin, `CentralBuffer` and `DataStore` nodes. The other two are related to the structural organization, but not the behavior. They will not affect the verification of real-time properties.

The most important aspect for object nodes semantic mapping is to keep the object type dependency and the object values in the TPN. To keep the object type dependency, each type of object can be considered as a variable in the memory and represented by a TPN place. However, if the object values are also kept, it will be very expensive to generate the state class graph during the model checking. This work ignores the object values in the object nodes, which is reasonable for the verification of real-time properties. This issue is detailed in the following part, then we provide the mapping semantics for the upper bound of object node, as it is a common feature for all types of object nodes, at last we provide the mapping semantics for each type of object nodes.

#### 3.4.4.1 Discussion on Object Value Issue

Each type of object can be considered as a variable in the memory and represented by a TPN place. Ideally, the upper bound and the object values of object nodes are mapped using `tts` variables (`tts` has been presented in the section 2.2 of Chapter 2). For example, if the upper bound of an object node is $N$, we can define $N$ variables to represent each value. During the model checking, the $N$ variables will generate $2^N$ markings, which will lead to a combinatorial explosion problem. Therefore, it is very expensive to keep the object values during model checking.

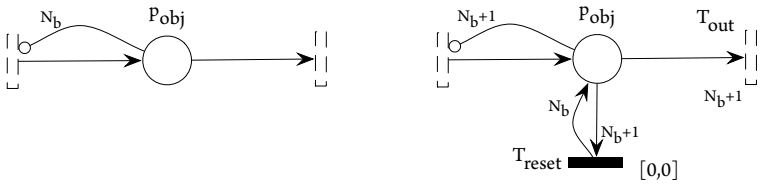
On the other hand, this issue is not a concern in the verification of the architecture design. The object values can be used as input of an action or an activity to compute the output value, or be used to compute the guards for the outgoing flows. The first one is the concern in the verification of detailed design in the V-model, not the architectural design. In the second case, the object value is not a concern either. We illustrate it using the decision node as an example. In the semantic mapping of decision node (page 72),

we have ignored the guard conditions. One of the principles of model checking is to check all the possible execution traces in a model. The guard conditions limit the execution traces relying on object values. Only if all the possible values are available, the execution traces will be complete. The guard conditions are ignored to make sure that all the possible execution traces can be checked. The only side-effect is that it may cause false alarms. This compromise is a must for the model checking techniques. Otherwise, the combinatorial explosion problem is easily met.

#### 3.4.4.2 Upper Bound of Object Node

*The upper bound of an object node is the maximum number of tokens allowed in the node. Objects cannot flow into the node if the upper bound is reached.* (page 422 of UML Spec.) The upper bound is a common feature in all types of object nodes. By default, the upper bound value is not defined in UML. This means that the object node is unbounded. We define its mapping semantics and use it in the mapping semantics for the object nodes.

Here we need to explicitly define the meaning of *"object cannot flow into the node"*. If the object node works like a buffer, it means that the new object value is blocked and thus cannot enter into the object node. Otherwise, if the object node works like a store, the new object value should enter into the object node and replace the old one. Thus, we define mapping semantics for upper bound of buffer-like and store-like object nodes respectively in Fig. 3.4.9.




**(a) Buffer-like Object Node** **(b) Store-like Object Node**

**Figure 3.4.9:** Upper Bound Mapping Semantics

Assume the upper bound is $N_b$. The place $P_{obj}$ represents the object node. For buffer-like object node (Fig. 3.4.9 (a)), an inhibitor arc between $P_{obj}$ and its incoming transition will limit the incoming tokens. Once $P_{obj}$ accumulates $N_b$ tokens, the inhibitor arc will prevent the $(N_b + 1)^{th}$ token to enter in $P_{obj}$, until some tokens in $P_{obj}$ are consumed by the outgoing transitions.

For store-like object node (Fig. 3.4.9 (b)), the $(N_b + 1)^{th}$ token should be allowed to enter into $P_{obj}$,

thus the weight on the inhibitor arc is $N_b + 1$, instead of $N_b$. However, this $(N_b + 1)^{th}$ token will replace the $N_b^{th}$ token already in the store, thus the transition $T_{reset}$ and its incoming and outgoing arcs are added. This structure ensures that once $P_{obj}$ accumulates $N_b + 1$ tokens, it is immediately reduced to $N_b$. This reset function does not take any time. A potential firing conflict might occur between the transitions $T_{out}$ and $T_{reset}$ when both of them are enabled. According to the feature of store-like object nodes, the $T_{out}$ should have the priority. This can be solved by adding a priority arc between $T_{reset}$ and $T_{out}$. However, the TPN with priority arcs does not support the generation of state class graph with markings, and thus will largely increase the size of state space. We do not recommend the use of priority arcs. The priority between $T_{reset}$ and $T_{out}$ in fact impacts the values of the object read by $T_{out}$. As the values of object nodes are ignored in this work, this priority can also be ignored here.

The InputPin and OutputPin work in both buffer-like and store-like manner. The CentralBuffer node works in the buffer-like manner. The DataStore node works in the store-like manner. We will detail their use in the following sections.

#### 3.4.4.3 Input Pin & Output Pin

*A pin is a typed element and multiplicity element that provides values to actions and accepts result values from them.* (page 287 of UML Spec.) Basically, a pin is mapped to a TPN place. Then the multiplicity and upper bound of tokens should be considered.

*Input pins are object nodes that receive values from other actions through object flows.* (page 406 of UML Spec.) *An action cannot start execution if an input pin has fewer values than the lower multiplicity. The upper multiplicity determines the maximum number of values that can be consumed by a single execution of the action.* (page 279 of UML Spec.) In the context of this thesis, we limit the modeling capacity by making the lower and upper multiplicity have equal value.

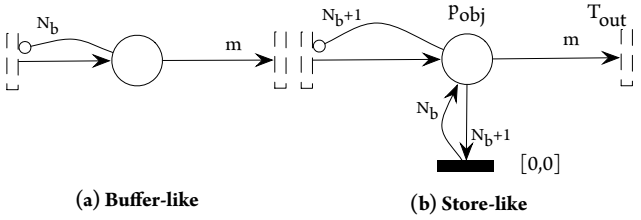
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **InputPin** |  |   |

**Figure 3.4.10:** Input Pin Mapping Semantics

Fig. 3.4.10 shows the mapping semantics for `InputPin`. If the `InputPin` is bounded by $N_b$, both buffer-like and store-like upper bounded mapping semantics are defined. The multiplicity is mapped to the weight $m$ on the normal arc between $P_{obj}$ and $T_{out}$.

*Output pins are object nodes that deliver values to other actions through object flows.* (page 425 of UML Spec.) *For each execution, an action cannot terminate itself unless it can put at least as many values on its output pins as required by the lower multiplicity on those pins. The values are actually put in the pins once the action completes. Values that may remain on the output pins from previous executions are not included in meeting this minimum multiplicity requirement.* (page 287 of UML Spec.)

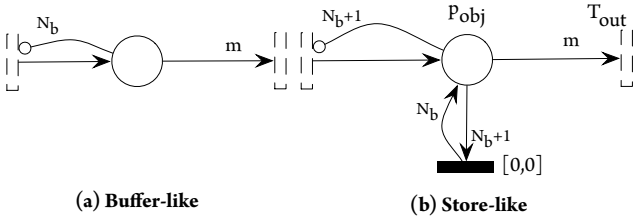
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **OutputPin** |  |   |

**Figure 3.4.11:** Output Pin Mapping Semantics

Fig. 3.4.11 shows the mapping semantics for `OutputPin`. If the `OutputPin` is bounded by $N_b$, both buffer-like and store-like upper bounded mapping semantics are defined.

#### 3.4.4.4 Central Buffer Node

*A central buffer node is an object node for managing flows from multiple sources and destinations. It accepts tokens from upstream object nodes and passes them along to downstream object nodes. Central buffer nodes give*

*additional support for queuing and competition between flowing objects.* (page 377 of UML Spec.)

The behavior of `CentralBuffer` is like a transient storage. Its mapping semantics is defined in Fig. 3.4.12. If the `InputPin` is bounded by $N_b$, the upper bound mapping semantics is the same as the buffer-like semantics defined in Fig. 3.4.9 (a). The `CentralBuffer` nodes do not have a multiplicity feature. The incoming and outgoing transitions of the object place $P_{obj}$ are the transitions mapped from the object flows (see object flow mapping semantics in Section 3.4.5).

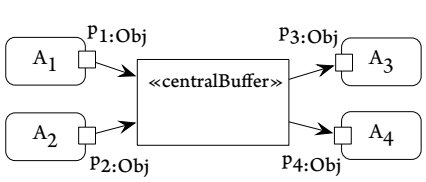
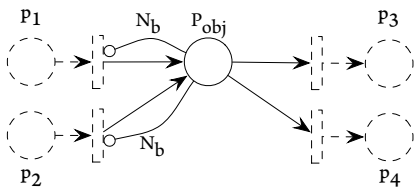
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **CentralBuffer** |  |  |



**Figure 3.4.12:** Central Buffer Mapping Semantics

#### 3.4.4.5 DataStore Node

*A data store node is a central buffer node for non-transient information. A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object.* (page 385 of UML Spec.)

The mapping semantics is defined in Fig. 3.4.13. If the `InputPin` is bounded by $N_b$, the upper bound mapping semantics is the same as the store-like semantics defined in Fig. 3.4.9 (b). The `DataStore` nodes do not have a multiplicity feature either.

The tokens flowing out of `DataStore` nodes are copies of tokens that remain in the `DataStore` node, so the values behave as if they are being read from the store. Here the "read object" semantics is defined using the *read arc*. This mapping semantics will guarantee that data persists in the current execution of activity, because the tokens in the place $P_{obj}$ will not be consumed through the read arc.

A state explosion issue may be caused by the `DataStore` node. If the design model does not make explicit the mechanism of disabling the use of `DataStore`, the data in it will be read infinitely many times, which leads to an unbounded error, i.e. the places $P_3$ and $P_4$ in Fig. 3.4.13 may become unbounded. Therefore,

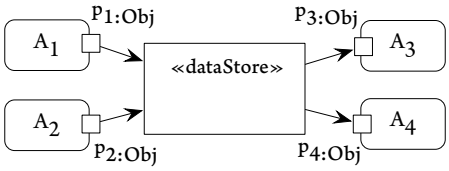
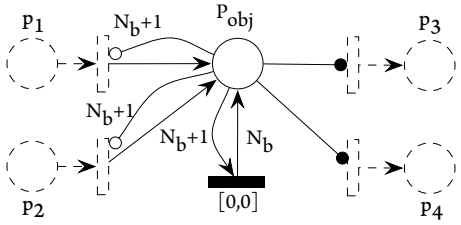
| Node Type | Notation | Mapping TPN |
|---|---|---|
| **DataStore** |  |  |



**Figure 3.4.13:** DataStore Mapping Semantics

when the UML model is designed, we need to be careful to use the DataStore. The mechanism for enabling and disabling the DataStore is a must to ensure the system is bounded.

### 3.4.5 Connections

A connection can be either control flow or object flow.

*A control flow is an edge that starts an activity node after the previous one is finished. Objects and data cannot pass along a control flow edge. Tokens offered by the source node are all offered to the target node.* (page 382 of UML Spec.) ControlFlow shows the flow of control from one node to the next. Control flow is mapped to an arc from the place $P_{OUT}$ of its source to the transition $T_{IN}$ of its target (Fig. 3.4.14).

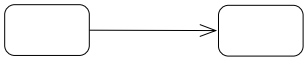
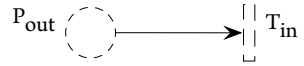
| Path Type | Notation | Mapping TPN |
|---|---|---|
| **ContorlFlow** |  |  |



**Figure 3.4.14:** Control Flow Mapping Semantics

*An object flow is an activity edge that can have objects or data passing along it.* (page 416 of UML Spec.) The mapping semantics of object flow is defined in Fig. 3.4.15. An object flow is mapped to a transition and its incoming and outgoing arcs. The following MARTE stereotype is used to specify the communication time of object flow: MARTE::MARTE_Foundations::GRM::CommunicationMedia::Packet T. This time specification is mapped to the time constraint on the TPN transition.

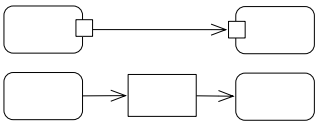
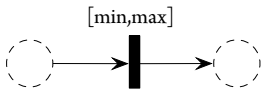
| Path Type | Notation | Mapping TPN |
|---|---|---|
| **ObjectFlow** |  |   |

**Figure 3.4.15:** Object Flow Mapping Semantics

## 3.5 State Machine Diagram Mapping Semantics

*The State Machine package defines a set of concepts that can be used for modeling discrete behavior through finite state- transition systems. In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. These two kinds of state machines are referred to here as behavioral state machines and protocol state machines. Behavioral state machine can be used to model the behavior of individual entities (e.g., class instances). The state machine formalism described in this sub clause is an object-based variant of Harel statecharts [Har87]. Protocol state machines are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger.* (page 551 of UML Spec.)

Protocol state machine is generally used to specify the classifier's behavior along with object's lifecycle or protocol usage. It specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier's operations. Protocol state machine can be simply considered as a specialization of behavioral state machine. Therefore in the context of this thesis, we only provide the mapping semantics for behavioral state machine, and all state machine (SMD) wording refers in fact to the behavioral state machine.

*State machines can be used to express the behavior of part of a system. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine.* (page 589 of UML Spec.)

Semantically speaking, the SMD defined in UML is an extension of Mealy machines [Mea55] and Moore machines [Moo56] that allows actions both on transition and state entry/exit. The activities can eventually depend on several resources to enable its execution. To define the mapping semantics for SMD, we first give a quick overview of the elements in SMD:

- **Vertex**: *A vertex is an abstraction of a node in a state machine graph. In general, it can be the source or*

*destination of any number of transitions.* (page 608 of UML Spec.)

- **Transition**: *A transition is a directed relationship between a source vertex and a target vertex.* (page 597 of UML Spec.)
- **State**: *A state models a situation during which some (usually implicit) invariant condition holds. Three kinds of states are distinguished: simple state, composite state and submachine state.* (page 575 of UML Spec.)
- **Pseudostate**: *A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph.* (page 565 of UML Spec.) According to specific semantics, the following kinds of pseudostates are distinguished: initial, deep history, shallow history, join, fork, junction, choice, entry/exit point and terminate pseudostates.
- **Final state**: *A special kind of state signifying that the enclosing region is completed.* (page 557 of UML Spec.)
- **Region**: *A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.* (page 573 of UML Spec.)

**Event pool and run-to-completion**: *The event pool for the state machine is the event pool of the instance according to the behavioral context classifier. The semantics of event occurrence processing is based on the run-to-completion (RTC) assumption, interpreted as run-to-completion processing. The processing of a single event occurrence by a state machine is known as a run-to-completion step.* (page 590 of UML Spec.) During semantic mapping, the event pool and the RTC issues should be explicitly specified.

**Flattening and mapping semantics**: Hierarchically nested states and orthogonal regions do not extend the semantic expressiveness. They help the designer in the writing of sophisticated models for complex systems. Other evolutions with respect to classical state diagram introduce some common elements from activity diagram, like fork/join, choice and junction. This is originally aimed to allow designer to model synchronization and control flow between SMDs in a more intuitive way. Without them, these behaviors can only be modeled as a group of emit/received events. Nevertheless, these pseudostates do not either extend the semantic expressiveness of SMD.

*Flattening* word is used when the mapping requires converting a nested SMD to an unnested SMD, which will ease the mapping afterwards. The final target is to have a SMD with only simple states, final states, transitions and unnested pseudostates.

*Mapping* word is only used for translating an unnested SMD model to a TPN model.

In this section, we first explicit the semantics of event pool (Section 3.5.1), then provides some general semantics for states in Section 3.5.2, then provide the flattening semantics (Section 3.5.3), at last provide the mapping semantics (Section 3.5.4).

## 3.5.1 Event Processing & Event Pool

Before presenting the flattening and mapping method, some important event related semantics must be clarified: the meaning of events, the processing method of events, and the event pool in state machines.

### 3.5.1.1 Event & Event Type

*The detection of an (event) occurrence by an object may cause a behavioral response. For example, a state machine may transition to a new state upon the detection of the occurrence of an event specified by a trigger owned by the state machine, or an activity may be enabled upon the receipt of a message. When an event occurrence is recognized by an object, it may have an immediate effect or the event may be saved in an event pool and have a later effect when it is matched by a trigger specified for a behavior.* (page 454 of UML Spec.)

Without specific priority, there are two categories of events defined in the system: *external event* and *internal event*. *Internal events* are local ones which are emitted explicitly in the system, either from the same state machine instance, or from an other instance of the same state machine, or from an instance of a different state machine. *External events* are events whose reception target is defined in the system specification, but not its emission source. For example, in an aircraft system (Fig. 3.5.1), the radio system's job is to generate a *call received* event to the onboard computer when it receives a *phone call* from outside. In this case, the *phone call* is an external event and the *call received* is an internal event.

It is important to distinguish these two event types because for a fixed period, external events are considered as infinite in terms of occurrence and without any information of arrival time. The internal events, however, has a finite occurrence bound because they are generated by a finite system. In the context of this thesis, we suppose the occurrence of both system events and environment events is finite.

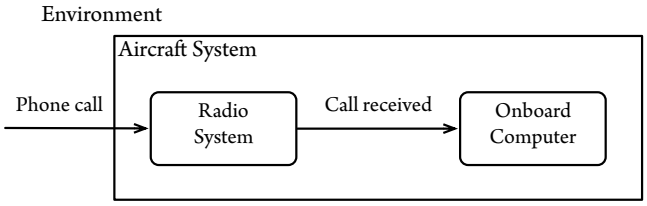



**Figure 3.5.1:** Event Categories Example: System & Environment

#### 3.5.1.2 Event Processing

To clarify how events are processed in state machines, we should first answer the following six questions:

- $Q_1$: How an event occurrence is processed between concurrent state machines? Can this event occurrence be processed simultaneously by these state machines?
- $Q_2$: How an event occurrence is processed between concurrent states in orthogonal regions? Can this event occurrence be processed simultaneously by these regions?
- $Q_3$: How an event occurrence is processed between concurrent states in the same region? Can this event occurrence be processed simultaneously by these states?
- $Q_4$: How successive event occurrences are processed? Can an event occurrence be processed concurrently with the previous event occurrence?
- $Q_5$: How an event occurrence is processed within a stable state in a state machine? Can the state machine pass between two states without finishing the processing of an event occurrence?
- $Q_6$: How an event occurrence is processed by two conflict transitions originating from the same state? Can both transitions fire simultaneously?

The UML Specification 2.4.1 provides explicit semantics for the questions $Q_2 - Q_6$, while the question $Q_1$ is not explicitly defined.

Answers for $Q_2$ and $Q_3$: **An event occurrence can be processed simultaneously by the orthogonal regions, but it cannot be processed simultaneously by the states in the same regions.** *In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event occurrence — as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled,*

*the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below. The order in which selected transitions fire is not defined. Each orthogonal region in the active state configuration that is not decomposed into orthogonal regions (i.e., "bottom-level" region) can fire at most one transition as a result of the current event occurrence. When all orthogonal regions have finished executing the transition, the current event occurrence is fully consumed, and the run-to-completion step is completed.* (page 591 of UML Spec.)

Answer or $Q_4$: **An event occurrence cannot be processed concurrently with the previous event occurrence.** *The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.* (page 590 of UML Spec.)

Answer for $Q_5$: **The state machine cannot pass between two states without finishing the processing of an event occurrence.** *The processing of a single event occurrence by a state machine is known as a run-to-completion step. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The run-to-completion step is the passage between two state configurations of the state machine.* (page 590 of UML Spec.)

Answer for $Q_6$: **Only one transition can be fired when two transitions originating from the same states are conflict.** *It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in conflict with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.* (page 591 of UML Spec.)

For the $Q_1$, as the UML specification does not explicitly define an event occurrence can be processed simultaneously by two concurrent state machines, or can only be processed by one of the state machines in conflict, we should suppose both cases are possible. We discuss these two cases in the following section for event pool.

#### 3.5.1.3 Event Pool

*The event pool for the state machine is the event pool of the instance according to the behavior context classifier, or the classifier owning the behavioral feature for which the state machine is a method.* (page 590 of UML Spec.)

We provide the explicit semantics for the above two kinds of event pool in Fig. 3.5.2. In (a), each state machine instance has an event pool, while in (b), the classifier owning the state machines has a universal event pool.

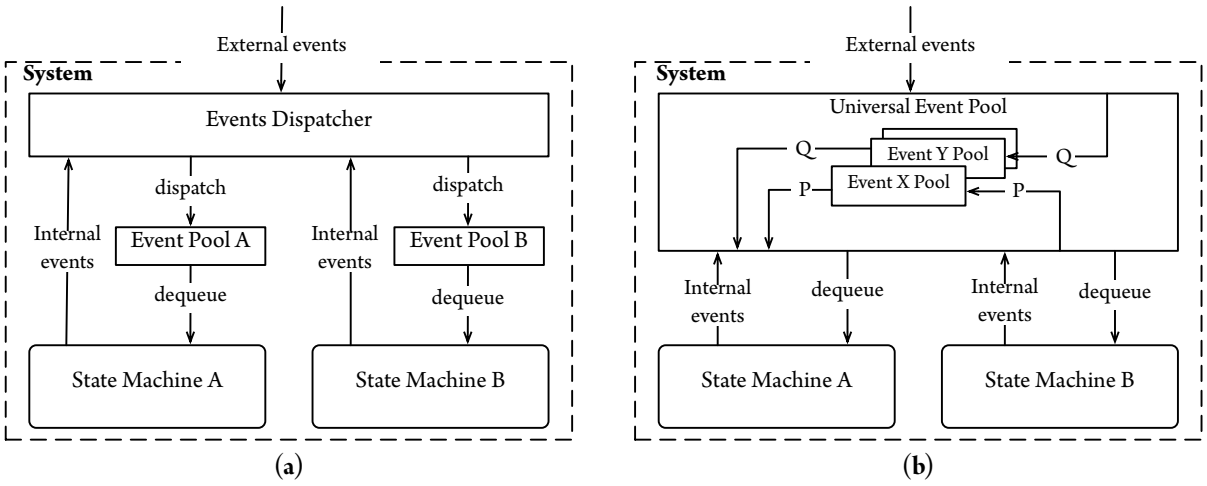



**Figure 3.5.2:** Event Pool Model

According to the above analysis for the question $Q_1$, both event pool semantics in (a) and (b) are expected to be able to handle both possible cases: an event occurrence can be processed either simultaneously by two concurrent state machine or only by one of the state machine in conflict. However, the semantics in (a) can only handle the former one, while the semantics in (b) can handle both, as analysis below.

The semantics in (a) signifies that each state machine instance (not including submachine state) is dynamically equipped with an event pool during execution. All events, no matter external events or internal events, will first be dispatched globally via the system's events dispatcher, then stored in this event pool. The events will be consumed sequentially by state machines. This semantics can handle the case such that state machines can process an event occurrence simultaneously. For example, an event occurrence of type *P* is sent to the system with concurrent state machine *A* and state machine *B*. *P* will be multi-dispatched to

both *A* and *B*. If both *A* and *B* are waiting for *P*, their transitions can fire simultaneously at the moment of reception. However, when several state machines compete for the same event occurrence, this semantics will not work. For example, when *P* is sent to *A* and *B*. *P* is not consumed neither by *A* nor by *B* for the given instant. The dispatcher does not know *P* should be dispatched to which event pool. The dispatcher must make a decision to multi-dispatch *P* to both *A* and *B*, otherwise, the whole system is blocked. Then, the conflict semantics between *A* and *B* is lost.

The semantics in (b) can solve this problem. In (b), the system has only one universal event pool, where all the events are pending to be consumed. If there is competition, the dequeue mechanism will decide randomly which state machine will get the event instance at this given instant. The universal event pool cannot be really a unique pool at system-level, because different state machine can react to the events in a concurrent way. State machine *A* is expecting an event instance of type *P*, and state machine *B* is expecting one of type *Q*; event instance *P* is pending before *Q*. If there is only one event pool, *B* cannot start handling *Q* until *A* dequeues *P*, which violates the original concurrent semantics. The universal event pool is in fact a set of event pool which are instantiated by event type.

**Capacity limit of event pool.** Another important implicit semantics of event pools is that they have a limited capacity. Sometimes an event arrives at an inconvenient time, when state machines are in a state that cannot handle this event. In many cases, the nature of the event is such that it can be postponed, within limits, until the system enters another state, in which it is better prepared to handle the original event. This limits is the fundamental guarantee for a real finite state system to behave as expected without pool memory explosion. Once the capacity is reached, either the new appended event will be ignored, or the old pending events will be dropped off. All kinds of clearance mechanisms are possible in real system design. In the context of this thesis, two general strategy are discussed in the section 3.5.4.5: *time out* and *size out*. But for the verification purpose, we suppose the event instances of the given system is restricted to the capacity of event pool, otherwise, it is considered as a design error.

### 3.5.2 State in General

As some flattening semantics is based on the inner behaviors of states, before discussing the flattening semantics, we provide some general semantics for states.

**Active States** *A state can be active or inactive during execution. A state becomes active when it is entered as a*

*result of some transition, and becomes inactive if it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (e.g., self transition).* (page 579 of UML Spec.)

**State Entry and Exit** *Whenever a state is entered, it executes its entry behavior before any other action is executed. Conversely, whenever a state is exited, it executes its exit behavior as the final step prior to leaving the state.* (page 579 of UML Spec.)

**Behavior in State (do-activity)** *The behavior represents the execution of a behavior, that occurs while the state machine is in the corresponding state. The behavior starts executing upon entering the state, following the entry behavior. If the behavior completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition (see below) the state will be exited. Upon exit, the behavior is terminated before the exit behavior is executed. If the state is exited as a result of the firing of an outgoing transition before the completion of the behavior, the behavior is aborted prior to its completion.* (page 579 of UML Spec.)

### 3.5.3 Flattening Semantics

The purpose of flattening is to convert a nested SMD to an unnested SMD, and therefore ease the mapping afterwards. The final target is to have a SMD with only simple states, final states, transitions (local and internal) and unnested pseudostates. During the flattening, not only will the topology change, but also the actions associated with original states and transitions will be modified. The nested SMD elements handled by the flattening include: regions, states (composite state and submachine state), external transitions, nested pseudostates (entry/exit point, shallow/deep history, and fork/join). We start discussing from the composite state, which exhibits more complex semantics than the other elements. The key to flattening a nested structure is to define the entering and exiting semantics from the topmost vertices to the innermost vertices.

*A composite state either contains one region or is decomposed into two or more orthogonal regions. Each region has a set of mutually exclusive disjoint sub-vertices and a set of transitions. A given state may only be decomposed in one of these two ways. A composite state is either a simple composite state (with just one region) or an orthogonal state (with more than one region).* (page 576 of UML Spec.) We first discuss the flattening semantics for the simple composite state, and then discuss the flattening semantics for the orthogonal composite state.

#### 3.5.3.1 Simple Composite State

A substate is defined as the state enclosed within a region of a composite state. When a substate does not contain any other state, it is called *direct substate*; otherwise, it is referred to as an *indirect substate*. A semantic variation point about the default entry rule is defined in the specification of UML.

**Semantic variation point (default entry rule)** *If a transition terminates on an enclosing state and the enclosed regions do not have an initial pseudostate, the interpretation of this situation is a semantic variation point. In some interpretations, this is considered an ill-formed model. That is, in those cases the initial pseudostate is mandatory.*

*An alternative interpretation allows this situation and it means that, when such a transition is taken, the state machine stays in the composite state, without entering any of the regions or their substates.* (page 576 of UML Spec.)

In the context of this thesis, we use the former interpretation: the initial pseudostate is mandatory in such cases.

**Entering a non-orthogonal composite state** The specification of UML differentiates the following cases:

- **Default entry** *Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state. In this case, the default entry rule is applied (see Semantic variation point (default entry rule)). If there is a guard on the trigger of the transition, it must be enabled (true). (A disabled initial transition is an ill-defined execution state and its handling is not defined.) The entry behavior of the composite state is executed before the behavior associated with the initial transition.* (page 580 of UML Spec.)

  The flattening semantics is defined as Fig. 3.5.3. The default entry rule is applied: the outside state *A* transits to the outgoing state of the initial pseudostate (substate *B*).

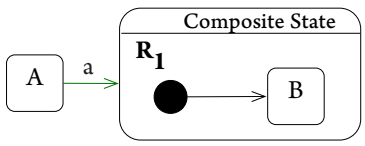
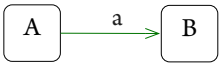
| Nested Machine | Notation | Flattening Semantics |
|---|---|---|
| **Simple Composite State Default Entry (with Initial node)** |   |   |

**Figure 3.5.3:** Default Entry Flattening Semantics for Simple Composite State

- **Explicit entry** *If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate.* (page 580 of UML Spec.)

  The flattening semantics is defined as Fig. 3.5.4. As the transition from state *A* goes to the substate *B*, then *B* becomes active, and the entry code of the composite state is executed before the entry code of *B*.

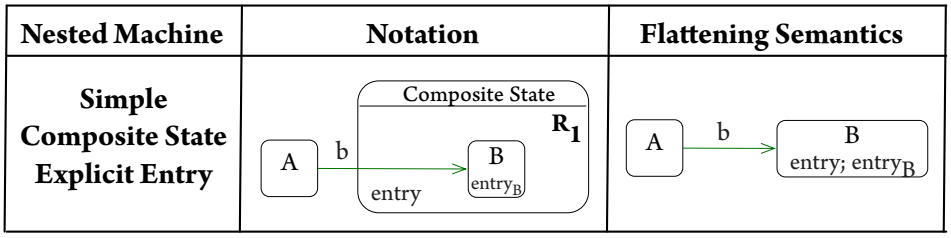



**Figure 3.5.4:** Explicit Entry Flattening Semantics for Simple Composite State

- **Shallow history entry** *If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the default history state is entered. This is the substate that is target of the transition originating from the history pseudostate. (If no such transition is specified, the situation is ill-defined and its handling is not defined.) If the active substate determined by history is a composite state, then it proceeds with its default entry.* (page 580 of UML Spec.)

  Since shallow history is a reference to the most recent substate, the flattening algorithm must have a mechanism to remember which is the most recent substate. As illustrated in Fig. 3.5.5, in the flattened version, the shallow history does not exist any more. It is replaced by some newly defined guards and actions on the associated transitions. More precisely, a variable *last active (LAST)* under the namespace of the given composite hierarchy is declared for this composite state, and all the inner transitions which will cause the substate to be activated/deactivated will have a supplementary action: record which substate is the most recent activated. The default value of the variable *LAST* when no substate has ever been activated is to use the initial pseudostate. All incoming and outgoing transitions of the shallow history will be copied and linked to each substate, with a supplementary guard defined: only when the target state is the most recent activated state will the guard be enabled.

All these guard conditions are mutually exclusive, thus at any time at most only one of these incoming/outgoing copies will be able to fire. The guard on the outgoing copies are optional as they will always be evaluated as true.

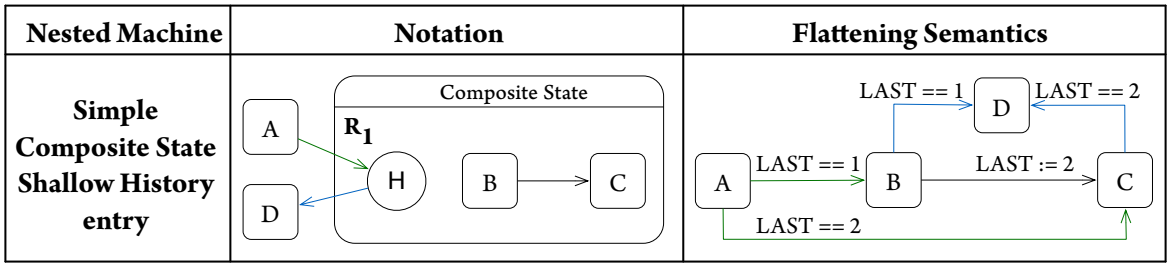



**Figure 3.5.5:** Shallow History Entry Flattening Semantics for Simple Composite State

- **Deep history entry** *The rule here is the same as for shallow history except that the rule is applied recursively to all levels in the active state configuration below this one.* (page 580 of UML Spec.)

  A deep history is like the shallow history with an extended behavior that can remember any level of nesting of the composite states. However, as the flattening process will run in a bottom-up way, it means that for each deep history, all its sibling substates have already been flattened, which makes it semantically and structurally equal to a shallow history. The only difference is that when doing the transition copy, shallow history can only cover the flattened substates originally at the same level, while deep history can cover all of them without restriction.

  The values of variable *LAST* is from 1 to n, where *n* is the sum of *direct substate* numbers in the enclosing composite states configured by the deep history pseudostate.

- **Entry point entry** *If a transition enters a composite state through an entry point pseudostate, then the entry behavior is executed before the action associated with the internal transition emanating from the entry point.* (page 580 of UML Spec.)

  The flattening semantics is defined as Fig. 3.5.6. For a transition entering a composite state, no matter if it directly links to the composite state or bypasses through entry point, the *entry action* of the composite state must always be executed. Compared to the direct connection, the extra semantics introduced by the entry point is the trigger on the outgoing transition of the entry point. To keep

this trigger, we define a special state $S_{entry}$ to represent the entry point. $S_{entry}$ has an entry behavior (the entry behavior of the composite state), but no do or exit behaviors.

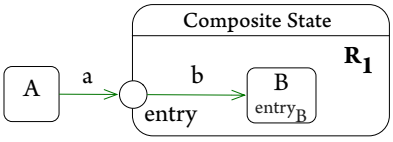
| Nested Machine | Notation | Flattening Semantics |
|---|---|---|
| **Simple Composite State Entry point entry** |  |  |

**Figure 3.5.6:** Entry Point Entry Flattening Semantics for Simple Composite State

### Exiting a non-orthogonal composite state

*When exiting from a composite state, the active substate is exited recursively. This means that the exit activities are executed in sequence starting with the innermost active state in the current state configuration.* (page 581 of UML Spec.) According to the UML specification, the following exiting cases are differentiated:

- **Default exit** As the UML specification does not define default exit rule, by default it is considered a well-formed model without explicit exiting notation. The default exit semantics is defined as Fig. 3.5.7. The outgoing transition of the composite state is copied as the outgoing transition of each inner substates except the initial pseudostate.

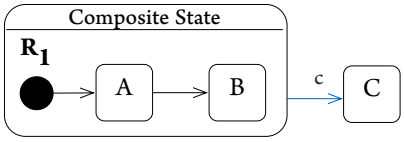
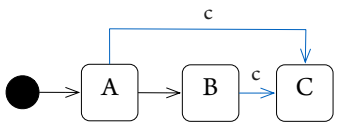
| Nested Machine | Notation | Flattening Semantics |
|---|---|---|
| **Simple Composite State Default Exit** |  |  |

**Figure 3.5.7:** Default Exit Flattening Semantics for Simple Composite State

- **Explicit exit** If the transition goes to the outer state of the composite state, then that outer state becomes active and the exit code of the substate is executed before the execution of the exit code of the composite state. The flattening semantics is defined as Fig. 3.5.8.

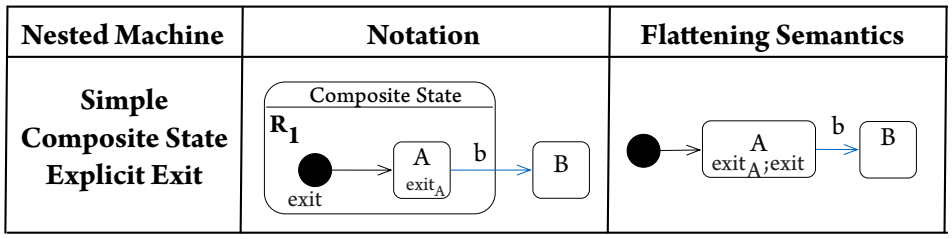



**Figure 3.5.8:** Explicit Exit Flattening Semantics for Simple Composite State

- **Shallow history & deep history exit** The shallow history and deep history exit have been defined in the entry parts.

- **Exit point exit** *If, in a composite state, the exit occurs through an exit point pseudostate the exit behavior of the state is executed after the behavior associated with the transition incoming to the exit point.* (page 581 of UML Spec.) *An exit point pseudostate is an exit point of a state machine or composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state.* (page 567 of UML Spec.) The flattening semantics is defined as Fig. 3.5.9.

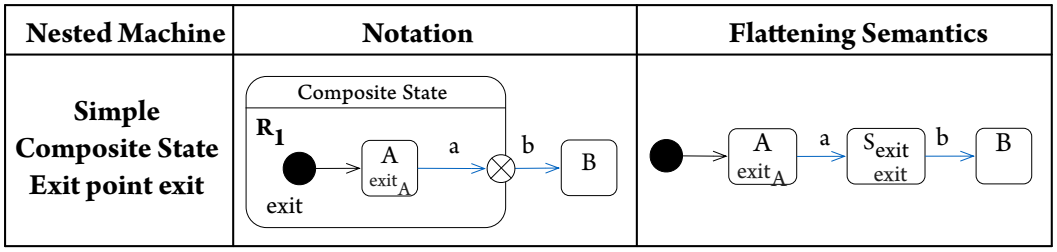



**Figure 3.5.9:** Exit Point Exit Flattening Semantics for Simple Composite State

Entry point and Exit point are dual semantic elements to describe compositional event handling border. Similar to entry pseudostate, a special state $S_{exit}$ is defined to represent the exit point. $S_{exit}$ has an exit behavior (the exit behavior of the composite state), but no entry or do behaviors.

- **Final state exit** *A special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed,*

*then it means that the entire state machine is completed.* (page 557 of UML Spec.)

The flattening semantics is defined as Fig. 3.5.10. To keep the trigger on the incoming transition of the final state, a special state $S_{final}$ is defined to represent it. $S_{final}$ has an exit behavior (the exit behavior of the composite state), but no entry or do behaviors.

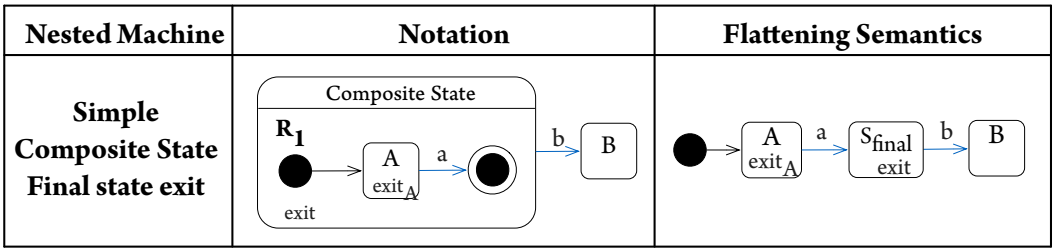



**Figure 3.5.10:** Final State Exit Flattening Semantics for Simple Composite State

**Flattening semantics for the actions and guards on transitions** In the above flattening semantics, if there are actions defined on the outgoing transition of inner exit state, they will be sequentially combined to the target flattened transition. The guards will also be integrated by conjunction.

### 3.5.3.2 Orthogonal Composite State

Regions address the modeling of concurrency. The word *orthogonal* implies that each region in the composite state is executed concurrently. When several sub-systems are executed concurrently, the number of state in the whole system is the product of the number of state in each concurrent sub-system. This leads to a combinatorial increase in the number of state of the associated state machine. This feature can be considered as a redundant semantic modeling element taking the idea of `Part` from the composite structure diagram, except that its scope is inside the state machine. Although the orthogonal regions do not add any semantic expressiveness to classic state diagrams, its flattening semantics requires some details.

We propose a flattening semantics for orthogonal composite states using Fork & Join pseudostates. According to the UML specification, the fork and join pseudostates can only be used in orthogonal regions. For the flattening purpose, we need to allow them to be use in the non-orthogonal structure. This adaption does not change the behavdonk semantics of the state machine.

**Entering an orthogonal composite state** *Whenever an orthogonal composite state is entered, each one of its orthogonal regions is also entered, either by default or explicitly. If the transition terminates on the edge of the*

*composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork), these regions are entered explicitly and the others by default.* (page 581 of UML Spec.)

We provide the flattening semantics for the default and explicit entries.

- **Default entry** By default, each concurrent region starts executing from the initial pseudostates. Instead of linking the incoming transition to the outgoing states of the initial pseudostate, a fork pseudostate is created as a delegate to maintain the concurrent semantics. The flattening semantics is defined as Fig. 3.5.11, where the outgoing transition of state *A* links to the fork pseudostate. Completion transitions are created, called anonymous transitions. They have no defined event triggering them. This means that such transition will immediately fire when a state being the source of a completion transition becomes active. These anonymous transitions link the fork nodes to the target substates of the initial nodes. All the entry/do/exit behaviors of substates and sub transitions are kept as-is.

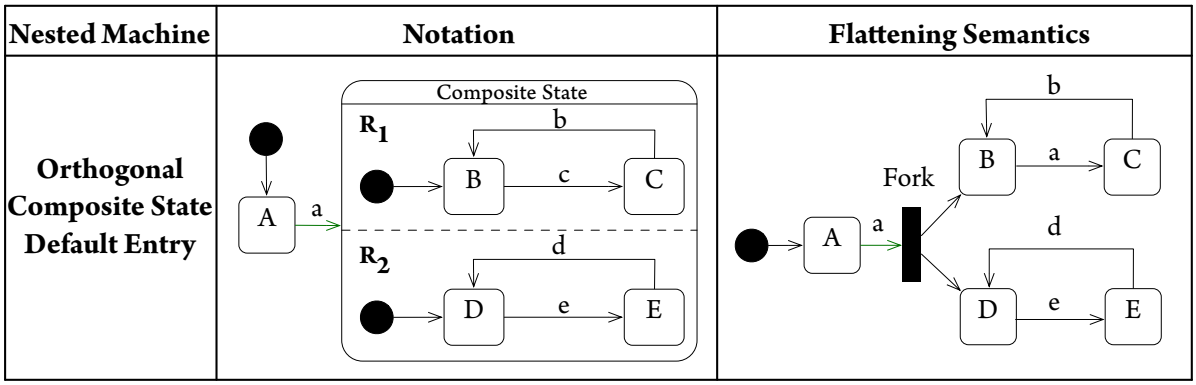



**Figure 3.5.11:** Default Entry Flattening Semantics for Orthogonal Composite State

- **Explicit entry** If the transitions goes to substates in one or more regions (in case of a fork), this explicit entry is defined as Fig. 3.5.12, where the fork node links the explicit substates through the anonymous transitions.

**Exiting an orthogonal composite state** *When exiting from an orthogonal state, each of its regions is exited. After that, the exit activities of the state are executed.* (page 581 of UML Spec.)

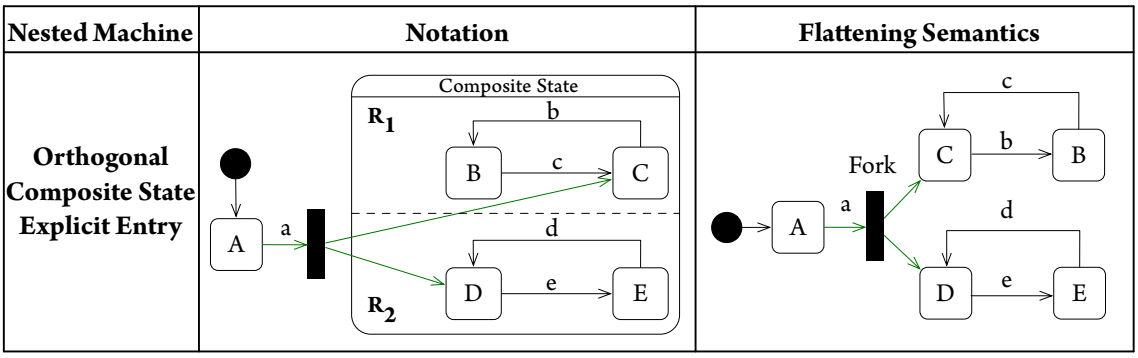



**Figure 3.5.12:** Explicit Entry Flattening Semantics for Orthogonal Composite State

Three kinds of exit semantics are provided: default, explicit and final state exits. We only handle the exit semantics here, while the entry semantics has been handled in the previous part.

- **Default exit** The flattening semantics for default exit is defined as Fig. 3.5.13.

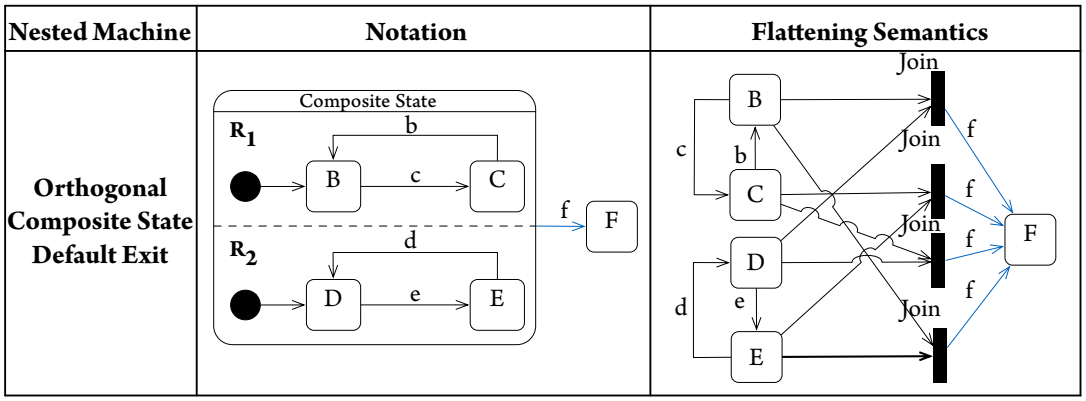



**Figure 3.5.13:** Default Exit Flattening Semantics for Orthogonal Composite State

For the outgoing transitions, a Cartesian product of join pseudostates is created, using the join node's outgoing transition to hold the original semantics of the composite state's outgoing transition (blue ones). At any time when the composite state is active, only one of these composed join node will be enabled to respond to the outgoing trigger.

- **Explicit exit** The flattening semantics for the explicit exit is defined as Fig. 3.5.14. Only the explicit

substates in a region are combined with the substates in non-explicit regions and then linked to the join nodes.

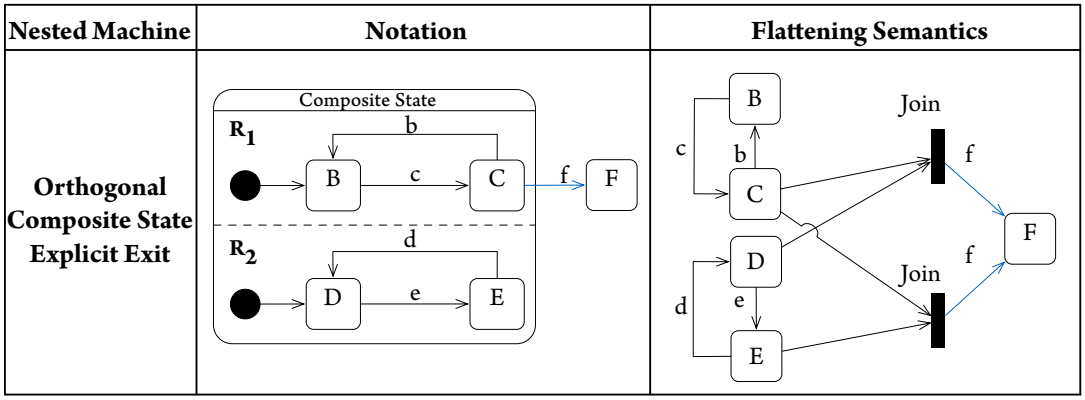



**Figure 3.5.14:** Explicit Exit Flattening Semantics for Orthogonal Composite State

- **Final state exit** The flattening semantics for the final state exit is defined as Fig. 3.5.15. A special state $S_{final}$ is created for the final state. The combination of $S_{final}$ and the substates in other regions are linked to the join nodes.

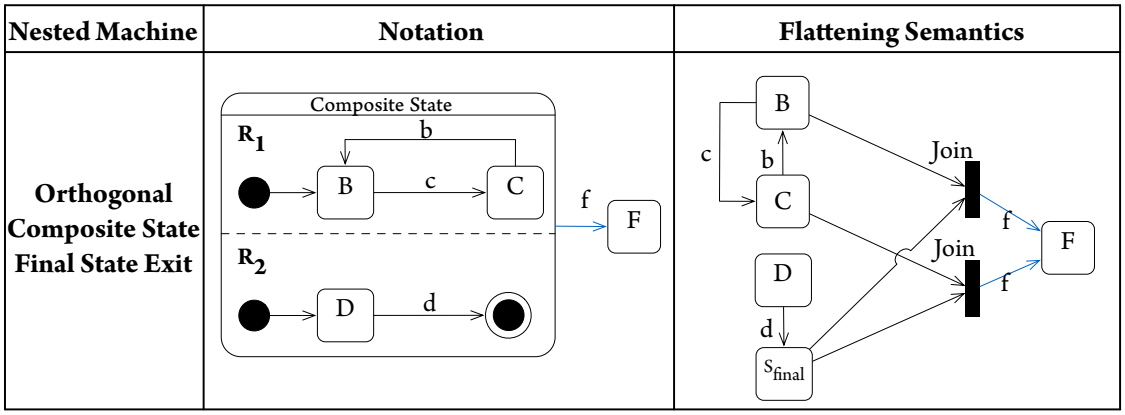



**Figure 3.5.15:** Final State Exit Flattening Semantics for Orthogonal Composite State

#### 3.5.3.3 Submachine State

*A submachine state is semantically equivalent to a composite state. The regions of the submachine state machine are the regions of the composite state. The entry, exit, and behavior actions and internal transitions are defined as part of the state. Submachine state is a decomposition mechanism that allows factoring of common behaviors and their reuse.* (page 576 of UML Spec.)

The only semantic difference, in terms of RTC, is when a submachine state is nested, whether it is used *in behavioral state machine* or *in protocol state machine*.

**In behavioral state machine (as integrated)** At each reuse, the submachine state structure is *copied* to the nested structure. Therefore it is in fact a part of its root state machine, which means it must respect the same run-to-completion processing as the other parts. In this case, a submachine state can share the same flattening semantics for composite state.

**In protocol state machine (as communicated)** *The states of protocol state machines are exposed to the users of their context classifiers. A protocol state represents an exposed stable situation of its context classifier: When an instance of the classifier is not processing any operation, users of this instance can always know its state configuration.* (page 577 of UML Spec.)

In this scenario, each time the submachine state is entered, a new instance will be implicitly created to handle the event coming afterward. Therefore the given submachine state will have an independent run-to-completion scope. Its inner events can be handled concurrently with those at root state machine level.

In the context of this thesis, we only focus on behavioral state machines, thus the second case will not be discussed. By default, we rely on the submachine states in the behavioral state machine (as integrated) for the semantic mapping afterwards.

#### 3.5.3.4 Fork & Join Pseudostates

*Join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers.* (page 567 of UML Spec.)

*Fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers.* (page 567 of UML Spec.)

Fork pseudostate models the execution of concurrent aspects in transitions. The incoming transition

is split into two or more transitions terminating on orthogonal target vertex. Join pseudostate is the dual element of fork. Because these two elements are always used within orthogonal regions, the way orthogonal regions are flattened will impact the meaning of fork & join.

We have discussed the use of fork nodes in the explicit exit for orthogonal composite state (Fig. 3.5.14). For the region flattening algorithm using fork and join pseudostates, as the concurrent execution and synchronization are provided by the fork and join nodes respectively, the regions can be removed without jeopardizing the original semantics (Fig. 3.5.16).

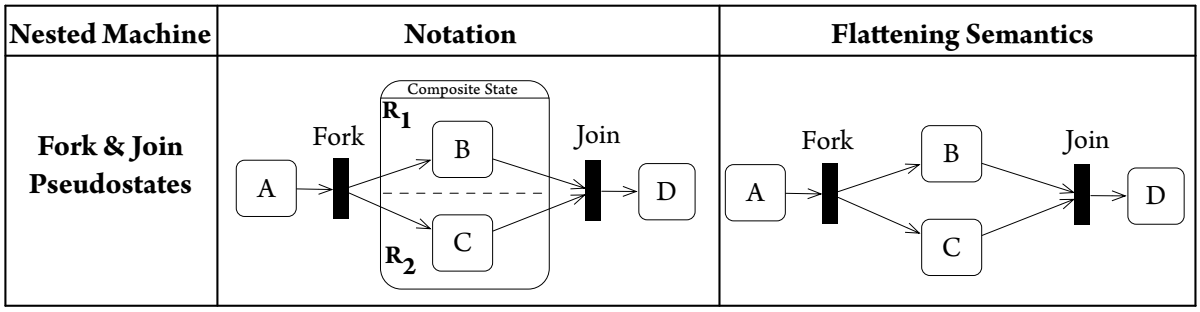



**Figure 3.5.16:** Fork & Join Pseudostate Flattening Semantics

### 3.5.4 Mapping Semantics

After flattening, the remaining SMD elements for which the mapping must be defined are: *State (simple state), Final state, Transition (local and internal), and Pseudostates (Initial, Terminate, Junction, adapted Choice, Fork & Join without regions)* .

The objective is to map the unnested SMD to a TPN, which formally defines its execution semantics. The relatively complicated semantics for the unnested SMD is the transitions and states involving inner behaviors such as effect, exit, entry and do. On the other hand, the run-to-completion (RTC) processing must be under consideration.

We first present some general semantics for the transitions and states in Section 3.5.4.1. The mapping semantics for the RTC semantics and the inner behaviors are respectively provided in Section 3.5.4.2 and Section 3.5.4.3. We define the mapping semantics for states and transitions in Section 3.5.4.4, and discuss the clearance mechanisms for the event pool in Section 3.5.4.5. As a special kind of simple state, the map-

ping semantics for final states is provided in Section 3.5.4.6. At last, the mapping semantics for pseudostates is provided in Section 3.5.4.7.

### 3.5.4.1 Transition & State in General

*A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.* (page 597 of UML Spec.) A transition can be associated with several triggers (the triggers may fire the transition), at most one guard and at most one effect behavior. From the viewpoint of the target and source states, there exist three kinds of transitions: external, internal and local transitions:

- ***Internal*** *implies that the transition, if triggered, occurs without exiting or entering the source state. Thus, it does not cause a state change.* (page 606 of UML Spec.)
- ***Local*** *implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite state, and these will be exited and entered.* (page 606 of UML Spec.)
- ***External*** *implies that the transition, if triggered, will exit the source vertex.* (page 606 of UML Spec.)

**Transitions and states for mapping** In the flattening step, all the external transitions have been flattened to local transitions. Therefore, we only need to provide mapping semantics for the local and internal transitions. The flattening semantics for composite states and submachine states has been defined in the previous section, thus there exists only simple states in the unnested SMD.

### 3.5.4.2 Run-to-Completion (RTC) Semantics

**Run-to-completion processing** *The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.* (page 590 of UML Spec.)

**Run-to-completion step** *Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed*

*while the state machine is in some intermediate and inconsistent situation. The run-to-completion step is the passage between two state configurations of the state machine.* (page 590 of UML Spec.)

We use the Fig. 3.5.17 to illustrate the mapping for the RTC semantics. In order to explain the execution sequence, the UML notation only has two states A, B, and the transition between them. In state A, the do and exit activities are defined. In state B, the entry and do activities are defined. The transition has a trigger, a single guard and an effect action.

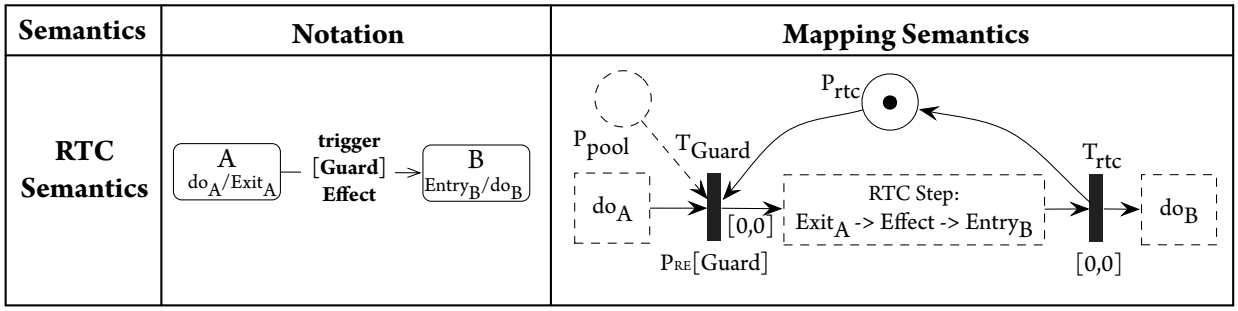



**Figure 3.5.17:** Run-to-Completion Semantics

The execution sequence which ensures the RTC step is described as follows:

1. The transition is triggered by the event arrival. At this moment, the event instance has not yet been consumed.

2. The guard constraint on the SMD transition is evaluated. If it is satisfied, the event instance is consumed, otherwise the event instance will stay in the event pool and the SMD transition will not fire.

3. When the transition matches both preconditions to fire, the following actions will successively execute:

   (a) the exit behavior of the source state A

   (b) the effect action defined on the SMD transition

   (c) the entry behavior of the target state B

The mapping semantics for the RTC semantics contains both the RTC processing and RTC step. The squares in dotted line stand for respectively the behaviors of $do_A$, RTC *step*, and $do_B$. The SMD transition

$T$ is mapped to a pair of TPN transitions ($T_{guard}$ and $T_{rtc}$) with intermediate behaviors RTC *step* and a RTC place $P_{rtc}$. The place $P_{rtc}$ has an initial token. It is used to indicate whether the RTC processing is finished. The $T_{Guard}$ transition inherits the guard constraint from $T$ using the PRE functions of tts. If there is a single trigger on $T$, the place $P_{pool}$ is used as the event pool for the target type of event occurrence. If the event arrives but the guard is not satisfied, the event occurrence will not be dispatched, thus stays in the event pool. According to the semantics of RTC step, after the trigger and the guard of transitions are both satisfied, the behaviors of $Exit_A$, *Effect* and $Entry_B$ must be executed sequentially without interruption. When $P_{rtc}$ has a token, it means the RTC processing is finished. Meanwhile, the $do_B$ behavior can be executed.

#### 3.5.4.3 Do/Exit/Entry/Effect Behaviors Mapping

The optional entry/do/exit/effect behaviors in a state or on a transition need to be mapped to TPN. In the context of this thesis, the inner behavior can be an activity diagram or an unnested state machine or an action language expression (C in our prototype). We use the mapping semantics in Fig. 3.5.18 for these behaviors.

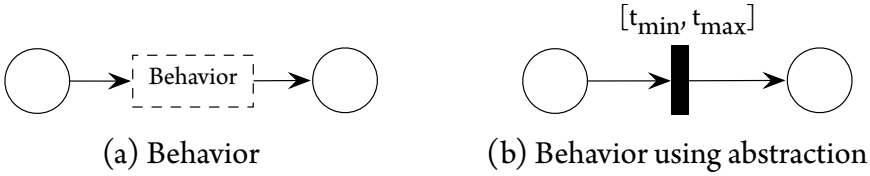



**Figure 3.5.18:** Do/Exit/Entry/Effect Behavior

In figure (a), the frame behavior stands for the activity of entry/do/exit/effect. If the inner behavior will not impact or be impacted by the other parts of the whole system, this behavior can be abstracted using a TPN transition, which gives the minimum and maximum execution time of this behavior, as shown in figure (b). To generalize our discussion, we use the figure (a) as the mapping semantics for the inner behaviors.

**Initialization of Inner Behaviors** The initialization of the do/exit/entry/effect inner behaviors is intuitive, which means it starts from the initial node in the activity or state machine.

**Termination of Inner Behaviors** As the entry/exit/effect behaviors are atomic and cannot be interrupted by the firing of transitions or by the other external behaviors, they will complete the behaviors and produce

a completion event. In the activity diagram, either when all the flows reach the Flow Final nodes, or when the whole activity reaches the Activity Final node, then the completion event will be produced. In the unnested state machine, either when all the Final states are reached, or when the Termination pseudostate is reached, then the completion event will be produced.

However, the do behavior is more complex. *The behavior represents the execution of a behavior, that occurs while the state machine is in the corresponding state. The behavior starts executing upon entering the state, following the entry behavior. If the behavior completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition the state will be exited. Upon exit, the behavior is terminated before the exit behavior is executed. If the state is exited as a result of the firing of an outgoing transition before the completion of the behavior, the behavior is aborted prior to its completion.* (page 579 of UML Spec.)

Therefore, the termination of do behavior must satisfy two semantics:

- If the behavior completes while the state is still active, it raises a completion event.
- If the state is exited as a result of the firing of an outgoing transition before the completion of the behavior, the behavior is aborted prior to its completion.

The first semantics is similar to the termination semantics for the other behaviors. We focus on the second semantics. To model this interruption semantics in TPN, as we do not know when the do activity is interrupted by the firing of the outgoing transitions, all possible behavior should be modeled. A stopwatch arc can be used for this purpose, but this will potentially lead to the state space explosion problem in the model checking. On the other hand, in the context of this thesis, as we focus on real-time embedded systems, it is reasonable to forbid this arbitrary interruption. All the behaviors in the critical systems mus be explicitly specified. If the do activity can be interrupted, the interruption point must be predefined in the specification. From this point of view, in the context of this thesis, we only adopt the first semantics, which means the outgoing transitions can be fired only if the do activity is completed.

#### 3.5.4.4 Mapping semantics for Transition & State

This section provides the mapping semantics for local/internal transitions and simple states, and then discusses the mapping semantics for single and multiple triggers on the transitions.

**Local Transition.** The mapping semantics for the local transition and its associated states is illustrated by Fig. 3.5.19. In the SMD model, state A contains $Do_A$ and $Exit_A$ activities, and state B contains $Entry_B$ and

$Do_B$ activities. The transition from A to B contains at most one trigger (multiple triggers will be discussed later), a guard and an effect action.

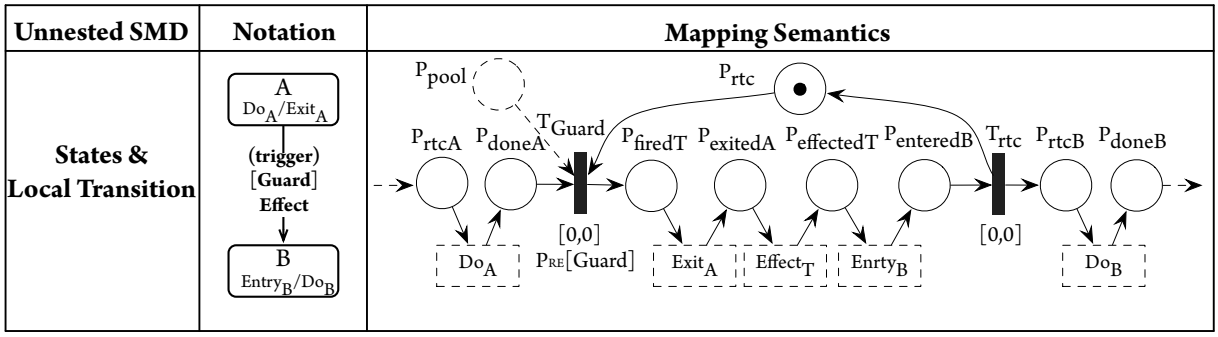



**Figure 3.5.19:** Local Transition Mapping Semantics

In the mapping semantics, the chain [place $P_{rtcA}$ -> behavior $Do_A$ -> place $P_{doneA}$] models that state A has completed its do activity. The SMD transition $T$ is mapped to a pair of TPN transitions ($T_{guard}$ and $T_{rtc}$) with intermediate places and behaviors. The place $P_{rtc}$ has an initial token. It is used to indicate whether the RTC processing is finished. When $P_{rtc}$ has a token, it means the RTC processing is finished. The $T_{Guard}$ transition inherits the guard constraint from $T$ using the PRE functions of tts. If there is a single trigger on $T$, the place $P_{pool}$ is used as the event pool for the target type of event occurrence. If the event arrives but the guard is not satisfied, the event occurrence will not be dispatched, thus stays in the event pool. When $T$ is fired (represented by the place $P_{firedT}$), the activities $Exit_A$, $Effect_T$ and $Entry_B$ are executed. After entering state B (represented by the place $P_{enteredB}$), the transition $T_{rtc}$ produces a RTC event through the place $P_{rtc}$. Meanwhile, the $Do_B$ behavior can be executed.

We provide a mapping semantics for the local transition with abstraction in Fig. 3.5.20. The $Do_A$ behavior is abstracted as the transition $T_{doA}$ with minimum and maximum execution time [t1, t2]. Similarly, the $Exit_A$, $Effect_T$, $Entry_B$ activities are abstracted together using the transition $T_{rtc}$, and the $Do_B$ activity is abstracted using the transition $T_{doB}$.

**Internal Transition.** An internal transition can be seen as a special kind of local transition without entry and exit behaviors, and the source and target states are the same one. Fig. 3.5.21 illustrates its mapping

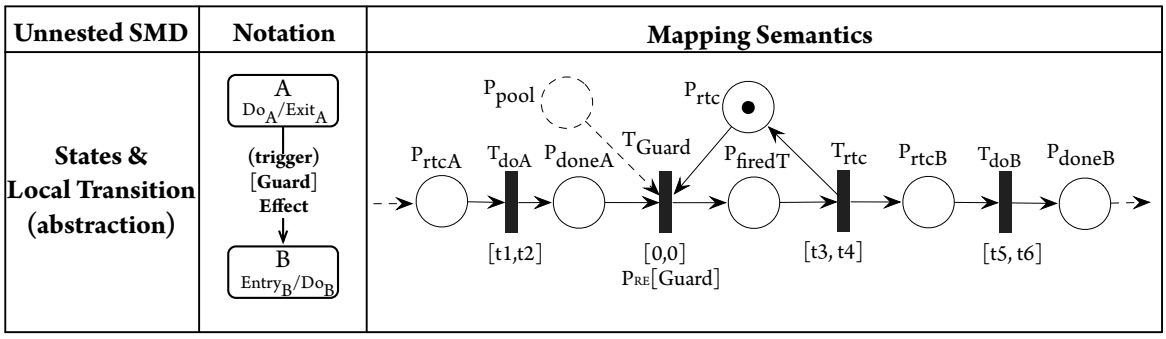



**Figure 3.5.20:** Abstract Local Transition Mapping Semantics

semantics. According to its semantics, the exit and entry activities are eliminated, and the transition $T_{rtc}$ is linked to the place $P_{rtc}$ of itself.

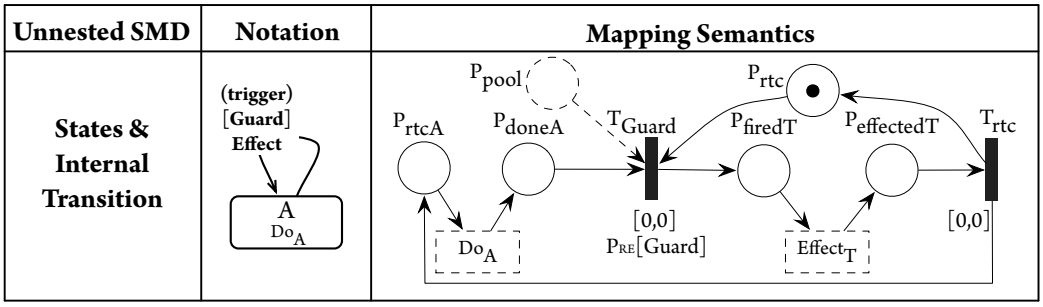



**Figure 3.5.21:** Internal Transition Mapping Semantics

**Single Trigger Transition.** The mapping semantics for the single trigger transition is illustrated by Fig. 3.5.22. The event pool place $P_{pool}$ receives instances of event $a$ from all the producers, and then provides to all the consumers.

**Multiple Trigger Transition.** We have discussed the processing semantics for multiple triggers transition at the beginning of this section (see the answer for $Q_6$ in page 88): **Only one transition can be fired when two transitions originating from the same states are conflict.** If a transition has several possible triggers to enable it, they are under an exclusive "or" logic, which is represented by duplicating the SMD transition.

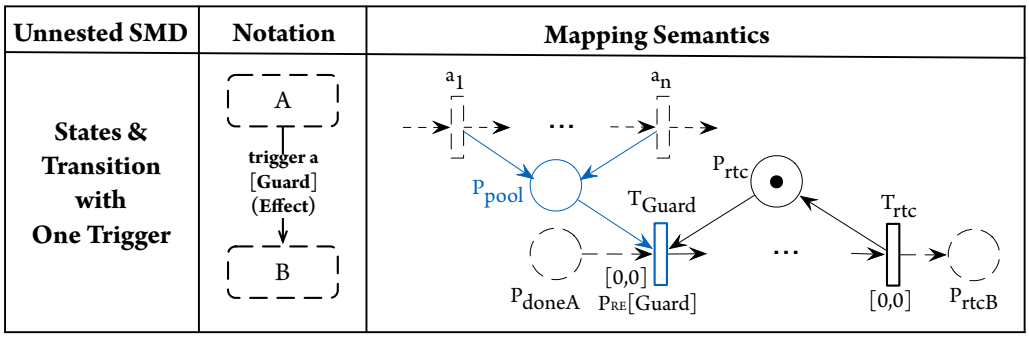



**Figure 3.5.22:** Single Trigger Transition Mapping Semantics

The copied transitions will inherit the same guard and behaviors while keeping respectively the single event trigger. The mapping semantics for the single trigger transition is illustrated by Fig. 3.5.23. In the TPN, the duplicated SMD transitions are mapped to two guard transitions $T_{guard\_a}$ and $T_{guard\_b}$. As the event pool is instantiated by event type, in this case, two event pool places $P_{pool\_a}$ and $P_{pool\_b}$ are created to receive respectively the instances of events *a* and *b*. More precisely, if both events *a* and *b* are available at the moment to trigger the transition, as only one event can be finally consumed after transition's firing (ensured by the shared place $P_{doneA}$), it is up to the event dispatch mechanism to decide the priority. The UML specification does not give any details about this priority definition, therefore in the context of this thesis, it is assumed that the dispatch is arbitrary.

#### 3.5.4.5 Event Pool Clearance Mechanisms

When explicit events are introduced, the impact of event pool must be considered. The pool, instantiated by event type, is represented by a single empty place for the whole system, not for each state machine instance. (This has been discussed in the section 3.5.1.3) This place, with a global visibility of a given event type, on the one hand can consolidate all the emission of the system, and on the other hand can dispatch event instances with competition mechanism for all event consumers. We provide the mapping semantics for the single trigger and the multiple trigger transitions.

As mentioned at the beginning of the section, once an event pool concept is introduced, the clearance mechanism must be defined. Otherwise for those events which will arrive always at inappropriate time, the pool would keep growing and produce an overflow, especially for those systems which are designed to run

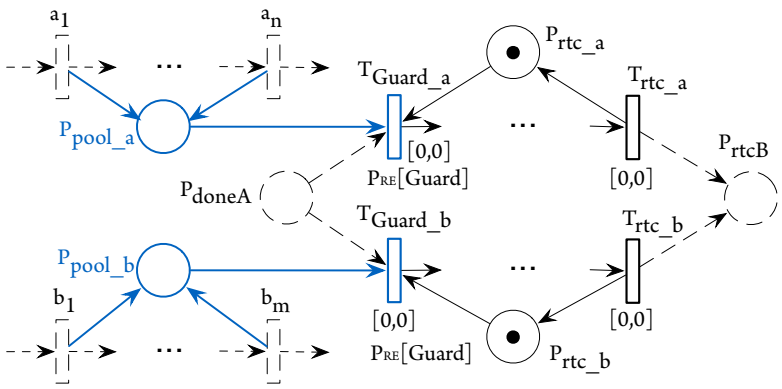
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **States & Transition with Multi Trigger** | A<br>**trigger a,b**<br>**[Guard]**<br>**(Effect)**<br>B |  |

**Figure 3.5.23:** Multiple Trigger Mapping Semantics

infinitely. In the context of this thesis, two generic strategies are proposed and implemented: *time out* and *size out* strategies (Fig. 3.5.24):

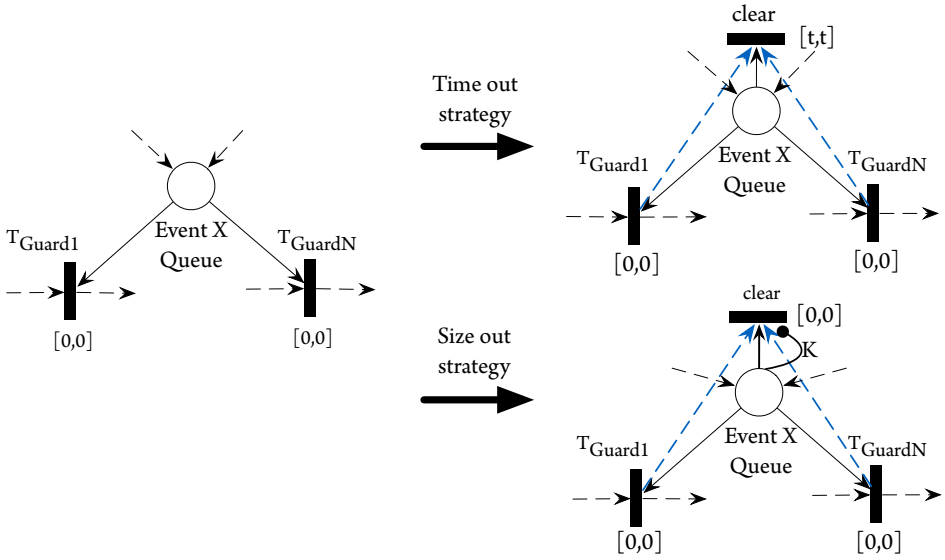



**Figure 3.5.24:** Mapping Semantics: Event Pool Clearance Mechanism

- **Time Out Strategy**: This solution (Fig. 3.5.24, right top) adds an outgoing TPN transition to the event pool place, with execution time [**t,t**] in which **t** is the maximum time that an event instance could stay in the pool if not consumed. The new TPN transition **clear** will compete event with real event consumers, which means if an event can be handled at a given time, it is not guaranteed to be handled because the event pool may be cleared before. This violates the original semantics and therefore need constraints to forbid the **clear** to compete events with normal $\mathbf{T_{Guard}}$ transitions.

  TINA tool box offers the analysis of TPN with priorities, which is a kind of constraints between transitions, called transition priority. The blue arrows in Fig. 3.5.24 stand for this: the source transition ($\mathbf{T_{Guard_1}}$, ..., $\mathbf{T_{GuardN}}$) will always be prior to the target transition (**clear**) if both are enabled. Thus the event pool will only clear the time out event when there is no state machine at the ready state to receive it.

- **Size Out Strategy**: This solution removes event instances from the event pool when it reaches its maximal capacity (see Fig. 3.5.24, right bottom). This strategy should provide a detailed dequeue policy, like FIFO, LRU, etc. However, since for different event instance, their use is always the same in terms of triggering the corresponding state machines. Therefore the only criteria that matters (which can influence system's behavior) is the event instance number in the pool.

  Like time out strategy, this solution adds an outgoing **clear** transition to the event pool place, but with a different execution time [**0,0**]. This implies that once the pool is full, the clearance work will start immediately. Of course it will encounter the same problem of token competition if no transition priority is defined. The control of pool's capacity is implemented by a read arc from pool place to **clear**, using **K** as the capacity parameter.

- **Without Clearance Mechanisms**: The time out and size out strategy in TPN introduce priority arcs. TINA supports different abstraction used for building state class graph. A TPN with priority arc will be unfolded using state preservation abstractions. Without priority arcs, a marking preservation abstraction is possible. The marking preservation abstraction is the highest abstraction, which makes the model checking more efficient. If the priority arcs are supposed to be avoided, the clearance mechanism can be replaced by on-the-fly checking to allow detecting potential overflows. This method simply observes the arrived event amount in the event pool. If the amount is out of bound, the on-the-fly checking stops. It indicates that the system design itself possibly has some vulnerabil-

ity in the interaction with the environment.

In the context of this thesis, we do not use any clearance mechanisms for the event pools. Instead, in the real-time critical systems, we take as granted the event instances of the given system are always restricted to the capacity of event pool, otherwise, this is a design error.

#### 3.5.4.6 Final State

*Final state is a special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed.* (page 557 of UML Spec.)

In the flattening step, the final states in regions have been flattened and replaced by a special state $S_{final}$ with triggers but no entry or do activity (see pages 3.5.3.1 and 3.5.3.2). The final states in the topmost region are kept. These final states are mapped to a TPN place without initial token or outgoing transitions, as shown in Fig. 3.5.25.

| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| Final State |  |  |

**Figure 3.5.25:** Final State Mapping Semantics

#### 3.5.4.7 Pseudostates

We discuss the mapping semantics for the initial, terminate, junction, choice, fork and join pseudostates in this section. The mapping semantics for the initial and terminate pseudostates is simple. The mapping semantics for the fork and join pseudostates is not complex either, as they have the same behaviors as existing TPN elements. The mapping semantics for the junction and choice pseudostates needs to be detailed, as the RTC processing is relatively complicated for the compound transitions constructed with them. The definition of a common semantics is a key point of the semantic mapping: *Transitions outgoing pseudostates may not have a trigger (except for those coming out of the initial pseudostate).* (page 598 of UML Spec.) This point is important when mapping the junction, choice, fork and join nodes.

**Initial.** *An initial pseudostate represents a default vertex that is the source for a single transition to the default*

*state of a composite state. There can be at most one initial vertex in a region.* (page 566 of UML Spec.)

An initial pseudostate activates the state machine instance at the beginning. It is represented by a place with one token and no outgoing transitions. *The outgoing transition from the initial vertex may have a behavior, but not a trigger or guard* (page 566 of UML Spec.). As the outgoing transition from the initial vertex is specific, we provide the mapping semantics for the initial pseudostate and its outgoing transition in Fig. 3.5.26.

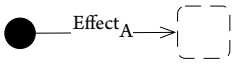
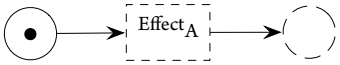
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **Initial & Outgoing Transition** |   |   |

**Figure 3.5.26:** Initial Pseudostate and Outgoing Transition Mapping Semantics

**Terminate.** *Entering a terminate pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate.* (page 567 of UML Spec.)

The terminate pseudostate is similar to the activity final node in the activity diagram. It is represented by a place without initial token. In order to stop all the executions in the state machine, the inhibitor arcs are used to link all the TPN transitions, as shown in Fig. 3.5.27. Once the terminate place is filled with token, the inhibitor arcs will halt all the transitions.

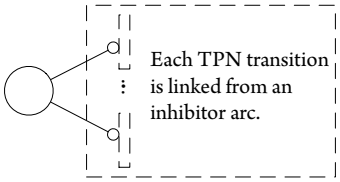
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **Terminate Pseudostate** |  |   |

**Figure 3.5.27:** Terminate Pseudostate Mapping Semantics

**Junction & Choice.** Junction and choice pseudostates are both used to chain multiple transitions. Junction vertices are used to construct static conditional branches while choice vertices are used to construct

dynamic conditional branches. The distinction between these two conditional branches is reflected in the RTC processing.

*In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined. If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above). In other words, for guard evaluation, a choice point has the same effect as a state.* (page 600 of UML Spec.)

Therefore, if there are choice points in a compound transition, guards downstream of a choice point are evaluated if and when the choice point is reached. If there are junction points in a compound transition, both guards that precede a junction point and the guards downstream of the junction point are evaluated before the junction point is reached. The RTC processing means that *an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.* (page 590 of UML Spec.) This RTC semantics must be ensured during the semantic mapping.

*Choice vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a dynamic conditional branch. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run- to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined "else" guard for every choice vertex.)* (page 567 of UML Spec.)

The mapping semantics for the choice node is illustrated in Fig. 3.5.28. To simplify the discussion, we suppose there is no exit behavior in state A, and there is no entry behavior in states B and C. The choice node is mapped to the TPN place $P_{choice}$. The transition $T_{guardA}$ and the behavior $Effect_A$ stands for the behavior of the incoming transition of choice node. If the incoming transition has a single trigger, the place $P_{pool}$ is used to represent the event pool. Once the RTC step is finished, the RTC token refills both the places $P_{rtcB}$ and $P_{rtcC}$. This mapping semantics guarantees that the behavior of $Effect_A$ may impact the guards on the outgoing transitions. Meanwhile, it ensure the RTC processing.

*Junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a*

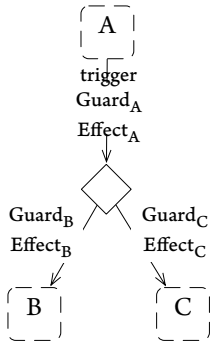
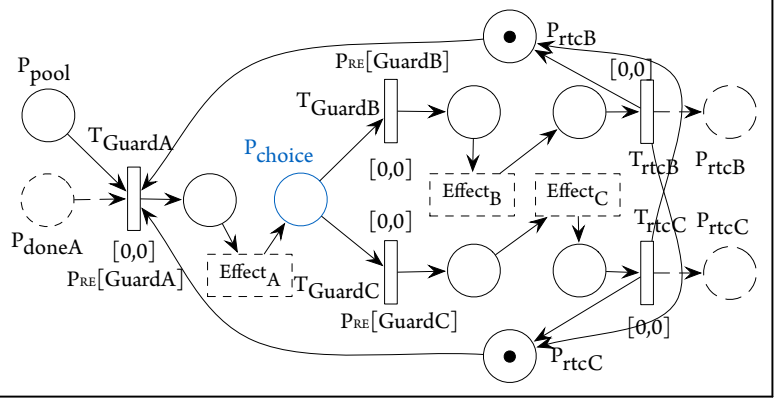
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **Choice Pseudostate** |  |  |



**Figure 3.5.28:** Choice Pseudostate Mapping Semantics

*merge). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted "else" may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.)* (page 566 of UML Spec.)

The mapping semantics for the junction node is illustrated by Fig. 3.5.29. As the guards are evaluated before the firing of incoming transitions, the guards on each conditional branch are combined. In the figure, $\text{Guard}_A$ and $\text{Guard}_B$ are combined in one branch, while $\text{Guard}_A$ and $\text{Guard}_C$ are combined in another branch. As the trigger is on the incoming transition, the event pool place $P_{poolA}$ provides events to both transitions $T_{GuardAB}$ and $T_{GuardAC}$. This mapping semantics guarantees that all guards are evaluated statically before the firing of transitions. Meanwhile, it ensures the RTC processing.

**Fork & Join.** *Fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers.* (page 567 of UML Spec.)

The mapping semantics for the fork node is illustrated by Fig. 3.5.30. To simplify the discussion, the exit behavior of state A and the entry behaviors of states B/C are omitted. The fork node is mapped to

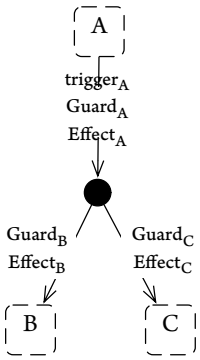
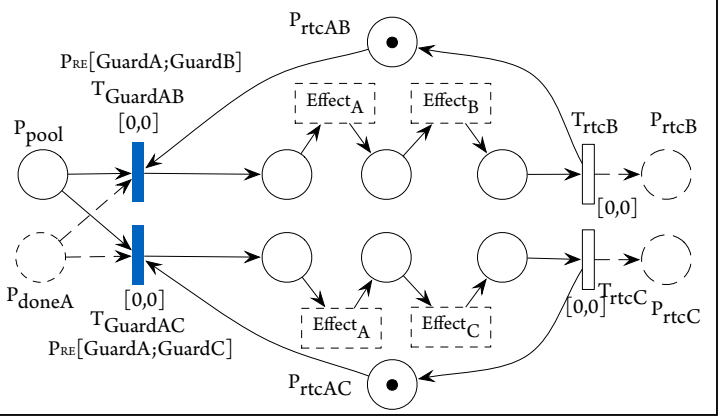
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **Junction Pseudostate** |  |  |



**Figure 3.5.29:** Junction Pseudostate Mapping Semantics

a TPN transition $T_{fork}$ (blue color) with time constraint [0,0]. The RTC processing needs to be explicitly mapped. The RTC is mapped to the places $P_{rtcB}$ and $P_{rtcC}$ which link transitions $T_{rtcB}$ and $T_{rtcC}$ to the common transition $T_{Guard}$ where the guard of incoming transition is specified using PRE[Guard]. This mapping semantics ensures that only when both outgoing transitions complete, will the compound transition finish the RTC processing.

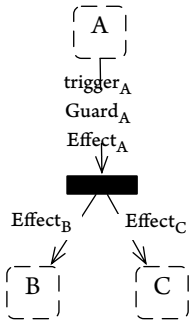
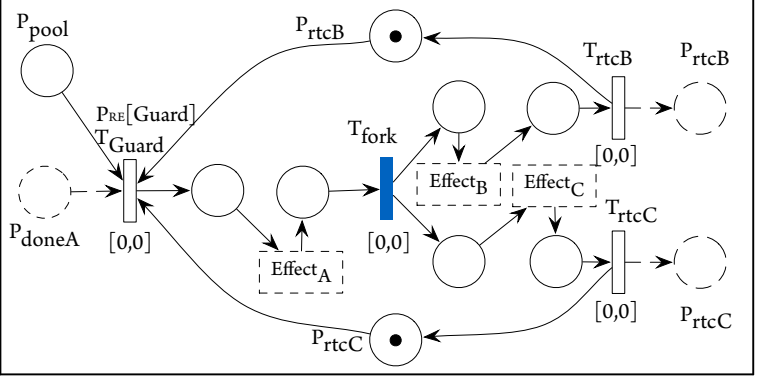
| Unnested SMD | Notation | Mapping Semantics |
|---|---|---|
| **Fork Pseudostate** |  |  |



**Figure 3.5.30:** Fork Pseudostate Mapping Semantics

*Join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions.*

*The transitions entering a join vertex cannot have guards or triggers.* (page 567 of UML Spec.)

The mapping semantics for the join node is illustrated by Fig. 3.5.31. To ease the discussion, the exit behaviors of state A/B and the entry behavior of states C are omitted. The join node is mapped to the transition $T_{GuardC}$, which is also the guard transition. As the incoming transitions cannot have triggers, the RTC place $P_{rtcC}$ links the transition $T_{rtcC}$ to the guard transition $T_{GuardC}$. As all the incoming and outgoing transition of join node do not have triggers, it is not necessary to maintain the RTC semantics here.

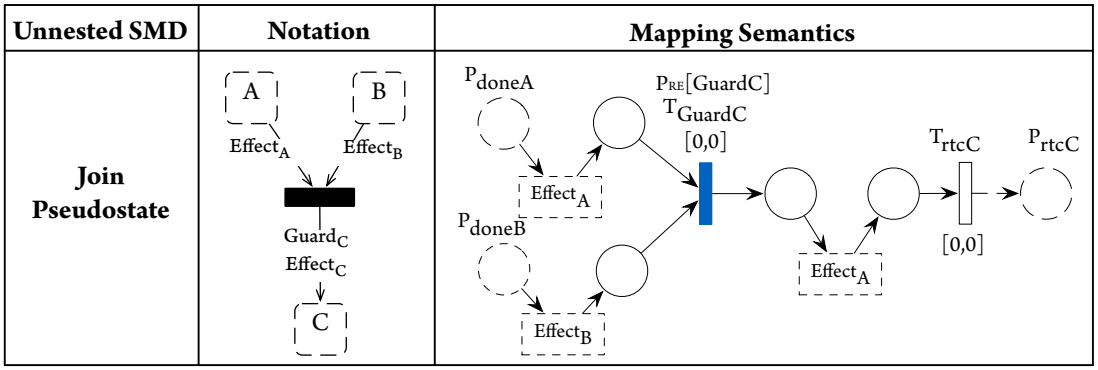



**Figure 3.5.31:** Join Pseudostate Mapping Semantics

## 3.6 Resource Mapping Semantics

In the UML-MARTE model, the behaviors (activity and state machine) consumes the resources such as the CPU, the memory, etc. The scheduling policy applied by the scheduler will impact the real-time requirements. Thus, if the target system relies on some external resources, the real-time behavior for the resources scheduling needs to be explicitly specified in the TPN model.

The MARTE profile `MARTE::MARTE_Foundations::GRM::Scheduler:schedPolicy` provides some typical scheduling policies for real-time embedded systems, such as Earliest Deadline First, FIFO, Fixed Priority, Least Laxity First, Round Robin, Time Table Driven. It also allows users to define their own scheduling policy. Mapping semantics for these well-known scheduling policies to TPN model could introduce some semantic ambiguities. For example, when using Fixed Priority scheduling policy, there is no explicit indication in the UML-MARTE level to specify what should be the scheduler's behavior if two requests have the

same priority; but as this information is mandatory for the TPN modeling, then a semantic gap is potentially created.

Besides, the exact behavior of some dynamic scheduling policy could not be mapped to TPN in a trivial way. For example the EDF/preemptive policy always need to compute for each reassignment cycle the process which is the closest to its deadline. This requires a dynamic comparison between the amount clock/time state of each transition and the given reference, which is unfeasible neither in classical TPN nor in TPN with data extension.

As modeling of scheduler policy is not the focus of this thesis, we do not aim in our work to provide the mapping semantics for any specific scheduling policy. Instead, we propose a generic scheduling algorithm with preemption option. This scheduling algorithm is used to decide for the given time $T$, which resource instance(s) will be allocated to which requester(s).

### 3.6.1 Generic Resource Scheduling

A resource is a 3-tuple $(I, S, Q)$, in which:

- $I$ refers to identification, which indicates the *type* of the resource.
- $S$ is the scheduler used to respond to the requirement of the resource. A scheduler has a preemption option.
- $Q$ is the instance amount of the provided resource.

For example, a 4-core CPU with preemptive scheduling policy is modeled as $(\textit{CPU-CORE, preemption, 4})$.

In the MARTE profile, the following properties are used to specify the resource, scheduler and allocations:

- **Preemption**: `MARTE::MARTE_Foundations::GRM::Scheduler:isPreemptible`
- **Scheduling policy**: `MARTE::MARTE_Foundations::GRM::Scheduler:schedPolicy`
- **Resource amount**: `MARTE::MARTE_Foundations::GRM::Resource:resMult`
- **Required amount**: `MARTE::MARTE_Foundations::GRM::Scheduler:resMult`
- **Allocations**: `MARTE::MARTE_Foundatins::Alloc::Allocate`

A generic resource scheduling semantic pattern is defined as shown in Fig. 3.6.1. The content of the square in dotted line is a behavior model consuming the target resource in a preemptive or non-preemptive manner. In this thesis, we only provide the mapping semantics for the resource scheduling in an activity diagram using event-trigger actions. The same mapping principle is applicable to the other kinds of behavior model (time-trigger actions in the activity and state machine).

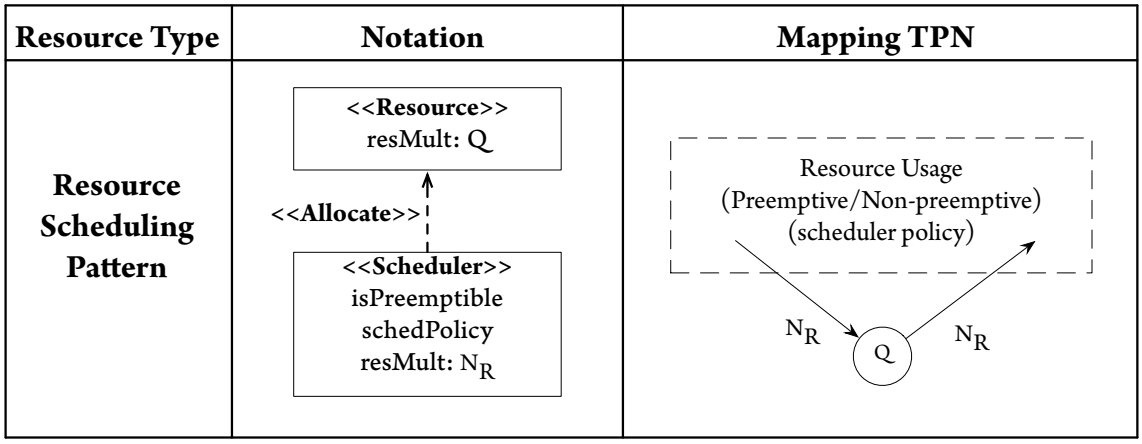



**Figure 3.6.1:** Generic Resource Scheduling Mapping Semantics

### 3.6.2 Non-preemptive Resource Scheduling

The mapping semantics for the non-preemptive resource scheduling in the activity diagram using an event-trigger action is illustrated by Fig. 3.6.2. The mapping semantics for the event-trigger action has been presented in page 75. The resource place $P_{res}$ contains $Q$ instances of a given type of resource. $P_{res}$ linked to the transition REQUIRE_RES represents the fact that the given action requires $N_R$ instances of resource, and $P_{res}$ linked from the transition RELEASE_RES represents the fact that the $N_R$ instances of resources should be released and returned to the resource place.

### 3.6.3 Preemptive Resource Scheduling

TPN with stopwatch are commonly used to cope with preemptive modeling. However, it is very expensive in terms of reachability graph generation when performing the model checking to assess real-time properties.

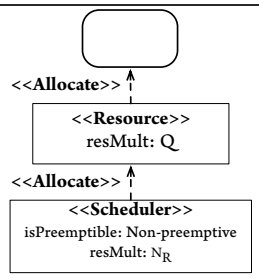
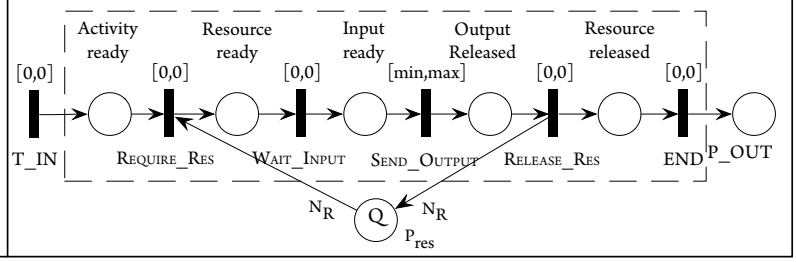
| Scheduling Type | Notation | Mapping TPN |
|---|---|---|
| **Action Node (event-tigger) Non-preemptive Resource Scheduling** |  |  |

**Figure 3.6.2:** Non-Preemptive Resource Scheduling Semantics

We propose a solution to mitigate this issue. The objective is to model the same semantics without using stopwatch mechanism. The idea is to use the time slice of the preemptive scheduler as the time unit to segregate the action's execution. The transition of execution (with time constraint $[t_{min}, t_{max}]$) is divided into the structure presented in Fig. 3.6.3. The resource place $P_{res}$ containing $Q$ instances connects to each transition (the two direction arrows in blue color) to represent the preemptive scheduling, where

- $t_s$ is the time slice of the scheduler;
- $K = \lfloor t_{min}/t_s \rfloor$ is the minimal number of occurrence times of $t_s$;
- $S = \lfloor t_{max}/t_s \rfloor$ is the maximal number of occurrence times of $t_s$;
- $A = t_{min} - K \cdot t_s$ stands for the left time from $t_{min}$ after $K$ occurrence of $t_s$;
- $B = t_{max} - S \cdot t_s$ stands for the left time from $t_{max}$ after $S$ occurrence of $t_s$.

The frame $K$ represents the possible execution time $[t_{min}, (K+1) \cdot t_s[$; the $K+1$ frame represents the possible execution time $[(K+1) \cdot t_s, (K+2) \cdot t_s[$; the last $S$ frame represents the possible execution time $[S \cdot t_s, t_{max}]$. All the execution time from $t_{min}$ to $t_{max}$ is covered. We give an example (Ex. 3.1) to explain the mapping semantics.

**Example 3.1 (Preemptive Scheduling Example)** *Suppose the execution time of a given action is [10, 20], and the time slice of the scheduler is 3. According to the above mapping semantics, K = 3, S = 6, A = 1, and B = 2. The possible execution time and slice occurrences are listed in the Table 3.6.1.*

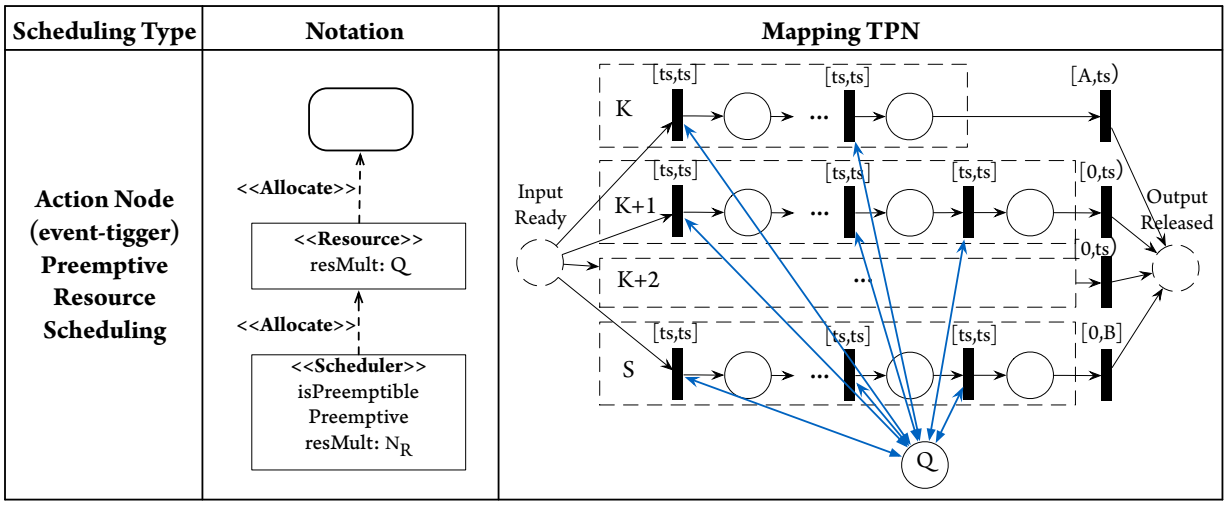



**Figure 3.6.3:** Preemptive Resource Scheduling Semantics

**Table 3.6.1:** Preemption Scheduling Example

| Frame K (K = 3, 4, 5, 6) | Execution Time | Slice Occurrence Time Point |
|---|---|---|
| 3 | $[10, 12[$ | 3, 6, 9 |
| 4 | $[12, 15[$ | 3, 6, 9, 12 |
| 5 | $[15, 18[$ | 3, 6, 9, 12, 15 |
| 6 | $[18, 20]$ | 3, 6, 9, 12, 15, 18 |

## 3.7 Time Semantics in Multi-Clock Modeling

For real-time analysis in multi-clock modeling, one of the clocks must be a reference clock. Then, other clocks can be compared with this reference clock. *Clock tick* is the smallest unit of time recognized by a device. *Clock drift* refers to the phenomena where a clock does not tick exactly at the same time as the reference clock. From the viewpoint of real-time analysis, the main difference between single-clock and multi-clock modeling is that the *clock drifts* should be taken into account in multi-clock modeling environment. In single-clock modeling, it is not mandatory to distinguish the notions of *tick* and *clock cycle* (the amount of time between two ticks of a clock), because the difference between the *clock cycle* and the physical time is of the same proportion for both *clock cycle* and *tick* at any given time. If a clock drift occurs, it is

also effective for every part in the system. In multi-clock modeling, however, the semantic mapping need to exhibit a correct semantics for clock drifts as each clock drifts independently from the others.

The main idea is to assume a global physical clock and project each time consumption and drift on this precise time reference. In our study, we use the physical time notion as the exact reference for both single-clock and multi-clock modeling. The physical time and the verification tools we rely on both dense time and discrete, thus our approach can handle both dense time and discrete time problems.

In the single-clock context, the measured execution time is directly used after a global normalization of the time units. For example, if action A takes $[3.4, 4.7]$ *ms* and actions B $[78.9, 463.5]$ $\mu s$, the corresponding min time and max time on the TPN transition are respectively $[34000, 47000]$ and $[789, 4635]$, with the common unit of 0.1 $\mu s$ to keep all the results natural numbers. All time values in time constraints should be integers, as the TINA model checker requires this convention.

In the multi-clock context, the measured execution time needs to be first mapped to tick numbers from the global physical clock, and then the physical model time is deduced by associating each clock's drift. We use the same example but respectively give the corresponding clock properties: let clocks A and B theoretically tick every 1 $\mu s$, and their backward drift and forward drift are both 1%, therefore action A's tick number is $[3400, 4700]$ and action B's is $[78.9, 463.5]$. As tick number must be integer, a rounding strategy must be designed without introducing unreasonable conversion error. We use the floor function for $t_{min}$ and ceiling function for $t_{max}$. Therefore, we have A for $[3400, 4700]$ and B for $[78, 464]$ as tick numbers after the rounding. It is possible to take a more precise unit, but the more precise the more states will be created when analyzing the real-time properties, because it increases the time difference between the max and min time values. Thus there exist a compromise between the precision and the scalability of verification.

As the corresponding tick time range is $[0.99, 1.01]$ $\mu s$ due to the mentioned clock property, action's physical time duration is computed by multiplying this range and action's tick number range. Following the same principle of unit normalization, the final min time and max time are $[336600, 4747000]$ and $[7821, 46763]$ respectively, with the common unit of 0.01 $\mu s$. Compared with the actions in mono-clock modeling, the precision of execution time is increased.

The drawback is that, as the method assumes each component has an independent clock, it can be too constraining for those devices which share a clock in a multi-clock modeling. The reason why we decided to choose this abstraction is that in the verification view point, this will only lead to a false-violation, which means if a time property is verified under independent-clock hypothesis, it must also be true for a shared-

clock system. This sufficient but not necessary condition in practice may only cause a performance trade-off in system design, but never gives out a wrong verification result when property's proof is positive.

## 3.8 Discussion

### 3.8.1 Verification of Model Transformation

The automatic model transformation referred to in this work is in fact a semantic mapping, which preserves all the property-related semantics of the source UML-MARTE model. Regarding the objective of the verification of real-time properties at architecture level, this abstraction is justified because it is not mandatory to preserve all the information, for example, the object values.

A concern with this method is whether the model transformation (semantic mapping) is correct. In other words, how to verify this model transformation (semantic mapping). Indeed, this is a crucial question.

What to verify? Some surveys of the state-of-the-art about the verification of code generation [Dav03, Nec11] and the verification of model transformation [CS13, PSS98] summarized the following expected properties:

- **Language-related properties** includes terminate, determinism, typing, and preservation of execution semantics properties.
- **Transformation-related properties** includes source/target conformance, syntax relations, semantics relations and functional behaviors properties.

The verification of model transformation for the UML-MARTE model is not trivial. Generally, the best way to verify if the model transformation preserves the intended semantics is to compare the state space graph of the source and target formal models. As shown in Fig 3.8.1, a formal specification must then be defined for UML models as a reference semantics. The execution semantics is then compared with this reference semantics. However, since UML is semi-formal, a formal definition is needed to establish the reference, which is one of the work in this thesis. Our proposal relies on a translation to a formal model instead of a direct formal specification of an operational semantics that would allow to build the state space at the UML level. This does not change the fundamental issue: how to validate this formal specification?

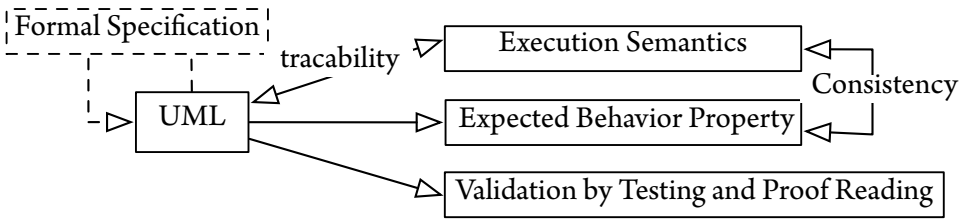



**Figure 3.8.1:** Verification of Model Transformation

A solution may mitigate the problem by mapping the UML-MARTE model to different formal models and verifying if they converge into the same formal semantics. Nevertheless, whether the semantics is lost between a semi-formal model and a formal one can only be assessed using testing and human proof reading.

Another possible solution is derived from translation validation that have been experimented for the same purpose for AADL in the QUARTEFT project. This method allows to verify that some important intended behavioral properties conform to the execution semantics. For example, we can define TPN observers to assess the run-to-completion processing semantics. More precisely, when an event occurrence is being processed, the other occurrences of this event cannot be accepted. However, when the behavior property specification and the execution semantics are both wrong in the same way, this method does not work. Then some test cases must be used to validate the execution semantics.

As a future research direction, the expected behavior properties would be defined and used to verify the conformance between the execution semantics and the behavior specification. This can validate some key execution semantics in the UML models.

### 3.8.2 Boundedness and Decidability Issue

The main objective of this thesis is to propose a set of methods that may improve the efficiency of model checking in order to verify properties in large scale systems. The mapping translates the end user model to the verification model, on which the desired properties will be assessed. We need to discuss here whether the proposed mapping semantics can ensure boundedness and decidability in the verification TPN models.

Before discussing this issue, we recall the research background of this thesis. We rely on the UML-MARTE design models that have finite states and finite event occurrences. In other words, the design model is bounded. In fact, a practical correct engineering system must be K-bounded, otherwise it is not possible to implement it with limited resources. Therefore instead of checking the boundedness, it becomes a K-

boundedness problem for this thesis. Since K-bounded TPN without stopwatch arcs is decidable, and no stopwatch is used in the model mapping, therefore this mapping will not introduce any unboundedness or undecidability.

*A TPN is bounded if the marking of each place is bounded by some integer.* [BV07] In the semantic mapping method, we have used inhibitor arcs for the Activity Final and Object nodes in the activity diagram, and the Terminate pseudostate in the state machine diagram. The question is whether the inhibitor arcs will make the TPN model unbounded.

We need to discuss this problem taking into account the following two aspects:

- **Inhibitor arcs in Activity Final nodes and Terminate pseudostates.** The mapping semantics are defined respectively in page 70 and page 113. In this case, the inhibitor arcs are used to terminate all the transitions in the TPN model when the whole system enters the final flow state, which potentially decrease the size of the state space of a TPN model.

  When the control flows have not yet arrived at the final TPN place, this place is empty, which means it cannot affect the behavior of the control/data flows. Once the final flow place is filled, all the transitions in the whole TPN model are stopped, and thus no new tokens can be produced. Therefore, the TPN model is still bounded.

- **Inhibitor arcs in object nodes.** The mapping semantics is defined from page 79 to page 83. In data flows, the TPN place representing UML object node will become unbounded only in the following cases: the producer of tokens continues to send tokens to the object TPN place, while the speed of consumption is rather slow. Obviously, this is caused by a boundedness design error. A well designed real-time system must avoid generating an unbounded amount of data or must possess a clearance mechanism to restrict the capacity of the object store. Therefore, before verifying real-time properties, a verification on boundedness should be performed.

*State reachability and boundedness is proven to be undecidable for arbitrary TPN. However, state reachability is decidable for bounded TPN, which is sufficient for virtually all practical purposes.* [GLM+05] Therefore, in the context of this thesis, as the TPN model is bounded, and the state reachability is decidable.

## 3.9 Conclusion

This chapter presented the methodology for the property-driven approach, which provides the basis for our semantic mapping contributions. The objective is to automatically map UML-MARTE models to executable TPN models on which efficient model checking can be performed afterwards to verify real-time properties.

The main contributions of the current chapter are summarized as follows:

1. **Specification of the mapping semantics for composite structure diagrams** ( Section 3.3).
   The composite structure diagram connects different sub-system behaviors through the communication medias. The mapping semantics is provided for the entities Part, Port and Connector.

2. **Specification of the mapping semantics for activity diagrams** (Section 3.4) [GPC12b].
   The activity diagram emphasizes the sequence and conditions for coordinating lower-level behaviors. The mapping semantics is provided for the UML-AD control nodes, event-trigger and time-trigger actions, objects, and connections. In order to standardize the mapping semantics for the asynchronous behavior, we extend the original semantics for action by defining an asynchronous semantics using the MARTE profile, and then map it to the TPN model. It is a general pattern in the reactive asynchronous system, and thus can be reused in the modeling and verification.

3. **Specification of the mapping semantics for state machine diagrams** (Section 3.5).
   We investigate the behavioral state machine in this thesis. Two aspects are considered when the mapping semantics is defined. First, hierarchically nested states and orthogonal regions do not extend the semantic expressiveness. They help the designer in the writing of sophisticated models for complex systems. The nested SMD can be converted to an unnested SMD. This is the work of flattening. Second, the unnested SMD with only simple states, final states, transitions and unnested pseudostates are mapped to the TPN model. This is the work of mapping.

4. **Specification of the mapping semantics for resource scheduling** (Section 3.6).
   In this work, we do not aim to provide the mapping semantics for any specific scheduling policy. Instead, we propose a generic scheduling algorithm with preemption option. This scheduling algorithm is used to decide for the given time $T$, which resource instance(s) will be allocated to which requester(s). The specification and verification of specific scheduling policies can be a future research work.

5. **Implementation of the tool for semantic mapping in the property-driven verification toolset.** The mapping semantics defined in this chapter has been implemented as a tool in the real-time property verification toolset. The implementation coverage library is provided in Appendix A.

# 4

# Specification of Real-Time Property


## Résumé

Les langages de modélisation doivent fournir des éléments pour décrire la structure et le comportement des systèmes ainsi que leurs exigences. Plusieurs approches ont été proposées pour les propriétés temps réels : des extensions des logiques exploitées par les outils de vérification telles LTL, CTL, mu-calcul qui sont éloignées du point de vue utilisateur du système; des patrons de propriétés issus d'une analyse du domaine comme ceux proposés par Dwyer et Konrad; des relations d'ordre partiel entre les événements se produisant dans le système comme la partie CCSL (Clock Constraint Specification Language) du standard MARTE. Ces deux dernières approches ont été conçues pour l'utilisateur final et ne sont pas forcément adaptées à une mise en oeuvre efficace pour les outils de vérification de modèle. Ce chapitre définit un ensemble minimal de patrons de propriétés temps réel atomiques qui sera utilisé pour traduire les propriétés qualitatives et quantitatives temps réel exprimées par le concepteur. L'objectif est de faciliter les activités de vérification sans réduire l'expressivité des spécifications.

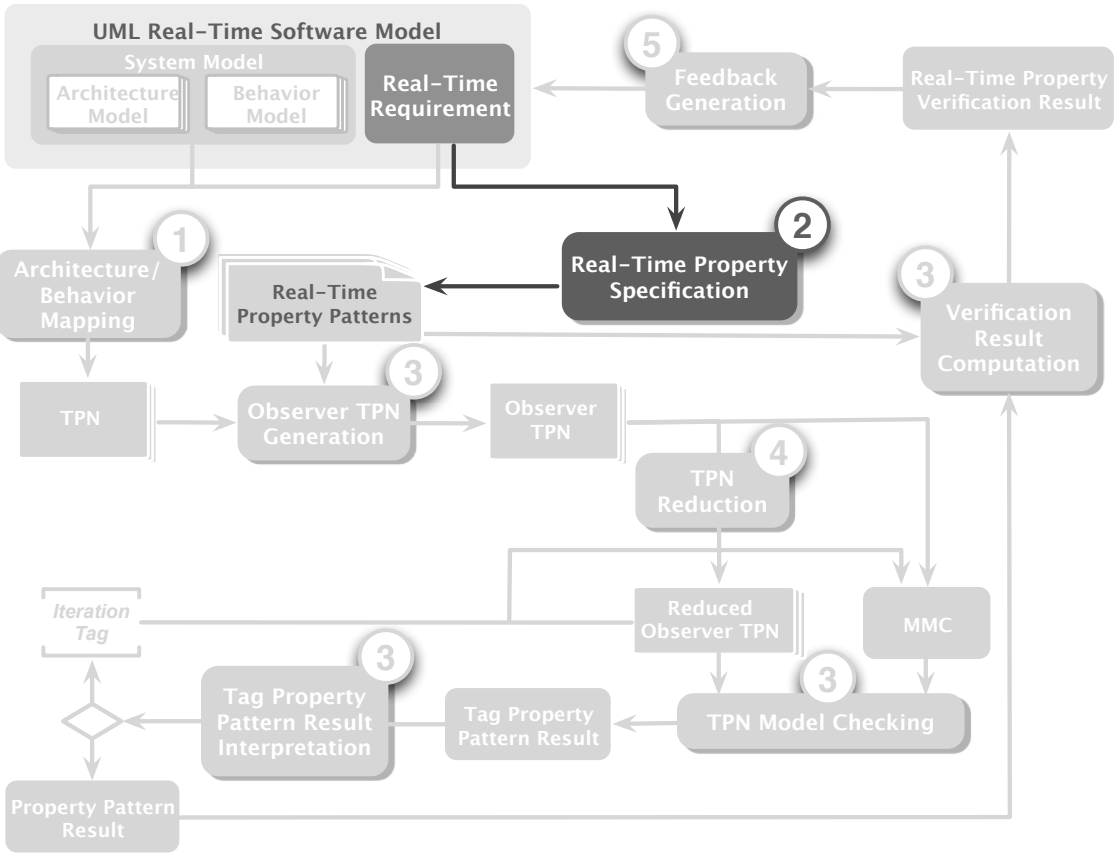
Selon les exigences exprimées dans plusieurs projets de recherche impliquant de nombreux partenaires industriels, les besoins en terme de vérification de propriétés temps réel comprennent: les pires et meilleurs temps d'exécution, les pires et meilleurs temps de traversée des moyens de communication, la durée d'un état, les contraintes liées à la synchronisation, la coïncidence, la précédence, etc.

Ces exigences peuvent être sémantiquement décomposées en un ensemble de propriétés élémentaires que nous appelons les patrons de propriété pour la vérification. Ils peuvent faciliter l'utilisation de méthodes formelles en particulier pour les utilisateur non-experts en fournissant des solutions récurrentes à la spécification et la vérification. Ils permettent de décomposer des propriétés complexes comme une composition de propriétés élémentaires qui reposent sur un plus petit espace d'état et réduisent ainsi le coût de vérification.

Les patrons de propriété habituels sont dérivés du travail de Dwyer et Konrad. Ils ciblent l'expressivité pour les utilisateurs finaux qui spécifient les exigences temps réel, mais ne garantissent généralement pas l'atomicité sémantique ou la facilité de la vérification. Nous proposons un ensemble minimal de patrons atomiques de propriétés temps réel dans le but de diminuer la complexité de la vérification. L'intégralité des exigences temps réel exprimées par des patrons de Dwyer et Konrad et une part importante de celles exprimées en CCSL peuvent être traduites sous la forme d'une composition de ces patrons élémentaires. Cette décomposition est automatique et donc transparente aux utilisateurs.

Ces patrons de propriété sont minimaux parce qu'ils sont sémantiquement atomiques et ne peuvent pas être exprimés sous la forme d'une composition d'autre éléments atomiques. Nous fournissons la traduction des patrons de Dwyer et Konrad. Cela signifie que nos patrons de propriété atomique sont sémantiquement complet par rapport aux travaux de Dwyer et Konrad.

Nous avons également traduit une partie de la spécification de CCSL ainsi qu'une variante de CCSL basée sur les tâches en nos patrons de propriétés. C'est une seconde illustration de l'expressivité des patrons que nous proposons.




*Progress Map 2:* Property Specification using Real-Time Property Patterns

This chapter defines a minimal set of real-time property patterns used to specify both qualitative and quantitative real-time properties, for the purpose of verification-ease and semantic completeness (Progress map 2). Classic property patterns based on Dwyer's and Konrad's pattern systems target expressiveness for the end-users that specify real-time requirements, but this usually does not ensure that they are semantically atomic or easy to verify (Challenge 2 in page 22). We define a minimal set of atomic real-time property patterns in the order to decrease the verification complexity. All end-user dedicated real-time requirements are expressed as compositions of these patterns. The common requirements based on Dwyer's and Konrad's patterns and CCSL language will be automatically mapped to our patterns using a predefined metamodel and a mapping library. We also define a small extension for task level CCSL specification and translate them into our property patterns (Contribution 2 in page 22). All the patterns defined in this chapter will be checked

in an efficient way using the proposals in Chapter 5.

## 4.1 Introduction

According to the user's point of view expressed in several collaborative research projects involving industrial partners such as P [1], TOPCASED [2], OPEES[3], QUARTEFT [4], SPICES [5], SPACIFY [6], and CESAR [7], etc, the real-time requirements commonly used in a real-time concurrent system include the worst/best case execution time, worst/best case traversal time, state duration, the scheduling related constraints such as synchronization, precedence, coincidence, etc. [Kop11]. These requirements can be semantically decomposed into a set of elementary properties that we call property patterns. Design patterns are widely used in many engineering domains, because they are thought as a means of leveraging the experience of expert system designers [VHJG95]. Property patterns can fulfill a similar purpose: on one hand ease the use of formal methods especially for the non-expert users by providing the recurrent solutions to specification and verification problem; and on the other hand decompose complex properties into a set of simpler ones that rely on a smaller state space and thus decrease the verification cost.

In this chapter, we present a set of real-time property patterns used to specify real-time requirements.

**Property based on Dwyer's and Konrad's works**

Dwyer et al. initially proposed qualitative temporal property patterns for finite-state verification [DAC98, DAC99]. They focused on logical time properties, thus no concept of quantitative real-time constraints such as time interval and duration were present in their pattern system. Dwyer et. al. also performed a large-scale study in which specifications containing over 500 temporal requirements were collected and analyzed. They noticed that over 90% of these could be classified under one of the proposed patterns [DAC99], which encouraged others to use Dwyer's pattern system and to extend this study.

The following works on quantitative time property patterns [KC05, GL06, ADZLB12] extended Dwyer's patterns, with additional real-time constraints. In [KC05], Konrad created mappings of quantitative time

[1] http://www.open-do.org/projects/p/
[2] http://www.topcased.org/
[3] http://www.opees.org/
[4] http://projects.laas.fr/fiacre/
[5] http://www.spices-itea.org/public/news.php
[6] http://spacify.gforge.enseeiht.fr/
[7] http://www.cesarproject.eu/

property patterns into three real-time temporal logics: MTL,TCTL [Alu91], and RTGIL [MRK+97], and then defined a pattern template to ease the reuse. [GL06] provided a catalogue of patterns for real-time extension that handled a less expressive set of patterns (without some modifiers). Also based on Dwyer's property patterns, [ADZLB12] proposed a set of real-time properties that introduced the time constraints *Interval* and *Duration*, using 4 scope modifiers and 4 categories of patterns. They did not implement all the scope and categories (e.g. *Precedence, Bounded Existence, Chain Response* and *Chain Precedence*), because they aimed to apply their approach on the modeling language Fiacre[8] [BBF+07], which does not require all the patterns but only the most commonly used ones.

From the viewpoint of property verification, we advocate that the property patterns in Dwyer's pattern system are not atomic. Let's take a end-to-end real-time requirement as example (see Ex. 4.1).

**Example 4.1 (Verification Pattern Example)** *For events A and B, Within time interval I ($[T_{min}, T_{max}]$), the real-time property is **Exist** A **After** B **Within** I. Its semantics can be represented by the logic formula:*

$$(\neg B) \vee (B \wedge A \wedge (\boldsymbol{T}_{AB} \geq T_{min}) \wedge (\boldsymbol{T}_{AB} \leq T_{max})),$$

*where $\boldsymbol{T}_{AB}$ is the time interval from the first occurrence of A to the first occurrence of B. It can be decomposed into 3 atomic properties: Exist B, $\boldsymbol{T}_{AB} \geq T_{min}$ and $\boldsymbol{T}_{AB} \leq T_{max}$.*

**Property based on CCSL**

UML by itself is an untimed model. Many extensions were proposed inside and outside OMG. MARTE was introduced to provide a generic time expressiveness. To explicit keywords that denote usual concepts of the domain (periodic, sporadic, sampling, etc), Mallet et al. introduced the Clock Constraint Specification Language (CCSL) [AM08]. It offers a rich set of constructs to specify time requirements and constraints based on sets of instantaneous clocks (events) and clock constraints.

**Need for a verification-ease property specification method**

Relying on the decomposition in Ex. 4.1, the real-time requirements can be translated to and checked with a set of atomic properties. We aim to define such a minimal set of atomic property patterns that targets the ease of both specification and verification. The properties expressed using Dwyer/Konrad's patterns

[8] http://projects.laas.fr/fiacre/

and CCSL languages can be automatically translated to the verification targeted atomic property elements, which will then be assessed using the observer-based verification approach.

We explain some core concepts in Section 4.2, then give a quick overview of property pattern approaches in Section 4.3. We introduce the catalog of real-time property patterns in Section 4.4. The metamodel of real-time property pattern and the mapping library are presented in Section 4.5. In order to assess the expressiveness of the real-time verification patterns, we apply our approach to CCSL constraints, and propose a small extension based on CCSL to deal with the task level constraints in Section 4.6.

## 4.2 Preliminaries

Before defining the pattern-based approach, it is mandatory to clarify some core concepts used in the specification: qualitative & quantitative property, occurrence & predicate & scope, event & state.

### 4.2.1 Qualitative & Quantitative Property

Real time is a quantitative notion of time that is measured using a physical (real) clock. In contrast to real time, logical (virtual) time deals with a qualitative notion of time that is expressed using event ordering relation such as *before, after, sometimes, eventually, precedes*, etc. A real-time system verification implies that all quantitative and qualitative time requirements should be satisfied. Real time can be seen as a particular case of logical time where the events generated by a physical clock are taken as time reference. In the context of this thesis, we focus on the quantitative time properties because, on the one hand, the qualitative aspects have been studied in many works; and, on the other hand, the introduction of physical clocks will increase the complexity of model checking, which is the problem we aim to study using property driven approach.

### 4.2.2 Occurrence & Predicate & Scope

A common pattern for specifying a property is composed of three elements: occurrence, predicate and scope. Predicate describes what must occur, and scope describes when it must occur. Occurrence is a concept about the bounded existence. The occurrence of a predicate could be specified as existence, absence, always (exist), or (exist) bounded occurrence. Given a temporal property **Exist** A **After** B **Within** *I*, Exist is the occurrence, A is the predicate, while After B Within I is the scope.

### 4.2.3 Event & State

The verification of end-user requirements covers three levels of model: the design model (UML in our case), the verification model (TPN in our case), and the model checking state class graph (marking graph in our case). We should distinguish the concepts of event and state from these three levels.

An event is an instantaneous and atomic occurrence of an action at a point in time. Relying on different model granularities, an event can be:

- **At UML model level**:
  - A communication event: send, receive, read, write, etc.
  - The execution of a transition from one state to another
- **TPN model level**: a TPN transition.
- **State class graph level**: a transition between states.

State is a universal concept through the whole system, regardless of the modeling granularity. A state represents a situation during which some invariant conditions hold. The system remains in the state for some time.

Before presenting the details about the proposed property pattern approach, we need to clarify a convention on the use of event and state. In the pioneer work of Dwyer et al., a complete set of qualitative property patterns were defined targeting specification activities. Thus, there was no need to distinguish the use of event and state in the predicate and scope. For example, when a property is specified as *Exist A Before B*. A and B could be state or event. From the viewpoint of verification, the predicate *Exist A* is supposed to support both state and event. Nevertheless, to avoid ambiguity, the scope *Before* a state is usually understood as *Before* the enter event of the state. Therefore, using scope with state is only a specification requirement, while it is redundant for the verification. For this reason, in the context of this thesis, the scope in a property can only be used with events, while the predicate can be used both with events and states.

## 4.3 Property Pattern Approach

Dwyer's property pattern system was based on eight patterns (*Absence, Existence, Bounded Existence, Precedence, Response, Chain Precedence* and *Chain Response*) and five scope modifiers (*Global, Before, After, Be-*

*tween* and *After-Until*). Konrad and Cheng extended Dwyer's patterns to specify both qualitative and quantitative requirements. Konrad's property patterns are organized in an hierarchy in Fig. 4.3.1 [KC05], where the grey frame part corresponds to Dwyer's patterns.

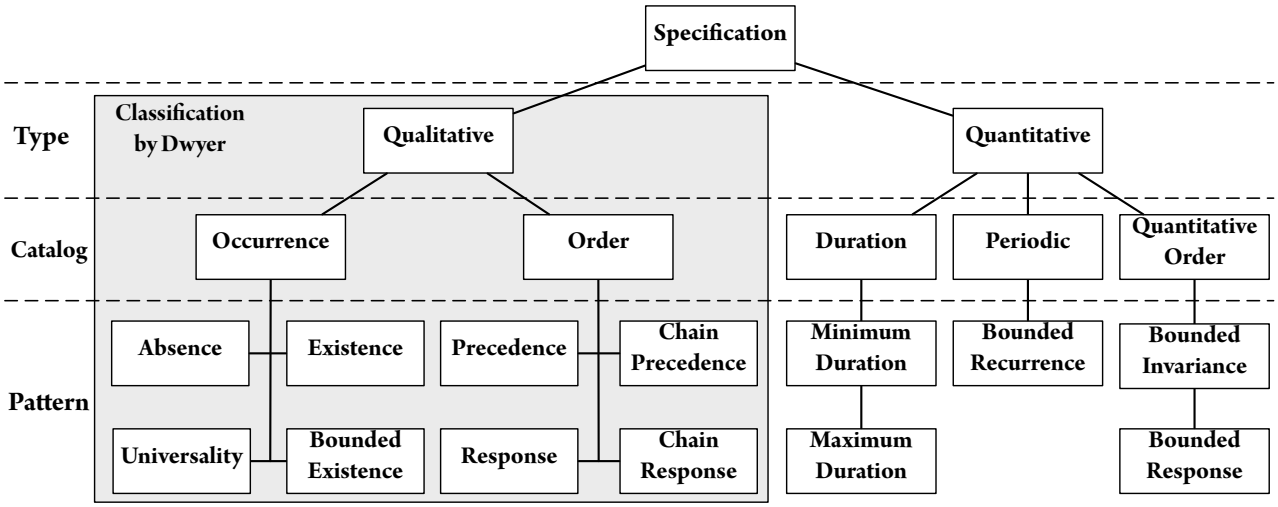



**Figure 4.3.1:** Pattern Hierarchy

### 4.3.1 Qualitative Property Patterns

Dwyer's qualitative patterns are briefly described as follows. In the descriptions, for brevity, we use the term *predicate* to mean *a state in which the given state formula is true, or an event from the given disjunction of events occurrences.*

- **Absence** A given predicate must not occur within a scope.
- **Existence** A given predicate must occur within a scope.
- **Bounded Existence** A given predicate must occur k times within a scope. Variants of this pattern specify at least k occurrences and at most k' occurrences of a state/event.
- **Universality** A given predicate occurs throughout a scope.
- **Precedence** A predicate P must always be preceded by a predicate Q within a scope.

- **Response** A predicate P must always be followed by a predicate Q within a scope.
- **Chain Precedence** A sequence of states/events $P_1$, ..., $P_n$ must always be preceded by a sequence of states/events $Q_1$, ..., $Q_m$. This pattern is a generalization of the **Precedence** pattern.
- **Chain Response** A sequence of states/events $P_1$, ..., $P_n$ must always be followed by a sequence of states/events $Q_1$, ..., $Q_m$. This pattern is a generalization of the **Response** pattern.

In our work, we focus on the first six patterns, because the chain patterns can be split into the atomic verification patterns. Moreover, the bounded existence, precedence, and response patterns can be specified using absence, existence, and universality patterns and some basic predicates.

The five qualitative scope modifiers defined by Dwyer are:

- **Global** The predicate must hold during the whole system execution.
- **Before** The predicate must hold up to a given event.
- **After** The predicate must hold after the occurrence of a given event.
- **Between** The predicate must hold between the occurrence of event *P* and the occurrence of event *Q*.
- **After-Until** Similar to **Between**, but the predicate must hold even if event Q never occurs.

To represent the periodic semantics, we add a new scope modifier **Periodically**:

- **Periodically** The predicate must hold at least once every period.

### 4.3.2 Real-Time Suffix

As shown in Fig. 4.3.1, five quantitative modifiers were introduced in Konrad's quantitative temporal property patterns: *Minimum Duration, Maximum Duration, Bounded Recurrence, Bounded Response* and *Bounded Invariance*.

In our work, considering that the specification should ease the verification and the composition of atomic patterns, instead of quantitative modifiers, we introduce three real-time suffixes to use together with some predicates and scopes. These real-time suffixes have equivalent semantics to the ones defined by Konrad.

- **At least** The predicate must hold at least *T t.u.*. This suffix is used together with state predicate.
- **At most** The predicate must hold at most *T t.u.*. This suffix is used together with state predicate.
- **Within** The predicate must hold within time interval *I* ($[T_{min}, T_{max}]$). This suffix is used together with scope *Before* and *After*.

With these suffixes and new scope *Periodically*, we can specify real-time properties such as:

- **Exist** event A **After** event B **Within** I;
- **Always** state S holds **At least (At most)** T *t.u.* **Before** event E;
- **Absent** event A **Precedes** event B, **Periodically**.

After clarifying the patterns and scope modifiers, all the real-time requirements in real-time reactive systems can be specified by combining these patterns, scope modifiers and real-time suffixes. The translation is given in the Appendix B.

## 4.4 Catalog of Real-Time Property Patterns

This catalog covers the atomic verification patterns used to check real-time requirements. In the context of the thesis, as we rely on model checking for the verification, we focus on finite-state systems, whose execution time can be finite or infinite.

The common method to specify a property for a given system is composed of two aspects: pattern and scope. In the whole system's state graph, the scope operator is first executed to select all states that belong to this scope. These state candidates are then qualified by the given pattern definition. In order to ease the verification, we define a minimal set of patterns (including predicates and scopes). All the other real-time requirements are in fact a composition of these elementary patterns.

Our real-time property patterns can specify all the requirements based on Dwyer's pattern system with additional quantitative time suffixes. We provide the translation from all possible Dwyer's and Konrad's property patterns to our patterns in Appendix B. Since it is proved that 90% of real-time requirements can be specified using Dwyer's patterns, we can say that most real-time requirements are also covered by our approach.

In the proposed pattern system (see Fig. 4.4.1 ), the real-time requirements will be specified using real-time property patterns (either atomic pattern or composite pattern). Composite patterns can be easily build using binary operators (or, and, imply). Atomic patterns contain three elements: *occurrence modifier*, *basic predicate* and *scope modifier*. Basic predicates are based on state and event modifiers while scope modifiers are only based on event modifiers.

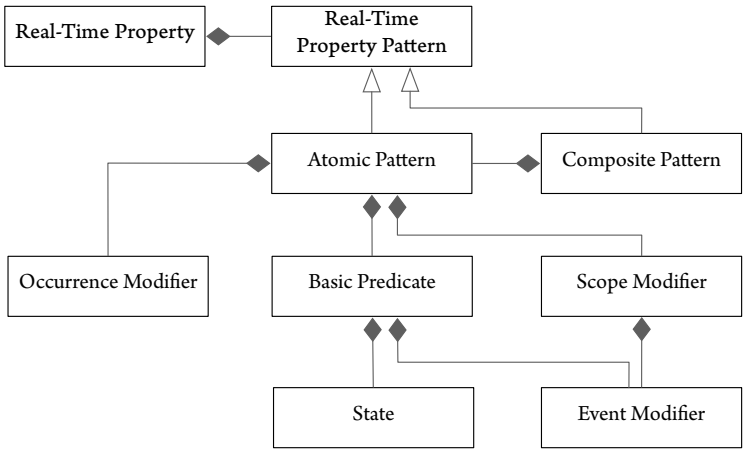



**Figure 4.4.1:** Temporal Property Verification Pattern System

In the following section, we define the set of occurrence modifiers, event modifiers, basic predicates, and scope modifiers.

### 4.4.1 Occurrence Modifier

Occurrence modifiers are used to specify the occurrence times of given event/state modifiers within some scope. All the temporal properties are in one of the three cases: Exist, Absent, and Always:

- **Exist** *predicate* in *Scope*: the given *predicate* must occur within a scope.
- **Absent** *predicate* in *Scope*: the given *predicate* must not occur within a scope.
- **Always** *predicate* in *Scope*: the given *predicate* occur through a whole scope.

### 4.4.2 Basic Event Modifier

Predicates are based on events and states. An event can be an atomic element $E$, but in most context it is more complex structure, e.g. event $E^i$ is the $i^{th}$ occurrence of event $E$. After analyzing possible usage of the event in a temporal property, we propose the following basic event modifiers. The event modifiers can be extended, which means the basic event $E$ can be replaced by the other event modifiers.

- $\mathbf{E}^i$ The $i^{th}$ occurrence of $E$ (Fig. 4.4.2). When using this modifier, the occurrence of $E$ must be finite under the observed execution. Whether an event is finite or not is checked by a predicate presented in the following section. In the context of this thesis, $E^1$ stands for the first occurrence of $E$, while $E$ stands for the event type. By default, when no occurrence is specified, $E$ is regarded as $E^1$.

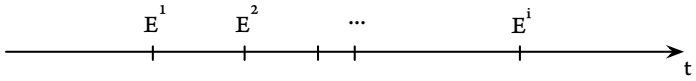



**Figure 4.4.2:** $i^{th}$ Occurrence of $E$

- $\mathbf{E}^{-k}$: The event standing for the delay of $k$ times occurrence of event $E$ (Fig. 4.4.3). This event modifier will be used to specify the temporal property between $E$ and the event of its delay occurrence, etc.

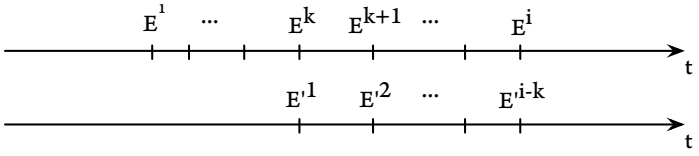



**Figure 4.4.3:** $k$ Times Occurrence Delay of $E$

- $\mathbf{E}^{/k}$: The event standing for the occurrence of the same event $E$ but ticking $k$ times slower than $E$ (Fig. 4.4.4). is This modifier will be used to specify sub-occurrences of $E$, etc.

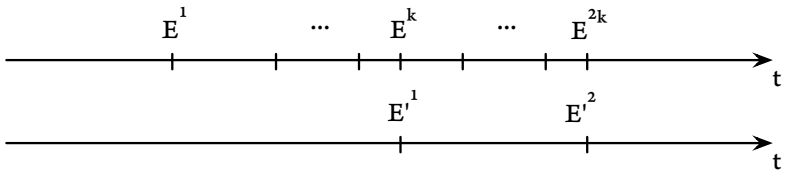



**Figure 4.4.4:** Sub-Occurrence of $E$

- **I+T**: *T t.u.* measured from the initialization of the system for verifying the worst/best case execution time of a system (Fig. 4.4.5).

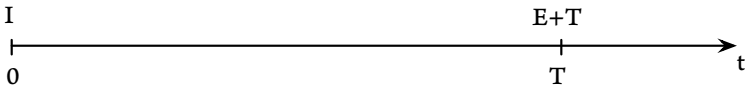



**Figure 4.4.5:** T after System Initialization

- **E+T**: *T t.u.* after $E$ (Fig. 4.4.6). When using this modifier, the occurrence of $E$ must be finite. This modifier will be used to specify the scope *within* and the predicate *at least (at most)*. For example, after $E$ within I ($[t_{min}, t_{max}]$) will be specified as after $E + t_{min}$ and before $E + t_{max}$.

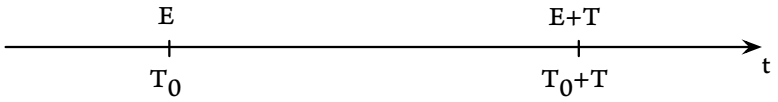



**Figure 4.4.6:** T after $E$

- $\mathbf{S}^S$: The event standing for entering state $S$ (Fig. 4.4.7).

- $\mathbf{S}^E$: The event standing for exiting state $S$ (Fig. 4.4.7).

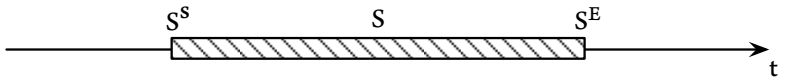



**Figure 4.4.7:** Entering and Exiting event of State

### 4.4.3 Basic Predicate

Basic predicates are based on events and states. An event stands for the above event modifiers or composite events. In the temporal properties, we are concerned with the following basic predicates:

- **O**($E^i$) = **True**: $E^i$ has occurred.
- **isFinite**($E$) = ***True***: The occurrence of event $E$ is finite.
- **Freq**($E_A$) · $\mathbf{N}_A$ = **Freq**($E_B$) · $\mathbf{N}_B$ : There exists equivalent occurrences between $E_A$ and $E_B$. This predicate is used to verify the temporal properties in periodic scheduling, in both finite and infinite time execution. Usually, it appears together with the scope **Periodically**. Suppose two periodic events $E_A$ and $E_B$ exhibit respectively occurrence frequency $F_A$ and $F_B$. There exists minimal coefficients $N_A$ and $N_B$ ($N_A, N_B \in \mathbb{Z}^+$) that makes $F_A \cdot N_A = F_B \cdot N_B$. $N_A$ and $N_B$ can be computed using the Least Common Multiple (`lcm`) and the Greatest Common Divisor (`gcd`).

$$N_A = \frac{\texttt{lcm}(F_A, F_B)}{\texttt{gcd}(\texttt{lcm}(F_A, F_B), F_A)} \tag{4.1}$$

$$N_b = \frac{\texttt{lcm}(F_A, F_B)}{\texttt{gcd}(\texttt{lcm}(F_A, F_B), F_b)} \tag{4.2}$$

  A temporal property may require to limit the time difference between two periodic events. If these two events exhibit the same frequency, $N_A$ and $N_B$ are equal. Otherwise, $N_A$ and $N_B$ should be introduced to identify the corresponding occurrence between $E_A$ and $E_B$.

- **T**($E_A$, $E_B$) $>$ **t**: Semantically, it is equivalent to T($E_A$) - T($E_B$) > $t$. If $E_A$ and $E_B$ have equivalent occurrences, the time interval between each equivalent occurrences of $E_A$ and $E_B$ is **At least** $t$+1 *t.u.*. This predicate can be used together with scope **Periodically**. If $E_A$ and $E_B$ occur only once, T($E_A$,

$E_B) >$ t stands for T$(E_A^1, E_B^1) >$ t. The event $E_A$ and $E_B$ can be refined using event modifiers, such as T$(E_A^{/k}, E_B)$, to specify more complex temporal property.

- **T$(E_A, E_B) <$ t**: Semantically, it is equivalent to T$(E_A)$ - T$(E_B) < t$. If $E_A$ and $E_B$ have equivalent occurrences, the time interval between each equivalent occurrences of $E_A$ and $E_B$ is **At most** *t*-1 *t.u.*. This predicate can be used together with scope **Periodically**. If $E_A$ and $E_B$ occur only once, T$(E_A, E_B) <$ t stands for T$(E_A^1, E_B^1) <$ t. The event $E_A$ and $E_B$ can be refined using event modifiers, such as T$(E_A^{/k}, E_B)$, to specify more complex temporal property.

- **S** = *True* : The state *S* holds.

- **D$(S) \geq t$** : The duration of a given *S* is **At least** *t t.u.*.

- **D$(S) < t$**: The duration of a given *S* is **At most** *t*-1 *t.u.*.

### 4.4.4 Basic Scope Modifiers

- **Global**: the scope is the whole system execution.

- **Before** $E^i$: Before the $i^{th}$ occurrence of *E*. When using this scope, **isFinite$(E)$** must be true.

- **After** $E^i$: After the $i^{th}$ occurrence of *E*. When using this scope, **isFinite$(E)$** must be true.

- **Between** $E_A$ **and** $E_B$: If $E_A$ and $E_B$ are infinite, the scope between should be redefined as between the equivalent occurrences of $E_A$ and $E_B$. The event $E_A$ and $E_B$ can be refined using event modifiers, such as Between $E_A^{/k}$ and $E_B^{-m}$, to specify more complex temporal property. If $E_A$ and $E_B$ occur only once, Between $E_A$ and $E_B$ t stands betwwen $E_A^1$ and $E_B^1$. The event $E_A$ and $E_B$ can be refined using event modifiers, such as between $E_A^{/k}$ and$E_B$, to specify more complex temporal property.

The scope **After** $E_A$ **Until** $E_B$ can be represented by the above ones:

- When $E_B$ occurs after $E_A$, it is equivalent to **Exist** $E_B$ **After** $E_A \wedge$ **Between** $E_A$ **and** $E_B$;

- When $E_B$ does not occurs after $E_A$, it is equivalent to **Absent** $E_B$ **After** $E_A \wedge$ **After** $E_A$.

## 4.5 Metamodel and Mapping Library

All temporal properties based on Dwyer's patterns with additional quantitative time suffixes can be mapped to our patterns. The complete mapping library is provided in Appendix B.

We present in Fig. 4.7.1 the metamodel of the proposed pattern system. It is defined within Eclipse modeling Framework (EMF) to ease the integration with UML models.

Relying on this metamodel and mapping library, the mapping process can be performed automatically.

## 4.6 Pattern Composition : Application to CCSL Constraints

In order to assess the expressiveness of the real-time verification patterns, we apply our approach to a commonly used temporal property specification language: CCSL from the UML MARTE standard. We first introduce what is CCSL in Section 4.6.1, then present the concept of time tolerance in verification in Section 4.6.2. The CCSL constraints are then translated using the proposed real-time property patterns in Section 4.6.3. We have defined task level constraints based on CCSL. These constraints are specified using our verification patterns in Section 4.6.4.

### 4.6.1 What is CCSL

UML by itself is an untimed model. Many extensions were proposed inside and outside OMG. MARTE was introduced to provide a generic time expressiveness. To explicit keywords that denote usual concepts of the domain (periodic, sporadic, sampling, etc), Mallet et al. introduced the Clock Constraint Specification Language (CCSL) [MAL10]. It offers a rich set of constructs to specify time requirements and constraints based on sets of instantaneous clocks (events) and clock constraints. The property pattern that we have defined in the previous parts can also be expressed by CCSL.

A CCSL specification consists of clock declarations and a set of binary clock relations. These relations apply to clocks or clock expressions. CCSL constraints are classified into four categories:

1. coincidence-based constraints (also known as synchronous constraints),
2. precedence-based constraints (also known as asynchronous constraints),
3. mixed constraints, which combine synchronous and asynchronous constraints,

4. NFP (Non Functional Property) chronometric constraints, which is pertinent for chronometric clocks only and used for quantitative timed properties.

**TimeSquare**: The CCSL parser is provided in the tool TimeSquare[9], which is a Model Development Kit (MDT) provided as a set of Eclipse plugins. TimeSquare allows to define CCSL constraints in the UML models, and then simulate these constraints using the generated model traces in Papyrus MDT[10].

### 4.6.2 Time Tolerance in Verification

CCSL deals with logical time. It defines the coincidence between two clocks $A$ and $B$ as they occur at the same time, which indicates $T(A) - T(B) = 0$. Although the coincidence means that something occurs simultaneously (with no time difference), in real-time system the strict simultaneous property is rarely achieved. Thus, the design requirements are usually associated with time tolerance. In order to be more realistic, we introduce the concept of time tolerance for all CCSL-based constraints. The time tolerance is denoted by $\delta$, which can be expressed using the NFP chronometric constraints.

### 4.6.3 Specification of CCSL Constraints

The real-time property patterns proposed in our work can be seen as a translation bridge between the front-end specification language (in our case the properties in Dwyer's and Konrad's works) and the back-end verification language (in our case TPN observers and the logic formulae). CCSL is also a front-end specification language for expressing the event-based real-time properties. Therefore, the CCSL constraints can be translated to our property patterns in order to be verified using the model checking afterwards. In this section, we translate the CCSL coincidence-based constraints (sub-clocking and tight sub-clocking), precedence-based constraints (precedence and strict precedence) and their derived constraints (equality, exclusion, alternation, and synchronization) using our property patterns.

Chronometric constraints are special kinds of logical constraints, with a specific clock called IdealClock or Real-Time clock in CCSL. In our discussion, the coincidence-based and precedence-based constraints are extended by the physical clocks (denoted by the time tolerance $\delta$) using the NFP chronometric constraints.

[9] http://timesquare.inria.fr/
[10] https://www.eclipse.org/papyrus/

The original definitions of the CCSL constraints are referenced from the work of Mallet et al. in [MAL10, AM09].

**Clock** A clock is a 5-tuple $\langle \mathcal{I}, \prec, \mathcal{D}, \lambda, u \rangle$, where

- $\mathcal{I}$ is a set of instants,
- $\prec$ is a quasi-order relation on $\mathcal{I}$, named strict precedence,
- $\mathcal{D}$ is a set of labels,
- $\lambda : \mathcal{I} \rightarrow \mathcal{D}$ is a labeling function,
- $u$ is often called `tick`, it can be `processorCycle` as well or any other logical activation of a behavior.

The ordered set $\langle \mathcal{I}, \prec \rangle$ is the temporal structure associated with the clock. $\prec$ is a total, irreflexive, and transitive binary relation on $\mathcal{I}$.

A discrete-time clock is a clock with a discrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a fashion that respects the ordering on $\mathcal{I}$: let $\mathbb{N}^* = \mathbb{N} \backslash \{0\}$, idx : $\mathcal{I} \rightarrow \mathbb{N}^*, \forall i \in \mathcal{I}, idx(i) = k$ if and only if $i$ is the $k^{th}$ instant in $\mathcal{I}$.

**Time Structure** A time structure (TS) is a pair $\langle C, \preccurlyeq \rangle$ where $C$ is a set of clocks, $\preccurlyeq$ is a binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named precedence, $\preccurlyeq$ is reflexive and transitive. From $\preccurlyeq$ we derive four new relations:

- *Coincidence* ($\equiv \triangleq \preccurlyeq \cap \preccurlyeq^{-1}$),
- *Strict precedence* ($\prec \triangleq \preccurlyeq \setminus \equiv$),
- *Independence* ($\| \triangleq \overline{\preccurlyeq \cup \preccurlyeq^{-1}}$), and
- Exclusion ($\# \triangleq \prec \cup \prec^{-1}$).

#### 4.6.3.1 CCSL Coincidence-based Constraints

There exists a mapping $h$ from $\mathcal{I}_a$ to $\mathcal{I}_b$ which is injective and order preserving. $a$ is said to be a sub-clock of $b$, and $b$ is a super-clock of $a$.

**Sub-clocking** Let $a, b$ be two clocks. The clock relation $a$ `isSubClockOf` $b$ means that $a$ is a sub-clock of $b$ (denoted as $a \boxed{\subset} b$). This means that each instant in $a$ is coincident with exactly one instant in $b$. An example of sub-clocking is given by Fig. 4.6.1. $a$ `isSubClockOf` $b$ iff:

$$\begin{aligned}&\Big((\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a)(\exists l \in \mathbb{N}^*, b[l] \in \mathcal{I}_b)(a[k] \equiv_{TS} b[l] = h(a[k]))\Big) \wedge \\ &\Big((\forall k_1, k_2 \in \mathbb{N}^*, a[k_1], a[k_2] \in \mathcal{I}_a)(a[k_1] \prec a[k_2] \Rightarrow h(a[k_1]) \prec h(a[k_2]))\Big)\end{aligned} \tag{4.3}$$

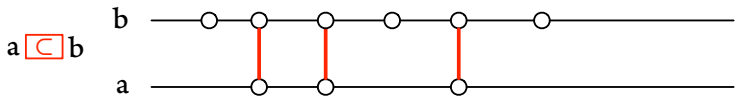



**Figure 4.6.1:** Example of Sub-clock

Specification using real-time property pattern:

$$\begin{aligned}&\Big((\forall k \in \mathbb{N}^*, a^k \in \mathcal{I}_a)(\exists l \in \mathbb{N}^*, b^l \in \mathcal{I}_b)\Big((\mathbf{T}(a^k, b^l) < \delta) \wedge (\mathbf{T}(b^l, a^k) < \delta)\Big)\Big) \wedge \\ &\Big((\forall k_1, k_2 \in \mathbb{N}^*, a^{k_1}, a^{k_2} \in \mathcal{I}_a)(\mathbf{T}(a^{k_2}, a^{k_1}) > \delta \Rightarrow \mathbf{T}(h(a^{k_2}), h(a^{k_1})) > \delta)\Big)\end{aligned} \tag{4.4}$$

**Tight sub-clocking** $a \boxed{\Subset} b$ is a sub-clocking relation in which the image of $\mathcal{I}_a$ by $h$ is an interval of $\mathcal{I}_b$. An example of tight sub-clocking is given by Fig. 4.6.2. $a \boxed{\Subset} b$ iff:

$$\begin{aligned}&\Big((\exists j \in \mathbb{N})(\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a)\Big((b[j+k] \in \mathcal{I}_b) \wedge (a[k] \equiv_{TS} b[j+k])\Big)\Big) \wedge \\ &\Big((\forall k_1, k_2 \in \mathbb{N}^*, a[k_1], a[k_2] \in \mathcal{I}_a)(a[k_1] \prec a[k_2] \Rightarrow h(a[k_1]) \prec h(a[k_2]))\Big)\end{aligned} \tag{4.5}$$

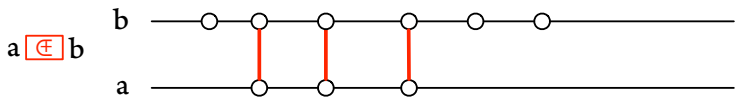



**Figure 4.6.2:** Example of Tight Sub-clock

Specification using real-time property pattern:

$$\Big((\exists j \in \mathbb{N})(\forall k \in \mathbb{N}^*, a^k \in \mathcal{I}_a)\Big((b^{j+k} \in \mathcal{I}_b) \wedge (\mathbf{T}(a^k, b^{j+k}) < \delta) \wedge (\mathbf{T}(b^{j+k}, a^k) < \delta)\Big)\Big) \wedge \Big((\forall k_1, k_2 \in \mathbb{N}^*, a[k_1], a[k_2] \in \mathcal{I}_a)(\mathbf{T}(a^{k_2}, a^{k_1}) > \delta \Rightarrow \mathbf{T}(h(a^{k_2}), h(a^{k_1})) > \delta)\Big) \tag{4.6}$$

#### 4.6.3.2 Derived Coincidence-based Constraints

**Exclusion** $a$ `exclusiveWith` $b$ (denoted as $a$ $\boxed{\#}$ $b$) means that $a$ and $b$ have no coincidence instants. $a$ `exclusiveWith` $b$ iff.

$$(\forall j \in \mathbb{N}^*, a[j] \in \mathcal{I}_a)(\forall k \in \mathbb{N}^*, b[k] \in \mathcal{I}_b)\Big(\neg(a[j] \equiv_{TS} b[k])\Big) \tag{4.7}$$

Specification using real-time property pattern:

$$(\forall j \in \mathbb{N}^*, a^j \in \mathcal{I}_a)(\forall k \in \mathbb{N}^*, b^k \in \mathcal{I}_b)\Big(\neg\Big((\mathbf{T}(a^j, b^k) < \delta) \vee (\mathbf{T}(b^k, a^j) < \delta)\Big)\Big) \tag{4.8}$$

**Equality** $a$ $\boxed{=}$ $b$ is a typical synchronous clock relation derived from tight sub-clocking.

$$a \boxed{=} b \Leftrightarrow (a \boxed{\Subset} b) \wedge (b \boxed{\Subset} a)$$

Hence, there is a bijection between instants of $a$ and $b$. This bijection is order preserving and the instants are point-wise coincidence: $\forall k \in \mathbb{N}^*, a[k] \equiv b[k]$. An example of equality is given by Fig. 4.6.3. $a$ $\boxed{=}$ $b$ iff:

$$\Big((\forall k \in \mathbb{N}^*, a[k] \in \mathcal{I}_a)\Big((b[k] \in \mathcal{I}_b) \wedge (a[k] \equiv_{TS} b[k])\Big)\Big) \wedge \Big((\forall k \in \mathbb{N}^*, b[k] \in \mathcal{I}_b)\Big((a[k] \in \mathcal{I}_a) \wedge (a[k] \equiv_{TS} b[k])\Big)\Big) \tag{4.9}$$

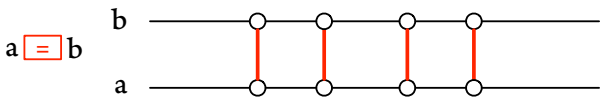



**Figure 4.6.3:** Example of Equality

Specification using real-time property pattern:

$$\begin{aligned}&\Big((\forall k \in \mathbb{N}^*, a^k \in \mathcal{I}_a)\Big((b^k \in \mathcal{I}_b) \wedge (\mathbf{T}(b^k, a^k) < \delta) \wedge (\mathbf{T}(a^k, b^k) < \delta)\Big)\Big) \wedge \\ &\Big((\forall k \in \mathbb{N}^*, b^k \in \mathcal{I}_b)\Big((a^k \in \mathcal{I}_a) \wedge (\mathbf{T}(b^k, a^k) < \delta) \wedge (\mathbf{T}(a^k, b^k) < \delta)\Big)\Big)\end{aligned} \tag{4.10}$$

#### 4.6.3.3 CCSL Precedence-based Constraints

The clock constraint Precedence distinguishes two forms: the strict precedence and the non strict precedence. Intuitively, this means that each instant in $b$ follows one instant in $a$.

**Strict precedence**: An example of strict precedence is given by Fig. 4.6.4. $a$ strictly precedes $b$ (denoted $a \prec b$) iff:

$$(\forall i \in \mathcal{I}_b)(k = idx_b(i)) \Longrightarrow (a[k] \prec_{TS} b[k])) \tag{4.11}$$

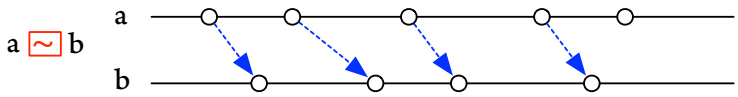



**Figure 4.6.4:** Example of Strict Precedence

Specification using real-time property pattern:

$$(\forall i \in \mathcal{I}_b)(k = idx_b(i)) \Longrightarrow (\mathbf{T}(b^k, a^k) > \delta) \tag{4.12}$$

**Precedence** An example of precedence is given by Fig. 4.6.5. $a$ precedes $b$ (denoted as $a \preccurlyeq b$) iff:

$$(\forall k \in \mathbb{N}^*, b[k] \in \mathcal{I}_b)\Big((a[k] \in \mathcal{I}(a)) \wedge (a[k] \preccurlyeq_{TS} b[k])\Big) \tag{4.13}$$

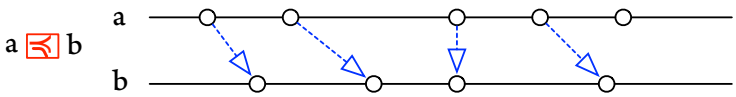



**Figure 4.6.5:** Example of Precedence

Specification using real-time property pattern:

$$(\forall k \in \mathbb{N}^*, b^k \in \mathcal{I}_b)\Bigg((\mathbf{T}(b^k, a^k) > \delta) \vee \Big((\mathbf{T}(a^k, b^k) < \delta') \wedge (\mathbf{T}(b^k, a^k) < \delta')\Big)\Bigg) \tag{4.14}$$

#### 4.6.3.4 Derived Precedence-based Constraints

**Alternation** An example of alternation is given by Fig. 4.6.6. A time structure TS satisfies $a$ `alternatesWith` $b$ (denoted as $a \sim b$) iff:

$$(\forall i \in \mathcal{I}(a))(k = idx_a(i)) \Longrightarrow (a[k] \prec_{TS} b[k] \wedge b[k] \prec_{TS} a[k+1]) \tag{4.15}$$

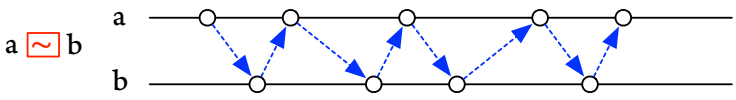



**Figure 4.6.6:** Example of Alternation

Specification using real-time property pattern:

$$(\forall i \in \mathcal{I}(a))(k = idx_a(i)) \Longrightarrow \Big((\mathbf{T}(b^k, a^k) > \delta) \wedge (\mathbf{T}(a^{k+1}, b^k) > \delta)\Big) \tag{4.16}$$

**Synchronization** An example of synchronization is given by Fig. 4.6.7. A time structure TS satisfies $a$ `synchronizesWith` $b$ (denoted as $a \bowtie b$) iff:

$$(\forall k \in \mathbb{N}^*)(a[k] \prec_{TS} b[k+1]) \wedge (b[k] \prec_{TS} a[k+1]) \tag{4.17}$$

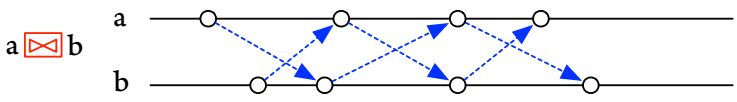



**Figure 4.6.7:** Example of Synchronization

Specification using real-time property pattern:

$$(\forall k \in \mathbb{N}^*)\Big((\mathbf{T}(b^{k+1}, a^k) > \delta) \wedge (\mathbf{T}(a^{k+1}, b^k) > \delta)\Big) \tag{4.18}$$

### 4.6.4 Specification of CCSL-based Task Level Constraints

The concept of *event* is regarded as a *clock* in CCSL. In the scheduling of reactive systems, some task temporal constraints are required. A task in the scheduling is defined as the smallest computable unit, which consumes time and modifies resources (consumes and produces). It contains two inner events, $A_S$ (starting event) and $A_E$ (ending event). A task could be executed infinitely or finitely according to the design. Task properties can be expressed as relations such as two tasks must be coincident in each period during the infinite-time scheduling.

#### 4.6.4.1 Coincidence Constraint

**Definition 4.1 (Coincidence)** *Tasks A and B are coincident iff the $n^{th}$ occurrence of A occurs simultaneously with the $n^{th}$ occurrence of B ($n \in \mathbb{N}$). It is equivalent to say that the $n^{th}$ occurrence of $A_S$ occurs simultaneously with the $n^{th}$ occurrence of $B_S$, and the $n^{th}$ occurrence of $A_E$ occurs simultaneously with the $n^{th}$ occurrence of $B_E$. In Fig. 4.6.8(a), A and B are coincident, but they are not coincident in (b) due to the overlap between $A_E^i$ and $B_S^{i+1}$.*

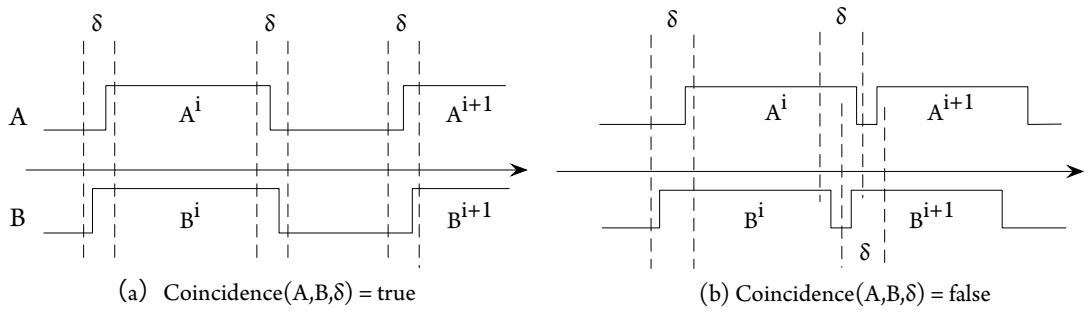



**Figure 4.6.8:** Coincidence Constraint

**Specification 4.1 (Coincidence Constraint)**

$(Freq(A_S) = Freq(B_S)) \wedge (Freq(A_E) = Freq(B_E))$

$(T(A_S^i, B_S^i) < \delta) \wedge (T(A_E^i, B_E^i) < \delta)$

$(T(A_S^{i+1}, B_E^i) > \delta) \wedge (T(B_S^{i+1}, A_E^i) > \delta)$

#### 4.6.4.2 Synchronization Constraint

**Definition 4.2 (Synchronization)** *Synchronization is a weakened coincidence relation without preventing a simultaneously execution. The only concern is that the execution order must persist. In Fig. 4.6.9(a), A and B are synchronized, but they are not synchronized in (b) due to the overlap between $A_E^i$ and $B_S^{i+1}$.*

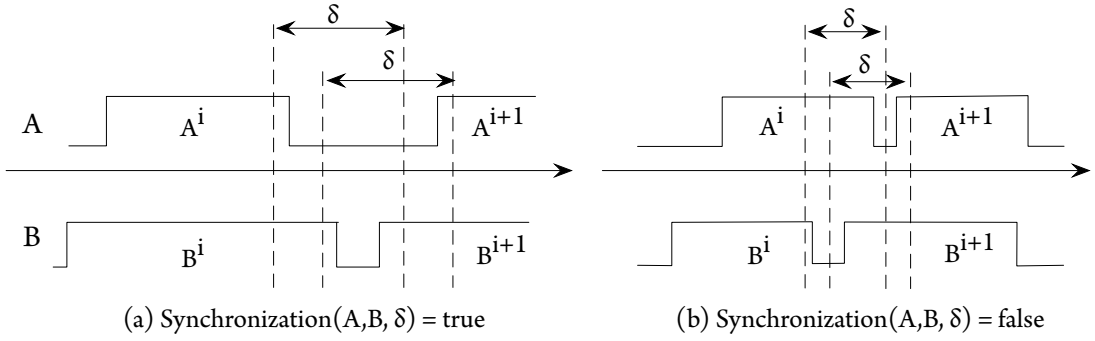



**Figure 4.6.9:** Synchronization Constraint

**Specification 4.2 (Synchronization Constraint)**
$(Freq(A_S) = Freq(B_S)) \wedge (Freq(A_E) = Freq(B_E))$
$(T(A_S^{i+1}, B_E^i) > \delta) \wedge (T(B_S^{i+1}, A_E^i) > \delta)$

#### 4.6.4.3 Exclusion Constraint

**Definition 4.3 (Exclusion)** *Task A and B are excluded iff all the presence of A does not occur simultaneously with any presence of B. It could be considered as another form of coincidence with some time offset. As In Fig. 4.6.10 (a) task A and B are excluded, but in (b) they are not excluded due to the overlap between $A_E^i$ and $B_S^{j+1}$*

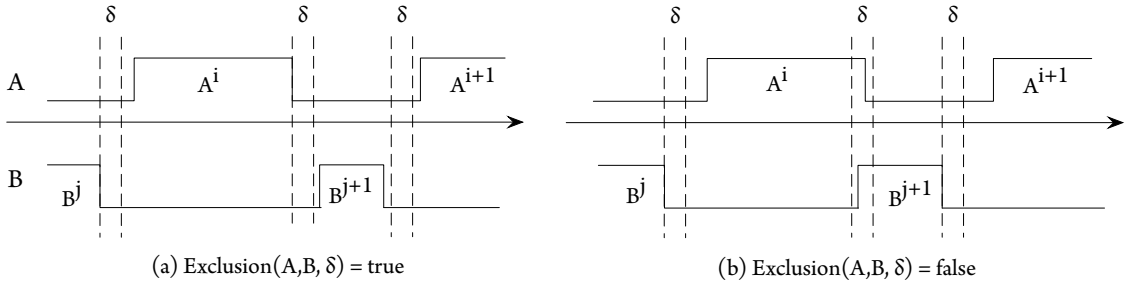



**Figure 4.6.10:** Exclusion Temporal Constraint

**Specification 4.3 (Exclusion Constraint)**
$T(B_S^j, A_S^i) > \delta \Rightarrow (T(B_S^j, A_E^i) > \delta) \wedge (T(A_S^{i+1}, B_E^j) > \delta)$
$T(B_S^j, A_E^i) > \delta \Rightarrow T(A_S^{i+1}, B_E^j) > \delta$
$T(B_E^j, A_S^i) > \delta \Rightarrow T(B_S^j, A_E^i) > \delta$
$T(B_E^j, A_E^i) > \delta \Rightarrow (T(B_S^j, A_E^i) > \delta) \wedge (T(A_S^{i+1}, B_E^j) > \delta)$

#### 4.6.4.4 Sub-occurrence Constraint

**Definition 4.4 (Sub-occurrence)** *Task B is a sub-occurrence of task A, iff B is k ($k \in \mathbb{N}^+$) times slower than A, which indicates the $i^{th}$ occurrence of A and the $j^{th}$ occurrence of B occur simultaneously, where always $j \leqslant i$. In Fig. 4.6.11 (a) B is the sub-occurrence of A, while in (b) is not due to the overlap between $A_S^{i+1}$ and $B_E^j$.*

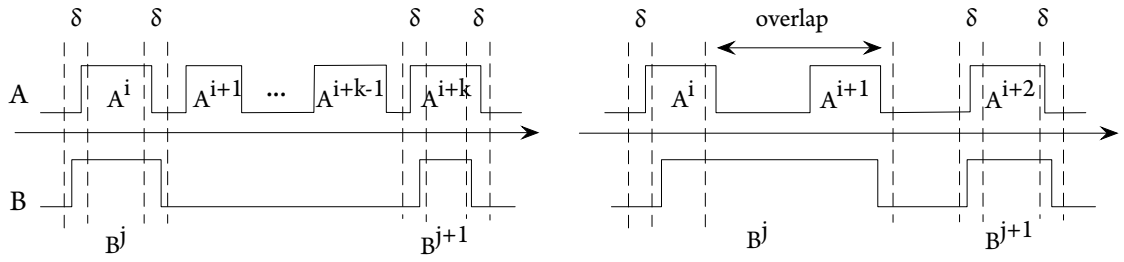



**Figure 4.6.11:** Sub-occurrence Constraint

**Specification 4.4 (Sub-occurrence Constraint)**
$(Freq(A_S^{i/k}) = Freq(B_S^j)) \wedge (Freq(A_E^{i/k}) = Freq(B_E^j))$
$(T(A_S^{i/k}, B_S^j) < \delta) \wedge (T(A_E^{i/k}, B_E^j) < \delta) \wedge (T(A_S^{i/k+1}, B_E^j) > \delta) \wedge (T(B_S^{j+1}, A_E^{i/k}) > \delta)$

#### 4.6.4.5 Precedence Constraint

**Definition 4.5 (Precedence)** *Task A precedes task B iff at any time, the occurrence time of A is superior or equal to the occurrence time of B. This implies $A_S^i$ must precede $B_S^i$, however it is not necessary to also have $A_E^i$ precedes $B_S^i$ in all context. Three strictness levels are defined, $L_1$(least strict), $L_2$(strict), $L_3$(most strict) (see Fig. 4.6.12 ).*

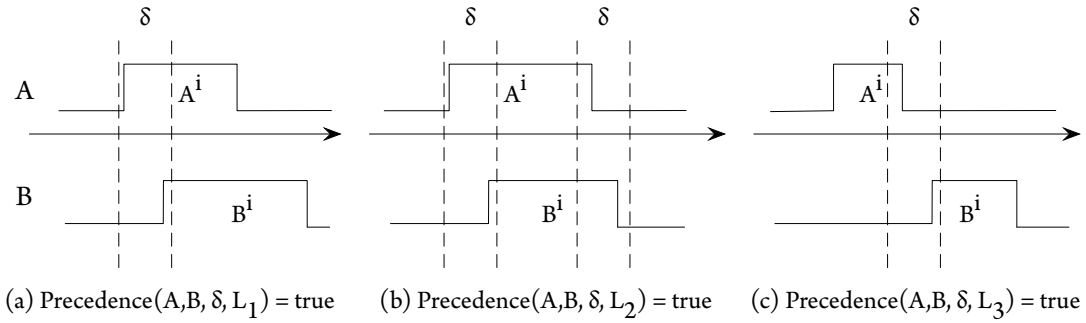



**Figure 4.6.12:** Precedence Constraint

**Specification 4.5 (Precedence Constraint)**
$L_1 : T(B_S^i, A_S^i) > \delta$
$L_2 : (T(B_S^i, A_S^i) > \delta) \wedge (T(B_E^i, A_E^i) > \delta)$
$L_3 : T(B_S^i, A_E^i) > \delta$

## 4.7 Conclusion

Classic property patterns based on Dwyer's and Konrad's pattern systems target expressiveness for the end-users that specify real-time requirements, but this usually does not ensure that they are semantically atomic or easy to verify. We define a minimal set of atomic real-time property patterns in the order to decrease the verification complexity. All end-user dedicated real-time requirements are expressed as compositions of these patterns. The proposed set of property patterns is minimal because its elements are semantically atomic and cannot be expressed as a composition of other atomic elements. We provide the translation from all possible Dwyer's and Konrad's property patterns to our property patterns. This means our atomic property patterns are semantically complete with respect to Dwyer's and Konrad's work. The common requirements based on Dwyer's and Konrad's patterns will be automatically mapped to our patterns using a predefined metamodel and a mapping library. Our property patterns decompose complex properties into a set of simpler ones that rely on a smaller state space and thus decrease the verification cost. We also rely on these elementary patterns for event-based CCSL specification and a small extension for task-based CCSL specification, which is one of the standardized specifications of timed model in MARTE. All the patterns defined in this chapter will be checked in an efficient way using the proposals in Chapter 5.

The main contributions of the current chapter are summarized as follows:

1. **Real-time requirement specification relying on real-time property patterns.**
   We have defined a minimal set of atomic real-time property patterns to specify real-time requirements. These patterns target verification easiness. We propose to specify a temporal property using occurrence modifiers, basic event modifiers, basic predicates and basic scope modifiers. The composition of these elements covers all the properties based on Dwyer's qualitative patterns and Konrad's quantitative patterns, and the CCSL part of UML MARTE standard.

2. **Automatically mapping real-time requirements to real-time property patterns.**
   All the temporal properties based on Dwyer's patterns with additional real-time suffixes can be mapped

to our patterns. The complete mapping library is provided in Appendix B. The metamodel is defined within Eclipse modeling Framework (EMF) to ease the integration with UML model. Relying on this metamodel and the mapping library, the mapping process is performed automatically.

3. **Applying real-time property patterns to CCSL-based task constraints.** [GPC12a]
   CCSL deals with logical time. Based on CCSL, we have introduced the concept of time tolerance, which is a mandatory for the real system modeling. We have translated CCSL constraints using our real-time property patterns. In the scheduling of reactive systems, some task temporal constraints are required. We propose a small extension of CCSL constraints to deal with the task level constraints, and then specify them using the real-time property patterns.

4. **Implementation of real-time property specification tool in the verification toolset.** The property specification tool has been implemented in the verification toolset. This tool allows to automatically translate real-time requirements based on Dwyer's and Konrad's work into our property patterns. It is also possible to parse and translate CCSL constraints into our property patterns. For now, the prototype of this tool does not cover all the property mapping. It will be completed in the near future.

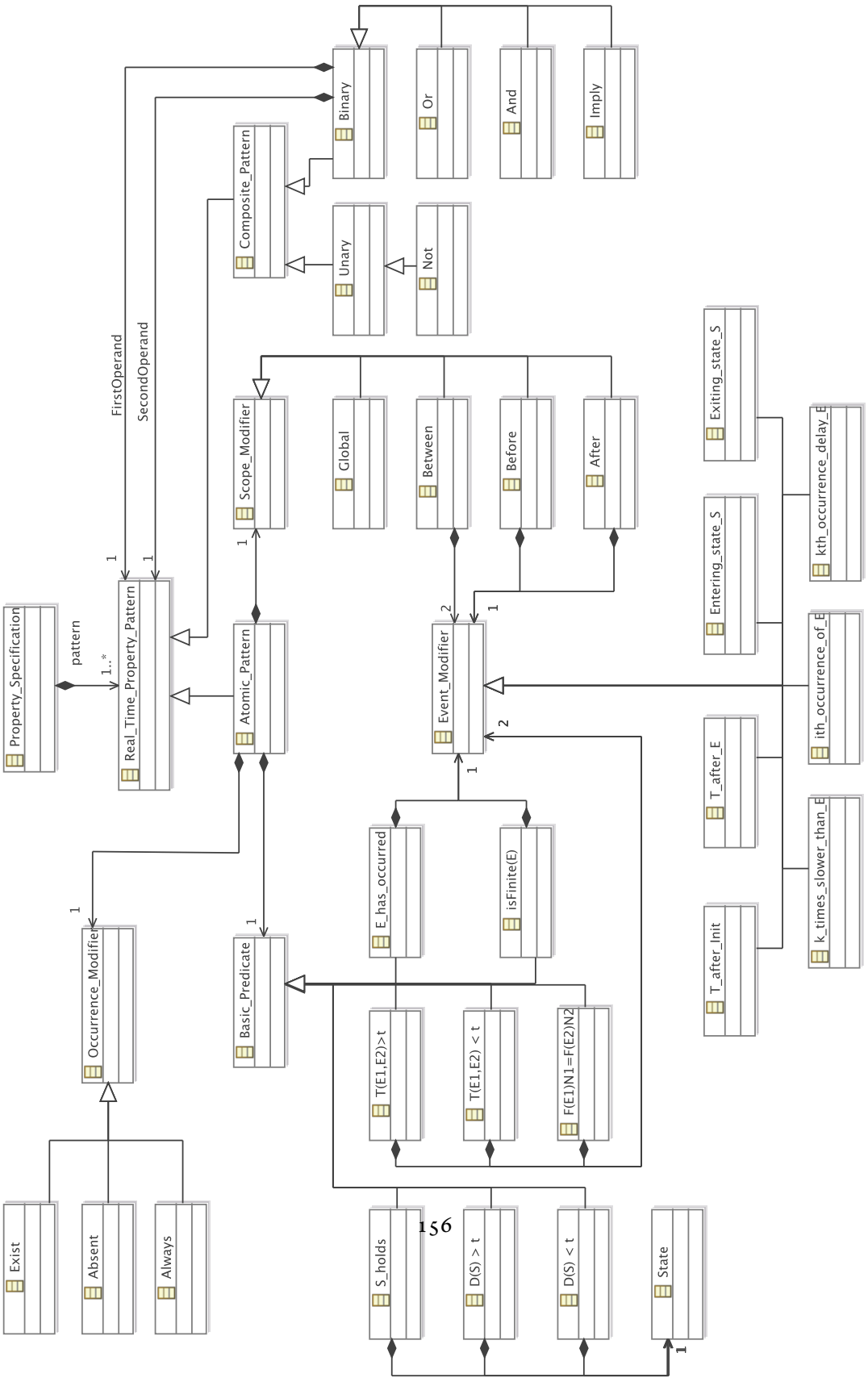



**Figure 4.7.1:** Metamodel of Temporal Property Pattern

# 5

# Property Verification based on TPN/tts Observers


## Résumé

Ce chapitre propose une approche pour la vérification de modèles spécifiés en réseaux de Petri temporisés exploitant des observateurs pour évaluer la satisfaction des patrons élémentaires de propriété temps réel proposé dans le chapitre précédent.

La plupart des exigences qualitatives exprimées par les utilisateurs industriels peuvent être traduites formellement en exploitant les patrons de propriété de Dwyer en utilisant différents formalismes logiques. De nombreux outils de vérification permettent d'évaluer les propriétés spécifiées dans ces formalismes. Les exigences quantitatives exprimées en utilisant les patrons de Konrad peuvent aussi être spécifiées de la même façon en exploitant des extensions temporisées des formalismes logiques. Ces méthodes de spécification pour les propriétés quantitatives sont intéressantes théoriquement pour étudier la complétude des formalismes, mais leur utilisation pratique est limitée en raison de la capacité des outils de vérification. Ces outils ne fournissent pas de le même support pour les propriétés quantitatives que pour les propriétés

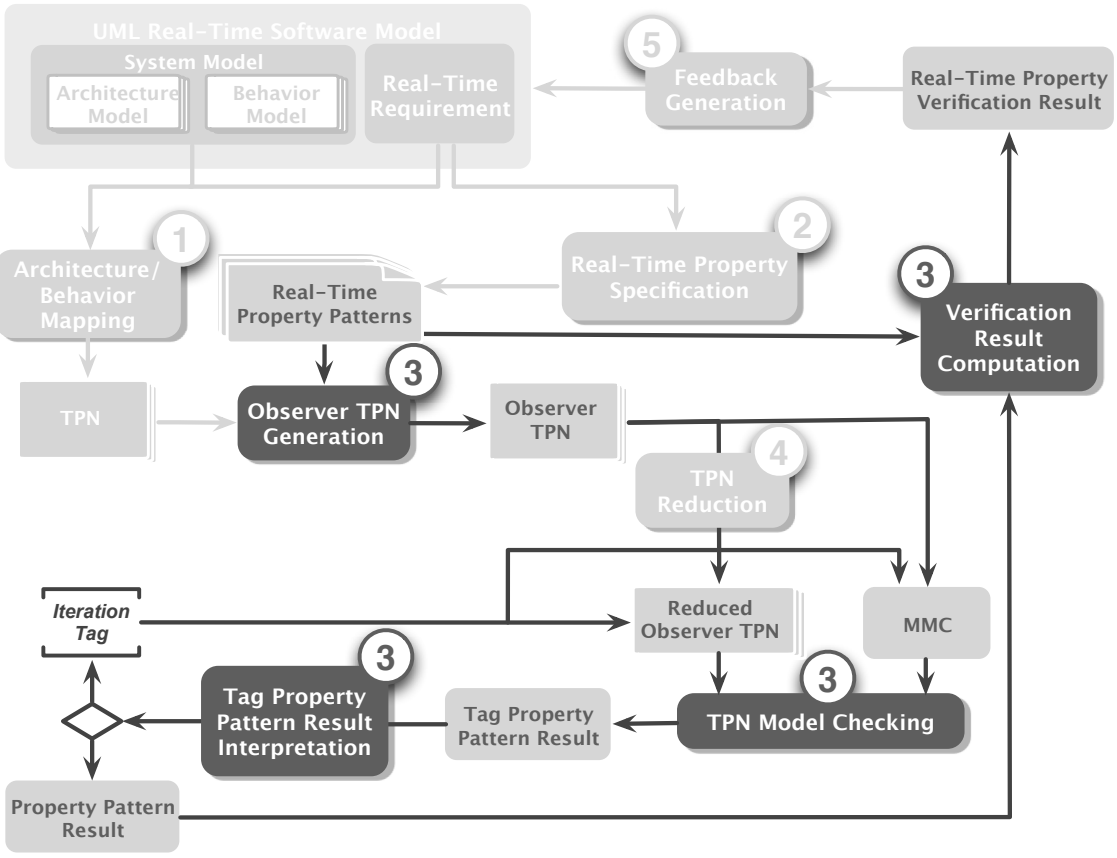
qualitatives. Habituellement, ce problème est résolu en utilisant des observateurs : des parties de modèle qui conduisent dans des états particulier si les contraintes sur le temps sont satisfaites. La satisfaction des propriétés temporisées est alors réduites à l'accessibilité de ces états . Les observateurs sont exécutés en même temps que le modèle en cours d'évaluation. L'accessibilité est alors exprimée par des formules de logique non temporise. Les contributions de ce chapitre sont :

- Les exigences temps réel sont exprimées en utilisant un ensemble minimal de modificateurs élémentaires (occurrence, événement, prédicat de base, et domaine), qui sont sémantiquement atomiques. Toutes les propriétés temps réel exprimées à l'aide des patrons de Dwyer et Konrad et une partie de celles exprimées en CCSL peuvent être traduites en ces modificateurs élémentaires. L'ensemble d'observateurs proposés dans ce chapitre correspond à ces modificateurs élémentaires. Pour préciser les deux dernières phrases: Ces observateurs sont atomiques et ne peuvent pas être exprimés avec les autres observateur. Les autres observateurs sont obtenus par composition des observateurs élémentaires. L'ensemble des observateurs est minimal et complet par rapport aux patrons de propriété définis dans le chapitre 4.

- Etant donné qu'ils sont ajoutés dans le modèle du système, les observateurs peuvent augmenter la taille de l'espace d'état, parce qu'ils introduisent de nouveaux états et de nouvelles transitions. Ces structures d'observation supplémentaires, bien que que ne modifiant pas la sémantique originale du système, peuvent changer la taille de l'espace d'états, et du graphe des classes d'états généré et donc influencer sur les performances de la vérification. Il est donc nécessaire de concevoir des observateurs qui minimisent cet impact. Par contre, cela peut demander d'exploiter des caractéristiques des réseaux de Petri temporisés qui peuvent réduire les capacités d'abstraction du graphe d'état, ce qui conduit à un coût beaucoup plus élevé pour la vérification. Par conséquent, la conception de ces observateurs doit effectuer un compromis entre ces deux aspects. Dans cette thèse, la priorité est de préserver le niveau le plus élevé d'abstraction du graphe d'état, puis de minimiser ensuite le nombre de transitions et d'états introduits par les observateurs.

- Lors d'une vérification de modèle classique, l'utilisateur attend une réponse positive ou négative concernant la satisfaction de l'exigence exprimée. Pour les propriétés quantitatives, toutefois, les utilisateurs souhaitent souvent déterminer les bornes exactes de satisfaction d'une propriété et pas seulement vérifier que les bornes proposées sont correctes. Nous proposons une méthode itéra-

tive dichotomique pour approcher progressivement les bornes exactes en combinant la vérification à base d'observateurs avec un moteur de recherche dichotomique.




*Progress Map 3:* Observer-based Real-Time Property Verification

In this chapter, we introduce a TPN/tts observer-based verification approach to assess the satisfaction of real-time property patterns defined in the previous chapter (Progress map 3). Temporal logics such as TCTL can be used to express real-time properties, but are limited in practice due to the capability of model checking tools we are relying on (Challenge 3 in page 22). Usually the end users are more capable to use observers than logical formulae. Our approach adds 12 event-based TPN observers and 4 state-based observers in the observed system to check each real-time property pattern using the accessibility assertions in the modal $\mu$-calculus (MMC) and the muse model checker from the TINA toolset. This observer-based verification approach takes advantage of high abstraction of state class graph to minimize model checking complexity, and thereby improve the verification efficiency. This verification approach can also be used as to search for the bound values for properties such as BCET/WCET. This may help the users improve and refine

their design model. (Contribution 3 in page 24)

## 5.1 Introduction

A qualitative time requirement can be mapped to Dwyer's property patterns using various logic formalisms. For example, the property *Exist S responds to P* is specified as $EG(P \Rightarrow AF(S))$ in CTL or as $\Diamond(P \Rightarrow \Diamond S)$ in LTL. Many model checking tools allow to check properties specified in these formalisms. Therefore, qualitative time properties expressed using Dwyer's patterns can be assessed with the help of these tools.

A quantitative time requirement expressed using Konrad's patterns can also be specified in a similar way. For example, the time-bounded response property *Exist S responds to P within k t.u.* can be specified as $EG(P \Rightarrow AF_{<k}(S))$ in TCTL. Such a specification method for quantitative properties are interesting for theoretical purpose, but its practical use would be limited due to the capability of model checking tools we are relying on, as these tools do not provide the same support for timed temporal logics as for untimed temporal logics.

Usually, the above problem is solved using observers. Observers are executed concurrently with the model under assessment. Some transitions will be fired and the expected states will be reached if and only if some timed conditions are satisfied. These behaviors will be observed to check the satisfaction of real-time properties.

In Chapter 3, we have mapped UML-MARTE to TPN models, used to assess the expected real-time properties. In this chapter, we present the real-time property verification approach based on TPN observers. The observer approach is commonly used in model checking. A similar work is carried out by Abid et al. [AZB13]. Before comparing the work of Abid and our works, the concept of Time Transition System should be precised. We have presented in the state of the art (see Section 2.3.2) that TINA toolset allows data handling on TPN based on classical imperative programming by adding common features like guard variable definition and modification to each transition. The variable's value set is composed of an extension of reachability graph's state, which unifies the verification processes and makes it transparent to the TPN user while enlarging the modeling capacity. The TPN with data handling and priority extension is called Time Transition System (tts). Each transition in tts has two associated functions:

- Pre represents a transition guard: the transition will be enabled only when both TPN's marking pre-condition and the guard are satisfied.

- ACT is performed when the transition is fired. It can modify the data that are used to compute the guards.

In the work of Abid et al., a set of observers was defined using the expressiveness of tts. They did not give an automatic method to generate observers. Rather, they defined a set of observers at tts level for their property patterns. After selecting the "most efficient ones", they proved that the observers were correct. Their observers largely rely on the priority arc in tts. This degrades the abstraction level when generating state class graph for model checking. In our work, we distinguish the state-based and event-based observers. According to the real-time property patterns defined in Chapter 4, most of the real-time requirements target the events, and the others target the states. We design and implement the event-based observers at TPN level (12 observers), the state-based observers at tts level (4 observers), and select the "most efficient ones". The criteria for efficiency is the time and resource consumed by a TPN system and its observers. Our observers take advantage of the highest abstraction (marking abstraction in our case) provided by TINA toolset to minimize the size of the state space in the generated state class graph.

Some related works defined TPN observers for verifying some specific properties. The work of [GDRA+12] proposed a worst case time interval TPN observer. However, this observer added extra semantics to the observed system, which leaded to a change on the system's original behavior. According to our understanding, this extra semantics should be avoided. Compared to this work, our work focuses on a set of observers, which can be used to verify all the real-time requirements specified using the pattern system of Dwyer and Konrad or CCSL. From this point of view, the proposed set of observers is complete.

The contributions of this chapter are summarized as follows:

- **First, the proposed set of atomic observers is complete and minimal.** As presented in Chapter 4, a real-time requirement is expressed using a set of elementary modifiers (occurrence, event, basic predicate and scope modifiers), which are semantically atomic. The set of elementary modifiers is minimal, which means all the real-time properties expressed using Dwyer's and Konrad's patterns or CCSL can be specified using our elementary modifiers. The proposed set of observers corresponds to these elementary modifiers. Therefore, the observers are atomic and composable. The set of observers is minimal and complete.

- **Second, the observers can minimize the size of the state space in the generated state class graph.** As an additional part added in the original system, the observers will increase system's original state space, because they introduce some possible execution traces related to the observation.

These additional observer structures, although do not add extra semantics or remove original semantics for system's behavior, they can change the size of state space of the generated state class graph and thus impact the performance of the model checking. The less transitions an observer is composed of, the better performance will the model checking achieve. Nevertheless, minimizing the transitions in an observer may require to use some TPN/tts features which may degrade of abstraction level of state class graph. The lower the abstraction level is, the worse the performance will be. Therefore observer design must promote a trade-off between the above two aspects. In our approach, the priority is keeping the highest abstraction (marking abstraction) of the state class graph, then minimize the transitions in the observers. We rely on the $\mu$-calculus and the muse model checker from the TINA toolset to assess the real-time properties based on our proposed observers.

- **Third, use observers to compute the bound values of quantitative property.** When performing model checking, an observer can give an answer such as YES or NO for the satisfaction of the given property. For quantitative properties, however, users usually expect to know, instead of *whether the property is bounded by* $[t_{min}, t_{max}]$, that the exact bound of the property$[t_{min}, t_{max}]$. In order to fill this gap of usage, we propose an iterative dichotomy method that will gradually approach the bound value by combining the observer with a binary search engine.

We introduce the catalog of TPN/tts observers in Section 5.3; present the verification process using a running example in Section 5.4; present the computation benefit of the observer approach in Section 5.5; at last we explain the method used to guarantee verification scalability in Section 5.6.

## 5.2 Design Principles of TPN/tts Observers

### 5.2.1 Structure of Observer

A TPN/tts observer is a complementary structure linked to or independent from the original system. As shown in Fig. 5.2.1, in order to assess an event-based real-time property, a TPN/tts observer is associated with the original system, by adding arcs to the transition $T_A$ in component A and the transition $T_B$ in component B. If the real-time property is based on states, the tts observers are independent from the system (represented by the dotted links in Fig. 5.2.1). The observer contains a place $P_{tester}$, which can assess that the real-time property is satisfied using the accessibility assertions in the state-event modal $\mu$-calculus

(MMC) formulae in ktz format. MMC allows to check whether a marking exists and whether a transition is fireable in the ktz. According to the real-time requirement, one or more MMC formulae are generated, such as $\Box(P_{tester} = 1)$ or $\Diamond(P_{tester} = 0)$, etc. The MMC formulae will be checked on-the-fly using the muse model checker from the TINA toolset.

We do not use the LTL formulae. To use LTL, we need to generate the state class graph preserving LTL semantics, which is less abstract than the state class graph preserving marking semantics and thus will decrease the analysis efficiency.

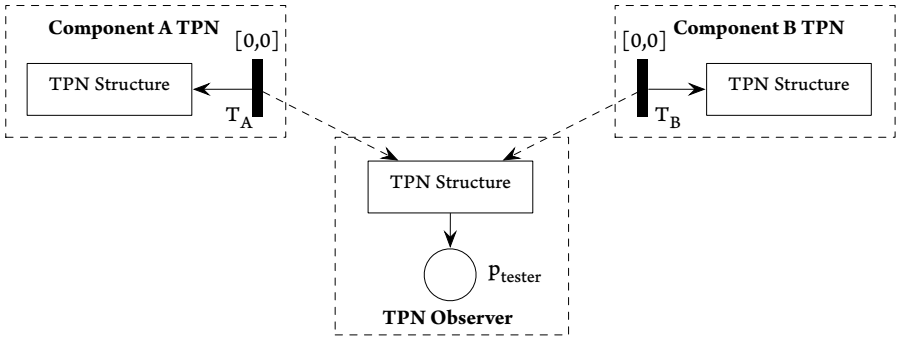



**Figure 5.2.1:** Observer Structure

### 5.2.2 Soundness of Observer

The soundness here means that the observers should not impact the system's behavior by introducing extra semantics or removing original semantics. We ensure this principle by designing the TPN observers associated with the TPN transitions, but not TPN place. Indeed, associating with places would add or remove tokens and may change system's execution traces.

The manner that the observers are associated with the transitions in this approach does not impact the system behavior. As shown in the observer structure of Fig. 5.2.1, the TPN observer links from the transitions of the system components. It works as a "read-only" mode. Thus, no extra TPN tokens or time constraints will be added to the original system. As the tts observers are independent from the system, it will not introduce extra behavioral semantics to the system either.

### 5.2.3 Efficiency of Observer

As model checking may require a huge amount of resource, the design and implementation of the efficient observers must follow four principles to ensure the generation of smaller state space:

- First, the system model with integrated observers should be able to perform the possible highest abstraction (marking abstraction in our case) when generating the state class graph. This high abstraction state class graph must preserve all the information related to the verification of the targeted property.

  This principle is ensured by designing the observers using the TPN/tts features acceptable by the state class graph preserving marking. For example, we cannot use the priority arc, which is an extended element of tts. It allows to set priorities between the TPN transitions, and this feature makes the design of observers much easier. However, when generating the state class graph at marking abstraction level using the tina tool, the priority arcs cannot be preserved. Therefore, in our work, we design observers without priority arcs to ensure that the marking abstraction can be performed. As we verify the properties using observers, this transforms quantitative problems to reachability problems. We use the $\mu$-calculus (MMC) as the logic formalism in the model checker muse. The reachability graph generating from the marking abstraction contains all the mandatory reachability information. In other words, it preserves all the information related to the verification of the targeted real-time properties.

- Second, the state/transition number of the added TPN/tts observers should be as small as possible to minimize the size of the state space of the system integrated with observers.

  There exist many different methods to define the same observer. The use of some kinds of TPN arcs may ease the design, but may also generate more states/transitions in the state class graph. There are five kinds of arcs in TPN: regular, read, inhibitor, stopwatch and stopwatch inhibitor arcs. The stopwatch and stopwatch inhibitor arcs may increase the state/transition graph size. Therefore, in our work, we do not use these two to reduce the size of the state space.

- Third, the checking of each property pattern must be independent in terms of state class graph generation to promote parallel computation. This principle is also used to ensure the performance of the property verification.

## 5.3 Catalog of TPN/tts Observers

According to the requirement specification patterns defined in Chapter 4, an atomic real-time property consists of three parts: an occurrence modifier, a predicate and a scope. Predicates are based on events and states, while scopes are only based on events. The observers are supposed to have three classifications: event modifier observer, basic predicate observer and scope modifier observer. These three kinds of observers are specified in this section. The occurrence modifiers are not used as observers, their uses are illustrated after the definition of observers.

### 5.3.1 Event Modifier Observers

Predicates are based on events and states. An event can be atomic or composite ones. We regard the event compositions as event modifiers. An example of composite event is given here: *t t.u. after event* $E^{i-k}$. This composite event is built from three event modifiers: $E^i$ (the $i^{th}$ occurrence of event $E$), $E^{i-k}$ (the $k^{th}$ occurrence delay of current event $E^i$), and $E^{i-k} + t$ (the *t t.u.* from current event $E^{i-k}$).

To ease the composition of event modifiers, first of all we define a generic event observer pattern (see Fig. 5.3.1). $E$ is an observable transition (standing for an event) in the system model. The frame *Observer* contains the observer structure that links transition $E$ to transition $E'$. $E'$ stands for the target event modifier of the observer, and it works also as an extensible transition used to connect other observers if required.

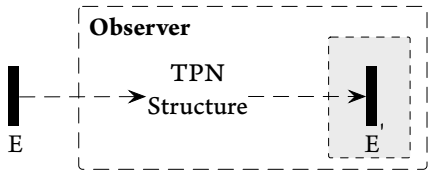



**Figure 5.3.1:** Generic Observer Pattern

#### 5.3.1.1 $E^i$: $i^{th}$ Occurrence of $E$

When using this modifier, the occurrence of $E$ under observation must be finite under the observed execution. The observer in Fig. 5.3.2 represents this event modifier. When $E$ has occurred $i$ times, the place $P_{occ}$ will have accumulated $i$ tokens, then the transition $E^i$ will be enabled. This makes the transition

$E^i$ become the $i^{th}$ occurrence of event $E$. The place $P_{once}$ with one token controls the occurrence times of $E^i$. By default, $E^i$ occurs only once. The place $P_{once}$ can also be removed to allow $E^i$ occurring more than once.

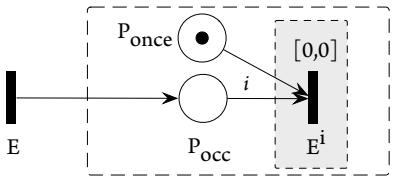



**Figure 5.3.2:** Event Observer: $i^{th}$ Occurrence of $E$

#### 5.3.1.2 $E^{-k}$: $k^{th}$ Occurrence Delay of $E$

The transition $E^{-k}$ stands for the delay of $k$ times occurrence of event $E$. In other words, $E^{-k}$ represents the event that delays $k$ times occurrence compared to $E$. In Fig. 5.3.3, the place $P_{occ}$ contains the tokens that represent the number of occurrence of $E$. Each time $P_{occ}$ has accumulated $k$ tokens, the read arc will enable the transition $E^{-k}$ that will consume one token from $P_{occ}$ marking.

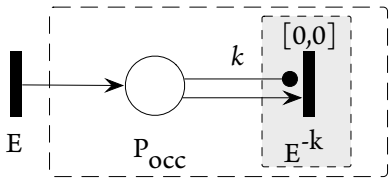



**Figure 5.3.3:** Event Observer: $k$ Times Occurrence Delay of $E$

#### 5.3.1.3 $E^{/k}$: $k$ Times Slower Sub-occurrence of $E$

The transition $E^{/k}$ stands for the sub-occurrence of event $E$, with a frequency $k$ times slower than $E$. When $E$ occurs $k$ times, $E^{/k}$ will be enabled once. In Fig. 5.3.4, each time when the place $P_{occ}$ accumulates $k$ tokens, the transition $E^{/k}$ is fired. Simultaneously, all the $k$ tokens in place $P_{occ}$ are consumed.

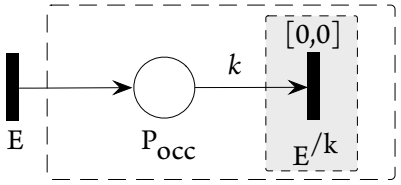



**Figure 5.3.4:** Event Observer: Sub-occurrence $k$ Times Slower than $E$

#### 5.3.1.4 I+t: Time Passed Since System Initialization

**I+T** stands for the absolute time instant measured from the initial state of the system. In Fig. 5.3.5, the observer consists of two parts. One is the place $P_{Init}$ that stands for the initialization of the whole system. As $P_{init}$ has not ingoing arcs, it starts at the same instant as all other initial places of the system. The other is the transition $E'$ that represents the moment when time has passed $t$ *t.u.* since system's initialization.

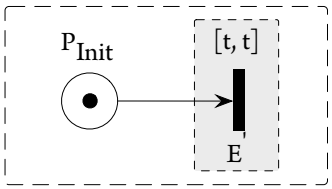



**Figure 5.3.5:** Event Observer: Time Passed since System Initialization

#### 5.3.1.5 E+t: Time Passed Since $E$

$E + t$ stands for the moment when $t$ *t.u.* has passed since each occurrence of $E$ (see Fig. 5.3.6).

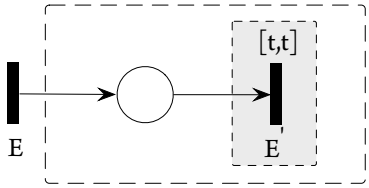



**Figure 5.3.6:** Event Observer: Time Passed since $E$

#### 5.3.1.6 $S^S$ & $S^E$: Entering and Exiting Events of a State

This observer is used to represent the entering and exiting events of a given system state. It uses the date manipulation functions of `tts`. A state referred to in the observer corresponds to a marking assertion. In Fig. 5.3.7, the transitions $S^S$ and $S^E$ represent respectively the entering and exiting events of state S. When a system enters the targeted state $S$, the assertion $S$ in the PRE condition of the transition $S^S$ is true, which will enable $S^S$. The token in place $P_S$ transits to place $P_E$. Similarly, when the system exits state $S$, the the assertion $\neg S$ in the PRE condition of the transition $S^E$ becomes true, thus $S^E$ is enabled.

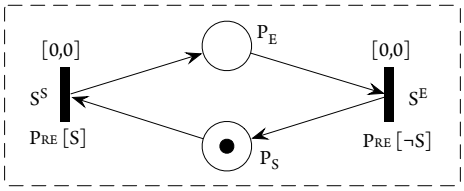



**Figure 5.3.7:** Event Observer: Starting and Ending Event of S

### 5.3.2 Basic Predicate Observers

The pattern of predicate observers is defined in Fig. 5.3.8. The transition $E_M$ is a(n) (composite) event modifier. The `TPN` structure stands for the observer structure. Each predicate observer is verified using one or several `MMC` assertions.


**Figure 5.3.8:** Predicate Observer Pattern

#### 5.3.2.1 $\mathbf{O}(E^i)$ = true: $E^i$ has occurred

In Fig. 5.3.9, the place $P_{occ}$ linked from transition $E_M$ is used to observe the occurrence times of an event. Once the transition $E_M$ is fired, the token in place $P_{occ}$ will be observed. The `MMC` assertion $P_{occ} \geq i$ is used

to check whether $E_M$'s $i^{th}$ occurrence has occurred.

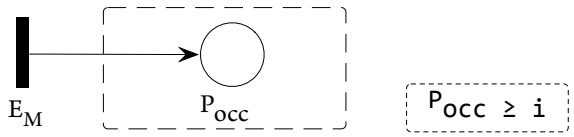



**Figure 5.3.9:** Predicate Observer: Occurrence of $E^i$

#### 5.3.2.2 isFinite($E$) = *True*: Bounded Occurrence of $E$

This predicate is used to detect whether the occurrence of an event is finite. In Fig. 5.3.10, the place $P_{occ}$ accumulates the occurrence times of event $E_M$. If the transition *Overflow* is not fired, it signifies no overflow is detected, because $E_M$ does not exceed the bound $Occ_{max}$ (a predefined threshold value). We conclude that $E_M$ is bounded during system's execution.

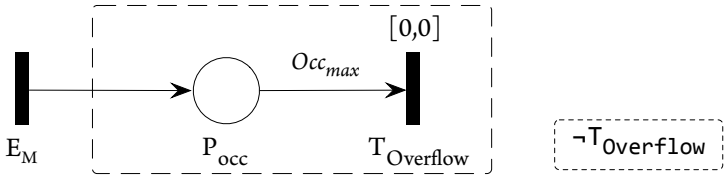



**Figure 5.3.10:** Predicate Observer: Occurrence of $E$ is bounded

#### 5.3.2.3 Freq($E_A$) · $N_A$ = Freq($E_B$) · $N_B$: Equivalent Occurrence between $E_A$ and $E_B$

This predicate is used to identify equivalent occurrences between two periodic events with different (or equal) frequencies. Suppose two periodic events $E_A$ and $E_B$ exhibit respectively occurrence frequency $F_A$ and $F_B$. There exists minimal coefficients $N_A$ and $N_B$ ($N_A, N_B \in \mathbb{Z}^+$) that makes $F_A \cdot N_A = F_B \cdot N_B$. $N_A$ and $N_B$ can be computed using the Least Common Multiple (lcm) and the Greatest Common Divisor (gcd).

$$N_A = \frac{\texttt{lcm}(F_A, F_B)}{\texttt{gcd}(\texttt{lcm}(F_A, F_B), F_A)} \tag{5.1}$$

$$N_b = \frac{\texttt{lcm}(F_A, F_B)}{\texttt{gcd}(\texttt{lcm}(F_A, F_B), F_b)} \tag{5.2}$$

A real-time property may require to limit the time difference between two periodic events. If these two events exhibit the same frequency, $N_A$ and $N_B$ are equal. Otherwise, $N_A$ and $N_B$ should be introduced to identify the corresponding occurrence between $E_A$ and $E_B$.

In Fig. 5.3.11, places $Tester_A$ and $Tester_B$ accumulate respectively the occurrence times of $E_A$ and $E_B$. The tokens in *Tester* places will be consumed through the transition *Diff* when the tokens in $Tester_A$ are superior or equal to $N_A$ and the tokens in $Tester_B$ are superior or equal to $N_B$. Once $Tester_A$ contains $N_A + 1$ tokens, it stands for $E_A$ executes at least one occurrence faster than $E_B$. This exception will be detected using the *Overflow* transitions. The checking assertion is: $\neg(Overflow_A \vee Overflow_B)$.

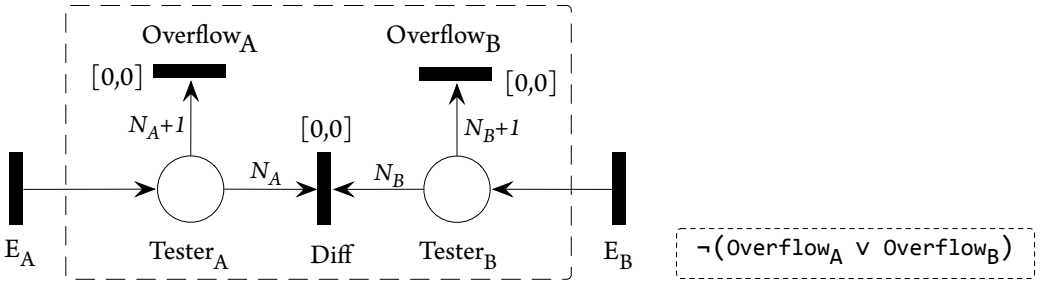



**Figure 5.3.11:** Predicate Observer: Same Frequency between $E_A$ and $E_B$

#### 5.3.2.4 T($E_A$, $E_B$) > t: Minimum Time Interval between Events

This observer is used to check that the time interval between the equivalent occurrences of $E_A$ and $E_B$ is at least *t*. $E_A$ and $E_B$ can be periodic or aperiodic. Semantically, it is equivalent to T($E_A$) - T($E_B$) > *t*. It has a similar structure as the observer for equivalent occurrence between events, except that a transition $T_{Delay}$ is added. $T_{Delay}$ stands for the time delay for event $E_A$. The following MMC assertion should be satisfied to check this predicate: $\neg(Overflow_A \vee Overflow_B) \wedge \neg((Tester_B = N_B) \wedge (Tester_A < N_A))$. When $E_A$ and $E_B$ are aperiodic, $N_A = N_B = 1$.

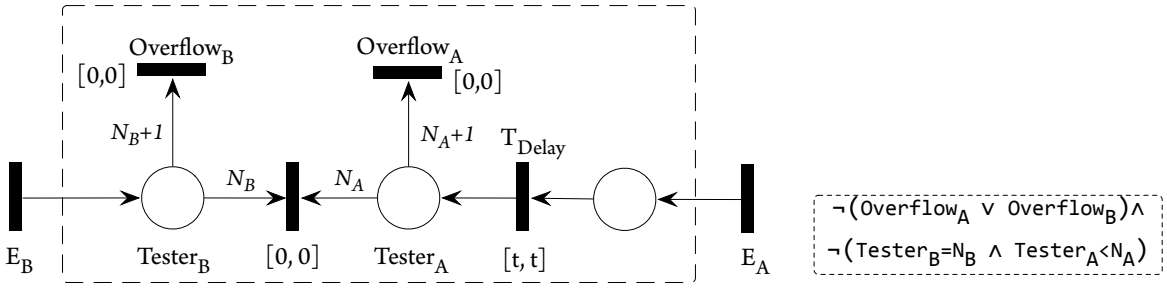



**Figure 5.3.12:** Predicate Observer: Minimum Time Interval between $E_A$ and $E_B$

#### 5.3.2.5 $T(E_A, E_B) < t$: Maximum Time Interval between Events

This observer is used to check the time interval between the equivalent occurrences of $E_A$ and $E_B$ is at most $t$. Semantically, it corresponds to $T(E_A)$ - $T(E_B) < t$. The following assertion should be satisfied: $\neg Overflow_B$.

If the assertion $\neg(Overflow_A \vee Overflow_B)$ is true, then $|T(E_A) - T(E_B)| < t$ is satisfied. When $E_A$ and $E_B$ are aperiodic, $N_A = N_B = 1$.

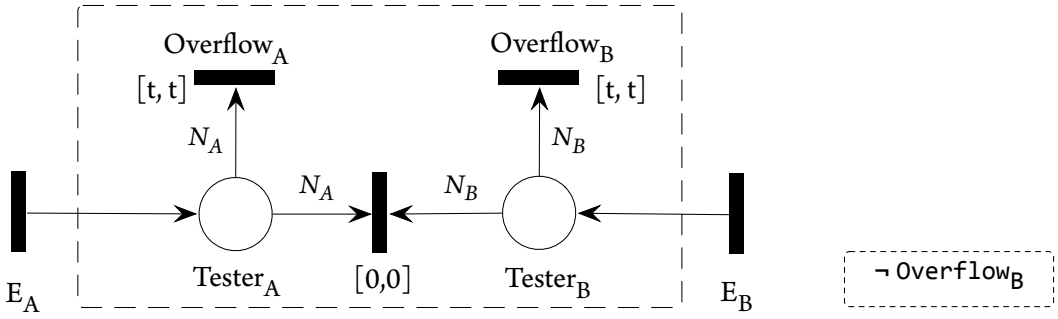



**Figure 5.3.13:** Predicate Observer: Maximum Time Interval between $E_A$ and $E_B$

#### 5.3.2.6 $D(S) \geq t$: Minimum Time Duration of State

The most direct and efficient way of designing observer for state duration is to use the PRE function of TPN. In Fig. 5.3.14, the transition with constraint [t,t] will fire when state $S$ holds at least $t$ *t.u.*. The transition with constraint [0,0] will fire when state $S$ does not hold any more. This transition is used to

clear the marking in *Tester* place, because state *S* may hold several times in the whole system's execution. The MMC assertion of checking is: $S \wedge (Tester = 1)$

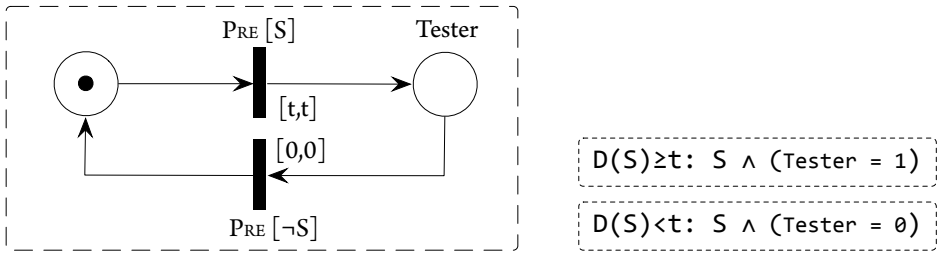



**Figure 5.3.14:** Predicate Observer: Time Duration of State

#### 5.3.2.7 D(*S*) < *t*: Maximum Time Duration of State

The maximum time duration of state uses the same observer as the minimum time duration (Fig. 5.3.14), but different assertion: $S \wedge (Tester = 0)$.

### 5.3.3 Scope Modifier Observers

Scope modifiers include Global, Before $E^i$, After $E^i$, and Between $E_A$ and $E_B$.

#### 5.3.3.1 Global

Global scope modifier does not need an observer in TPN. When applied in the verification, it is sufficient to indicate that the scope is all states of the whole TPN, denoted as $\mathcal{A}$.

#### 5.3.3.2 Before $E^i$ & After $E^i$

The scopes before and after are represented by the same observer (Fig. 5.3.15) but different logic formulae. The place *Tester* accumulates the occurrence times of event *E*. We use $Tester < i$ ($E^i$ has not yet occurred) to check Before $E^i$ and use $Tester \geq i$ ($E^i$ has occurred) to check After $E^i$.

#### 5.3.3.3 Between $E_A$ and $E_B$

**Between** $E_A$ **and** $E_B$ means between the equivalent occurrences of $E_A$ and $E_B$. If both $E_A$ and $E_B$ are periodic events, their occurrence frequencies must be equal. If $E_A$ and $E_B$ occur only one time, by default

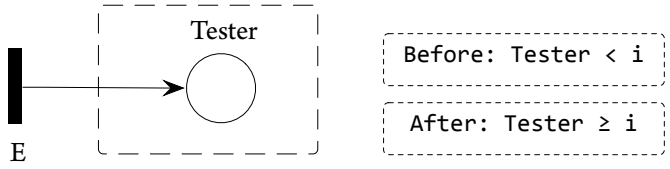



**Figure 5.3.15:** Scope Observer: Before E & After E

their frequencies are equal.

In Fig. 5.3.16, the places $Tester_A$ and $Tester_B$ accumulates the difference of the occurrence times between $E_A$ and $E_B$. The observer $(Tester_A = 1) \wedge (Tester_B = 0)$ stands for Between $E_A$ and $E_B$.

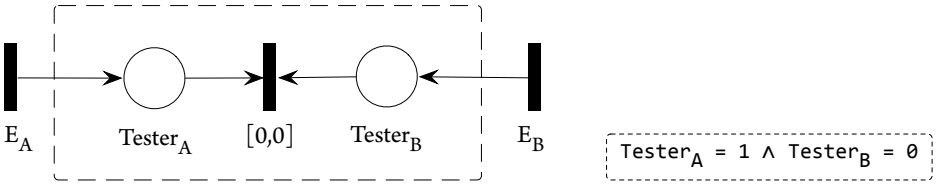



**Figure 5.3.16:** Scope Observer: Between two Events

### 5.3.4 Occurrence Modifiers

An occurrence modifier can be *Exist, Absent, and Always*. It is used together with predicates and scopes to assess a real-time property. The use of observers is not mandatory. Assume that in the state class graph, $N(P)$ is the number of states that match the predicate $P$, $N(S)$ is the number of states that match the scope $S$, and $N(P \wedge S)$ is the number of states that match both the predicate and the scope. According to the semantics of *Exist, Absent, Always* defined in Chapter 4, we have the following assertions:

- **Exist** *Predicate* in *Scope*: $\begin{cases} N(P \wedge S) \geq 1 & \text{if } N(S) > 0; \\ True & \text{if } N(S) = 0. \end{cases}$
- **Absent** *Predicate* in *Scope*: $N(P \wedge S) = 0$
- **Always** *Predicate* in *Scope*: $N(P \wedge S) = N(S)$

**Note**: When $N(S) = 0$, according to classical semantics [1], the predicates for Exist, Absent and Always should be true. The assertion of Absent and Always satisfies this definition by default. The assertion of Exist is extended by adding the assertion "True, if N(S)=0".

## 5.4 Observer-based Verification Example

We illustrate the observer-based verification method using a simple example (see Ex. 5.1).

**Example 5.1 (Observer-based Verification Example)** *In Fig. 5.4.1, two concurrent processes are specified in the TPN model. Both of them execute only once. The desired real-time property $\mathcal{P}$ is **Always** $E_A$ **After** $E_B$ **Within** $[1, 2]t.u.$.*

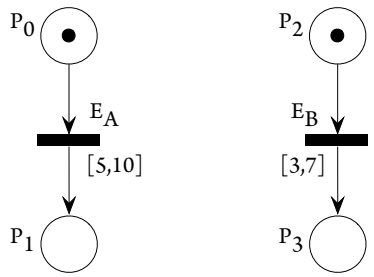



**Figure 5.4.1:** Observer-based Verification Example

### 5.4.1 Example Verification

The first step is to verify whether $\mathcal{P}$ is satisfied using observer-based approach. $\mathcal{P}$ is a safety property. It can be mapped to a real-time property pattern presented in Chapter 4, where the occurrence modifier is *Always*, predicate is $E_A$ *occurs*, and scope modifier is *After* $E_B$ *Within I*. The scope *After* $E_B$ *Within* $[t_{min}, t_{max}]$ should be mapped to *Between* $E_B + t_{min}$ *and* $E_B + t_{max}$ (consulting Appendix B).

According to the observers presented above, we add a composite observer on $E_B$ and an atomic observer on $E_A$ (see 5.4.2). The composite observer consists of three atomic observers:

[1] If the scope is false, the predicate is always true.

- **obs**$_1$: $T_1$ linked from $E_B/P_4$ stands for event modifier $E_B + t_{min}$,
- **obs**$_2$: $T_2$ linked from $E_B/P_5$ stands for event modifier $E_B + t_{max}$,
- **obs**$_3$: $P_6/T_3/P_7$ linked from $T_1/T_2$ observes the scope modifier *Between* $T_1$ *and* $T_2$.

**obs**$_4$ is the atomic observer associated to $E_A$. It observes the occurrence of $E_A$. Then the property $\mathcal{P}$

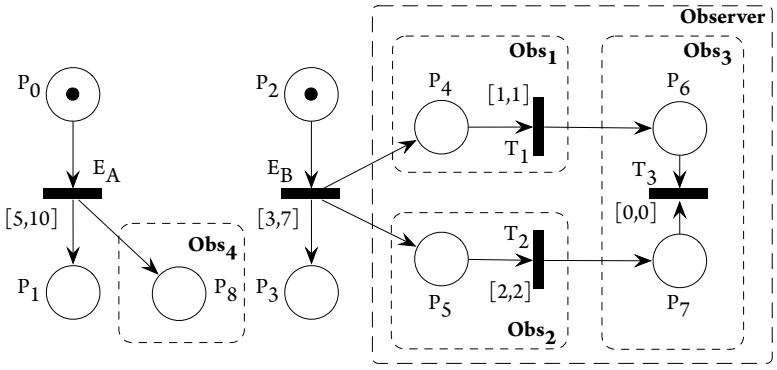



**Figure 5.4.2:** Verification of Example

(**Always** $E_A$ **After** $E_B$ **Within** [1,2]) will be verified using the following MMC formulae:

- The predicate ($E_A$ occurs) assertion $P$ is $P_8$.
- The scope (**After** $E_B$ **Within** [1,2]) assertion $S$ is $P_6 \wedge \neg P_7$.
- According to the definition of occurrence modifier **Always**, if $N(P \wedge S) = N(S)$, $\mathcal{P}$ is satisfied.

## 5.4.2 Verification Result

The above MMC assertion is checked on the TPN model (with observers) using the model checker muse. Marking abstraction is used to generate state class graph. It generates ktz format reachability graph with 10 states and 13 transitions (Fig. 5.4.3)). When checking assertion $P \wedge S$, there is one satisfied state ($S_5$), thus $N(P \wedge S) = 1$. When checking assertion $S$, there are 2 satisfied states ($S_4$ and $S_5$), thus $N(S) = 2$. As $N(P \wedge S) \neq N(S)$, property $\mathcal{P}$ fails.

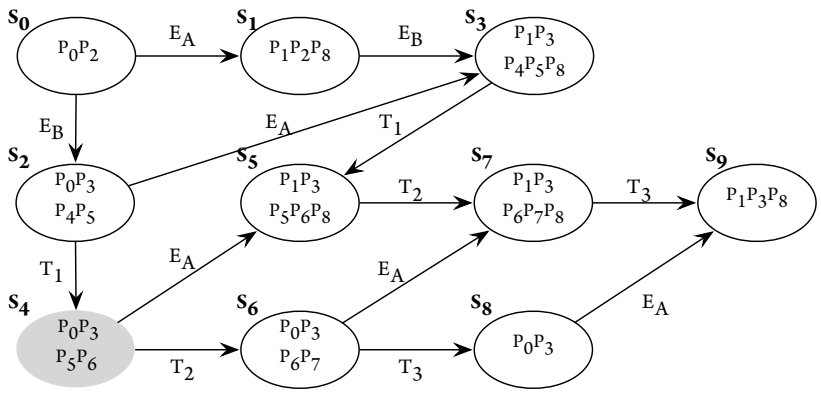



**Figure 5.4.3:** Reachability Graph of Verification Example

$N(P \wedge S) < N(S)$ implies the state(s) satisfying $\neg P \wedge S$ is(are) violation state(s). Therefore, check $\neg P_1 \wedge P_6 \wedge \neg P_7$, and the result shows there exist one violation state in `ktz`. It is the state $S_4$ with marking $P_0P_3P_5P_6$.

## 5.5 Computing Bound Value of Quantitative Property

When performing model checking, an observer can give an answer such as Yes or No for the satisfaction of the given property. For quantitative properties, however, instead of *whether the property is bounded by* $[t_{min}, t_{max}]$, the users usually expect to know *what are the exact bounds* $[t_{min}, t_{max}]$ *for that property*. The property pattern verification approach not only can check the satisfaction of the property, but also can be extended to compute the bound value of the quantitative property. This provides an assistant to help the users refine and improve their design model.

This service is implemented using an iterative method (shown in the progress map 3) that will gradually approach the bound values by combining the observer into a binary search engine.

In order to find the lower bound of the given quantitative property $\mathcal{P}$, a series of qualitative property *Always* $\mathcal{P} > k$ *in scope Global* will be recurrently assessed by the model checker. If for $k$ the answer is true, but for $k-1$ the answer is false, the lower bound is then $k$ (assuming $\mathcal{P}$ belongs to natural number). The same principle is applied to search the upper bound value by changing the query from $\mathcal{P} > k$ to $\mathcal{P} < k$.

To search the exact value as a natural number is like searching for a element in an ordered set. In order to minimize the search time, a binary search strategy is used. This reduces the computation complexity from

$\mathcal{O}(N)$ to $\mathcal{O}(\log_2 N)$, where $N$ is the predefined lower(upper) bound that should be large enough to cover all quantitative property's values in practice.

In a parallel computation environment, the search strategy can be more optimal. Given K CPU, the number of check will be reduced from $\mathcal{O}(\log_2 N)$ to $\mathcal{O}(\log_K N)$.

We take the example of Worst Case Execution Time (WCET) to explain the search algorithm.

**Example 5.2 (WCET Computation)** *The real-time property to be computed is: WCET of event E.*

### 5.5.1 WCET Property Verification

First of all, the expected property is specified using the set of real-time property patterns. This property is interpreted by the occurrence modifier: *Always*, the predicate: Maximum time interval between Init and *E*, and the scope modifier: *Global*. The observer is given in Fig. 5.5.1. With this observer, we use the following

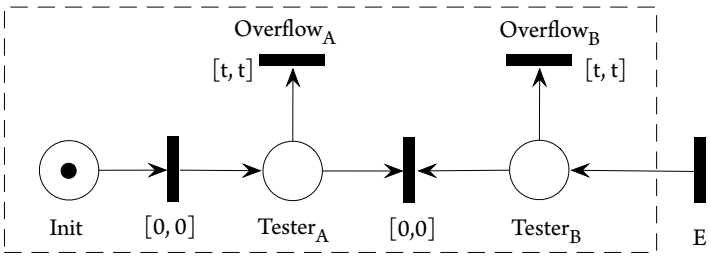



**Figure 5.5.1:** Property Computation Example: WCET

assertion to check WCET $< t$, where $\mathcal{A}$ represents the whole state space of the given TPN model including the observers.

$$N(\neg(Overflow_A \vee Overflow_B) \wedge \mathcal{A}) = N(\mathcal{A})$$

### 5.5.2 Computation of WCET Bound Value

If WCET $< t$ is true, then the WCET might be lower than $t$. We use a binary search algorithm to compute the bound value.

If WCET < $\frac{t}{2}$ is true, the bound value must be in the time interval $[1, \frac{t}{2}]$, otherwise, the bound value must be in the time interval $[\frac{t}{2}, t]$. According to these results, the next search will be performed on one of them until we find the exact WCET value.

### 5.5.3 Discussion: K-ary Searching Algorithm

K-ary searching algorithm follows the same principle: in each iteration, the original range [a, b] will be divided into K sections: [a, a+(b-a)/K], [a+(b-a)/K, a+2(b-a)/K], ..., [a+(K-1)(b-a)/K, b]. To simplify the discussion, we call the minimal value in each interval as $v_{min}$, and call the maximal value as $v_{max}$. Among these K sections, only one section will have the model checking result such that WCET < $v_{min}$ is false and WCET < $v_{max}$ is true. Therefore the new range for the next iteration is $[v_{min}, v_{max}]$. If $v_{max}$-$v_{min}$ = 1, the iteration is over and the WCET is $v_{max}$. For generalization, the initial range is always [0, N], where $N$ is the predefined lower(upper) bound that should be large enough to cover all quantitative property's value in practice.

### 5.5.4 Discussion: Cavity in Computation of Bound Value

A concern about this search method is the risk introduced by cavity intervals. An example is given to explain this concern (see Ex. 5.3).

**Example 5.3 (Cavity Discussion)** *The execution time of a given system is specified as two time intervals [2,8] and [12,18] (see Fig. 5.5.2). The property $\mathcal{P}$ (WCET < 20) is proved as true. Now the exact bound value of WCET is required, which is 18.*

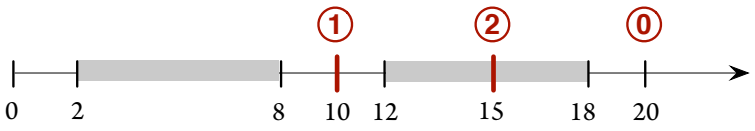



**Figure 5.5.2:** Cavity Discussion Example

Binary search algorithm is used to compute this exact bound value. Firstly, in the assertion WCET < t, t is set to be 10, which falls in the cavity [8,12]. Since there exists execution time [12,18], the transition *Overflow* in the observer (Fig. 5.5.1) will fire. The checking for WCET < 10 is thus false. Then we need to

check WCET < 15, whose checking result is also false. Then we try WCET < 18, which is still false. At last we try WCET < 19, whose result is true. The exact bound value is thus 18.

This example shows that the search algorithm using observers in model checking is sound even in the presence of cavities in the specified execution time.

## 5.6 Verification Scalability

Model checking techniques suffer from the state space explosion problem that makes it seem less useful for large-scale systems. In some systems, the size of the state space of the system grows exponentially along with the number of processes and variables. In our work, we use several methods to ensure the verification scalability, including property-driven semantic mapping (introduced in Chapter 3), on-the fly model checking (Section 5.6.1), state abstraction (Section 5.6.2), and TPN reduction (Chapter 6).

### 5.6.1 On-the-Fly Model Checking

The model checking toolset TINA embeds natively on-the-fly features.

On-the-fly methods allow the model checking to be performed without having the reachability graph fully generated. The property is checked along with the reachability graph's expansion. Once the property can already be cited as false according to the partial reachability graph, the check is suspended and the verification result become available immediately. Although in worst cases, all patterns (exist, absent, always) need a full generation to decide whether the property is satisfied, it would practically reduce both time and resource for general cases.

When the system behavior matches *Exist* pattern or violate *Absent* and *Always* pattern, the computation stops once it finds the first matched or violated state, thus it can stop earlier than classical method. The improvement of performance is obvious. However in worst cases, the footprint of checking property at every expansion step might make the whole computation time longer than classical methods.

### 5.6.2 State Abstraction

State abstraction is a technique applied at model checker level. Instead of preserving all possible states in the reachability graph to prepare for the verification, it proposes a series of abstraction for original states which can reduce the state expansion and therefore globally limit the memory footprint of computation.

Some properties of the system are guaranteed to be preserved in this abstracted version of the reachability graph.

The performance of abstraction is decided mainly by which kinds of property we want to preserve during the computation. Usually, the more universal the coverage is, the less abstraction of state can be done. We rely in this work on a model checker that makes a trade-off between the generality of method appliance (should be capable to verify most of the common properties) and the abstraction ratio (should be enough to make verification more scalable against larger system). In other words, the more properties can be verified, the less scalable the verification can be. Comparing to on-the-fly, the advantage of state abstraction is that it can reduce both computation time and resource in a stable way. This feature is important because industry deployment needs performance robustness.

TINA provides various state space abstractions for TPN state class graphs, following the techniques discussed in [BM83, BD91, Ber01, BV03]. All observers defined in this chapter do not require the preservation of the LTL/CTL/CTL * semantics in reachability graph because the assertion of pattern's satisfaction works on marking abstraction level, which does not required to preserve the firing sequence of transitions. Therefore the state abstraction level that we use is the highest possible comparing to our knowledge of the current theoretical progress in the TPN field.

## 5.7 Conclusion

This chapter presented a TPN/tts observer-based verification approach for checking real-time property patterns. Temporal logics such as TCTL can be used to express real-time properties, but such a logic formalism is limited in practice due to the capability of model checking tools we are relying on. Our approach adds efficient TPN/tts observers in the observed system to check each real-time property pattern. This observer-based verification approach takes advantage of high abstraction of state class graph to minimize model checking complexity, and thereby improve the verification efficiency. The verification approach can also be used as a computation method to search bound values for properties. This may help the users quickly adapt and refine their design model.

The main contributions of the current chapter are summarized as follows:

1. **First, the proposed set of atomic observers is complete and minimal.** As presented in Chapter 4, a real-time requirement is expressed using a set of elementary modifiers (occurrence, event, basic predicate and scope modifiers), which are semantically atomic. The set of elementary modifiers is

minimal, which means all the real-time properties expressed using Dwyer's and Konrad's patterns or CCSL can be specified using our elementary modifiers. The proposed set of observers corresponds to these elementary modifiers. Therefore, the observers are atomic and composable. The set of observers is minimal and complete.

2. **Second, the observers can minimize the size of the state space in the generated state class graph.** As an additional part added in the original system, the observers will increase system's original state space, because they introduce some possible execution traces related to the observation. This additional observer structures, although do not add extra semantics or remove original semantics for system's behavior, they can change the size of state space of the generated state class graph and thus impact the performance of the model checking. The less transitions an observer is composed of, the better performance will the model checking achieve. Nevertheless, minimizing the transitions in an observer may require to use some TPN/tts features which may degrade of abstraction level of state class graph. The lower the abstraction level is, the worse the performance will be. Therefore observer design must promote a trade-off between the above two aspects. In our approach, the priority is keeping the highest abstraction (marking abstraction) of the state class graph, then minimize the transitions in the observers. We rely on the $\mu$-calculus and the muse model checker from the TINA toolset to assess the real-time properties based on our proposed observers.

3. **Third, use observers to compute the bound values of quantitative property.** When performing model checking, an observer can give an answer such as YES or NO for the satisfaction of the given property. For quantitative properties, however, users usually expect to know, instead of *whether the property is bounded by* $[t_{min}, t_{max}]$, that the exact bound of the property$[t_{min}, t_{max}]$. In order to fill this gap of usage, we propose an iterative dichotomy method that will gradually approach the bound value by combining the observer with a binary search engine.

# 6

# Real-Time Property- Specific Reduction for TPN


## Résumé

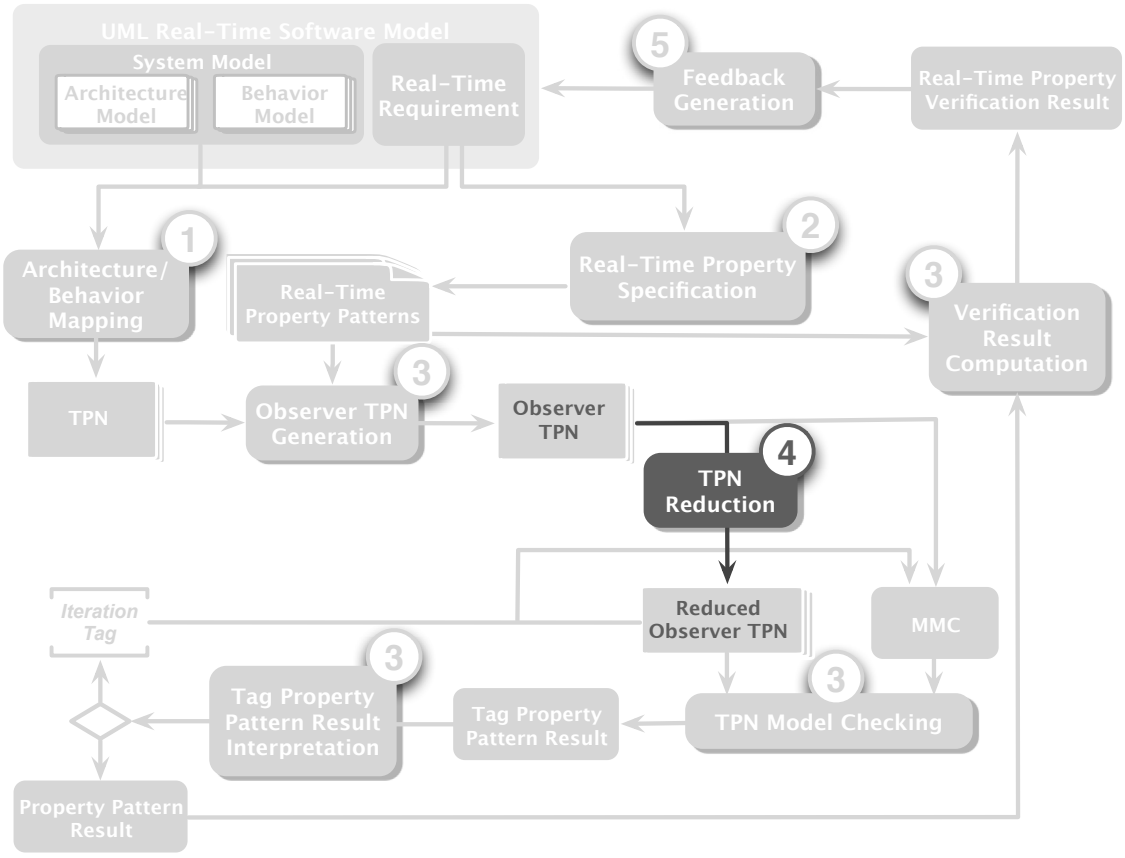
Ce chapitre propose une approche spécifique aux propriétés temps réels pour réduire l'espace d'état du réseau de Petri temporisé avant de générer le graphe d'états pour la vérification.

La vérification de modèle pour des systèmes asynchrones rencontre souvent des problèmes de passage à l'échelle car le nombre d'états dans l'execution du système augmente généralement exponentiellement avec la taille du système. Un système réaliste contient donc souvent un très grand nombre d'états et de transitions possibles. Les méthodes de vérification classiques rencontrent souvent ce problème car elles suivent le but, plus ou moins explicite, que de nombreuses propriétés de natures différentes seront évaluées en s'appuyant sur le même graphe d'états. Cette idée impose de construire les systèmes de transition les plus concrets et précis pour évaluer ensuite toute nature de propriétés. Les méthodes de réduction existantes suivent généralement cette philosophie pour préserver un espace d'état complet. Ces méthodes génériques ont améliorées significativement l'efficacité de la vérification de modèles, mais leurs améliorations sont de plus

en plus difficiles. Nous proposons de mettre de côté l'universalité des propriétés vérifiées, et d'introduire des méthodes de réduction spécifiques aux propriétés.

Les idées principales de cette partie sont, d'une part d'éliminer les parties du réseau de Petri sans rapport avec la propriété qui doit être vérifiée, et d'autres part de remplacer certaines parties pertinentes pour la propriété par des réseaux équivalents vis à vis de cette propriété mais conduisant à un plus petit nombre d'états et de transitions. La dépendance entre une partie dans un réseau de Petri temporisé et la propriété d'accessibilité considérée est dérivée de la causalité entre les transitions et les états dans le graphe d'états. Ceci impose une analyse basée sur la construction du graphe d'états qui est paradoxale car si nous pouvons générer le graphe d'états, il n'est plus nécessaire de le réduire. Nous proposons donc d'utiliser une sur-approximation de cette causalité sous la forme de la dépendance structurelle interne au réseau en imposant la divergence temporelle des parties éliminées et substituées.

Nous proposons, d'une part un algorithme pour rechercher dans un réseau de Petri les parties sans rapport avec les places, transitions et variables dont dépendent la propriété cible, et d'autre part un algorithme pour réduire les autres parties du réseau tout en préservant la propriété cible. La méthode de réduction pour les structures dont dépend la propriété consiste à diviser le réseau en sous-réseau de plus petite taille dont les relations avec le réseau global sont minimales, puis à construire une abstraction de chaque sous-réseau lorsqu'il présente un comportement régulier par rapport aux propriétés temps réels considérées. Ce comportement régulier est une abstraction de toutes les traces dans le graphe d'état du point de vue des observations. Nous proposons plusieurs structures régulières possibles et nous utilisons notre méthode de vérification pour montrer que la structure régulière permet de remplacer le sous-réseau et pour en calculer les caractéristiques temporisées. Cette approche est pertinente en terme de coût de vérification par la nature combinatoire de cette-ci : le coût de vérification sur une partie est en général beaucoup plus faible que le coût global. Il est donc possible d'étudier plusieurs structures régulières pour chaque partie du réseau avant que le coût cumulé soit supérieur au coût de vérification du réseau complet. Cette approche permet de réduire la taille de l'espace de chaque sous-réseau et de rendre calculable l'espace d'états du réseau complet avec les parties réduites avec les ressources usuellement disponibles . Cette méthode a donné de bons résultats expérimentaux pour la vérification des propriétés temps réel au niveau des modèles d'architecture. Ces travaux devront maintenant être étendus et expérimentés pour d'autres familles de propriétés et de systèmes.




*Progress Map 4:*Real-Time Property-Specific State Space Reduction

In the current chapter, we propose the real-time property-specific TPN reduction approach applied before generating the state class graph to verify the real-time properties using model checking (Progress Map 4). The verification of concurrent asynchronous systems using model checking usually encounter scalability problems very quickly along with the growth of system size (Challenge 4 in page 22). Our proposal is to build an equivalent of the original TPN, which exhibits the same property-specific behavior, and has less transitions and states. This reduces directly the scale of computation before generating the state space. The proposed reduction method is based on similar ideas as the partial order reduction, which is aimed at reducing the size of the state space that needs to be searched. The partial order reduction exploits the commutativity of concurrent executed transitions which result in the same state when expanding the state class graph. Our approach exploits the commutativity of sub-nets of TPN which result in the same property-specific be-

havior before expanding the state class graph. The approach is based on classic TPN model extended with data manipulation (tts) provided by the TINA toolset. To exploit the property-specific reduction, first a property relevance algorithm is applied to eliminate parts of property-irrelevant TPN structures, then the topology-implicit semantic equivalence and behavioral equivalence patterns are applied to identify the reducible sub-nets and reduce them using equivalent sub-nets. (Contribution 4 in page 24)

## 6.1 Introduction

The key issue that prevents a wide application of model checking in the industry is the scalability with respect to the size of the target system. A common system usually has thousands and even millions of states and transitions. Although a huge part of impossible transition firing sequences are eliminated during the building of system's behavior, the probable permutation of all others is still a very large number that will easily lead to combinatorial state space explosion.

Classic verification methodologies usually encounter scalability issue very quickly along with the growth of system size, because it follows an implicit purpose: once the reachability graph is generated, it can be reused to verify many different properties of the system, just by changing the assessed logic formulas. This consideration requires to build the most concrete and precise transition system to be used to assess any kind of properties. It makes sense if the assessed system does not change often the states and if there is a large number of requirements to assess. However, it is well known that the generation of state class graph for large scale models is the most expensive phase in terms of resource and time consumption. Theoretically, generating the reachability graph only once seems to be resource-saving by eliminating the effort of regenerating. However, this global-resource-saving principle implies an assumption that is sometimes false: that it will always be possible to generate the reachability graph with common available resources. The existing state space reduction methods, partial order reduction [Val91, GvLH+96], compositional reasoning [MC81, GL94], symmetry [CEFJ96, ES96], abstraction techniques [CGL94], on-the-fly model checking [Hol96, BRV04], etc., usually follow the same philosophy to produce a complete state space that preserves the mandatory semantics that allow the verification of all kinds of properties. These generic reduction methods have effectively improved the efficiency of model checking techniques. But their improvement is becoming more and more difficult. We thus might put aside the universality of the semantics expressed in the transition systems that allow to assess all kinds of properties, and take into account property specific reduction methods.

A typical system will run with a large amount of transitions and make the above prerequisites hard to maintain. The fundamental reason why this computation-oriented approach is not scalable is because it tries to preserve all information for the verification afterwards. In the context of the thesis, we focus on state space reduction approaches related to TPN models dedicated to real-time property verification. In Chapter 3 and Chapter 5, we have presented the use of the following approaches:

- **Modeling abstraction**: If the designer is sure that some components will not impact the property, there components will not be modeled.
- **Mapping abstraction**: After defining the execution semantics for the end-user model, the property-specific system model (UML-MARTE model) is mapped to the property-specific verification model (TPN model), which contains all the property-specific behaviors. Part of property-irrelevant information is eliminated using this abstraction.
- **State abstraction**: The tool tina provides abstraction options for the generated state class graph, which preserves different information. All observers defined in the precedent chapter do not require preserving the LTL/CTL/CTL * semantics in the state class graph because the logic assertions used to verify the properties require only the marking feature of the state class graph. Therefore, the state abstraction option used is the highest possible (marking abstraction) comparing to the current theoretical progress in TPN field, and the logic assertions are accessibility assertions.
- **On-the-fly checking**: Using on-the-fly model checking provided by the sift model checker.

Both on-the-fly and state abstraction techniques focus on reducing state space when performing model checking. Another reasonable thinking is to build an equivalent of the original TPN in term of property-specific behavior, but with less states and transitions. This reduces directly the scale of computation before expanding the state space. The assessed TPN will be reduced in several appropriate ways according to the target properties before being sent to the model checker.

Our approach focuses on the following contributions:

- The core idea of our proposal is to eliminate the property-irrelevant structures in a TPN model, then substitute the property-relevant structures by an equivalent with less states and transitions. The TINA toolset supports extended TPN with data handling called Time Transition Systems (tts), including the precondition function Pre and the action function Act. Our approach is based on this new TPN feature.

- The modeling and mapping abstractions only eliminate part of property-irrelevant behaviors. The relevancy between a TPN structure and the target property is decided by the causality between TPN transitions and states in the state class graph. In other words, measuring precisely the causality between TPN transitions requires a state class graph-based analysis. This leads to the following paradox that if we are able to generate the state class graph for the given TPN before the reduction, we may not need to reduce it any more. The solution is to use an over-approximation of causality, i.e. use TPN's structure dependence to deduce the causality. The relevance between two transitions at TPN level is the necessary condition that they are causally dependent. We propose an algorithm to search the TPN structures irrelevant to all the places, transitions and variables referred to by the target property, and reduce those structures irrelevant to the property.

- The property-relevant structure is reduced using the commutativity of sub-net in the TPN model which results in the same property-specific behavior. For some sub-net patterns recurring frequently in an asynchronous system, we define the topology-implicit semantic equivalence patterns to identify these sub-net and substitute them by the equivalent sub-net that exhibits the same property-specific behavior. This part of work is similar to the existing work of [SB96]. The work [SB96] targeted a general TPN model, while we define some new patterns that are derived from the asynchronous system.

- The topology-implicit semantic equivalence patterns are in fact simple behavior equivalent patterns. In some cases, it is complex to detect and extract a localized topology pattern. A novel real-time property-specific reduction method is proposed in this chapter. This method first identifies the possibly reducible sub-net, and then builds an equivalent net which exhibits the same property-specific behavior as the original net. Before substituting the equivalent sub-net, we use refinement functions to ensure that the substitution is correct.

In this chapter, Section 6.2 analyzes the relevancy between system components under observation and present reduction method for property-irrelevant TPN structures; Section 6.3 gives an overview of property-specific reduction method; Section 6.4 presents the topology-implicit semantic equivalence patterns; Section 6.5 illustrates the method using behavioral equivalence.

## 6.2 Reduction for Property-Irrelevant Structures

In a complex system, there are usually many components running concurrently. Some are **pseudo-concurrent** because they are eventually synchronized somewhere by data exchange. Others are of **real-concurrent**, for example, redundant sensors/actuators.

Fig. 6.2.1 is an example showing the relevancy between TPN system components integrated with property observers. It is shown that those real concurrent components are separated non connected graph (**A** and **C** in Fig. 6.2.1), and those pseudo-concurrent components are weakly connected graph with explicit cuts (**B, E, D, F** in Fig. 6.2.1).

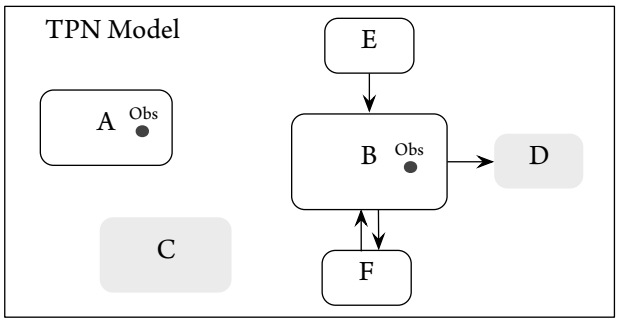



**Figure 6.2.1:** Relevancy between System Components

The TPN observers (points in **A** and **B**) for the given property are attached on sub-TPN **A** and **B**. If we are sure there is no places, transitions or variables referred to by the observers in **A** and **B**, we can remove **C** completely from the TPN model. The removal of **C** will not impact the observation in **A** and **B**.

The time divergence issue introduced by TPN must be discussed here. Suppose **C** is simply composed by the following structure in Fig. 6.2.2. As the time constraint of transition *T* is zero, this infinite loop does not consume time. This behavior blocks the time for the whole system. The transitions in the other components will never be activated if they are tagged with strictly positive time bound. Thus, we must ensure that the time is able to evolve in our target TPN model. In this case, the time divergence in **C** must be ensured before it is removed. Another solution is to avoid the time divergence issue in the modeling phase (in our case UML). The UML model represents an abstraction of the real system's behavior. The time divergence problem only occurs in the TPN system, because besides possessing their own clocks, the asynchronous TPN components also sharing a common clock. According to the mapping semantics defined in Chapter 3, if **C** does not

exchange data with the other parts in the UML model, no transitions or places will be added between them in the TPN model. In other words, **C** is also independent in the TPN model. Therefore, if a component is independent in TPN, we can deduce that it is also independent in UML. Meanwhile, if it is not relevant to the observer, it can be removed from both the UML and the TPN models.

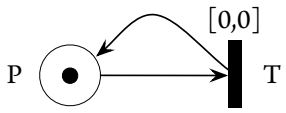



**Figure 6.2.2:** Time Divergence Issue

Once **C** is removed from the original TPN, the construction of state class graph will become easier because the number of transitions in the system has been reduced. The same action can be taken on **D** because **B** does not depend on **D**'s behavior due to the uni-directional cut from **B** to **D**. On the other hand, the reduction of **E** or **F** may change the observer's result (i.e. the original property of the system that we want to verify) of **B** because they have outgoing cuts to **B**.

The above illustration on relevancy between components with integrated observers gives two principles to define a formal method to reduce those property-irrelevant structures:

- **Identification**: Property-irrelevant sub-TPN can be identified by analyzing observer's dependency.
- **Reduction**: Property-irrelevant sub-TPN can be removed without changing the property.

### 6.2.1 Relevancy Analysis for TPN Extended with Data Handling

The relevancy between a TPN structure and the target property can be measured precisely using the causality between TPN transitions and states in the state class graph. This measurement leads to the following paradox that if we are able to generate the state class graph for the given TPN before the reduction, we may not need to reduce it any more. The solution is to use an over-approximation of causality, i.e. use TPN structure dependence to deduce the causality. The relevance between two transitions at TPN level is the necessary condition that they are causally dependent.

The behavior of a transition depends on its pre-conditions. In classic TPN, it refers to the incoming places of the transition. For example, in Fig. 6.2.3 (a), only $P_1$ and $P_2$ will impact T's behavior. In other

words, whenever $P_3$ will obtain markings or lose markings won't change the particular time sequence that T will fire. The direct outgoing transitions of T's incoming places, i.e. $T_1$ can also create dependencies, because with conflict, the time that $T_1$ fires will eventually change that of T. In the context of extended `tts` feature, the dependency between variables must also be included. Fig. 6.2.3 (b) shows that all transitions manipulating the variables (place marking is also taken into account as variables) which has been referenced in the precondition (PRE) of a property-relevant transition are also marked as property-relevant. Therefore, $T_Y$, $T_X$, and $P_4$ are tagged as relevant.

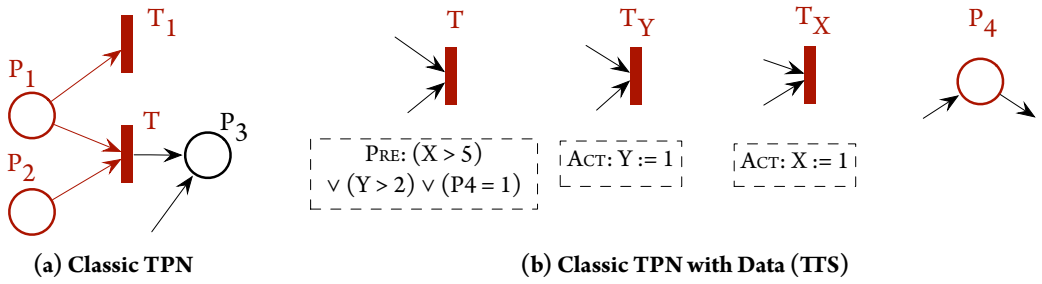



**Figure 6.2.3:** Relevant Structure for TPN Transition

The specification of a place dependency is quite simple. All incoming and outgoing transitions are taken into account, because both of them impact the marking of the place. Since `tts` extends only transition's semantic, the identification of the property-relevant elements for a given place in `TPN` with or without `tts` feature are done in the same way. (Fig. 6.2.4)

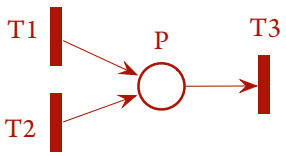



**Figure 6.2.4:** Relevant Structure for TPN Place

### 6.2.2 Algorithm for Reducing Property-Irrelevant Structure

According to the above analysis on relevant structures of TPN (with data) transition and place, we propose an algorithm to automate the reduction of irrelevant structures. Before providing the algorithm, we define three core functions.

**Transition dependency search $\mathbf{F}(t)$ = <$\mathbf{S_T}$, $\mathbf{S_P}$>:** $\mathbf{F}(t)$ extracts, for a given transition *t*, its direct dependent transition set $\mathbf{S_T}$ and place set $\mathbf{S_P}$,

- $\mathbf{S_P} = \mathbf{S_P^{'}} \cup \mathbf{S_P^{''}}$
- $\mathbf{S_P^{'}} = \bullet t$
- $\mathbf{S_P^{''}} = \{pl | pl \in \textsc{Pre}(t).v_P\}$ ($v_P$ are the places used as variables)
- $\mathbf{S_T} = \{tr | \textsc{Act}(tr).variables \subseteq \textsc{Pre}(t).variables\}$

where $\bullet t$ are ingoing places of the given transition *t*.

**Place dependency search $\mathbf{G}(p)$ = <$\mathbf{S_T}$>:** $\mathbf{G}(p)$ extracts, for a given place *p*, its direct dependent transition set $S_T$,

- $\mathbf{S_T} = \bullet p \cup p \bullet$

where $\bullet p$ and $p \bullet$ are respectively ingoing and outgoing transitions of the given place *p*.

**Property dependency search $\mathbf{I}(\mathcal{P})$ = <$\mathbb{P}$, $\mathbb{T}$>:** $\mathbf{I}(\mathcal{P})$ extracts, for a given target property $\mathcal{P}$, its dependent place set $\mathbb{P}$ and dependent transition set $\mathbb{T}$. A property is assessed using TPN observers and logic formulas (in our case $\mu$-calculus (MMC)). $\mathbb{P}$ and $\mathbb{T}$ are the set of places and transitions that respectively contains all the sets of place and transition in TPN observers (resp. $O_P$ and $O_T$), the sets of place and transition directly attached to observers (resp. $\psi_P$ and $\psi_T$) and the sets of place and transition referred to in logic formulas MMC (resp. $\phi_P$ and $\phi_T$), then

- $\mathbb{P} = O_P \cup \psi_P \cup \phi_P$
- $\mathbb{T} = O_T \cup \psi_T \cup \phi_T$

We present the pseudo-code of the algorithm (see Algo. 1) for searching property-relevant and property-irrelevant structures. The TPN system is defined as a set of places/transitions ($\mathbb{S}_P \cup \mathbb{S}_T$). The arcs are combined in the ingoings and outgoings of places and transitions. The input data are TPN model with observers ($O_P \cup O_T$) and logic formulas MMC corresponding to the target property $\mathcal{P}$. The desired outputs are the set of property-relevant structure and the set of property-irrelevant structure.

**Data**: System $\mathbb{S}_P \cup \mathbb{S}_T$ and observers $O_P \cup O_T$, property $\mathcal{P}$
**Result**: $\mathbb{S}_r$, $\mathbb{S}_{ir}$
$<\mathbb{S}_r^P, \mathbb{S}_r^T> := \mathbf{I}(\mathcal{P})$
**while** *not all elements in* $\mathbb{S}_r^P$ *and in* $\mathbb{S}_r^T$ *have been tagged* **do**
    p = $\mathbb{S}_r^P$.getUntagged();
    $\mathbb{S}_r^T$.addAll(**G**(p));
    p.tag();
    t = $\mathbb{S}_r^T$.getUntagged();
    $\mathbb{S}_r^P$.addAll(**F**(t).$S_P$);
    $\mathbb{S}_r^T$.addAll(**F**(t).$S_T$);
    t.tag();
**end**
$\mathbb{S}_r = <\mathbb{S}_r^P, \mathbb{S}_r^T>$;
$\mathbb{S}_{ir} = \overline{\mathbb{S}_r}$;

**Algorithm 1:** Relevant and Irrelevant TPN Structures Search

The search starts from the places and transitions depending on property, including TPN observers, transitions directly attached to observers, and places/transitions referred to by MMC logic formulas. Therefore, initially $\mathbb{S}_r^P := \mathbb{P}$ and $\mathbb{S}_r^T := \mathbb{T}$. The following iterative process tags the relevant elements using predefined functions $\mathbf{F}(\mathrm{t})$ and $\mathbf{G}(\mathrm{p})$. If not all the elements in $\mathbb{S}_r^P$ or $\mathbb{S}_r^T$ have been tagged as property-relevant:

- p = $\mathbb{S}_r^T$.getUntagged(): Get each place $p$ not tagged from $\mathbb{S}_r^P$.
- $\mathbb{S}_r^T$.addAll(**G**(p)): Search dependent transitions of $p$ and add them in $\mathbb{S}_r^T$.
- p.tag(): Tag place $p$ as property-relevant structure.
- t = $\mathbb{S}_r^T$.getUntagged(): Get each transition $t$ not tagged from $\mathbb{S}_r^T$.
- $\mathbb{S}_r^P$.addAll(**F**(t).$S_P$), $\mathbb{S}_r^T$.addAll(**F**(t).$S_T$): Search dependent places and transitions of $p$ and add them in $\mathbb{S}_r^P$ and $\mathbb{S}_r^T$.

- `p.tag()`: Tag $t$ as property-relevant structure.

In the end, the property-relevant set is the pair of the sets of relevant place and relevant transition; the property-irrelevant set is complementary to property-relevant set.

The algorithm that identifies all property-relevant and property-irrelevant elements through relevancy propagation is illustrated below using an example (see Ex. 6.1).

**Example 6.1 (Relevancy Propagation Example)** *Fig. 6.2.5 (a) is the original system TPN model, with observer $P_{obs}$ and the property assertion to be checked is $P_{obs} = 1$. We use Algo. 1 to tag the property-relevant structures and eliminate the property-irrelevant structure.*

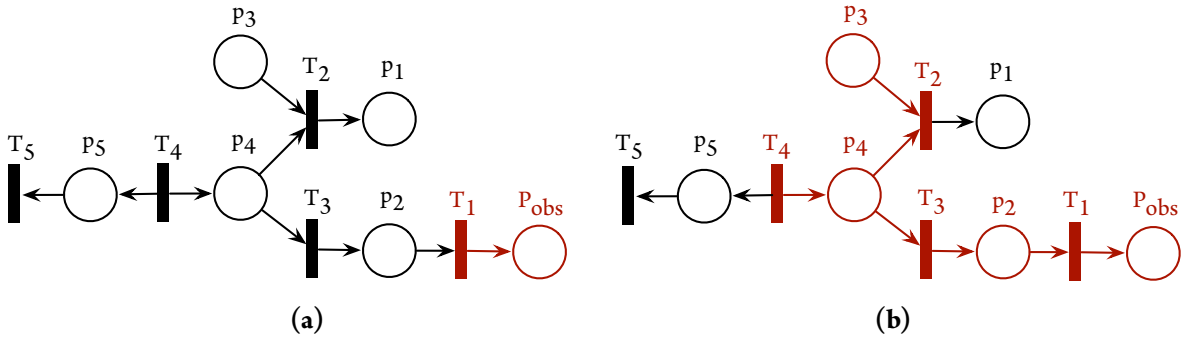



**Figure 6.2.5:** Example of Propagation of Property-Relevant TPN Structure

*The relevancy propagation process is illustrated below, the propagation result is Fig. 6.2.5 (b).*

1. *Initially, using $\mathbf{I}(\mathcal{P})$, the observer place $P_{obs}$ and its dependent transition $T_1$ are tagged as relevant (here we use red color as tag).*
2. *Using $\mathbf{F}(T_1)$, $P_2$ is tagged.*
3. *Using $\mathbf{G}(P_2)$, $T_3$ is tagged.*
4. *Using $\mathbf{F}(T_3)$, $P_4$ is tagged.*
5. *Using $\mathbf{G}(P_4)$, $T_2$ and $T_4$ are tagged.*

6. *Using* $\mathbf{F}(T_2)$, $P_3$ *is tagged.*

The removal action for property-irrelevant structures is trivial. All incoming and outgoing arcs of property-irrelevant TPN elements will be deleted along with the elements themselves before being passed to the model checker. For both Algo. 1 and the removal operation, the complexity is $\mathcal{O}(N)$, where N is number of elements (transition and place) of the given system.

## 6.3 Reduction for Property-Relevant Structures

The property-relevant structure is reduced using the commutativity of sub-net in the TPN model which results in the same property-specific behavior. The property-specific reduction method has similar idea to the partial order reduction.

The partial order reduction [CGP99, GvLH+96] is usually used in asynchronous concurrent systems, in which most of the activities in different processes are performed independently, without a global synchronization. The most successful techniques for dealing with this problem are based on the partial order reduction [GP93, Pel94, Val91]. The main idea of partial order reduction is to construct a reduced state class graph by eliminating the unnecessary behaviors. This method is based on the dependencies that exist between the transitions of a system. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. The reduction method then specifies the set of transitions that should be included in the reduced state class graph. The reduced behavior is a subset of the behavior of the full state class graph. Thus it does not add any information to the behavior of a system.

Compared to the partial order reduction, the proposed property-specific reduction exploits the commutativity of TPN structure before generating the state class graph, which result in the same property-specific behavior. The partial order reduction is performed at the state class graph level, while the property-specific reduction is performed at the TPN model level.

Another related work [DPC+09, DBRL12] by P. Dhaussy et al. proposed to improve model checking with context modeling. As requirements are usually related to specific use cases (context), they restricted the system behavior with a specific surrounding environment (modeling the context) describing the different configurations in which one wants to verify the system. It may take more effort for the engineers to explicitly and formally express more detailed requirements, but can make the model checking more efficient under a fully specified environment. This additional work may be required anyway with the integration of the DO-331 Model Based Development and Verification supplement in the safety critical system

development process, which enforces the writing of more precises requirements. They provide the Context Description Language (CDL) and the Observer Based Prover (OBP) toolset [1] based on this method. Both of our works aim to provide property specific reduction methods for formal verification. They focus on the modeling aspect, while we focus on the verification aspect. Both approaches could be integrated to provide an even better scalability.

For some sub-net patterns recurring frequently in an asynchronous system, we define the topology-implicit semantic equivalence patterns to identify these sub-net and substitute them by the equivalent sub-net that exhibits the same property-specific behavior. This part of work is similar to the existing work of [SB96, Ber83, B+86, Had90]. Berthelot originally developed set of reduction rules for general Petri nets [Ber83, B+86]. Haddad extended Berthelot's approach to Colored Petri nets [Had90]. Sloan et al. targeted the patterns in a general TPN model [SB96], while we define some new patterns that are derived from the asynchronous system.

The topology-implicit semantic equivalence patterns are in fact simple behavior equivalent patterns. In some cases, it is complex to detect and extract a localized topology pattern. A novel real-time property-specific reduction method is proposed in this chapter. This method first identifies the possibly reducible sub-net, and then builds an equivalent net which exhibits the same property-specific behavior as the original net. Before substituting the equivalent sub-net, we use refinement functions to ensure that the substitution is correct.

Before going to the detailed explanations, we define a symbolic system to ease the discussion:

- $T^+$ and $T^-$: for a given transition $T$, represent respectively $T$'s outgoing and incoming arcs.
- $P^+$ and $P^-$: for a given place $P$, represent respectively $P$'s outgoing and incoming arcs.
- $a.P$ and $a.T$: for a given arc $a$, represent respectively $a$'s associated place $P$ and transition $T$ [2]
- **card**(S): returns the number of elements in set S.
- **U**$(S)$: if **card**(S) = 1, returning the unique element in set S.
- $\mathbb{T}^{R(N)}$ and $\mathbb{P}^{R(N)}$: for a given TPN $N$, represent respectively the reducible transition and place sets.

[1] http://www.obpcdl.org/doku.php

[2] An arc has unique associated place and transition. Which one is source or target can be ignored here.

For the property-relevant TPN, we distinguish the reducible and non-reducible structure. The non-reducible structure is the places and transitions referred to by the observers used to verify the property. The other parts are all considered as reducible structure.

## 6.4 Reduction using Topology-Implicit Semantic Equivalence

For property-relevant structures, the most direct reduction method is to combine the sequential parts into reduced ones which have less states and transitions but retains the same properties (real-time, safety, etc.). Several topology patterns have been developed under this principle along with their corresponding reduction rules. All reducible elements must not be those directly associated with properties, including observers, transitions directly attached to observers, and places/transitions referred to in logic formulas MMC.

The topology-implicit semantic equivalence patterns include:

- **Redundant zero-time pattern**
  - Sequential pattern
  - Indirect initialization pattern
  - Shorten cycle pattern
- **Sequential encapsulation pattern**

### 6.4.1 Redundant Zero-Time Patterns

The objective of these patterns is to reduce the transitions with zero time interval that can increase the complexity of transition fire sequence's combination but do not provide any supplementary information for execution flow (Fig. 6.4.1, Fig. 6.4.2 and Fig. 6.4.3).

#### 6.4.1.1 Sequential Pattern

**Identification function** $\mathbf{F}_1(N) = <P_{pre}, T, P_{post}>$ will identify and give out a triple of TPN elements $<P_{pre}, T, P_{post}>$ that matches this sequential Redundant Zero-Time Pattern in the given TPN $N$:

- $T \in \mathbb{T}^{R(N)}, P_{pre} \in \mathbb{P}^{R(N)}, P_{post} \in \mathbb{P}^{R(N)}$

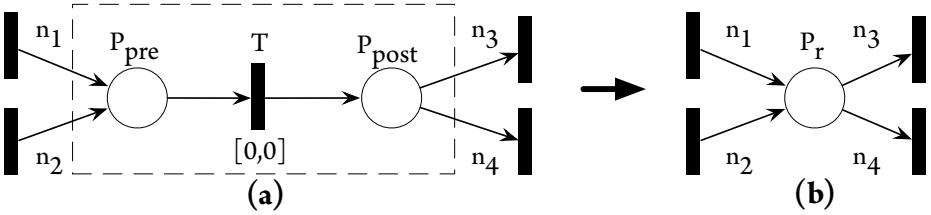



**Figure 6.4.1:** Redundant Zero-Time Pattern: Sequential

- $\begin{cases} \mathbf{card}(T^+) = \mathbf{card}(T^-) = 1 \\ \mathbf{card}(P^+_{pre}) = \mathbf{card}(P^-_{post}) = 1 \end{cases}$
- $\begin{cases} \mathbf{U}(T^+).weight = \mathbf{U}(T^-).weight = 1 \\ \mathbf{U}(T^-).P = P_{pre} \\ \mathbf{U}(T^+).P = P_{post} \\ \mathbf{U}(P^+_{pre}).T = \mathbf{U}(P^-_{post}).T = T \end{cases}$
- $P_{pre}.marking = P_{post}.marking = 0$
- $T.minTime = T.maxTime = 0$
- If $N$ is a tts, then $\textsc{Pre}(T) = \emptyset$ and $\textsc{Act}(T) = \emptyset$

**Reduction function $\mathbf{G}_1(P_{pre}, T, P_{post}, N) = <P_r, N'>$** will reduce the given triple $<P_{pre}, T, P_{post}>$ in the TPN $N$ to a new place $P_r$ in the new TPN $N'$ without zero-interval transition:

- $P_r$ is a newly created place, $P_r \notin N$
- $P_r.marking = 0, P_r \in \mathbb{P}^R(N')$
- $P^+_r = P^+_{post}$ and $P^-_r = P^-_{pre}$

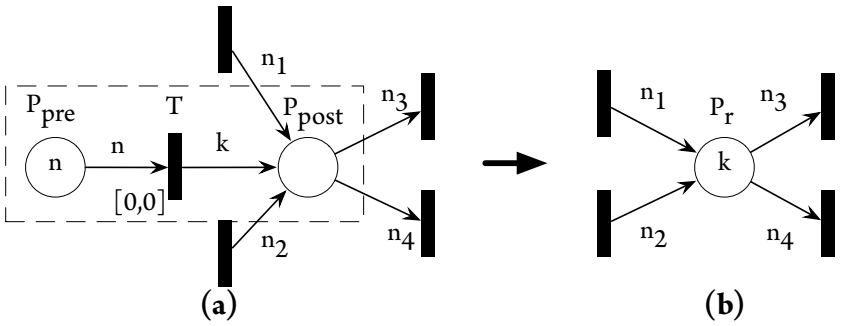



**Figure 6.4.2:** Redundant Zero-Time Pattern: Indirect Initialization

#### 6.4.1.2 Indirect Initialization Pattern

**Identification function** $\mathbf{F}_2(N) = <P_{pre}, T, P_{post}>$ will identify and give out a triple elements $<P_{pre}, T, P_{post}>$ that matches Indirect Initialization Pattern in a given TPN $N$:

- $T \in \mathbb{T}^{R(N)}, P_{pre} \in \mathbb{P}^{R(N)}, P_{post} \in \mathbb{P}^{R(N)}$
- $\begin{cases} \mathbf{card}(T^+) = \mathbf{card}(T^-) = \mathbf{card}(P^+_{pre}) = 1 \\ \mathbf{card}(P^-_{post}) \geq 1 \\ P^-_{pre} = \emptyset \end{cases}$
- $\begin{cases} \mathbf{U}(T^-).P = P_{pre} \\ \mathbf{U}(T^+).P = P_{post} \\ \mathbf{U}(P^+_{pre}).T = T \\ T \in P^-_{post}.T \end{cases}$
- $\begin{cases} P_{pre}.marking = \mathbf{U}(P^+_{pre}).marking \\ P_{pre}.marking \geq 1 \\ P_{post}.marking = 0 \end{cases}$
- $T.minTime = T.maxTime = 0$
- If $N$ is a tts, then $\textsc{Pre}(T) = \emptyset$ and $\textsc{Act}(T) = \emptyset$

**Reduction function** $\mathbf{G}_2(P_{pre}, T, P_{post}, N) = <P_r, N'>$ will reduce the given triple $<P_{pre}, T, P_{post}, N>$ in TPN $N$ to a new structure with only one place $P_r$ in the new TPN $N'$ without zero-interval transition.

- $P_r$ is a newly created place, $P_r \notin N$
- $P_r.marking = \mathbf{U}(T^+).weight, P_r \in \mathbb{P}^{R(N')}$
- $P_r^+ = P_{post}^+$ and $P_r^- = P_{post}^- - \{T\}$

#### 6.4.1.3 Shorten Cycle Pattern

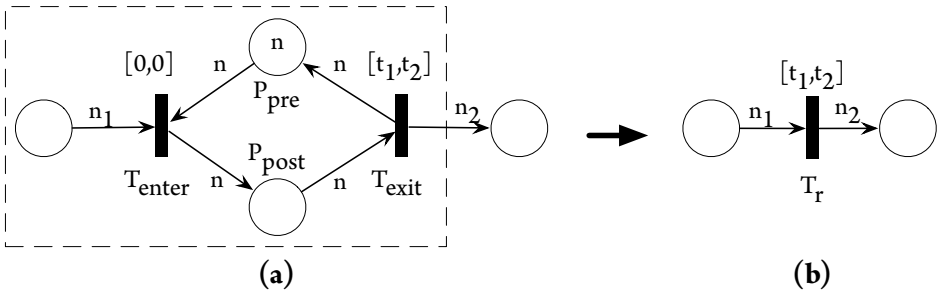



**Figure 6.4.3:** Redundant Zero-Time Pattern: Shorten Cycle

**Identification function** $\mathbf{F}_3(N) = <T_{enter}, P_{pre}, T_{exit}, P_{post}>$ will give out a 4-tuple of elements $<T_{enter}, P_{pre}, T_{exit}, P_{post}>$ that matches Shorten Cycle Pattern in a given TPN $N$:

- $T_{enter} \in \mathbb{T}^{R(N)}, T_{exit} \in \mathbb{T}^{R(N)}, P_{pre} \in \mathbb{P}^{R(N)}, P_{post} \in \mathbb{P}^{R(N)}$
- $\begin{cases} \mathbf{card}(T_{enter}^+) = \mathbf{card}(T_{exit}^-) = 1 \\ \mathbf{card}(P_{pre}^+) = \mathbf{card}(P_{pre}^-) = \mathbf{card}(P_{post}^+) = \mathbf{card}(P_{post}^-) = 1 \\ arc \in T_{enter}^- \Longrightarrow \mathbf{card}(arc.P^+) = 1 \end{cases}$
- $\begin{cases} \mathbf{U}(T_{enter}^+).P = \mathbf{U}(T_{exit}^-).P = P_{post} \\ \mathbf{U}(P_{pre}^+).T = \mathbf{U}(P_{post}^-).T = T_{enter} \\ \mathbf{U}(P_{pre}^-).T = \mathbf{U}(P_{post}^+).T = T_{exit} \\ P_{pre} \in T_{exit}^+, P_{pre} \in T_{enter}^- \end{cases}$

- $\begin{cases} P_{pre}.marking = \mathbf{U}(P^{-}_{pre}).weight = \mathbf{U}(P^{+}_{pre}).weight = \mathbf{U}(P^{-}_{post}).weight = \mathbf{U}(P^{+}_{post}).weight \\ P_{post}.marking = 0 \end{cases}$
- $T_{enter}.minTime = T_{enter}.maxTime = 0$
- If $N$ is a tts, then $\text{PRE}(T_{enter}) = \emptyset$ and $\text{PRE}(T_{exit}) = \emptyset$

**Reduction function** $\mathbf{G}_3(T_{enter}, P_{pre}, T_{exit}, P_{post}, N) = <T_r, N'>$ will reduce the given 4-tuple $<T_{enter}, P_{pre}, T_{exit}, P_{post}>$ in TPN $N$ to a new structure with only one combined transition $T_r$ in the new TPN $N'$:

- $T_r$ is a newly created transition, $T_r \notin N$
- $T_r.minTime = T_{exit}.minTime$ and $T_r.maxTime = T_{exit}.maxTime$, $T_r \in \mathbb{T}^{R(N')}$
- $T^{+}_{r} = T^{+}_{exit} - P_{post}$ and $T^{-}_{r} = T^{-}_{enter} - P_{post}$
- If $N$ is a tts, then $\text{ACT}(T_r) = \{\text{ACT}(T_{enter}); \text{ACT}(T_{exit})\}$

### 6.4.2 Sequential Encapsulation Pattern

The objective of this pattern is to reduce the transitions by combining sequential transitions into a unique one which encapsulates all necessary information from the original ones (time, preconditions, actions) (see Fig. 6.4.4).

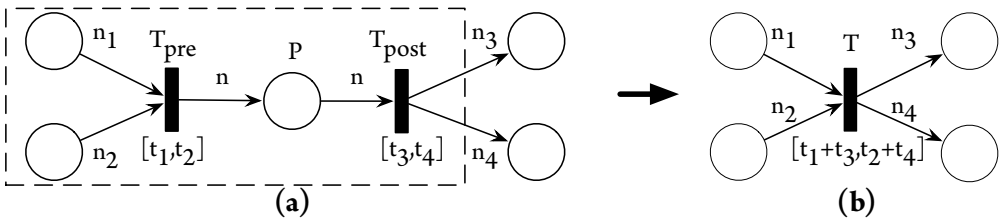



**Figure 6.4.4:** Sequential Encapsulation Pattern

**Identification function** $\mathbf{F}_4(N) = <T_{pre}, P, T_{post}>$ will identify and give out a triple of elements $<T_{pre}, P, T_{post}>$ that matches Sequential Encapsulation Pattern in a given TPN $N$:

- $P \in \mathbb{P}^{R(N)}$, $T_{pre} \in \mathbb{T}^{R(N)}$, $T_{post} \in \mathbb{T}^{R(N)}$
- $\begin{cases} \mathbf{card}(P^+) = \mathbf{card}(P^-) = 1 \text{ and } \mathbf{card}(T^+_{pre}) = \mathbf{card}(T^-_{post}) = 1 \\ arc \in T^-_{pre} \Longrightarrow \mathbf{card}(arc.P^+) = 1 \end{cases}$
- $\begin{cases} \mathbf{U}(P^+).weight = \mathbf{U}(P^-).weight \\ \mathbf{U}(P^-).T = T_{pre} \\ \mathbf{U}(P^+).T = T_{post} \\ \mathbf{U}(T^+_{pre}).P = \mathbf{U}(T^-_{post}).P = P \end{cases}$
- $P.marking = 0$
- If $N$ is a tts, then $\textsc{Pre}(T_{post}) = \emptyset$

**Reduction function** $\mathbf{G}_4(P_{pre}, T, P_{post}, N) = <T_r, N'>$ will reduce the given triple $<P_{pre}, T, P_{post}, N>$ in TPN $N$ to a new structure with only one transition $T_r$ in the new TPN $N'$:

- $T_r$ is newly created transition, $T_r \notin N$
- $T_r \in \mathbb{P}^{R(N')}$
- $T_r.minTime = T_{pre}.minTime + T_{post}.minTime$ and $T_r.maxTime = T_{pre}.maxTime + T_{post}.maxTime$
- $T^+_r = T^+_{post}$ and $T^-_r = T^-_{pre}$
- If $N$ is a tts, then $\textsc{Pre}(T_r) = \textsc{Pre}(T_{pre})$ and $\textsc{Act}(T_r) = \{\textsc{Act}(T_{pre}); \textsc{Act}(T_{post})\}$

## 6.5 Reduction using Behavioral Equivalence

The state space reduced by the topology-based patterns is limited. It still requires more reduction to generate large scale asynchronous systems. A novel reduction method based on property-specific behavioral equivalence is proposed. The topology-based method follows the same principle that the replacement must exhibit the same property-specific behavior (in our case real-time property-specific behavior). We can consider topology-based patterns as a subset of behavioral equivalence method, which focus on a straightforward behavior patterns.

We first provide an example in Section 6.5.1, and then give an overview of the proposal in Section 6.5.2. The one-way-out and generic behavioral equivalence pattern are respectively provided in Section 6.5.3 and Section 6.5.4. At last, some issues are discussed in Section 6.5.5.

### 6.5.1 Example of Behavioral Equivalence

The example from Ex. 6.2 illustrates the key ideas of the behavioral equivalence methods.

**Example 6.2 (Example of Behavioral Equivalence)** *Fig. 6.5.1 is a TPN model that cannot be reduced any more using topology-implicit semantic equivalence patterns. When generating the marking abstraction reachability graph from this net in TINA, it contains **177 states** and **365 transitions**. This system can be seen as two parts: part A standing for the structure in dotted frame, and part B standing for the other parts. Transition $t_4$ is considered as a portal transition between parts A and B.*

*Fig. 6.5.2 shows the reduction result using a behavioral equivalence method. Part A is still in the reduced structure, and part B of original net has been reduced. The reduced TPN only contains **3 states** and **3 transitions**, but it exhibits the same behavior as the original net for the outside observer, which looks at the periodic firing time of the portal transition $t_5$. In both nets, the firing time of $t_5$ exhibits the result in Table 6.5.1. For each occurrence n ($n \in \mathbb{N}$) of fired $t_5$, the firing time interval $[min_n, max_n]$ is $[5 + 17(n-1), 10 + 69(n-1)]$.*

*The key idea of behavioral equivalence reduction is to identify TPN structures, then replace it with behavioral equivalent but reduced structures. The system before reduction and after reduction have the same behavior from the viewpoint of transition property verification (in our case real-time property verification).*

| Occurrence | Time $[t_i^{min}, t_i^{max}]$ | Time Diff $[t_i^{min} - t_{i-1}^{min}, t_i^{max} - t_{i-1}^{max}]$ |
|---|---|---|
| 0 | $[0, 0]$ | - |
| 1 | $[5, 10]$ | $[5, 10]$ |
| 2 | $[22, 79]$ | $[17, 69]$ |
| 3 | $[39, 148]$ | $[17, 69]$ |
| ... | ... | ... |
| n | $[5+17(n-1), 10+69(n-1)]$ | $[17, 69]$ |

**Table 6.5.1:** Example Result: Same Firing Time in Both Net

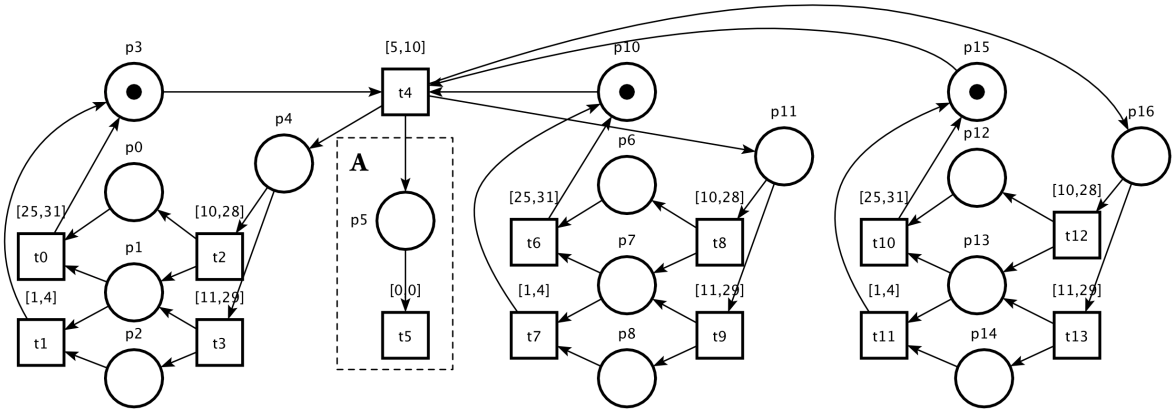



**Figure 6.5.1:** Example of Behavioral Equivalence

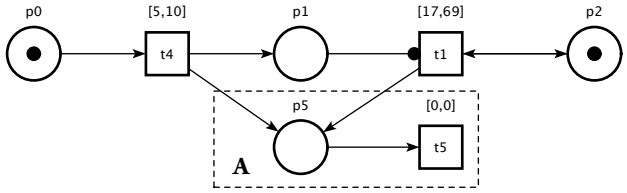



**Figure 6.5.2:** Example Result of Behavioral Equivalence

### 6.5.2 Approach Overview

An overview of the approach is illustrated in Fig. 6.5.3. We first identify and extract the reducible sub-blocks like *A*, *B*, and *C* from the whole system using *Identification* functions. These sub-blocks contain one outgoing and at most one incoming portal TPN transitions. In Fig. 6.5.3, *A* has one outgoing portal transition, while *B* and *C* have one incoming and one outgoing portal transitions. Then the state space of the reducible sub-blocks are reduced using the *Reduction* functions. The reduced sub-blocks ($A'$, $B'$, and $C'$) are built, and then used to replace the original sub-blocks after their soundness is assessed using *Refinement* functions. The *Identification*, *Reduction* and *Refinement* functions rely on the real-time property specification and observer-based verification approaches from our verification toolset.

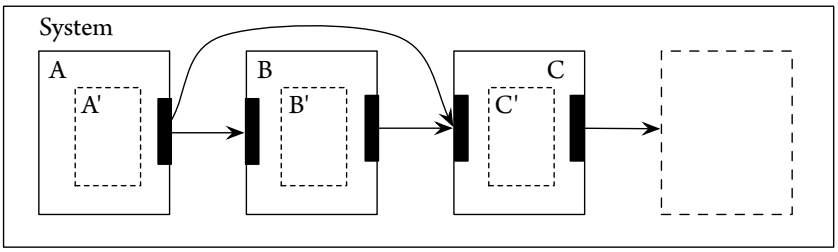



**Figure 6.5.3:** Overview of Behavior Equivalence Approach

### 6.5.3 One-Way-Out Behavioral Equivalence Pattern

Fig. 6.5.4 illustrates the concept of reduction by one-way-out behavioral equivalence. This TPN has 3 sub blocks: A, B and C. All of the three blocks are property-relevant but also reducible. Both of blocks A and B have a portal transition $T_A$ (resp. $T_B$) to block C. Block A has a feature that it only produces tokens via its unique boundary transition $T_A$ periodically or sporadically. The same characteristic can be found on block B. Therefore, from the viewpoint of C, regardless the complex inner behaviors of the blocks A and B, they are nothing but single transitions that may fire regularly under a pattern which sometimes feeds it by some tokens. This observation gives an opportunity to abstract and redefine this *regularity* to a fixed TPN structure that may contains less states and transitions than the one before reduction.

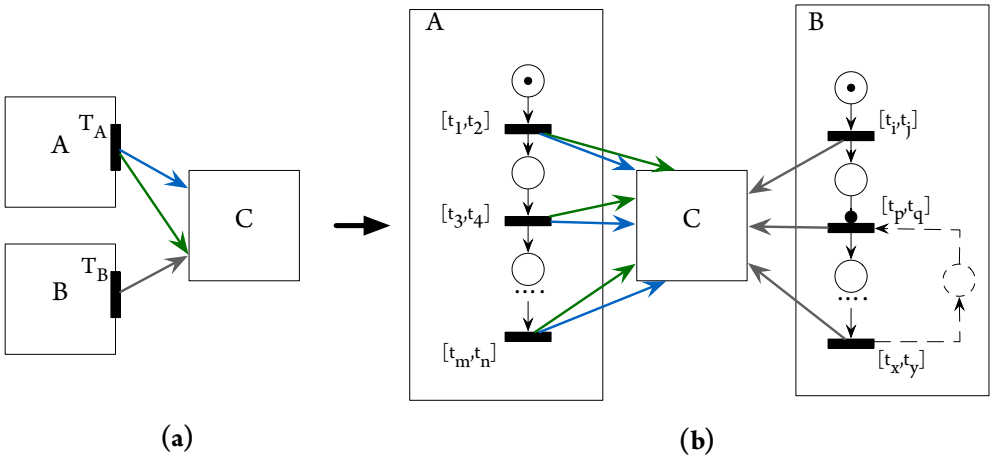



**Figure 6.5.4:** Reduction pattern of Behavioral Equivalence

Fig. 6.5.4 (b) shows how the behavioral pattern of a block like A or B can be simplified. Assuming there is a powerful but invisible observer that will record, for each fired transitions $T_A/T_B$, their occurrence time. Definitively, each occurrence $T_i$ must have a minimal and maximum time interval bound $T_i^{min}$ and $T_i^{max}$. $T_i^{min}$ and $T_i^{max}$ are derived by adding BCET (Best Case Execution Time) and WCET (Worst Case Execution Time) observers on transitions $T_A/T_B$ respectively. We have presented the method for computing WCET value in Section 5.5.2 of Chapter 5. The computation method for BCET is similar to the use of observer composition to represent minimum time interval between Init and current transitions.

The occurrence of $T_A/T_B$ can be finite (block A) or infinite (block B). The method to check whether a transition produces finite or infinite occurrence is to add an *isFinite(E)* observer (presented in Section 5.3.2 of Chapter 5) on the transition.

**Finite Occurrence Transition** If the occurrence is finite, the abstraction is relatively simple: it can be represented by a finite sequential section of transitions $\mathbb{T}_{seq} = \{T_i\}\ (i \in \mathbb{N})$ with adapted time interval $[min_i, max_i]$, $(t_0^{min} = t_0^{max} = 0)$ :

- $min_i = t_i^{min} - t_{i-1}^{min}$
- $max_i = t_i^{max} - t_{i-1}^{max}$

**Infinite Occurrence Transition** If the occurrence is infinite, the above sequential representation is no longer useful. Nevertheless, as our work focuses on finite-state systems, we can deduce that the original behavior of block A/B is finite in an infinite time scope. In other words, there must exist a repeating behavioral pattern in block A/B. In general, such a pattern is composed of a finite sequential section of transitions $\mathbb{T}_{seq} = \{T_i\}\ (i \in \mathbb{N})$ and a loop section of transitions $\mathbb{T}_{loop} = \{T_j\}\ (j \in \mathbb{N})$. The block B in Fig. 6.5.4 illustrates the composition of sequential and loop sections.

In the Fig. 6.5.2 of Ex. 6.2, the part $\{p_0, t_4, p_1, t_1, p_2\}$ corresponds to the reduction result of an infinite occurrence block. $p_0, t_4, p_1, t_1$ is the finite sequential section, and $t_1, p_2$ is the infinite loop section.

The reduction method includes 3 functions: **identification**, **reduction** and **refinement**.

**Identification function** $\mathbf{F}(N) = <A, T_{out}>$ identifies and extracts, for a given TPN $N$, the sub-net $A$ which matches the pattern that could be possible reduced (necessary condition) using one-way-out behavioral equivalence pattern, and the boundary exit transition set $T_{out}$:

**Data**: $A, t$
**Result**: $N_S$
$t_o^{min} := 0, t_o^{max} := 0$ ;
$N_S$.add(new Place(1)) ;
i := 0 ;
**while** *tHasOcc(i++)* **do**
    $t_i^{min}$ := getOccBCET(A,t,i) ;
    $t_i^{max}$ := getOccWCET(A,t,i) ;
    $T_i.min = t_i^{min} - t_{i-1}^{min}$ ;
    $T_i.max = t_i^{max} - t_{i-1}^{max}$ ;
    $N_S$.add($T_i$, new Place(0)) ;
**end**

**Algorithm 2:** Building Sequential Section

- A is a strongly connected graph, $A \subset N$, $T_{out} \subset A$
- $\forall p \in A, (p \in \mathbb{P}^{R(N)}) \wedge (p^+ \subset A) \wedge (p^- \subset A)$
- $\forall t \in A, (t \in \mathbb{T}^{R(N)}) \wedge (t^- \subset A)$
- $T_{out} = \{t | (t \in A) \wedge (t^+ \cap \overline{A} \neq \emptyset)\}$

**Reduction function G**$(A, t)$ = <$N_S, N_L$> extracts, for a given sub-net $A$ matching the above pattern and one of the transition $t$ in the $T_{out}$ set, the equivalent sequential structure $N_S$, and the eventual loop section $N_L$ if the occurrence of $t$ is infinite.

The sub-net $A$ is extracted from the whole system. This function first checks whether the occurrence of $t$ in sub-net $A$ is finite using the `isFinite()` observer. The `isFinite()` observer is given in the section 5.3.2 of chapter 5. In both cases, the $t_i^{min}$ and $t_i^{max}$ are measured using predefined `BCET` and `WCET` observers for the $i^{th}$ occurrence of fired $t$.

In the finite case, there is only a sequential section $N_S$. The set of sequential transitions $\mathbb{T}_{seq} = \{T_i\}$ $(i \in \mathbb{N})$ in $N_S$ is built using $t_i^{min}$ and $t_i^{max}$. Each transition $T_i$ in $\mathbb{T}_{seq}$ is associated with a time range $[T_i.min, T_i.max]$. The algorithm for building $N_S$ from $A$ using the transition $t$ is described in Algo. 2. Initially, $t_o^{min}$ and $t_o^{max}$ are set as 0. $N_S$ starts from an initial place with one token. Whether $t^i$ has occurred is checked using `tHasOcc(i)` function relying on the observer `O(`$t^i$`)`. For each new occurrence $(i)$ of fired $t$, a pair of `BCET` and `WCET` observers are added to $t$ in the sub-net $A$ to compute the $t_i^{min}$ and $t_i^{max}$. Then the time

range $[T_i.min, T_i.max]$ is associated to the transition $T_i$. $T_i$ is added in $N_S$, and an associated new place without token is also added in $N_S$.

In the infinite case, the key issue is to identify the occurrence of fired $t$ that divides the sequential section $N_S$ and the loop section $N_L$. The algorithm Algo. 3 is proposed to build the $N_S$ and $N_L$ sections by searching for the loop starting transition (*loopStartIndex*) and the length of loop (*loopLength*).

Since the occurrence of the fired $t$ is infinite, an occurrence bound value is predefined as *occThreshold* to stop the algorithm. Since the Identification function $\mathbf{F}(N)$ uses necessary conditions, the identified sub-net $A$ is considered as non-reducible if the loop section cannot be found using *occThreshold*. Another bound value *loopThreshold* judges whether the *loopStartIndex* and the *loopLength* are found. If the loop pattern holds for *loopThreshold* times, it is considered that this division of $N_S$ and $N_L$ is statistically correct. It is obvious that no matter how big that *loopThreshold* is, the assurance cannot reach 100%, because the execution is infinite. In order to make sure that the replacement refines exactly the same behavior as the original system, a pre-check (refinement) must be performed before integrating the reduced structure into the whole system.

**Refinement function $\mathbf{H}(A, t, N_S, N_L)$** checks the refinement between the identified reducible sub-net $A$ and the equivalent net $A' = N_S \cup N_L$, for a given transition $t$ in $T_{out}$, in the following way:

- Let $n_S$ be the length of sequential section $N_S$, and $n_L$ be the length of the loop section $N_L$.
- Let $T_S^m$ be the $m^{th}$ transition of $N_S$, and $T_L^k$ be the $k^{th}$ transition of $N_L$. $(m, k \in \mathbb{N}, 1 \leq m \leq n_S, 1 \leq k \leq n_L)$
- For the $i^{th}$ firing of $t$ in $A$, denoted as $t^i$, creates an occurrence observer for $t^i$ and adds it to $A$;
- Creates $n_S$ times the predicate observer of maximum time interval; adds these ones between the event modifier $t_i$ $(1 \leq i \leq n_S)$ in $N_S$ and the transition $T_S^m$ $(m = i)$ in $N_L$; checks if the following assertion holds for all $T_S^m$: $|T(t^i) - T(T_S^m)| \leq T_S^m.max - T_S^m.min$
- Creates an event modifier $t^p$, $(p = (i - n_S) \mod n_L)$ for the $i^{th}$ fired $t$ in $A$, and adds the observer of this modifier to $A$. As $i$ is infinite, $p$ is also infinite. Thus, $t^p$ stands for an infinite event.
- Creates $n_L$ times the predicate observer of maximum time interval; adds them between the event modifier $t^p$ in $N_S$ and the transitions $T_L^k$ $(k = i - n_S)$ in $N_L$; finds all the outgoing arcs of $t$ inside

**Data**: $A, t, occThreshold, loopThreshold$
**Result**: $N_S, N_L$
$t_{\circ}^{min}$ := 0, $t_{\circ}^{max}$ := 0 ;
$N_S$.add(new Place(1)) ;
occ := 0 ;
**while** *occ*++ $\leq$ *occThreshold* **do**
  $t_{occ}^{min}$ := `getOccBCET(A,t,occ)` ;
  $t_{occ}^{max}$ := `getOccWCET(A,t,occ)` ;
  **for** *loopStartIndex = 0; loopStartIndex < occ; loopStartIndex ++* **do**
    **for** *loopLength = 1; loopLength* $\leq$ *occ - loopStartIndex; loopLength ++* **do**
      match : = 0 ;
      **for** *index = loopStartIndex; index* $\leq$ *occ - loopLength; index++* **do**
        **if** *isSame*(<$t_{index}^{min}$, $t_{index}^{max}$>, <$t_{index+loopLength}^{min}$, $t_{index+looplength}^{max}$>) **then**
          match++ ;
        **end**
        **else** break;;
      **end**
      **if** *match* $\geq$ *loopThreshold* **then**
        **for** *k = 1; k < loopStartIndex; k++* **do**
          $T_k.min = t_k^{min} - t_{k-1}^{min}$ ;
          $T_k.max = t_k^{max} - t_{k-1}^{max}$ ;
          $N_S$.`add(`$T_k$`, new Place(0))` ;
        **end**
        **for** *k = loopStartIndex; k < loopStartIndex + loopLength; k++* **do**
          $T_k.min = t_k^{min} - t_{k-1}^{min}$ ;
          $T_k.max = t_k^{max} - t_{k-1}^{max}$ ;
          $N_L$.`add(`$T_k$`, new Place(0))` ;
          $N_L$.`connect(lastPlace,` $T_{loopStartIndex}$`)` ;
        **end**
        return ;
      **end**
    **end**
  **end**
**end**

**Algorithm 3:** Building Loop Section

$A$ (the outgoing arcs outside $A$ are not included), redirects the sources of these arcs from $t$ to the "merge" transition in the time interval observer (this will be detailed in the following example Ex. 6.3), perform the same redirection for $A'$; checks if $|T(t^p) - T(T_L^k)| \leq (T_L^k.max - T_L^k.min)$ holds for all $T_L^k$.

When verifying the loop section using refinement, as $A$ and $A'$ are both infinite, we propose a mechanism to ensure that the refinement is performed between two equivalent occurrence. We use an example (Ex. 6.3) to illustrate this issue and how the refinement works.

**Example 6.3 (Example of Refinement)** *In the Fig. 6.5.5 (a), the model $S_A$ is an identified sub-net of the whole system S. $E_A$ is the target portal transition. From the viewpoint of $E_A$, the net $S_A$ exhibits periodic behavior, with [3, 10] as the period. $S_B$ is the reduced net which exhibits the same property-specific behavior as $S_A$. From the viewpoint of $E_B$, $S_B$ also exhibits periodic behavior with [3,10] as the period. $S_B$ does not have the sequential section but only the loop section with one transition. Before replace $S_A$ by $S_B$ in S, the refinement is performed to verify $S_A$ and $S_B$ have equivalent property-specific behavior.*

*According to the above steps, as $n_S = 0$, we directly verify the equivalence of the loop section. In $S_B$, $n_L = 1$, thus, $p = 1$. In Fig. 6.5.5 (a), $E'_A$ stands for $t^1$. The **obs** is used to check $|E'_A - E_B| < 8$ between infinitely occurred $E'_A$ and $E_B$. We check the MMC assertion $\neg(Overflow_A \vee Overflow_B)$ in the state class graph preserving markings. The result is false. This is obvious. For example, if each time $E_A$ occurs at the instant 2, while $E_B$ occurs at instant 10, their occurrences quickly become non equivalent.*

*This problem is solved using a reset mechanism. In Fig. 6.5.5 (b), we first find all the outgoing arcs inside $S_A$ and $S_B$ (here arcs $a_A$ and $a_B$). Then we redirect the source of $a_A$ from $E_A$ to $T_{merge}$, and redirect the source of $a_B$ from $E_B$ to $T_{merge}$. In this way, after the comparison of each occurrence, the loop restarts. The MMC assertion $\neg(Overflow_A \vee Overflow_B)$ is true. In this way, it is proved that $S_B$ is equivalent to $S_A$ in terms of property-specific behavior.*

The reduction will likely make the verification more scalable if the reduced behavior has less states and transitions. However, this will not always be satisfied and we cannot predict the reduction effectiveness before refinement. This gain introduces additional costs: the computation of function $\mathbf{G}()$ and $\mathbf{H}()$. Although the reachability graph of sub-TPN will be much smaller, both $\mathbf{G}()$ and $\mathbf{H}()$ will be executed several times for a given sub-TPN. In this work, in order to avoid meaningless computation, the identification of sub-TPN matching the pattern will only check those strongly connected graphs with more than 10 transitions.

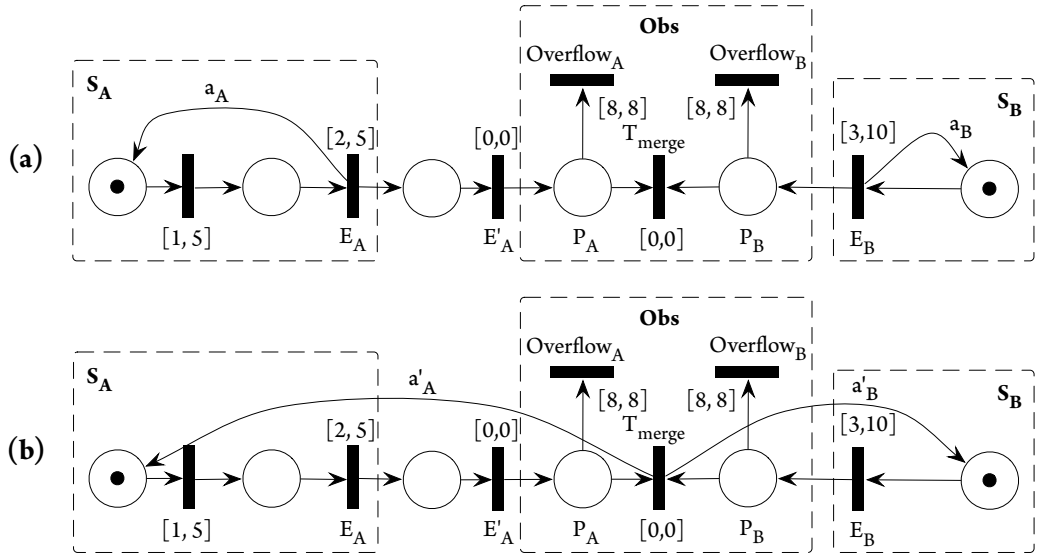



**Figure 6.5.5:** Example of Refinement

A more sophisticated gain-cost balance strategy could be studied in future works, like pattern recognition for target sub TPN together with machine learning on their improvement ratio.

### 6.5.4 Generic Behavioral Equivalence Pattern

The above one-way-out pattern of refinement gives hints for more generic patterns. First of all, a reducible sub-net must be independent of its surrounding behavioral context. It means that whether this structure is "knocked out" from the original TPN or not, it will exhibit exactly the same behavior whenever it is measured, in terms of occurrence times, firing time range of each transition and marking bound of each place. This feature is the key of behavioral equivalence-based reduction because it turns, during model checker's state exploration, the combination problem of $O(N \cdot M)$ into a divide-and-conquer problem of $O(N + M \cdot \delta)$, where $N$ is the state unfolding complexity caused by this sub-net, $M$ is the complexity caused by the other parts of the TPN, and $\delta$ is the complexity introduced by the refined component of the sub-net; it is expected (and often the case according the early test results) that $1 \leq \delta \ll N$.

Let $S$ be the system TPN, $A$ be the reducible sub-net of $S$, and $C$ be the complement structure of $A$ in $S$ (see Fig. 6.5.6 (a)). Once $A$ has been identified, the next task is to convert it into a reduced structure $B$,

which in order to have eventually $\delta \ll N$. Theoretically, the arcs are the interfaces between $A/B$ and $C$, and both transition and place can be the portal elements of $A/B$ to connect to $C$. In this thesis, we chose to use transitions as the portal elements because most of our observers are event/transition-based. This will ease both the reduction function $\mathbf{G}()$ when we "knock out" $A$ and find its behavioral equivalence parameters for building $B$, and the refinement function $\mathbf{H}()$ when we associate $A$ and $B$ to verify they are equivalent using observers.

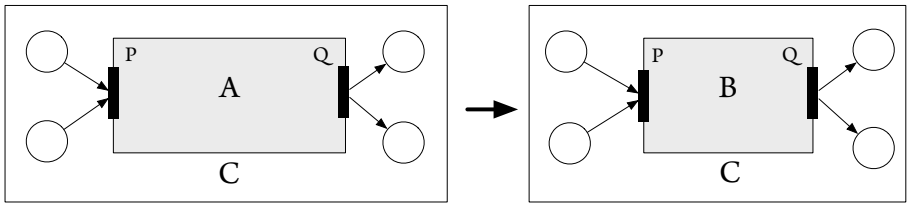



**Figure 6.5.6:** Generic Behavioral Equivalence Pattern

For the generic behavioral equivalence pattern, the two functions $\mathbf{G}()$ and $\mathbf{H}()$ are the same as the one given for the one-way-out pattern. We focus on how to identify the sub-nets matching this generic pattern. More precisely, an identification function is proposed to identify the sub-net structures which consist of two portal transitions such as transitions P and Q in Fig. 6.5.6.

Before presenting the identification function, two transition sets should be predefined.

**Definition 6.1 (Impact Set)** *Le E be a transition in a given TPN system S, the impact set of E is the set of TPN transitions that impacts the behavior of E in S, noted as* $\mathbb{S}_E^S$*. In other words, E causally depends on the elements in* $\mathbb{S}_E^S$*.*

**Definition 6.2 (Impacted Set)** *Let E be a transition in a given TPN system S, the impacted set of E is the set of TPN transitions that are impacted by E in S, noted as* $\mathbb{T}_E^S$*. In other words, the elements in* $\mathbb{T}_E^S$ *causally depends on E.*

**Computation of $\mathbb{S}_E^S$ and $\mathbb{T}_E^S$** Computing precisely the causality between TPN's transitions is a tough task, which requires a state class graph-based analysis. This leads to the following paradox: if we are able to generate the reachability graph for the given TPN before the reduction, we may not need to reduce it any

more. The solution is to use an over-approximation of causality, i.e. use the TPN's structure relevance to deduce the causality. The relevance between two transitions at TPN topology level is the necessary condition that they are causally dependent. Therefore we can reuse the algorithm Algo. 1 to compute these two sets.

Suppose $\mathbb{R}_E$ is the set including all relevant transitions of E in a system, we have

- $\mathbb{S}_E^S = \mathbb{R}_E$
- $\mathbb{T}_E^S = \{X | E \in \mathbb{R}_X\}$

We give an example (Ex. 6.4) to illustrate the defined impact and impacted sets.

**Example 6.4 (Example of Impact and Impacted Sets)** *In the net N of Fig. 6.5.7, the target transition is $T_3$. The impact set $\mathbb{S}_{T_3}^N$ is the set of relevant transitions $\{T_2, T_4\}$. The impacted set $\mathbb{T}_{T_3}^N$ is computed using the sets of relevant transitions of the other transitions. For $T_1$, its relevant transitions are $\{T_2, T_3, T_4\}$; for $T_2$, its relevant transitions are $\{T_3, T_4\}$. The transition $T_4$ does not have any relevant transition. As $T_3$ is relevant to $T_1$ and $T_2$, the impacted set $\mathbb{T}_{T_3}^N = \{T_1, T_2\}$.*

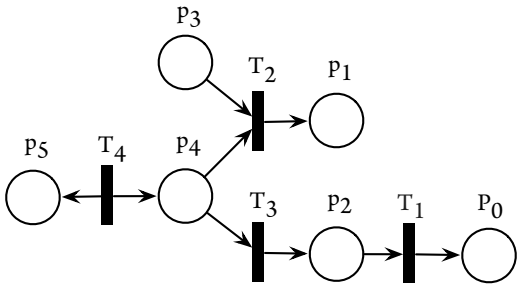



**Figure 6.5.7:** Example of Impact and Impacted Sets

**Identification function $\mathbf{F}_2(N)$ = <$A, P, Q$>** identifies and extracts, for a given TPN $N$, the sub-net $A$ that matches the pattern that could be possibly reduced by generic behavioral equivalence pattern, and the pair of portal transitions $P$ (incoming) and $Q$ (outgoing). $\mathbb{S}_P^N$ (resp. $\mathbb{S}_Q^N$) and $\mathbb{T}_P^N$ (resp. $\mathbb{T}_Q^N$) are the impact and impacted sets of $P$ (resp. $Q$). The algorithm Algo. 4 is used in $\mathbf{F}_2(N)$.

**Data**: $N$
**Result**: $A, P, Q$
**forall the** *transitions* $L, M \in S$ **do**
  **if** $\mathbb{T}_L^N \subset \mathbb{T}_M^N$ **then**
    $A = \mathbb{T}_M^N - \mathbb{T}_L^N + \{L\}$ ;
    $A' = A \backslash \{L, M\}$ ;
    found := true ;
    **forall the** $E \in A'$ **do**
      **if** $!\,(\mathbb{T}_E^{A'} \subseteq A \wedge \mathbb{S}_E^{A'} \subseteq A)$ **then**
        found = false ;
        break ;
      **end**
    **end**
    **if** *found* **then**
      $P = L, Q = M$ ;
      return ;
    **end**
  **end**
**end**

**Algorithm 4:** Generic Behavioral Equivalence Pattern Identification

$\mathbf{F}()$ guarantees that the whole structure $A$ will only casually impact part $C$ via transition $Q$, and will only be casually impacted by part $C$ via transition $P$ in Fig. 6.5.6. Therefore when the refinement is using P and Q as portal transitions, it will not change the behavior of the whole system.

## 6.5.5 Discussion

### 6.5.5.1 Behavior Coverage

In some cases, using the observed minimum/maximum time range to replace the old sub-structure do not refine fully the original behavior because there might be some "holes" in this range. For example, a transition can fire during [10,15] or [20,30], but never during [15,20]. If we use directly [10,30] as the [min, max] time range in refinement, the real-time behavior is extended. Therefore, a detailed observation must be taken into account to distinguish the case with and without these time-holes.

For a given observed range [min, max] of transition $T$ at its $i^{th}$ occurrence, the *check*$_k$ *exist* $T_i$ *between*

*k and k+1* will be executed for all $min \leq k < max$. If $check_k$ does not pass, the range will be broken into two sections: [min, k] and [k+1,max]. To be more general, if $check_{k_1}$, $check_{k_2}$, ... $check_{k_n}$ do not pass, the final refined equivalent time ranges of this occurrence will become $[min, k_1]$, $[k_1 + 1, k_2]$, ..., $[k_n + 1, max]$. Accordingly, the sequential transition of the equivalent sub-net will be refined to a sub-structure which contains all possible fireable time range, but also eliminates those impossible ranges.

Fig. 6.5.9 (a) shows that the transition $T$ in the reduced sub-net $A$ exhibits a firing time range $[t_3, t_4]$. But there exists time holes on this time range, as shown in Fig. 6.5.8. The transition $T$ should be replaced by the sub-range structure (grey part in Fig. 6.5.9 (b)). In each sequential branch of this structure, the second transition keeps the firing time constraint, while the first transition with time constraint [0,0] plays a role ensuring that this branch will be selected and fired equally.

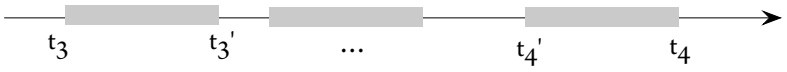



**Figure 6.5.8:** Behavioral Equivalence Pattern: Hole on Time Interval

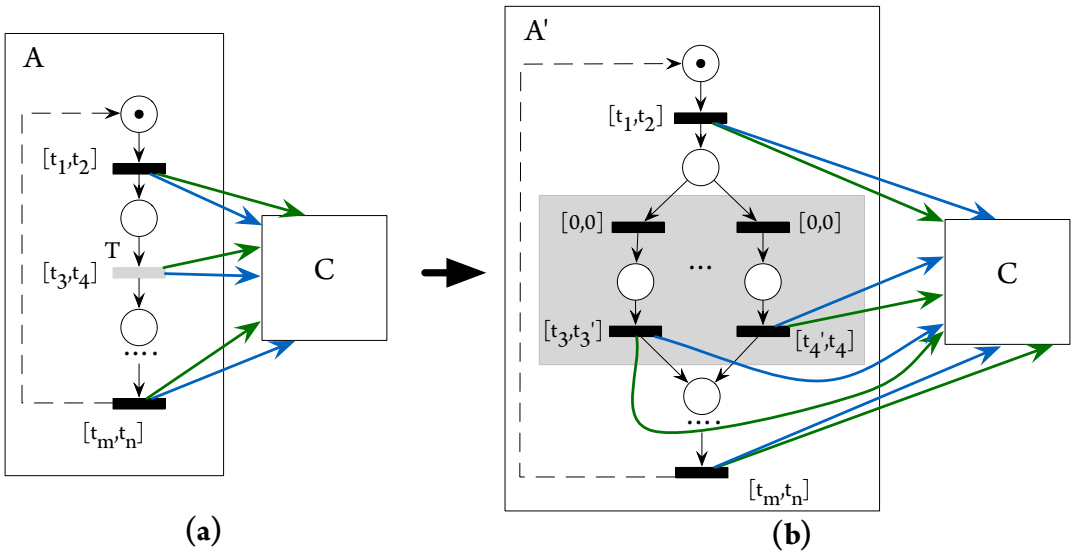



**Figure 6.5.9:** Behavioral Equivalence Pattern: Deal with Hole on Time Interval

#### 6.5.5.2 Predict the revenue-over-investment

More specifically, for a given TPN, there might be more than one reducible sub-components available. Different order of reduction will eventually impact the global verification time. A trivial strategy used in this work uses the sub-net's size (in terms of transition's number in the sub-net) to rank the order of reduction: the smaller one will be reduced first.

#### 6.5.5.3 Formal Proof and Future Research Direction

In this behavioral equivalent approach, we first identify and extract the reducible sub-blocks from the whole system using an *Identification* function. Then the state space of the reducible sub-blocks are reduced using a *Reduction* function. The reduced sub-blocks are derived, and are then used to replace the original sub-blocks after their soundness is assessed using a *Refinement* function.

At the time of writing this thesis, the *Reduction* and *Refinement* functions rely on the real-time property specification and observer-based verification approaches in our verification toolset. Suppose a reducible TPN sub-net is $N_s$ and its reduced sub-net is $N'_s$. We use the *Reduction* function to search for the sequential and loop sections that are used as the behavioral pattern of $N_s$, and then verify if this pattern behaves the same as the system's real behavior using the *Refinement* function. If verified, an $N'_s$ conforming to this pattern will replace the $N_s$. The reduction and refinement functions can be formally specified and proved. This should be further studied in the near future.

On the other hand, once an $N_s$ is identified, in order to compute the $N'_s$, some related TPN observers need to be associated to the $N_s$, and the corresponding state class graphs are then generated $m$ times ($m$ depends on the behavior of $N_s$). Indeed, this approach reduces the state space explosion problem in asynchronous systems using a time–memory tradeoff. But this approach can still be improved by decreasing the time used for the reduction and refinement. It is possible to build the sequential and loop sections for $N'_s$ by generating the state class graph only once and then analyzing its topology structure. This will be an interesting future research direction.

Boucheneb and Barkaoui proposed in [BB13] an effective method for reducing interleaving semantics redundancy in the reachability analysis of Time Petri Net. Their work showed that the union of state zones reached by different interleavings of the same set of transitions is not necessarily a state zone. They established sufficient conditions which ensure that this union is a state zone and showed how to compute this state zone without computing intermediate ones. It is possible to draw lessons from this work and propose more efficient property specific reduction methods for TPN models.

## 6.6 Conclusion

This chapter proposes TPN reduction approaches that are applied before verifying real-time properties by model checking. The classic verification methodology will encounter scalability problem very quickly along with the growth of system size, as it follows an implicit purpose: once the reachability graph has been generated, it can be reused to verify many properties of the system, just by changing the modal logic formulas. This consideration requires to build the most concrete and precise transition system to be able to assess any kind of properties. It makes sense if the assessed system does not change often and if there is a large number of requirements to assess. However, it is well known that the generation of reachability graph for large scale models is the most expensive phase in terms of resource consumption. Theoretically, generating reachability graphs only once seems to be resource-saving by eliminating the effort of re-generating. However, this global-resource-saving principle implies an assumption that is sometimes false: we can generate the reachability graph with common available resources.

We propose to create models equivalent to the original TPN in terms of property-related system behavior, but with less states and transitions. This reduces directly the model scale before performing state space generation. Compared to other TPN reduction techniques, our approach focus on the following four contributions:

1. **Reduction based on TPN with data manipulation features.**
   The TINA toolset supports extended TPN with data handling called Time Transition Systems (`tts`), including the precondition Pre and action Act functions. Our approach is based on this new feature. When eliminating property-irrelevant structures, all variables and place marking referred by the property-relevant structures should be preserved.

2. **Reduction driven by property verification.**
   The reduction is property-driven, which means that most structures irrelevant to the verification of a given property will be eliminated. Compared to existing techniques [SB96], our approach is driven by property, because TPN systems have been extended with property observers. This property-driven characteristic allows removing much more transitions in the TPN models, which directly signifies more reduced state space in model checking.

3. **Reduction based on topology-implicit semantic equivalence.**
   We propose practical topology-implicit semantic equivalence reduction patterns which are com-

monly found in TPN models with observer structures for real-time property verification. In this work, redundant zero-time patterns and sequential encapsulation patterns are proposed.

4. **Reduction based on behavioral equivalence.**
   In some cases, it is complex to detect and extract a localized topology pattern. However, it is mandatory to have the whole TPN reduced. Reduction methods based on behavioral equivalence are proposed. These methods identify sub-TPN which exhibits the same behavior as the original nets. Before reducing the pattern-matched structures, we use refinement functions to ensure the reduced net corresponds exactly to the original behavior.

# Part III

# Contribution to Fault Localization

# 7

# Automated Fault Localization in Model Checking


## Résumé

Ce chapitre présente une approche automatisée d'assistance à la localisation des éléments de modèle participants à la non satisfaction des exigences lors de la vérification de modèles. Il s'agit de faciliter et d'accélérer la mise au point de l'architecture des systèmes temps réels en localisant et en classant les éléments suspects dans un modèle pour lequel une exigence n'est pas satisfaite.

La génération d'un contre exemple dans le cas où une exigence n'est pas satisfaite est un service essentiel fourni par les outils de vérification de modèles. Ces contre exemples illustrent des comportements indésirables mais possibles dans le modèle considéré du système. Cette information peut aider les utilisateurs à corriger la conception défectueuse du système. Cependant, il est généralement très difficile de comprendre l'origine de l'échec en utilisant des contre exemples car d'une part ceux-ci sont en général très longs pour les erreurs qui ne sont pas triviales, et d'autre part une erreur peut résulter d'une combinaison de facteurs. Après identification de l'erreur, il est également complexe d'extraire des contre exemples les indices

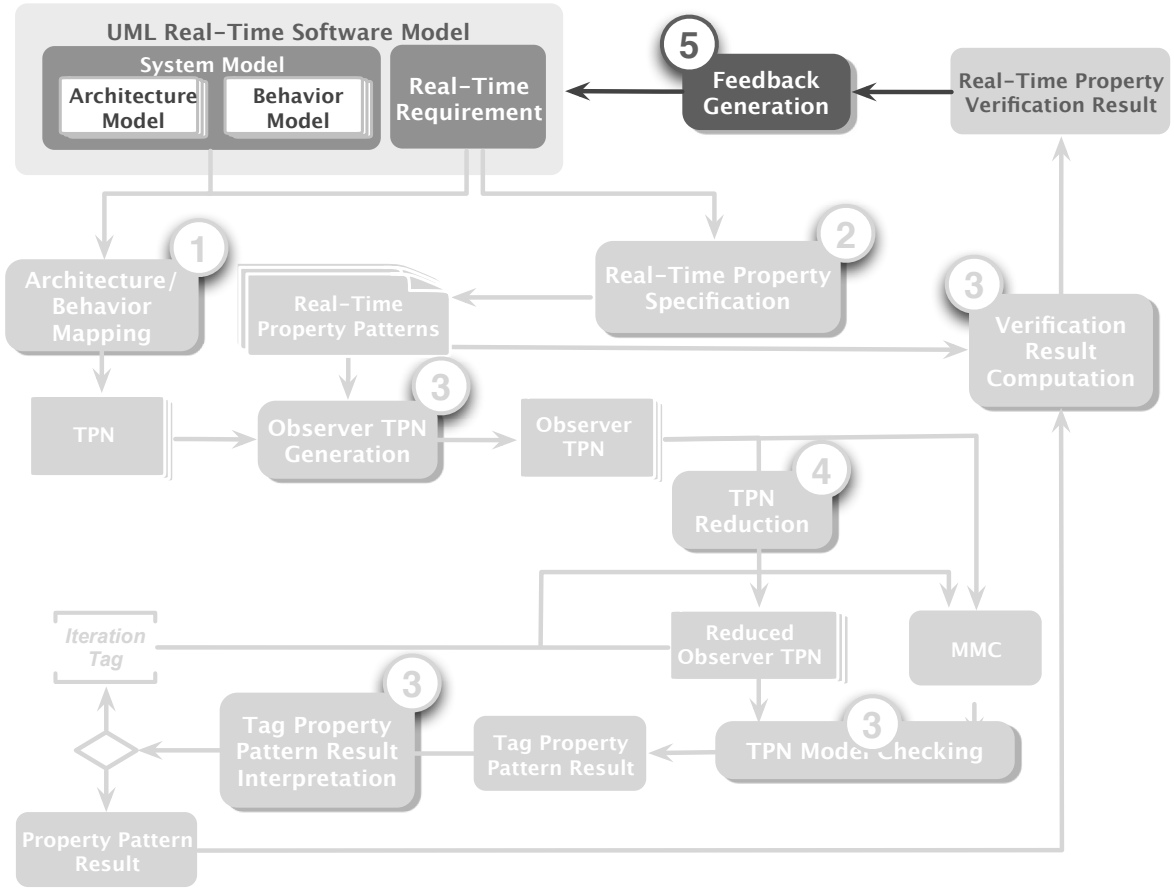
pour aider à améliorer la conception. Les contre exemples produits par les outils de vérification des modèles sont généralement des traces qui représentent des séquences d'états et de transitions qui conduisent à un état dans lequel l'exigence n'est pas satisfaite. D'une part, celles-ci sont souvent longues et difficiles à interpréter car l'origine réelle de l'erreur peut être une transition, voire une combinaison de transitions apparaissant à n'importe quelle position dans la trace, ce qui nécessite une longue analyse par les concepteurs. D'autre part, les contre exemples ne correspondent qu'à certains scénarios de défaillance spécifique qui ne sont pas forcément les plus pertinents pour la compréhension et la correction de l'origine de la défaillance. L'analyse peut être plus précise si elle s'appuie sur l'ensemble des scénarios. L'approche de vérification proposée s'appuie sur l'accessibilité qui construit l'ensemble des traces conduisant dans les états de défaillance ou ne conduisant pas dans les états souhaités. Elle permet donc de disposer de toutes les informations exploitables pour l'analyse des échecs.

Ces différents aspects nous permettent de conclure que la transmission aux utilisateurs des seuls contre exemples n'offre qu'une aide pratique très limitée pour identifier l'origine des défauts et corriger ceux-ci. Notre objectif est de signaler directement des éléments suspects car ils contribuent à un grand nombre de scénarios de défaillance. Pour cela, il est nécessaire de localiser et de classer les éléments potentiellement défectueux dans les modèles en s'appuyant sur les résultats de la vérification.

L'analyse de l'origine des défaillances dans la vérification de modèles est difficile à cause de l'utilisation d'abstractions. Au moment de la réduction, le conflit entre la précision du modèle et le coût de la vérification est un problème clé. L'abstraction est souvent nécessaire dans la vérification pour réduire la taille de l'espace d'états. Elle élimine certains éléments de la sémantique sans rapport avec les exigences mais peut également combiner certains aspects liés à celles-ci. Mais la distinction des parties combinées pourrait aider dans la compréhension de la défaillance.

Les techniques de localisation des défaillances actuellement disponibles pour la vérification de modèles produisent généralement un ensemble d'éléments suspects dans les modèles sans classement particulier. Dans cette partie, nous avons amélioré l'efficacité de l'analyse des défaillances en fournissant un facteur de suspicion, lorsqu'une exigence n'est pas satisfaite. Inspiré par la théorie de la divergence de Kullback-Leibler et la technique TF-IDF (Term Frequency - Inverse Document Frequency) une mesure exploitée dans l'exploration de données textuelles, le facteur de suspicion est utiliser pour classer les transitions suspectes. Nous construisons l'intégralité des traces d'erreur dans le graphe d'accessibilité en utilisant tous les états de défaillance. Le facteur de suspicion est ensuite calculé en utilisant la contribution à la défaillance de chaque transition sur toutes les traces d'erreur. Cette contribution est calculée à partir de l'entropie et

de l'entropie différentielle de transition. Nous appliquons cette approche aux réseaux de Petri temporisés et à la méthode de vérification à base d'observateurs pour obtenir toutes les traces d'exécution défaillantes et les états défaillant dans le graphe d'accessibilité préservant le marquage. L'approche proposée est illustrée à l'aide d'une étude de cas simple, et ensuite validé avec des métriques classiques sur un banc d'essai automatisé qui génère des modèles comportant des interblocages.




*Progress Map 5:* Automated Fault Localization in Model Checking

In this chapter, we present an automated fault localization approach based on model checking to ease and accelerate debugging by locating and ranking the suspicious elements in a model when a safety property is unsatisfied (Progress Map 5). Counterexamples produced by model checkers often stand for error traces, which represent sequences of system states and transitions that are often lengthy and difficult to understand, as they provide every steps (or an abstraction of steps) in the execution leading to the violation states. The origin of error might be anywhere along these traces and even a combination of transitions that are not contiguous, thus it requires a lengthy analysis by designers (Challenge 5 in page 22). Inspired by the `TF-IDF` (term frequency-inverse document frequency) measure and the `Kullback-Leibler Divergence` theory, we propose a suspiciousness factor to rank the potentially faulty transitions. We apply this approach to property-specific `TPN` model relying on observers-based verification approach presented in Chapter 5 to

provide all the faulty execution traces and the violation states in the marking reachability graph. Based on the mapping semantics from UML to TPN, the faulty transitions can be back-traced from TPN to UML. The approach is illustrated using a simple TPN case study, and then further assessed for its effectiveness and efficiency on an automated test bed. (Contribution 5 in page 24)

## 7.1 Introduction

One of the designers of model checking E. Clarke wrote in [Clao8]: "It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems." Generating a counterexample in case a formula is violated is a key service provided by model checkers. As exceptions of requirements, counterexamples are expected to display some unwanted but possible behaviors of the system to help the user(s) in correcting the faulty system design. However, it is usually an exhausting work to understand the origin of failure using counterexamples and to extract from them useful debugging clues to help improving the design. Counterexamples produced by model checkers often stand for error traces, which represent sequences of system states and transitions that are often lengthy and difficult to understand, as they provide every steps (or an abstraction of steps) in the execution leading to the violation states. More precisely, the origin of error might be anywhere along these traces and even a combination of transitions that are not contiguous, thus it requires a lengthy analysis by designers. On the other hand, the counterexamples derived from the assessment of temporal logic formulae usually only contain some specific failure scenarios. Even if the toolsets could generate all possible faulty scenarios, this is not a common feature. Without relying on all the possible error traces, fault analysis might not be precise enough in most cases.

Based on the above understanding, we advocate that feeding back end users with counterexamples provides limited help in understanding the origin of defects and in improving model design. Our ultimate goal is to detect and to provide the designer with ranked suspicious faulty elements. In other words, we aim to locate and rank the potentially faulty model elements relying on model checking results. The fact is, although model checking has been developed as a mature and heavily used verification and debugging technique, the automated fault localization analysis relying on model checking results is still mostly an open challenge.

Existing automated fault localization techniques in model checking usually produce a set of suspicious statements without any particular ranking. In this chapter, we introduce an automated fault localization ap-

proach based on model checking to ease and accelerate debugging by locating and ranking the suspicious elements in model checking when a safety property is unsatisfied. Inspired by the TF-IDF (term frequency-inverse document frequency) measure and the Kullback–Leibler Divergence theory, we propose a suspiciousness factor to rank the potentially faulty transitions. We construct error traces in the reachability graph using the violation states. The suspiciousness factor is then computed using the fault contribution of each transition on all the error traces. The fault contribution is computed using the entropy and differential entropy of transitions. We apply this approach to Time Petri Net (TPN) models relying on the observers-based verification approach presented in Chapter 5 to provide all the faulty execution traces and all the violation states in the state class graph preserving marking. This approach is illustrated using a simple TPN case study, and then further assessed for its effectiveness and efficiency using an automated test bed.

Compared to existing works, the main contributions are:

- Existing automated fault localization techniques usually produce a set of suspicious statements without any particular ranking. Our approach uses a suspiciousness factor based on information theory to rank all statements. Up to now, the current test results have shown that this approach is efficient and effective.

- It is based on a novel idea according to our knowledge and the communication reviews we got: using information theory to compute the entropy and differential entropy of error traces in the state class graph. We start with comparing the similarity between information retrieval for documents and fault localization for error traces. Then we propose an algorithm based on Kullback–Leibler Divergence to compute fault contribution of TPN transitions on error traces.

- The automated fault localization is applied at the reachability graph level, thus it can be used together with different verification models (TPN, TA, etc.). It provides fault location feedback for safety properties.

- It has been integrated in the temporal property verification framework for UML-MARTE real-time designs. This fault localization approach is applied to the property-specific verification models (in our case TPN). These property-specific TPN models are derived from the end-user model (UML-MARTE) using the property-specific mapping. After performing the property-specific reduction for TPN, the marking abstraction state-class graph is generated to assess the temporal properties using the observer-based model checking. If the property is dissatisfied, the violated states are used to locate the origin

of error. Based on the mapping semantics from the UML model to the TPN model, the faulty transitions in the marking graph are back-traced from the TPN model to the UML model.

This chapter is organized as follows: Section 7.2 states the targeted problem; Section 7.3 gives some preliminaries; Section 7.4 provides an automated fault localization approach based on the reachability graph; Experimental results derived from a set of test cases are presented in Section 7.5 to assess the effectiveness and efficiency of the approach; Section 7.6 introduces the algorithm for back-tracing faulty transitions from the verification model to end user model.

## 7.2 Problem Statement

### 7.2.1 Abstraction Issue

Fault localization in model checking is challenging as models usually have concurrent and indeterministic behaviors with many possible execution traces. These behaviors are mostly related to the use of abstraction in their design. Without appropriate information, fault localization may not be precise enough. Given a sequential, or synchronized concurrent, program which exhibits less execution traces, various debugging methods are available to detect and locate faulty statements. In model-based diagnosis, the use of abstraction is mandatory to reduce the state space explosion problem. At the time of writing, the conflict between model precision and verification cost is a key issue in model checking and model-driven engineering (MDE), therefore a compromise is made to remove the unnecessary information for some verification purpose while keeping all the property-related information.

There exist mostly three levels of abstraction:

- First, abstraction lies in the design models at early phases of MDE. For example, the specification of the execution time interval for each action and the condition of transition between actions is enough to verify the worst-case execution time (WCET) of an activity, while the detailed algorithm of an action might not be required.

- Secondly, abstraction lies in the mapping from design models to verification models. This one also targets verification-ease. The property-irrelevant elements can be reduced to decrease the state space size for model checking.

- At last, abstraction lies in the reachability graph. Usually, several types of abstraction preserving different kind of information are provided by model checkers when generating a reachability graph. In order to scale as much as possible, we choose in our work the highest possible abstraction.

### 7.2.2 Fault Localization Issue

Sometimes it is difficult, even for seasoned experts, to analyze the fault origin. We take a simple example (see Ex. 7.1) to illustrate this issue.

**Example 7.1 (Fault Localization Example)** *Assume a system consists of two concurrent processes A and B. Both execute only once. The execution time is [5,10] for A, and [3,7] for B. The expected temporal property $\mathcal{P}$ is **Always** A **After** B.*

*It is obvious that $\mathcal{P}$ is unsatisfied. The design fault occurs either on A or on B. To remove this violation, we can either replace the time constraint of A by [8,10], or replace the time constraint of B by [3,4]. However, without extra information, A and B exhibit the same suspicion. If an extra information is available, e.g. the best case execution time (BCET) of B is 5, then the time constraint of B cannot anymore be replaced by [3,4], thus the suspicion of B is largely decreased.*

This example is simple enough to be analyzed manually, while it is impossible for more complex system with thousands of transitions. Any modification on a transition may impact the verification result through time constraint propagation.

### 7.2.3 Existing Works

According to the survey from [Ali12], existing automated fault localization techniques in model checking usually produce a set of suspicious statements without any particular ranking.

[BNR03] proposed to analyze fault localization using one single counterexample that violated the expected properties in a particular case. Whenever a counterexample was found, the approach compared the error trace derived from the counterexample to all the correct traces that conformed to the requirement. On the observed error and correct traces, the transitions that led to the deviation from correct traces are marked as suspicious transitions. This technique has been implemented in the SLAM model checker [BR01].

[GV03] proposed to rely on multiple counterexamples. It defined the traces that started from initial states and ended with error states as negative traces, and the traces that did not take the error state as previ-

ous state as positive traces. It distinguished the transitions that existed in all positive traces; the transitions that appeared in all negative traces; the transitions that existed in one of positive traces but not in any negative traces; and the transitions that appeared in one of negative traces, but not in any positive traces. The algorithm then used the above marked transitions to identify the origin of failure. This method has been implemented in the `Java PathFinder` [HP00] toolset.

[Gro04] proposed to define a distance between the error trace and the successful traces. The distance was then used to find the closest successful trace to the counterexample. The causes of error were then derived from the comparison results between the closest successful trace and the counterexample. This method was implemented in the `Explain` toolset [GKL04b].

[CGS04] extended the concrete distance metric approach of [Gro04] to handle abstract executions of programs and properties expressed in `LTL`, resulting in improvements in both the distance metric used and the expressiveness of explanations over earlier work. This technique has been implemented in the `MAGIC` toolset [CCG+04].

[JM11b] proposed an approach that transformed the fault localization problem to a Max-SAT problem. It used only one failing trace and the corresponding input to build the Max-SAT formulation. This method has been implemented in the `BugAssist` toolset [JM11a].

### 7.2.4 Proposed Solution

Compared to the above existing works, our approach will improve the effectiveness of fault localization by providing a suspiciousness factor which is used to rank the suspicious transitions in verification models. The suspiciousness factor is computed using the fault contribution of each transition in the error traces derived from the reachability graph. Inspired by information retrieval theory, the basic idea is to compute the entropy and differential entropy of the transitions in the error traces.

At the time of writing, we have not yet combined our approach with the previous related works. This can be investigated in the future.

## 7.3 Preliminaries

### 7.3.1 Reachability Graph & Violation States

Reachability graphs are used to solve reachability problems in model checking. They contain all the states in the execution of a system and all the transitions between these states. In the TINA toolset, depending on the selected options, `tina` builds reachability graphs of different abstraction levels, expressed as Kripke transition systems (`ktz`). For example, the marking graph of a bounded Petri Net preserves marking reachability properties but not linear time temporal (`LTL`) properties.

Finding all violation states in the reachability graph is the first step for error localization. There exist two model checkers in the TINA toolset which provide (partially) this functionality. `muse` is a modal $\mu$-calculus model checker which can find all the violation states against the given modal $\mu$-calculus (`MMC`) formula in the reachability graph. `selt` is a State/Event `LTL` model checker which can give one counterexample (therefore not all the violation states) when the `LTL` formula is checked as false. If there is a need to find all violation states for `LTL` formula, one can apply the büchi automaton translating algorithm by iterating all the states in the reachability graph.

### 7.3.2 Error Traces

We aim to compute the fault contribution of each transition in the error traces. The error traces are constructed using the violation states in the reachability graph when a safety property is unsatisfied.

**Definition 7.1 (Fault Contribution)** *Fault Contribution ($C_F$) is a suspiciousness factor to evaluate a transition's suspicion level. It is used to rank the suspiciousness of transitions.*

The first step to locate the fault source is to enumerate all the violation states in the reachability graph. According to the observer-based model checking approach for `TPN` presented in Chapter 5, we use the marking graph as the reachability graph. A `TPN` state can be seen as a pair $(M, D)$, in which $M$ is a marking, and $D$ is a set of vectors called the firing domain. The `MMC` formula is used to check the marking existence, such as $(M_P = 1)$ or $(M_P = 0)$, where $M_P$ is the marking in the observation place $P$. Once the given `MMC` is violated, the set of violation states in the reachability graph is built.

**Definition 7.2 (Error Trace)** *There may exist several paths from the initial state $s_0$ to a violation state $s_v$ in the reachability graph. For all the states $\{s_i\}$ on each path, all the outgoing transitions of $s_i$ are gathered in a set called error trace $\pi$.*

We consider not only the transitions on the path that leads from $S_0$ to $S_v$ in the definition of error trace but also the direct outgoing transitions of all the states in the execution traces that lead to correct states. Indeed, in TPN, the transitions outgoing from the same place can mutually influence each other. A faulty transition can change the way a correct transition is fired if they are both outgoings from the same place. The correct transition will diminish the $C_F$ of the faulty transition.

**Example 7.2 (Error Trace Example)** *In Fig. 7.3.1, $s_0$ is initial state, $s_v$ is a violation state. In the execution trace from $s_0$ to $s_v$, there exist four states $\{s_0, s_1, s_2, s_3\}$ (apart from $s_v$). The state $s_3$ is in a correct trace. When the system is in state $s_2$, it is possible to transit to $s_7$ leading to a correct trace, or to $s_3$ leading to a violation state. If $s_7$ is removed from the graph, $s_3$ will have higher fault contribution for the violation state. The outgoing transitions of these four states are considered as error traces $\pi$, i.e., $\pi = \{t_0, t_1, t_2, t_1, t_5, t_4, t_2, t_3, t_4\}$.*

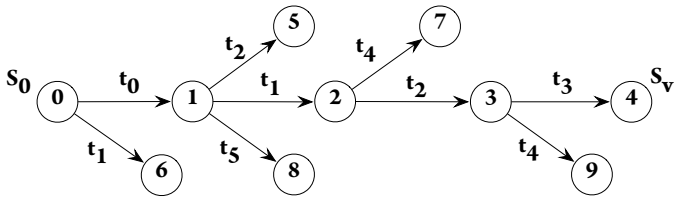



**Figure 7.3.1:** Error Trace Example

### 7.3.3 Kullback-Leibler Divergence Applied to Textual Documents

Kullback-Leibler Divergence (also called information divergence, information gain, relative entropy) [KL51] is a fundamental equation of information theory that qualifies the proximity of two probability distributions.

**Definition 7.3 (Kullback-Leibler Divergence)** *Kullback-Leibler Divergence (KL) is a measure in statistics that quantifies how close a probability distribution $P = \{p_i\}$ is to a model (or candidate) distribution $Q = \{q_i\}$.*

*The KL-divergence of Q from P over a discrete random variable is defined as*

$$D_{KL}(P \parallel Q) = \sum_i P(i) \ln \frac{P(i)}{Q(i)} \tag{7.1}$$

*Note: In the above definition,* $0 \ln \frac{0}{0} = 0$, $0 \ln \frac{0}{q} = 0$, *and* $p \ln \frac{p}{0} = \infty$.

Three properties are derived from it:

- **Asymmetry:** $D_{KL}(P \parallel Q) \neq D_{KL}(Q \parallel P)$.
- **Non-negative:** $D_{KL}(P \parallel Q) \geq 0$, $D_{KL}(P \parallel Q) = 0$ if $P$ matches $Q$ exactly.
- **Additive:** If $P_1, P_2$ are independent distributions, with the joint distribution $P(x, y) = P_1(x)P_2(y)$, and $Q, Q_1, Q_2$ likewise, then $D_{KL}(P \parallel Q) = D_{KL}(P_1 \parallel Q_1) + D_{KL}(P_2 \parallel Q_2)$.

`Kullback-Leibler Divergence` has many applications. We give an example of its application to text classification [BM98]. A textual document $d$ is a discrete distribution of $|d|$ random variables, where $|d|$ is the number of terms in the document. Let $d_1$ and $d_2$ be two documents whose similarity we want to compute. This is done using $D_{KL}(d_1 \parallel d_2)$ and $D_{KL}(d_2 \parallel d_1)$.

### 7.3.4 Term Frequency - Inverse Document Frequency

Another major application is the `TF-IDF` (Term Frequency - Inverse Document Frequency) algorithm [Jon72]. `TF-IDF` is a numerical statistic which reflects how important a term is for a given document in a corpus (collection) of documents. It is often used as a weighting factor in information retrieval and text mining. Variations of the `TF-IDF` weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance to a given user query [MRS08].

Suppose we have a collection of English textual documents and aim to determine which documents are most relevant to the query "the model checking". We might start by eliminating documents that do not contain the three words "the", "model", and "checking", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document and sum them all together; the number of times a term occurs in a document is called its term frequency (`TF`).

However, as the term "the" is very common, this might incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms

"model" and "checking". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less common words "model" and "checking". Hence an inverse document frequency (IDF) factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

To summarize the above explanation, TF-IDF is the product of two statistics, TF and IDF. TF stands for the frequency of a term in a document, and it reflects how important a term is in this document. IDF stands for the frequency of a term in different documents, and it reflects how distinguishable a term is to the document. The TF-IDF weight for a term both increases with the number of occurrences in a document (TF component); and the rarity of the term across the entire collection (IDF component).

## 7.4 Ranking Suspicious Faulty Transitions

Inspired by the TF-IDF algorithm, we propose a probabilistic fault localization approach based on the Kullback-Leibler Divergence. A relevance weight $C_F(t)$ is computed to assess the contribution of a transition $t$ in the error traces leading to violation states and thus its contribution to the fault.

### 7.4.1 Core Idea

In the TF-IDF algorithm, each term in the documents will contribute to the semantics of keywords. Some terms are considered as significant if they are more relevant to the semantics of keywords. This is similar to the fault contribution caused by a given transition in an error trace in model checking. Fig. 7.4.1 compares the similarity between semantic contribution of terms in documents and fault contribution of transitions in error traces. Some terms in documents have closer semantic relation to the keywords, the occurrence of these terms provide more semantic contributions to the occurrence of keywords. Similarly, the fault propagation depends on the topology of error traces, the occurrence of some transitions will provide more fault contributions to the occurrence of violation states.

The semantic contribution of a term in documents is measured by TF-IDF, where TF is the contribution of a term in single document, and IDF is the contribution of a term in a collection of documents. The fault contribution to the violation states $\{s_{vi}\}$ caused by a transition $t$ on error traces $\{\pi_i\}$ can also be evaluated by a similar measure $C_F(t)$, defined as TC-ITC (Transition Contribution - Inverse Trace Contribution). $C_F(t) = \mathtt{TC}(t) \cdot \mathtt{ITC}(t)$.

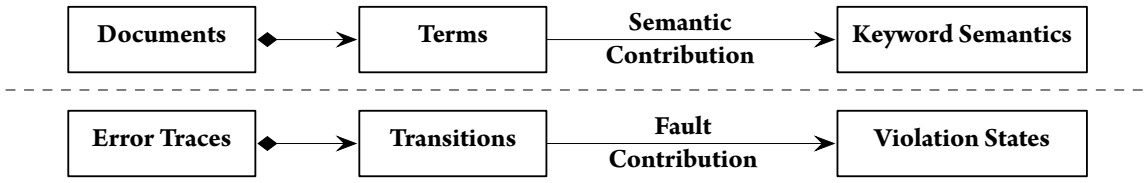



**Figure 7.4.1:** Comparison to TF-IDF

### 7.4.2 Fault Localization Example

We use a simple example (see Ex. 7.3) to present the whole fault contribution analysis algorithm.

**Example 7.3 (Fault Localization Example)** *Fig. 7.4.4 (without the observer part) is a TPN model with 10 transitions $\{t_0, t_1, \ldots, t_9\}$. It has two main execution paths (respectively through $t_1$ and $t_2$), both have a loop with a bound of 2. The expected temporal property is: system's BCET is bounded within a given time T, i.e. BCET $>$ T. We aim to automatically identify the potentially faulty transitions, and to rank them according to their fault contributions to the violation states.*

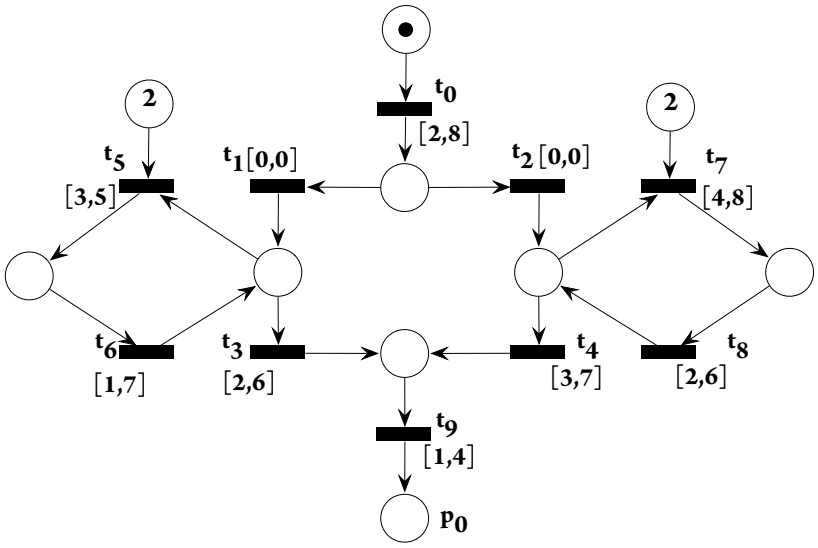



**Figure 7.4.2:** Example of Fault Localization Algorithm

### 7.4.3 TC-ITC Algorithm

#### 7.4.3.1 Building Error Traces

The length of error trace $L$ is defined as the number of states before the violation state $s_v$. The algorithm for enumerating all the error traces in the reachability graph is trivial, but the impact of state cycles in error traces needs to be discussed.

The reachability graph in Fig. 7.4.3 contains a state cycle $C_s$ ($s_1 \xrightarrow{t_1} s_3 \xrightarrow{t_4} s_4 \xrightarrow{t_3} s_1$). The error traces passing through $s_1$ may loop in $C_s$. Take one error trace as an example, the trace passing through states $s_0, s_1, s_3, s_4, s_6$ is

$$s_0 \xrightarrow{t_0} \{s_1 \xrightarrow{t_1} s_3 \xrightarrow{t_4} s_4 \xrightarrow{t_3} s_1\}_n \xrightarrow{t_8} s_6$$

where $n$ represents the number of time the cycle is repeated. A repetition will not increase the fault contribution as the system's behavior is restricted to the three states. Therefore, the cycle can be treated as a single point represented by a chain of transitions (here $t_1, t_4, t_3$). In other words, $n$ is taken to be 1.

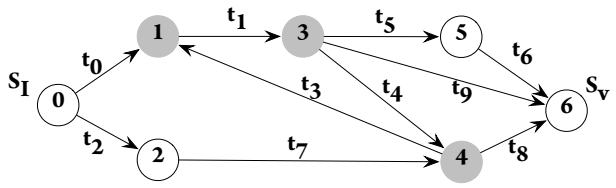



**Figure 7.4.3:** Cycle on Error Traces

To assess the expected temporal property, a TPN observer is added in the example (see Fig. 7.4.4). This observer is used to check the minimum time interval between two events. The observer is linked to transition $t_9$ because place $p_0$ is the end place of the running system. The MMC assertion to be checked is: $N(\neg(\neg p_1 \wedge p_2)) = N_A$, where $N(\neg(\neg p_1 \wedge p_2))$ is the number of states satisfying $\neg(\neg p_1 \wedge p_2)$, $N_A$ is the total number of possible states in the system's execution.

For the expected property BCET $> T$, when $T = 10$, the verification result is *False*. $N_A$ is 39, transition number is 50, while $N(\neg(\neg p_1 \wedge p_2))$ is 37. Therefore, there exist two violation states ($S_{11}$ and $S_{23}$) in the reachability graph (see Fig. 7.4.5). The error traces are:

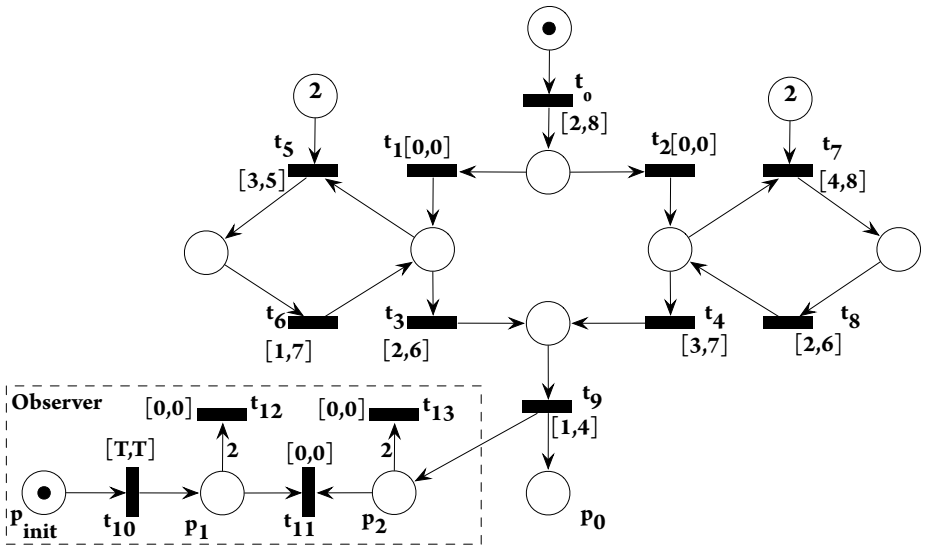



**Figure 7.4.4:** Verification of Fault Localization Example

$$\pi_1 = \{t_0, t_1, t_2, t_5, t_{10}, t_3, t_9, t_{10}\}$$
$$\pi_2 = \{t_0, t_1, t_2, t_4, t_{10}, t_7, t_9, t_{10}\}$$
$$\pi_3 = \{t_0, t_1, t_2, t_5, t_{10}, t_3, t_6, t_{10}, t_5, t_{10}, t_3, t_{10}, t_9\}$$

#### 7.4.3.2 Transition Contribution

**Definition 7.4 (Transition Contribution (TC))** *TC is a measure of the occurrence frequency of a transition $t$ in an error trace $\pi$. It reflects a transition's contribution to violation state $s_v$ on $\pi$. It is defined to be*

$$TC(t) = \frac{1}{M} \sum_{i=1}^{M} \frac{Q_i}{L_i} \tag{7.2}$$

*where $Q_i$ is number of occurrence times of $t$ on error trace $\pi_i$ of length $L_i$, and $M$ is the number of error traces.*

It uses the raw frequency of a transition in a error trace, i.e. the number of times that transition $t$ occurs in error trace $\pi$.

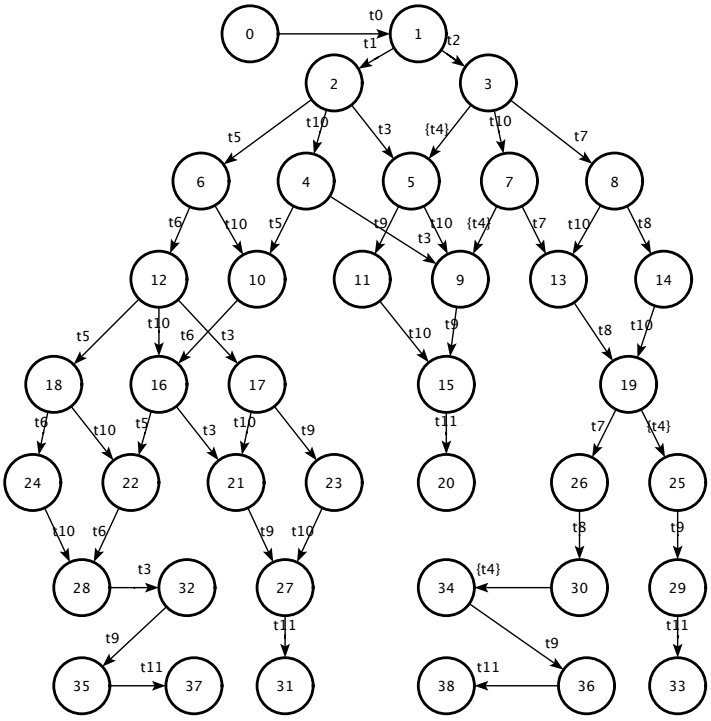



**Figure 7.4.5:** Reachability Graph of Fault Localization Example

#### 7.4.3.3 Inverse Trace Contribution

**Definition 7.5 (Inverse Trace Contribution ITC)** *ITC is a measure of whether a transition t is common or rare among all the error traces derived from all the violation states. It is defined to be*

$$ITC(t) = \log_2 \frac{M}{\sum_{i=1}^{M} X_i}, \tag{7.3}$$

*where* $X_i = \begin{cases} 1 & \text{if } t \text{ occurs at least one time in an error trace} \\ 0 & \text{otherwise} \end{cases}$ *and M is the total number of error traces.*

It is obtained by dividing the total number of error traces by the number of error traces containing the violation, and then taking the logarithm of that quotient.

#### 7.4.3.4 Fault Contribution of Transition

The weight TC-ITC is the product of the above two measures. In some cases, this product is 0, which does not mean it cannot be the fault source but only implies that the elements make the least contributions to the violation states and have the least probability comparing to the others and should be checked at last.

It is expected that the ranking of faulty possibility computed by the algorithm corresponds to manual analysis and human intuition, as it was shown for TF-IDF and other applications. We use the example from Fig. 7.4.4 to illustrate how they are matched. A more detailed automated benchmark is presented in the next section.

We give the computation result of the example in Fig. 7.4.6. The results show the fault contributions (normalized for comparing the trend) of each transition when $T$ varies from 5 to 50.

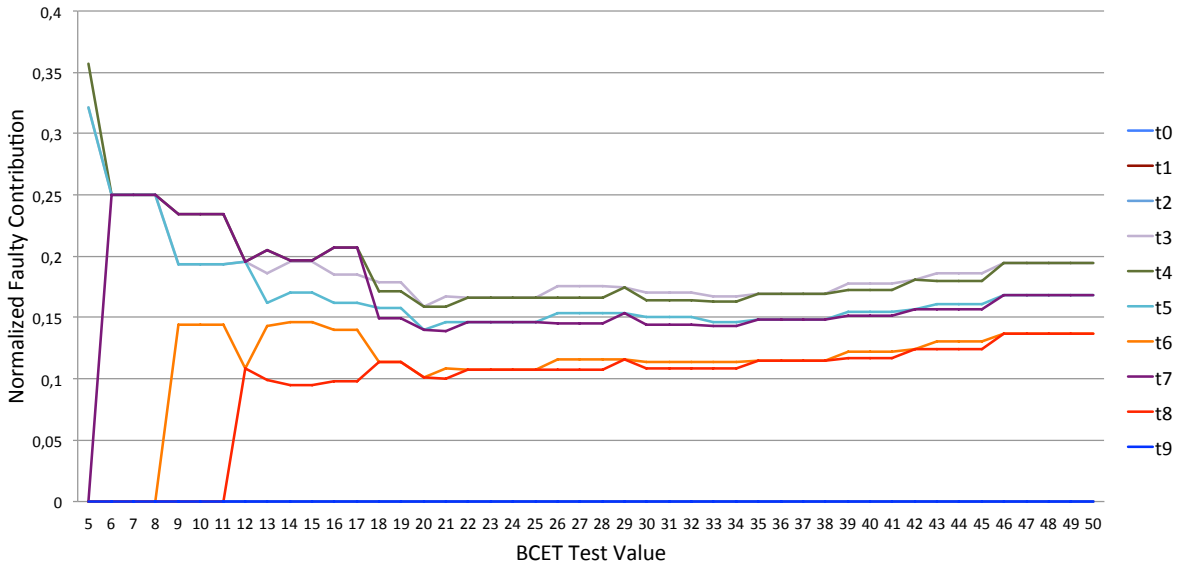



**Figure 7.4.6:** Feedback of Fault Localization Example

We explain the results in Fig. 7.4.6 in the following part:

- $1 \leq T < 5$: since the BCET of the system is 5, there will not be any violated state and accordingly no fault localization will be launched.
- $T \geq 47$: since the WCET of the system is 47, the reachability graph will not have any change after this

threshold, therefore the fault contribution of each transition will preserve the same value as $T = 47$.

- $5 \leq T < 47$: since $T$ represents the expected BCET of the system, all execution with time inferior to $T$ will be considered as violation. Without any other information, a reasonable heuristics can then be derived from this assertion: for BCET property, the less a transition can contribute/has contributed to the global execution time, the higher risk it will be the fault origin. Another intuition-valid rule is: when an element holds a more complex function, it has a higher risk to have design faults. To heuristically quantify the coefficient of these two different types of fault contribution is a subjective measure often context-dependent. In order to avoid this indecisive discussion, each time we encounter this situation in our example, we will just explain the two aspects without trying to combine them into one score for matching the ranking.

  We will now check whether the computed ranking of fault source matches these two references. We observe in this figure statistical trends that:

  - **Topologically symmetric pair ($t_3$, $t_4$) has a higher risk to be the fault cause than ($t_5$, $t_7$) and ($t_6$, $t_8$)**. This matches the heuristic rule because in whichever execution, $t_3$ and $t_4$ will only contribute once to the global execution time (i.e. [2,6] and [3,7] respectively), while $t_5$, $t_6$, $t_7$ and $t_8$ can at most execute twice and will contribute more (i.e [6,10], [2, 14], [8, 16] and [4, 12] respectively).
  - **In each symmetric pair of above, $t_3 \geq t_4$, $t_5 \geq t_7$ and $t_6 \geq t_8$.** This keeps demonstrating that it is always the one with the smallest execution time that get more risk to be the faulty one.
  - **$t_0$, $t_1$, $t_2$ and $t_9$ are equally the least suspicious elements.** This conforms to the intuition because in all execution paths, whether good or bad, $t_0$ and $t_9$ will always be executed therefore no information added for assessing the risk that they have to be the fault cause. For $t_1$ and $t_2$ it is a similar approach, because a design fault will either be on the left side or the right side, and in all execution paths of the left (resp. the right ) side, $t_1$ (resp. $t_2$) will always be executed.
  - **Pair ($t_5$, $t_7$) has a higher risk than ($t_6$, $t_8$).** Generally since $t_6/t_8$ has smaller execution time than $t_5/t_7$, it shall be more risky to be the error source according to the first heuristic rule. However, since $t_5/t_7$ plays a role that not only postpone the execution (like $t_6/t_8$), but also branch the execution path ($t_6/t_8$ do not have this function), their chance to be the fault cause will be re-distributed and raised as the second rule is engaged.

## 7.5 Experiments

To assess the success of a fault localization algorithm, many important criteria should be measured, such as effectiveness, precision, informativeness, efficiency, performance, scalability and information usefulness. In our work, we assess our approach by using two very significant criteria: effectiveness and efficiency.

**Effectiveness.** An effective fault localization method should point out the origin of failure. The effectiveness can be evaluated by the precision. According to the survey [WD09], the effectiveness can be assessed by a score called EXAM in terms of the percentage of statements that have to be examined until the first statement containing the fault is reached [EWDC10, WQ09, WSQG08, WWQZ08]. A similar score using the percentage of the program that need not be examined to find a faulty statement has been defined in [CZ05, JH05, RtPR03]. These two scores provide the same information, but the EXAM score is more direct and easier to understand. In this work, we use the EXAM score to assess the effectiveness of our approach, which is the percentage of transitions that have to be examined until the first faulty transition is found.

**Efficiency.** The fault localization techniques in model checking, like other techniques, should terminate in a timely manner, limited by some resource constraints. The efficiency can be assessed by the scalability and the performance.

In order to assess the effectiveness and efficiency of the proposed method, we have designed an automated test bed.

### 7.5.1 Automated Test Bed

The test bed will randomly generate systems which might have deadlocks, then apply the proposed analysis algorithm and check that it detects the introduced deadlocks. The general scenario we have chosen is deadlock in systems with concurrent use of resources. The main reason to use this common scenario as template is because it is relatively easy to create a scalable system with deadlock heuristically. Although the test bed only contains only one property, the effectiveness and efficiency evaluations will be meaningful for all safety properties, because the approach is based on the analysis of reachability graph.

For a given TPN system $S(P, R, M)$, $P$ are the processes which run infinitely and need a resource before the next task (a task is represented by a transition); $R$ are resources which are shared by all processes, but only accessible in an exclusive way; $M$ is a matrix to decide whether process $P_i$ will need to access resource $R_j$. Coffman identified four conditions that must hold simultaneously in order to have a deadlock [CES71]:

1. **Mutual exclusion condition** The resources involved are non-sharable.

2. **Hold and wait condition** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.

3. **No-preemptive condition** Resources already allocated to a process cannot be preempted.

4. **Circular wait condition** The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list.

According to these conditions, each process is designed to be moderately consuming the resource, i.e. it will use its resources exclusively, always release one before locking another. The order in which a resource is accessed in each process is however random, which establishes the necessary condition of deadlock. In practice, the first three conditions can be constructed statically when building the test case, while the fourth one can only be checked dynamically during the system's execution. Therefore, the generated TPN will not systematically guarantee that a "real" deadlock will occur. For example, among the 10,000 generated test cases that may have only one deadlock, the "real" deadlock occurs in 400 cases. Among the 10,000 generated test cases that may have 9 deadlocks, the "real" deadlocks occur only in one single case.

To improve the success of creating a deadlock in the system, we introduced another mechanism to enforce deadlocks: randomly let some processes during some tasks forget to release a resource it is locking. These tasks are then considered as the error source of system's deadlock.

With a generated system and its already known faulty transitions (release-forgot tasks), the test bed will apply our method to compute the fault contribution of each task.

### 7.5.2 Evaluation of Efficiency

We have generated thousands of test cases by assigning $P$ and $R$ values from 5 to 20, creating 1 to 9 faulty transitions, with all the other parameters totally random. To create systems with deadlocks, we generate 10,000 cases for each fault number from 1 to 9. After examining the circular wait condition, most of these cases are deadlock-free, therefore the number of deadlocked systems is in fact much smaller than 10,000. The exact number of deadlock test cases is shown as the second column in Table 7.5.1.

**Table 7.5.1:** Efficiency Evaluation

| System | | | Evaluation |
|---|---|---|---|
| **Fault Num.** | **Deadlocked Systems** | **Av. State/Transition** | **Average Time (s)** |
| 1 | 400 | 4949 / 15440 | 2.9092 |
| 2 | 517 | 2428 / 7130 | 1.1244 |
| 3 | 500 | 9884 / 31237 | 3.3533 |
| 4 | 402 | 8811 / 26663 | 2.5998 |
| 5 | 303 | 6756 / 18247 | 1.2196 |
| 6 | 504 | 27094 / 75808 | 5.064 |
| 7 | 757 | 104857 / 304741 | 15.0072 |
| 8 | 100 | 112306 / 283004 | 15.0289 |
| 9 | 1 | 241920 / 583200 | 36.906 |

The tests are performed on a 2.4 GHz Intel Core 2 Duo processor running Mac OS X 10.6.8. The system parameters and efficiency evaluation results are shown in Table 7.5.1. The average time of evaluation shows that the approach is efficient for large scale system.

### 7.5.3 Evaluation of Effectiveness

The effectiveness evaluation is shown in Table 7.5.2. We give out EXAM score, EXAM score variance, rank, and rank variance for the best and worst cases, and then show the average EXAM score and rank. The EXAM score varies from 0.7% to 13.3% for best cases, and varies from 3.9% to 18.6% for worst cases. In average, EXAM varies from 2.3% to 15.9% which corresponds to ranking results from 1 to 8. The stability is represented by the variance result. These experimental results shows our approach is effective. The user only need to assess 15.9% model elements in the worst case to find the error source.

## 7.6 Back-Tracing Fault Transitions in UML

In the above automated fault localization approach, we compute fault contribution $C_F$ for each transition in the error traces in the reachability graph. From the viewpoint of designers, the verification model TPN might be transparent to them, what they expect to get as feedback are guidance information in UML.

**Table 7.5.2:** Effectiveness Evaluation

| F. N. | Best Case | | | | Worst Case | | | | Average | |
|---|---|---|---|---|---|---|---|---|---|---|
| | EXAM | EXAM Var | Rank | Rank Var | EXAM | EXAM Var | Rank | Rank Var | EXAM | Rank |
| 1 | 0,13335 | 0,00134 | 3,25 | 1,79 | 0,18603 | 0,00244 | 4,33 | 1,63 | 0,15969 | 3,79 |
| 2 | 0,04229 | 0,00219 | 1,1 | 1,75 | 0,09574 | 0,00213 | 2,11 | 1,75 | 0,069015 | 1,605 |
| 3 | 0,02108 | 0,00106 | 0,75 | 1,52 | 0,05892 | 0,0009 | 1,75 | 1,52 | 0,04 | 1,25 |
| 4 | 0,00722 | 0,0004 | 0,26 | 0,49 | 0,039 | 0,00042 | 1,26 | 0,49 | 0,02311 | 0,76 |
| 5 | 0,02044 | 0,0017 | 0,83 | 2,95 | 0,0478 | 0,00162 | 1,83 | 2,95 | 0,03412 | 1,33 |
| 6 | 0,05369 | 0,00336 | 2,46 | 7,36 | 0,0766 | 0,0033 | 3,46 | 7,36 | 0,065145 | 2,96 |
| 7 | 0,08857 | 0,00372 | 4,61 | 10,9 | 0,10822 | 0,0037 | 5,61 | 10,9 | 0,098395 | 5,11 |
| 8 | 0,13091 | 0,00099 | 7,3 | 3,95 | 0,14905 | 0,001 | 8,3 | 3,95 | 0,13998 | 7,8 |
| 9 | 0,10169 | 0 | 6 | 0 | 0,11864 | 0 | 7 | 0 | 0,110165 | 6,5 |

In Chapter 3, we have mapped UML models to TPN using property-driven approach. Each element in the chosen UML diagram has been mapped to a TPN structure for the purpose of property-verification-ease. The algorithm to compute fault contribution of a UML element is trivial.

Suppose an UML element $E$ is mapped to a TPN structure $S =< P, T >$, where $P$ is the set of places and $T$ is the set of transitions, $T = \{t_1, t2, ..., t_i\}$. Then the fault contribution of $E$ is computed as below:

$$C_F(E) = \sum_i C_F(t_i) \tag{7.4}$$

## 7.7 Conclusion

Automated failure analysis and fault localization in model checking is a hard problem, due to information reduction caused by model abstraction. Yet, it is a key issue, as providing counterexamples is not enough to help designers in debugging models. This may require a great deal of human effort to locate faulty elements. Some works have provided good results by producing a set of suspicious faulty elements without particular ranking factor.

In this chapter, inspired by the theory of Kullback-Leibler Divergence and its successful application TF-IDF in data mining, we start with comparing the similarity between information retrieval for documents and fault localization for error traces. We propose to compute the fault contributions of each transition on

error traces. The fault contribution is the product of transition contribution (TC) and inverse trace contribution (ITC). Based on the mapping semantics between UML models and TPN model, the faulty transitions are then back-traced to UML model. The approach is illustrated using a simple case study, and then further assessed for its effectiveness and efficiency on a designed automated test bed.

The main contributions of the current chapter [GPC14a] is summarized as follows:

1. Existing automated fault localization techniques usually produce a set of suspicious statements without any particular ranking. Our approach uses a suspiciousness factor based on information theory to rank all statements. The current test results have shown that our approach is efficient and effective.

2. It is based on a novel idea according to our understanding: using information theory to compute the entropy and differential entropy of error traces in the reachability graph. We start with comparing the similarity between information retrieval for documents and fault localization for error traces. Then we propose an algorithm based on Kullback–Leibler Divergence to compute fault contribution of TPN transitions on error traces.

3. The automated fault localization is applied at the reachability graph level, thus it can be used together with different verification models (TPN, TA, etc.) if the verification toolset can provide several (in not all) erroneous execution traces and violated states. It provides fault location feedback for safety properties.

4. It has been integrated in the temporal property verification framework for UML-MARTE real-time designs. We apply this fault localization approach to the property-specific verification model (in our case TPN). This property-specific TPN model is derived from the end-user model (UML-MARTE) using the property-specific mapping. After performing the property-specific reduction for TPN, the marking abstraction state-class graph is generated to assess the temporal properties using the observer-based model checking. If the property is dissatisfied, the violated states are used to locate the origin of error. Based on the mapping semantics from the UML model to the TPN model, the faulty transitions in the marking graph are back-traced from the TPN model to the UML model.

# Part IV

# Industrial Application & Conclusion

# 8

# Application to Flight Management System


## Résumé

Dans ce chapitre, nous utilisons un cas d'étude avionique développé par Michael Lauer en 2013 pour tester les différentes méthodes proposées. Il s'agit d'une partie du système de gestion de vol FMS (Flight Management System) . Dans ce cas d'étude, nous étudions les exigences en terme de latence des communications et de fraîcheur des données.

Les architectures avioniques modulaires intégrées IMA (Integrated Modular Avionic) sont utilisées pour exécuter un ensemble d'applications partageant des ressources de calcul, appelées modules, communiquant par un réseau partagé AFDX (Avionics Full DupleX switched ethernet) et connecté à un ensemble de capteurs. Chaque fonction s'exécute au sein d'une partition du module. Le cas d'étude traite d'une sous-partie du système de navigation dont l'objectif est de contrôler l'affichage d'informations de navigation sur les écrans de pilotage. Le système de navigation interagit avec l'équipage au travers d'écrans et de claviers. Sur requête du pilote ou du copilote, saisie au moyen de leurs claviers respectifs, le système doit

afficher les informations du prochain point de navigation (way point) sur les écrans multi-fonction (Multi Function Display). Ces informations sont de deux types: des informations statiques et des informations dynamiques mises à jour périodiquement. Pour des raisons de sûreté, le FMS repose sur une architecture redondante. Sur chaque voie de cette redondance, le FMS est composé d'une fonctions KU (Keyboard and control Unit) comportant le clavier et d'une fonction MFD (Muti Functional Display) comportant l'écran. Le pilot peut entrer une requête d'affichage d'un point de navigation. Cette requête est reçue par la fonction KU. La requête est alors transmise aux gestionnaires de vol FM (Flight Manager) qui questionnent en parallèle la base de données de navigation NDB (Navigation DataBase). Celle-ci retourne aux FM les informations statiques du point de navigation qui sont ensuite périodiquement enrichies par chaque FM avec les informations dynamiques calculées en fonction des données de vol (vitesse, position, ...) produites par les centrales inertielles ADIRU (Air Data Inertial Reference Unit). Au final, chaque FM envoie périodiquement à chaque MFD les informations à afficher. Chaque ADIRU élabore les données de vol à partir de données de base fournies par des mesures envoyées par des capteurs.

Pour illustrer les exigences que doit satisfaire le système et évaluer nos propositions, nous en présentons deux: l'exigence de latence relative à une chaîne fonctionnelle, et l'exigence de fraîcheur relative à une chaîne fonctionnelle. L'exigence de latence permet de garantir que le système répond suffisamment rapidement à une sollicitation. L'exigence de fraîcheur permet de garantir qu'une donnée affichée du système dépend d'informations suffisamment récentes pour être pertinente.

Nous modélisons l'architecture du cas d'étude en utilisant le diagramme de structure composite de UML-MARTE, puis son comportement en utilisant les diagrammes d'activité et de machine d'état de UML-MARTE. Le modèle obtenu est ensuite traduit automatiquement en réseau de Pétri temporisé selon l'approche dédiée aux propriétés temps réels en utilisant la sémantique d'exécution définie dans le chapitre 3. Les exigences temps réels sont spécifiées en utilisant les patrons de propriété définis dans le chapitre 4. Après avoir réduit l'espace d'état par la méthode de réduction spécifique aux propriétés temps réels définie dans le chapitre 6, le graphe d'état préservant la sémantiques du marquage est généré pour évaluer les propriétés temps réel en utilisant les observateurs dans le réseau de Pétri temporisé défini par le chapitre 5. Les résultats obtenus sont identiques à ceux des travaux de M. Lauer à base de programmation linéaire en variable entière. Nous réalisons ensuite la même série d'expériences que M. Lauer afin d'évaluer si ce coût est acceptable sur des système de taille industrielle ce qui est le cas pour ce type de système d'avionique avec une structure relativement régulière qui se prête bien à la réduction.

In the current chapter, we use an avionic case study, a part of flight management system (FMS), to test the whole proposal from Part One (Property-driven approaches). In this case study, the latency and freshness requirements are assessed. We model the property-specific architecture of the case study using UML-MARTE composite structure, and model the property-specific behavior using the activity and the state machine diagrams. The UML-MARTE model is then mapped to a property-specific TPN model using the mapping semantics defined in Chapter 3. The real-time requirements are specified using the property patterns defined in Chapter 4. After performing the property-specific reduction for TPN presented in Chapter 6, the state-class graph preserving marking semantics is generated to assess the real-time properties using the observer-based model checking (Chapter 5). The experiment results show that our approach is able to analyze large scale systems more complex than the current real systems implemented in the Airbus A380 FMS.

## 8.1 Introduction

In the previous chapters, we have presented the contributions, including the property-driven approaches (the mapping from UML-MARTE to property specific TPN, the real-time property specification patterns, the real-time property verification based on observers in model checking, and the property specific reduction for TPN), and the fault localization in model checking. We have implemented and integrated the above approaches in the UMLMMC (UML-MARTE Model Checking) toolset.

In order to test the whole property-specific proposal, we use an avionic case study investigated by M. Lauer et al. [Lau12, LEBP11b, LEBP11a, LEPB10], which is a part of a flight management system (FMS). We rely on the system descriptions provided by Lauer et al.. The latency and freshness requirements are assessed in the case study. We model the architecture using UML-MARTE composite structure diagram, and model the behavior using activity and state machine diagrams. The UML-MARTE model is then mapped to property specific TPN model using the mapping semantics defined in Chapter 3. The real-time requirements are specified using the property patterns defined in Chapter 4. After performing the property-specific reduction for TPN presented in Chapter 6, the marking abstraction state-class graph is generated to assess the real-time properties using the observer-based model checking (Chapter 5). The experiment results show that our approach is able to analyze large scale systems more complex than the current real systems implemented in the Airbus A380 FMS.

In this chapter, Section 8.2 describes the FMS case study; Section 8.3 models the case study using UML-MARTE diagrams; Section 8.4 maps the UML-MARTE model to TPN model; Section 8.5 presents the property

verification results; Section 8.6 gives the results of scalability tests; Section 8.7 compares with the works of Lauer; Section 8.8 gives the conclusions.

## 8.2 Case Study: Flight Management System

A flight management system (FMS) is a fundamental component of a modern airliner's avionics. It consists of two units, a computer unit and a control display unit. The computer unit is integrated as a function on the hardware platform Integrated Modular Avionics (IMA). The control display unit provides human/machine interface for data entry and information display. An FMS manages part of the displays in the cockpit. It is a primary function of the in-flight management of the flight plan. It guides the aircraft along the flight plan using various sensors to determine the aircraft's position. A flight plan is a sequence of waypoints. From the cockpit, the FMS provides some information on a waypoint requested by the pilot and periodically refreshes dynamic data related to this waypoint (distance and estimated time of arrival).

### 8.2.1 Integrated Modular Avionics

The Integrated Modular Avionics (IMA) architecture is defined in the avionics context for sharing communication and computation resources while ensuring temporal and spatial segregation. An IMA system is a platform on which a set of functions is statically mapped. Avionics functions executing on the platform must fulfill safety requirements, one of them being a strong segregation. For that purpose, airframers and the ARINC corporation have proposed two standards. The standard ARINC 653 [C+97] specifies the management of computing resources (named modules): the scheduling of functions on each module is defined off-line by a periodic sequence of slots (named partitions) statically organized in a time-frame named the MAjor time Frame (MAF). Thus, each function periodically executes at fixed times. Modules however are globally asynchronous. The standard ARINC 664 [Spe05], implemented in AFDX (Avionics Full-Duplex Switched Ethernet) [Eng05] networks, describes the management of communication resources (switches and end-systems). Communication flows are statically segregated into Virtual Links (VL). Each VL is dedicated to a single function and implements a traffic shaper. It is characterized by a Bandwidth Allocation Gap (BAG), i.e., the minimal time interval separating two successive messages on the VL. These two standards globally define the IMA concept which has been implemented in the Airbus 380 and the Boeing 787 for instance. According to this definition, an IMA platform can be seen as a set of modules, switches and

links compliant to these standards.

The architecture of the FMS case study is represented in Fig. 8.2.1. Seven modules, from $Module_1$ to $Module_7$ (named $M_1$, ..., $M_7$ afterwards), are used to map the avionic functions. The functions are described in the follow part (Section 8.2.3).

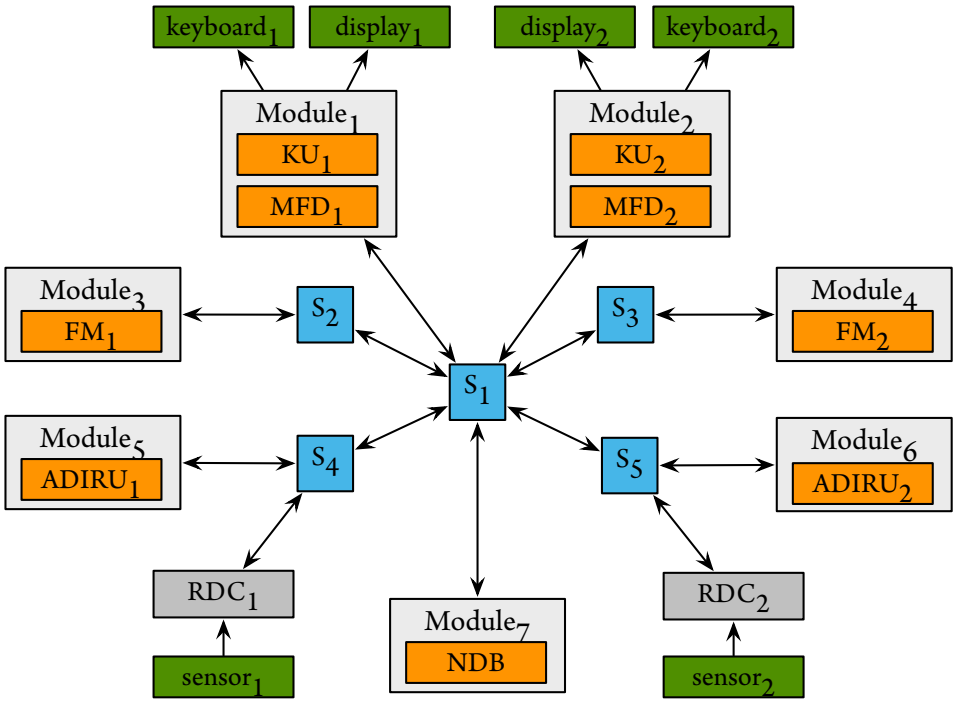



**Figure 8.2.1:** Architecture of the Case Study

Each function is allocated to a partition of a module. Each partition is described by the following real-time features: period of repetition, duration of the slot and offset in the MAF. We assume that each function has a worst case execution time less or equal to the duration of its partition. Allocation of both sides of the plane are symmetrical. The real-time parameters of modules and *RDC* are summarized in Table 8.2.1. The sensors capture the sampling data every 20 ms.

**Table 8.2.1:** Parameters of Partitions

| Partition | Period | Duration | Offset | Module |
|---|---|---|---|---|
| $KU_1$ | 50 | 25 | 0 | 1 |
| $MFD_1$ | 50 | 25 | 25 | 1 |
| $KU_2$ | 50 | 25 | 0 | 2 |
| $MFD_2$ | 50 | 25 | 25 | 2 |
| $FM_1$ | 60 | 30 | 0 | 3 |
| $FM_2$ | 60 | 30 | 0 | 4 |
| $ADIRU_1$ | 60 | 30 | 0 | 5 |
| $ADIRU_2$ | 60 | 30 | 0 | 6 |
| $NDB$ | 100 | 20 | 0 | 7 |
| $RDC_1$ | 50 | 10 | 0 | - |
| $RDC_2$ | 50 | 10 | 0 | - |

### 8.2.2 Avionics Full-Duplex Switched Ethernet

The Avionics Full-Duplex Switched Ethernet (AFDX) network is constituted by five switches $S_1$, ... $S_5$. The variables exchanged between functions through Virtual Links (VL). A VL defines a logical unidirectional connection from one source end-system to one or more destination end-systems. Each VL has a dedicated maximum bandwidth. This bandwidth is allocated by the System Integrator. For each VL, the End System should maintain the ordering of data as delivered by a partition, for both transmission and reception (ordinal integrity).

At the output of the End System, the flow of frames associated with a particular VL is characterized by two parameters: Bandwidth Allocation Gap (BAG) and Jitter. If the frames experienced no jitter from the scheduler, the BAG represents the minimum time interval between the first bits of two consecutive frames from the same VL. Table 8.2.2 lists the parameters of VL used in the case study.

The parameters of keyboards and sensors are given in Table 8.2.3, including the period time, the traverse time from sensors to equipments.

**Table 8.2.2:** Parameters of Virtual Links

| Virtual Link | Source | Destination(s) | Variable(s) | BAG (ms) | Route(s) |
|---|---|---|---|---|---|
| $VL_1$ | $KU_1$ | $FM_1, FM_2$ | $wpid_1$ | 32 | $\{S_1, S_2\}, \{S_1, S_3\}$ |
| $VL_2$ | $KU_2$ | $FM_1, FM_2$ | $wpid_2$ | 32 | $\{S_1, S_2\}, \{S_1, S_3\}$ |
| $VL_3$ | $FM_1$ | $MFD_1$ | $wpInfo_1, ETA_1$ | 8 | $\{S_2, S_1\}$ |
| $VL_4$ | $FM_1$ | $NDB$ | $query_1$ | 16 | $\{S_2, S_1\}$ |
| $VL_5$ | $FM_2$ | $MFD_2$ | $wpInfo_2, ETA_2$ | 8 | $\{S_3, S_1\}$ |
| $VL_6$ | $FM_2$ | $NDB$ | $query_2$ | 16 | $\{S_3, S_1\}$ |
| $VL_7$ | $NDB$ | $FM_1$ | $answer_1$ | 64 | $\{S_1, S_2\}$ |
| $VL_8$ | $NDB$ | $FM_2$ | $answer_2$ | 64 | $\{S_1, S_3\}$ |
| $VL_9$ | $RDC_1$ | $ADIRU_1$ | $pres_1$ | 32 | $\{S_4\}$ |
| $VL_{10}$ | $RDC_2$ | $ADIRU_2$ | $pres_2$ | 32 | $\{S_5\}$ |
| $VL_{11}$ | $ADIRU_1$ | $FM_1, FM_2$ | $speed_1$ | 32 | $\{S_4, S_1, S_2\}, \{S_4, S_1, S_3\}$ |
| $VL_{12}$ | $ADIRU_2$ | $FM_2, FM_1$ | $speed_2$ | 32 | $\{S_5, S_1, S_3\}, \{S_5, S_1, S_2\}$ |

**Table 8.2.3:** Parameters of Captors

| Sensor | Nature | Period (ms) | Traverse Time (ms) | Equipment |
|---|---|---|---|---|
| $key_1$ | sporadic | 60 | $[0.1, 02]$ | $M_1$ |
| $key_2$ | sporadic | 60 | $[0.1, 02]$ | $M_2$ |
| $sensor_1$ | sporadic | 20 | $[0.1, 02]$ | $RDC_1$ |
| $sensor_1$ | sporadic | 20 | $[0.1, 02]$ | $RDC_2$ |

### 8.2.3 Functions

In the cockpit, the pilot and the co-pilot use a personal keyboard and two displays to interact with the FMS. Information displayed on both screens must be similar although they are not processed by the same components. The FMS uses a redundant implementation of its functions which are segregated on each side of the plane (named side 1 and side 2).

There exist two main functional chains in the case study. The first one responds to pilot's sporadic request in Fig. 8.2.2. At any time, the pilot can request some information on a given waypoint. The $KU_1$ (*Keyboard and Cursor Control Unit*) controls the physical device used by the pilot to enter his requests.

When $KU_1$ receives a request ($req_1$), it broadcasts $wpid_1$ and $wpid_2$ to the *Flight Managers* $FM_1$ and $FM_2$ respectively. The *FMs* manage the flight plan, i.e., the trajectory between successive waypoints. When a request occurs, both query the *NDB* (*Navigation Database*) by sending $query_1$ (resp. $query_2$) to retrieve the static information on the waypoint such as the latitude and the longitude. The *NDB* separately answers each *FM* by sending a message $answer_1$ (resp. $answer_2$) containing the expected data. Upon reception of this message, each *FM* computes two complementary dynamic data: the distance to the waypoint, and the *ETA* (*Estimated Time of Arrival*). These data ($wpInfo_1$ and $wpInfo_2$ resp.) are periodically sent to respective *MFDs* (*Multi Functional Display*) which periodically elaborate the pages to be displayed on the screens.

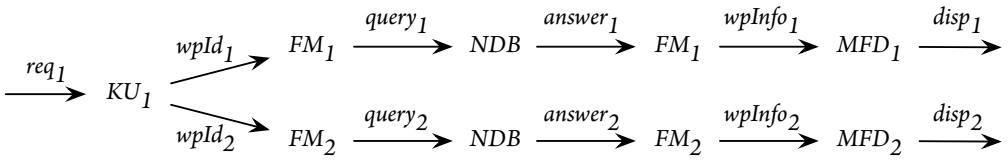



**Figure 8.2.2:** Functional Chain: Sporadic Response to Request

The second functional chain is used to periodically compute flight data ($disp_1$ and $disp_2$ resp.) refreshed on displays in Fig. 8.2.3. To compute these data, the *FM*s use the position and the speed of the aircraft ($speed_1$ and $speed_2$ resp.) which are periodically delivered by the *ADIRUs* (*Air Data Inertial Reference Unit*). The *ADIRUs* determine the speed and position of the aircraft thanks to data ($pres_1$ and $pres_2$ resp.) provided by sensors. Here, we only consider the data provided by one sensor per *ADIRU*. The sensors communicate through field networks. Interconnection with the AFDX network is managed by *RDCs* (*Remote Data Concentrator*).

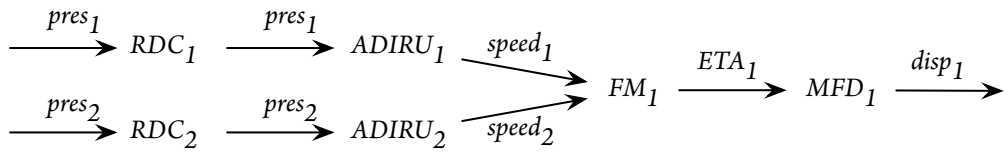



**Figure 8.2.3:** Functional Chain: Production of Periodic Data

From the above two functional chains in Fig. 8.2.2 and Fig. 8.2.3 we can see that a function can have

two behaviors with respect to its data: periodic or sporadic. Sensors, *RDCs*, *ADIRUs* and *MFDs* write there output data periodically using the last input data received (thus a data can be used several times before it is refreshed). For these functions, input data are stored in sampling ports in which data are continuously overwritten. On the contrary, the *KUs* and the *NDB* write there data sporadically in response to inputs. These functions use queuing ports to store their input data before reading them. The *FMs* use the two kinds of behaviors: periodic and sporadic.

### 8.2.4 Real-Time Requirements

Because of their critical nature, IMA systems must also satisfy strong real-time requirements. In this case study, we focus on 2 real-time requirements: latency and freshness.

#### 8.2.4.1 Latency Real-Time Requirement

The latency allows to guarantee that the system responds quick enough to a request. It corresponds to the time elapsed between an event at the beginning of a functional chain and the first event depending on it at the end of the chain, i.e. a sporadic input must result in an output before a given amount of time.

**Example 8.1 (Latency Requirement Example)** *An example of latency is given in Fig. 8.2.4. On the functional chain:* $\xrightarrow{req_1}$ $KU_1$ $\xrightarrow{wpId_1}$ $FM_1$ $\xrightarrow{query_1}$ $NDB$ $\xrightarrow{answer_1}$ $FM_1$ $\xrightarrow{wpInfo_1}$ $MFD_1$ $\xrightarrow{disp_1}$*, the maximum time between* $req_1$ *and the first occurrence of* $disp_1$ *depending on* $req_1$ *must be inferior to 700 ms.*

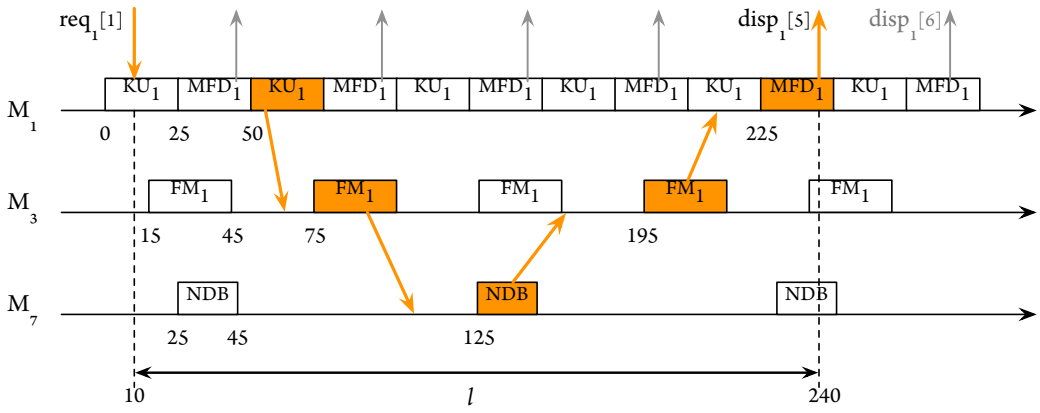



**Figure 8.2.4:** Latency Real-Time Requirement

#### 8.2.4.2 Freshness Real-Time Requirement

The freshness allows to ensure that a system variable depending on another variable is fresh enough. There exist two interpretations of the freshness requirement. One targets the time between an event at the end of a functional chain and the earliest dependent event at the beginning of the chain. The other corresponds to the time interval between an event at the end of a functional chain and the earliest previous event of the dependent event at the beginning of the chain. The work of Lauer et al. followed the former. In this thesis, we follow the later. Both are reasonable in the context of the case study. We explain the reason in the following example.

**Example 8.2 (Freshness Requirement Example)** *An example of freshness is given in Fig. 8.2.5. On the functional chain:* $\xrightarrow{pres_1} RDC_1 \xrightarrow{pres_1} ADIRU_1 \xrightarrow{speed_1} FM_1 \xrightarrow{ETA_1} MFD_1 \xrightarrow{disp_1}$*, the worst case of displaying ETA on the screen by MFD must not be superior to 400 ms. The former interpretation of freshness corresponds to interval* $f_1$*, while the later one corresponds to* $f_2$*. In the context of this case study, the output disp is used as the data to display on the screen of the pilots. From the viewpoint of the pilots, since the arrival of* $disp_1[1]$*, the displayed data on the screen has been updated by* $disp_1[1]$*. The* $disp_1[1]$*,* $disp_1[2]$ *and* $disp_1[3]$ *all depend on the input* $pres_1[1]$*. When* $disp_1[1]$ *arrives, the displayed data is updated again, as* $disp_1[4]$ *depends on* $pres_[4]$*. Therefore, we can use* $f_1$ *to measure the freshness between the output disp and its dependent input pres, or use* $f_2$ *to measure the freshness between the output disp and the previous one of its dependent input pres.*

## 8.3 Modeling and Semantics

We specify the models of the case study using UML-MARTE. The IMA architecture and AFDX network are specified using composite structure diagrams; the behavior of each module is specified using activity diagrams for the functional chain of latency real-time property; the behavior of each module is specified using state machine diagrams for the functional chain of freshness real-time property. Before modeling the system, an abstraction of the AFDX network is applied.

### 8.3.1 Abstraction of AFDX Network

We have discussed the importance of abstraction in the former parts. To analyze a large scale IMA system, the combinatorial complexity of a real-time property takes its root in the asynchronism of the modules, the

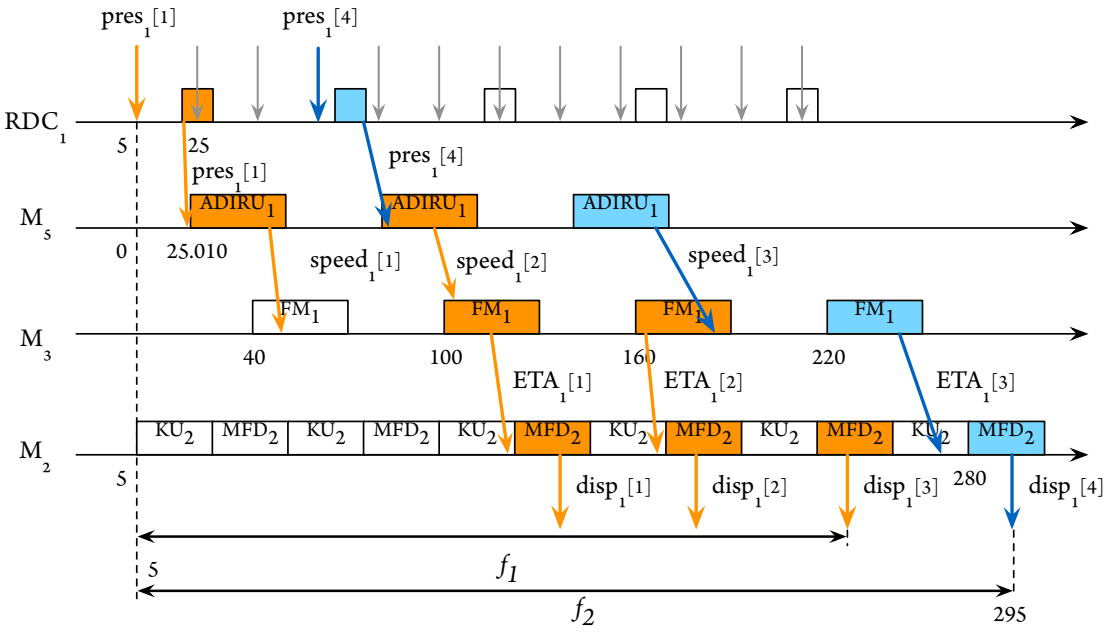



**Figure 8.2.5:** Freshness Real-Time Requirement

variability of the execution times and the indeterministic congestion in the network [LEBP11b]. Lauer et al. showed in [LEPB10] that taking into account all these factors in the evaluation of high level properties is intractable. They showed in [LEBP11b] that the complexity can be significantly reduced by characterizing the lower (resp. upper) bound of the network traversal time along the VL path as a time interval [a, b]. These bounds are determined by the trajectory approach [MM06] which has been successfully applied to AFDX networks in [BSF09].

In the FMS case study, Lauer et al. considered that each $VL_i$ was abstracted by a timed channel $c_i$ [0.12, 2] (in ms): each frame released by a $VL_i$ traffic shaper undergoes a delay between 0.12 *ms* and 2 *ms* to reach its destination. Note that this abstraction is an over-approximation because the bounds of the timed channels are determined with an over- approximative technique.

By applying this over-approximation abstraction method, the AFDX network of the case study in 8.2.1 is replaced by the channels in Fig. 8.3.1. The VLs with switches and ports are replaced by 16 temporal channels, $C_1$, ..., $C_{12}$. The temporal parameters of each channel are described in Table 8.3.1.

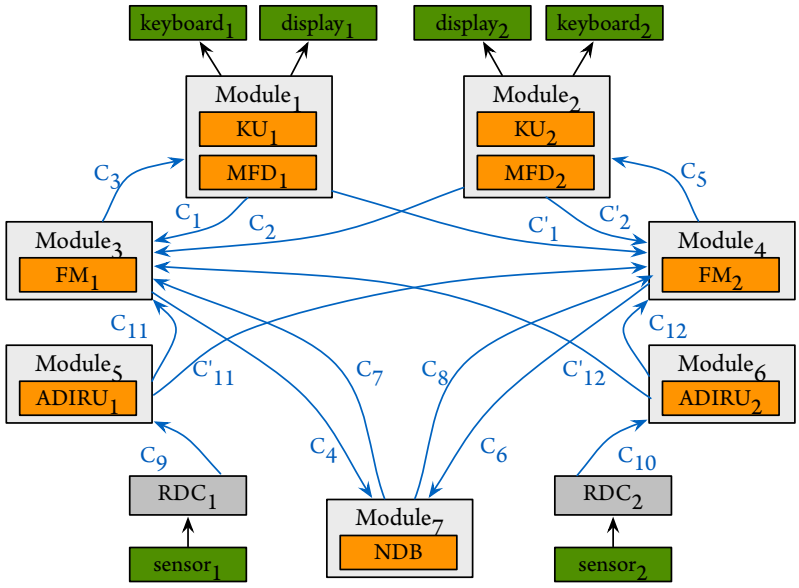



**Figure 8.3.1:** Abstract Network of Case Study

**Table 8.3.1:** Characters of Timed Channel

| Channel | Source | Destination | Variable(s) | BCTT ($\mu$s) | WCTT ($\mu$s) |
|---|---|---|---|---|---|
| $C_1$ | $KU_1$ | $FM_1$ | $wpid_1$ | 298 | 444 |
| $C'_1$ | $KU_1$ | $FM_2$ | $wpid_1$ | 298 | 444 |
| $C_2$ | $KU_2$ | $FM_1$ | $wpid_2$ | 298 | 444 |
| $C'_2$ | $KU_2$ | $FM_2$ | $wpid_2$ | 298 | 444 |
| $C_3$ | $FM_1$ | $MFD_1$ | $wpInfo_1$, $ETA_1$ | 310 | 490 |
| $C_4$ | $FM_1$ | $NDB$ | $query_1$ | 310 | 450 |
| $C_5$ | $FM_2$ | $MFD_2$ | $wpInfo_2$, $ETA_2$ | 310 | 490 |
| $C_6$ | $FM_2$ | $NDB$ | $query_2$ | 310 | 450 |
| $C_7$ | $NDB$ | $FM_1$ | $answer_1$ | 400 | 508 |
| $C_8$ | $NDB$ | $FM_2$ | $answer_2$ | 400 | 508 |
| $C_9$ | $RDC_1$ | $ADIRU_1$ | $pres_1$ | 150 | 156 |
| $C_{10}$ | $RDC_2$ | $ADIRU_2$ | $pres_2$ | 150 | 156 |
| $C_{11}$ | $ADIRU_1$ | $FM_1$ | $speed_1$ | 452 | 584 |
| $C'_{11}$ | $ADIRU_1$ | $FM_2$ | $speed_1$ | 452 | 584 |
| $C_{12}$ | $ADIRU_2$ | $FM_2$ | $speed_2$ | 452 | 584 |
| $C'_{12}$ | $ADIRU_2$ | $FM_1$ | $speed_2$ | 452 | 584 |

The traffic shaper is used to express the regulation of the packets on the VL to ensure the BAG. This regulation specifies the relation on the signals of the copy of the data emitted by the VL. According to the abstraction method proposed in [Lau12], the delay time caused by the traffic shaper is specified as $T_d = c \cdot \text{BAG}$, where BAG is defined in Table 8.2.3 and $c = 0 \ldots N_{\text{VL}} - 1$. $N_{\text{VL}}$ is the number of VL passing through a module on a functional chain which is specified by the system architecture.

## 8.3.2 Architecture Model

Using the above abstract method, we specify respectively the architecture of the system related to the verification of latency and freshness property using the UML-MARTE composite structure diagram.

The architecture related to latency real-time property is shown in Fig. 8.3.2.

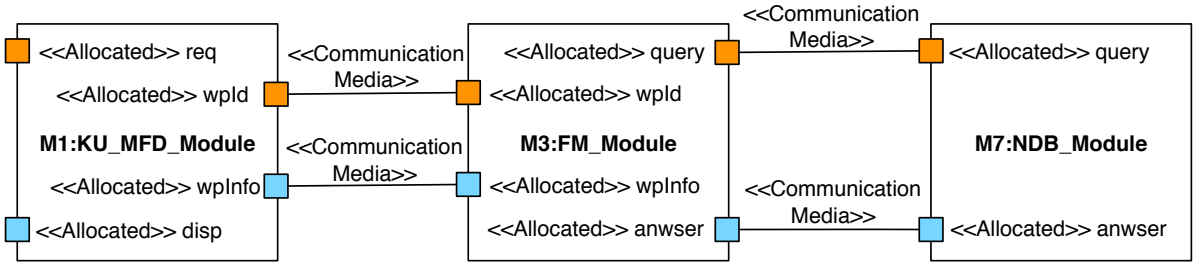



**Figure 8.3.2:** UML-MARTE Architecture for Latency Real-Time Property

The architecture related to freshness real-time property is shown in Fig. 8.3.3.

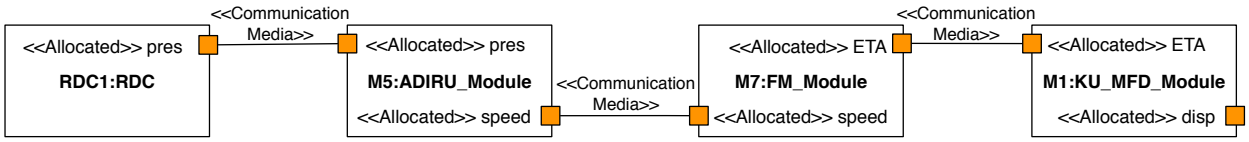



**Figure 8.3.3:** UML-MARTE Architecture for Freshness Real-Time Property

### 8.3.3 Behavior Model

As a rich modeling language, UML provides many diagrams and modeling entities for the specification of systems. From the viewpoint of modeling capacity, either activity or state machine diagrams can represent the system's abstract behavior. In the context of this thesis, in order to cover all the semantics mapping methods presented in Chapter 3, both activity and state machine diagrams are used to specify system's behavior respectively: system is modeled using activity diagram for verifying latency property, and using state machine diagram for verifying freshness property.

#### 8.3.3.1 Modeling System Behavior for Latency Functional Chain

Each function on avionic modules can be modeled as an asynchronous action in activity diagrams. The semantics of asynchronous action implies that only one type of computation occurs from its input to its output. For a given avionic function, however, it can handle more than one computation during its period. For example, the functional chain of sporadic response to request(Page 253) indicates that the function $FM_1$ has two types of computation:

1. Take the waypoint as input ($wpId_1$) and output the query for *NDB* ($query_1$)
2. Take the answer of *NDB* as input ($answer_1$) and output the waypoint's information ($wpInfo_1$)

Fig. 8.3.4 models the behavior of $FM_1$ module using activity diagram. To model $FM_1$, two separated asynchronous actions are used. The first one ($FM_1$) is connected to the Pins representing $wpId_1$ and $query_1$, and the second one ($FM1a$) to $answer_1$ and $wpInfo_1$. Since they are derived from the same avionic function, both their period and offset must be the same. According to avionic system's feature, not all modules will be powered at exactly the same time. This means not only that the offset for each avionic function will be probably different, but also that the value of offset is rather an interval than an exact value. In general, a `MARTE::DesignModel::HLAM::RtSpecification` will be created and shared by several asynchronous actions if they are derived from the same avionic function. This stereotype contains all information required for the real-time feature of given asynchronous actions.

#### 8.3.3.2 Modeling System Behavior for Freshness Functional Chain

The dependency of values between input and output through a functional chain could also be considered as a state-transition problem. For each avionic function $F_i$ in a given chain, at any given time $t$, the valid

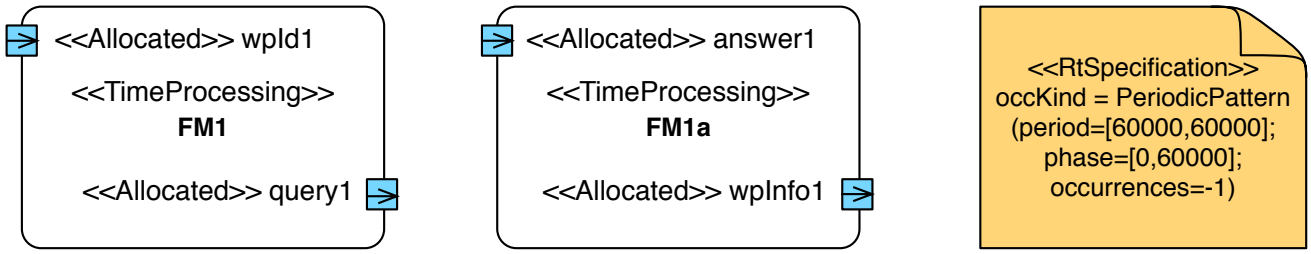



**Figure 8.3.4:** UML-MARTE Behavior for Latency Real-Time Property

input value $Input_i^t$ that $F_i$ depends on is unique. This dependency will not change until the first function $F_1$ on the functional chain generates a new output value, which is propagated to $F_i$, and $F_i$ generates the corresponding new output value when it is reactivated in a period. If we consider depending on a given input value as a state for an avionic function, then this state will change only when it receives an event standing for its output value based on the time that the next input is generated. The pre-requisites of the occurrence of this event are: the next input of $F_i$ has arrived, the next period of $F_i$ starts, and the computation of the new output is finished. These three pre-requisites are modeled as three states in a state machine diagram. For simplicity, we consider that the end of the computation happens simultaneously with the generation of the output by merging them into one event.

The modeling activity is split in two parts: modeling of avionic function and modeling of communication between functions.

**Avionic function** Fig. 8.3.5 models the behavior of the $FM_1$ module using state machine diagrams. In the context of this thesis, each $F_i$ where $i > 1$ is modeled respectively by a state machine of 7 states. Note: $kn1$ (resp. $kp1$) stands for $k-1$ (resp. $k+1$), $ni$ (resp. $np$) stands for next input (resp. next period), and $co$ stands for computing output.

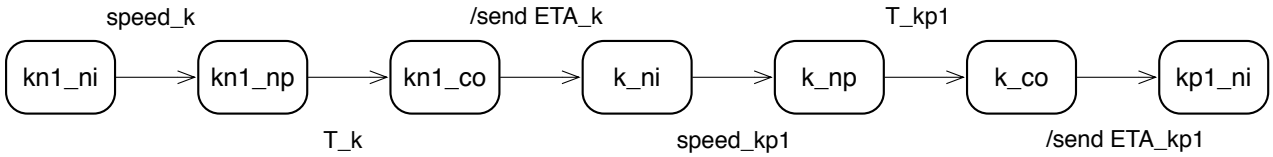



**Figure 8.3.5:** UML-MARTE Behavior for Freshness Real-Time Property

- State *kn1_ni*: $F_i$ depends on $\text{Input}_{k-1}$, waiting for next input ($Input_k$)
- State *kn1_np*: $F_i$ depends on $\text{Input}_{k-1}$, waiting for next period
- State *kn1_co*: $F_i$ depends on $\text{Input}_{k-1}$, computing $Output_k$
- State *k_ni*: $F_i$ depends on $\text{Input}_k$, waiting for next input ($Input_{k+1}$)
- State *k_np*: $F_i$ depends on $\text{Input}_k$, waiting for next period
- State *k_co*: $F_i$ depends on $\text{Input}_k$, computing $Output_{k+1}$
- State *kp1_ni*: $F_i$ depends on $\text{Input}_{k+1}$, waiting for next input ($Input_{k+2}$)

For $F_1$, since it is the root cause that changes the value dependency for the whole chain, it does not have the first two states. This abstraction of $k$ ($k \in \mathbb{N}$) generalizes the problem so the model can handle any sequence of input/output.

Accordingly, these states are sequentially connected by 6 transitions:

- The transition from state *kn1_ni* to state *kn1_np* (same for state *k_ni* to state *k_np*):
  - Triggered by event representing $Input_k$
  - The effect of the transition takes no time (or no effect)
  - No guard is defined
- The transition from state *kn1_np* to state *kn1_co* (same for state *k_np* to state *k_co*):
  - No trigger is defined
  - The effect of the transition takes a duration of $[0, T]$, where $T$ is the period of the avionic function
  - No guard is defined
- The transition from state *kn1_co* to state *k_ni* (same for state *k_co* to state *kp1_ni*):
  - No trigger is defined

– The effect of the transition takes a duration of $[B, W]$, where $B/W$ is the BCET/WCET of the avionic function. The effect will equally generate an event $Output_k$.

– No guard is defined

**Communication**

In this applicative scenario, the temporal aspect of transmission via network is not negligible. This can be modeled, for each pair of $Output_k$ and $Input_k$, by the connector between module ports using MARTE stereotype `MARTE::MARTE_Foundations::GRM::CommunicationMedia::Package T`.

### 8.3.4 Real-Time Requirement Specification

Both latency and freshness real-time requirements can be seen as maximum and minimum end-to-end time between two events, which corresponds to the specification: **Always** $\mathbf{T}(E_A, E_B) < t$ and **Always** $\mathbf{T}(E_A, E_B) > t$ in global execution. In order to test the scalability of the verification strategy, we compute the exact value of latency and freshness real-time property. Instead of verifying end-to-end maximum (minimum) time when given an over-estimated value, we compute the bounding value of both requirements.

#### 8.3.4.1 Latency Real-Time Property

Compute WCT (BCT), where always $\mathbf{T}(req, disp) <$ WCT$(>$ BCT$)$ in the global execution of the latency functional chain.

#### 8.3.4.2 Freshness Real-Time Property

Compute WCT (BCT), where always $\mathbf{T}(req, disp) <$ WCT$(>$ BCT$)$ in the global execution of the freshness functional chain.

## 8.4 Mapping UML-MARTE to TPN Model

### 8.4.1 Mapping of the Latency Functional Chain

Fig. 8.4.1 is the mapping of the latency function chain. The latency real-time requirement refers to $Module_1$ (functions $KU_1$ and $MFD_1$), $Module_3$ (function $FM_1$) and $Module_7$ (function $NDB$). $Module_1$ starts from

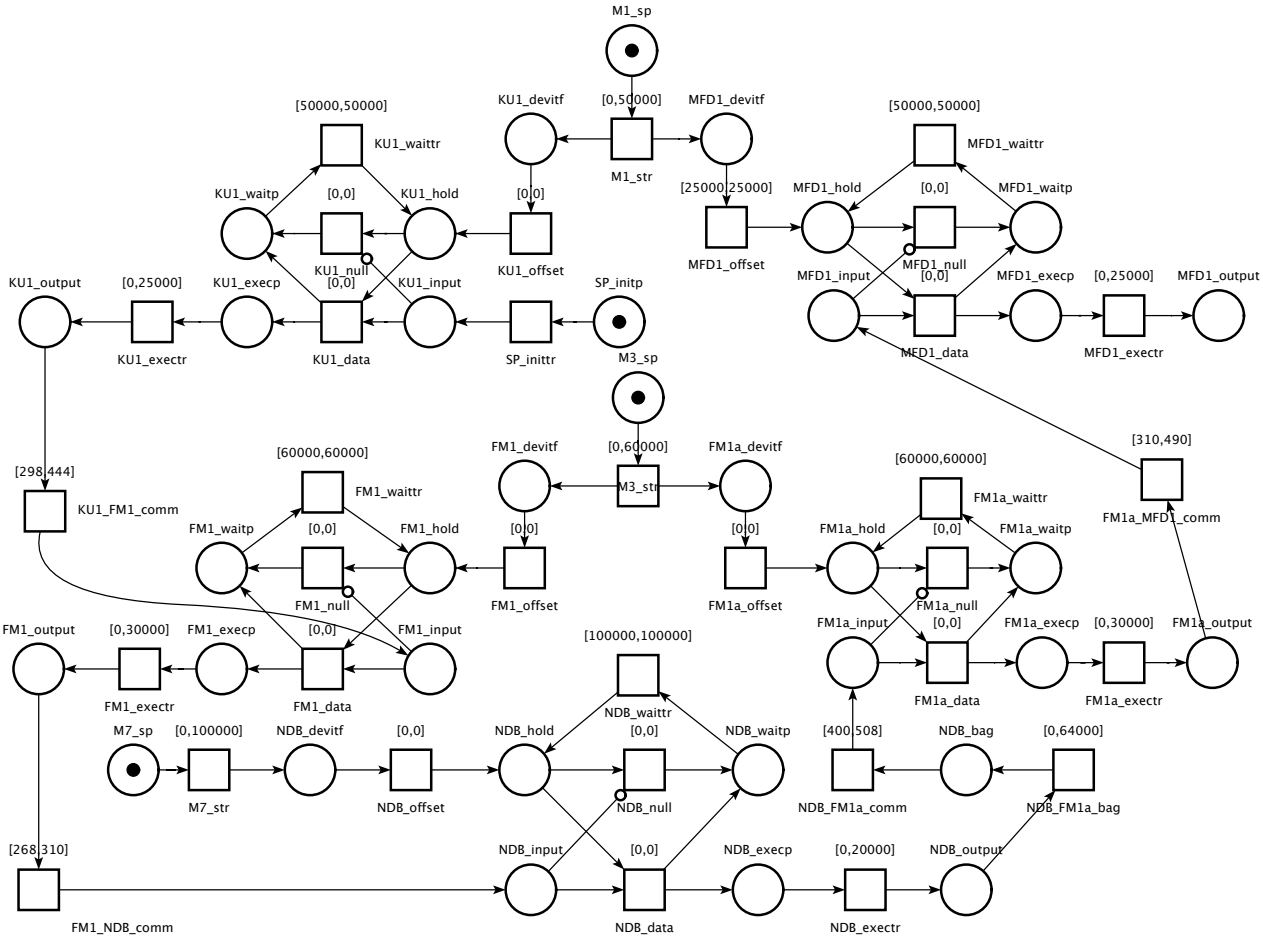



**Figure 8.4.1:** Mapping Result of System Related to Latency Property

place *M1_sp*. Through transition *M1_str*, functions $KU_1$ and $MFD_1$ run periodically, and the offset of $MFD_1$ is $25ms$ later than that of $KU_1$. $Module_3$ starts from place *M3_sp*. As the function $FM_1$ is used twice in the functional chain, two instances of $FM_1$ with the same offset are generated. $Module_7$ starts from place *M7_sp*. The variable produced by *NDB* will be sent to $FM_1$. It is possible for the variable to wait for the duration of a BAG before sending through the channel. The waiting time of BAG is represented by the transition $NDB\_FM_1\_bag$.

## 8.4.2 Mapping of the Freshness Functional Chain

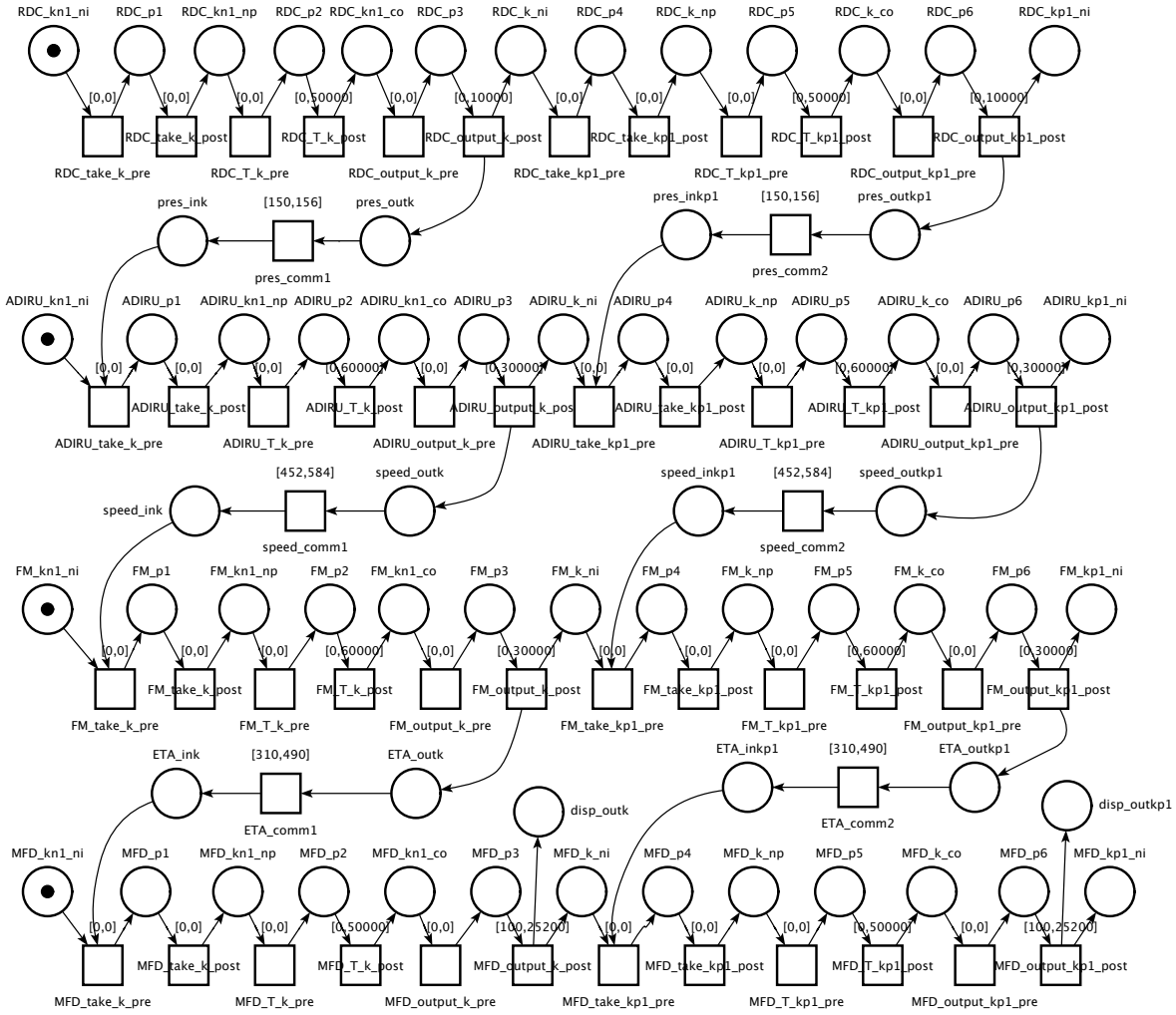



**Figure 8.4.2:** Mapping Result of System Related to Freshness Property

Fig. 8.4.2 is the mapping of freshness functional chain. Functions *ADIRU, FM, MFD* and *RDC* are referred in freshness functional chain. Two consecutive output of *RDC* (*pres*) are transmitted to *ADIRU* function, where variable *pres* is received and variable *speed* is transmitted to *FM* function. After computing the output variables, *ETA* is transmitted to *MFD* function, where the *disp* variables are computed.

## 8.5 Verification of Real-Time Property

According to the observer-based model checking verification approach presented in Chapter 5, two observers are used to compute WCT and BCT of latency and freshness real-time requirements. The observers are added on corresponding transitions in the original system to compute optimized value of properties (Fig. 8.5.1).

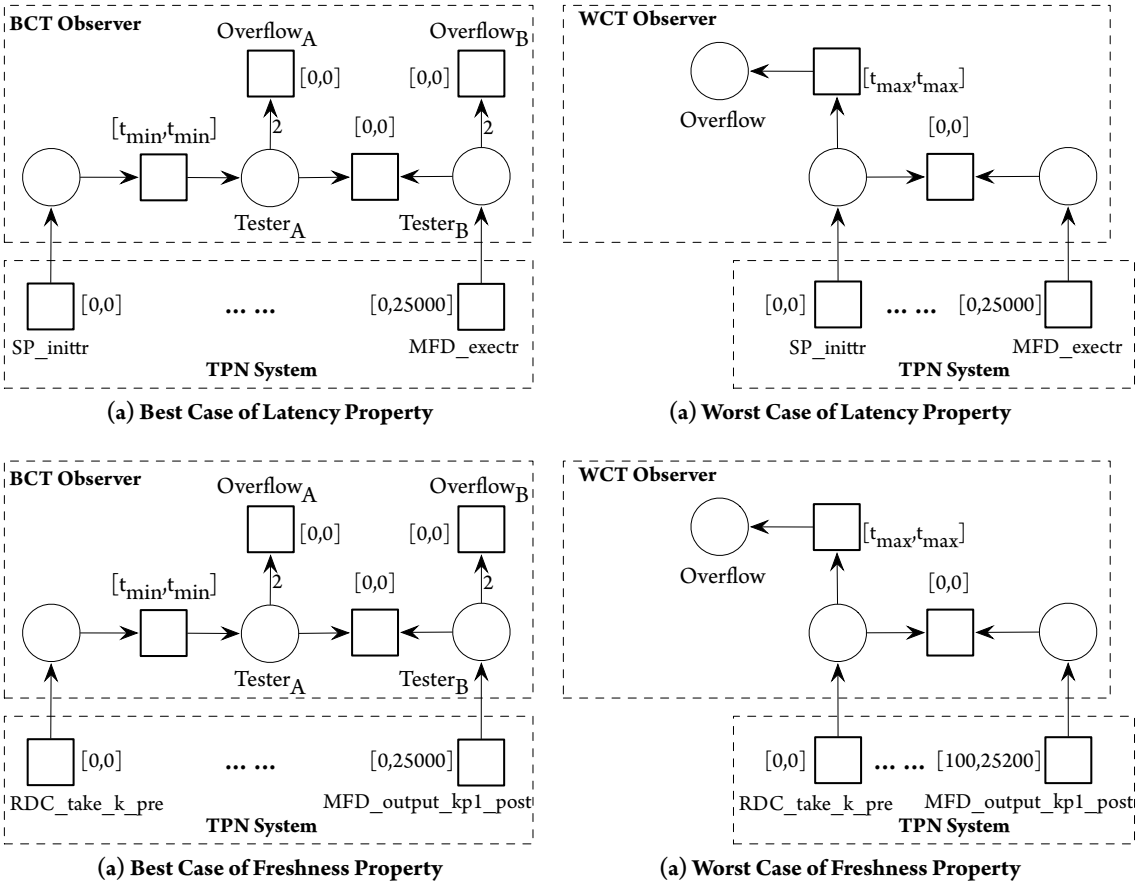



**Figure 8.5.1:** TPN Observer for Latency and Freshness Property

The computation results are shown in Table 8.5.1. For latency real-time property, the WCT (resp. BCT) is 450.4 (reps. 75.2) *ms*. For freshness real-time property, the WCT (resp. BCT) is 316429 (resp. 1012) *ms*. The original system for latency property without TPN observer has 9378 states and 23250 transitions.

TPN observers impact the size of the state space of the original system. The impact depends on $t_{min}$ and $t_{max}$ on the tester transition. By applying the reduction techniques, the number of states and transitions is significantly reduced. Take the WCT latency property for example, compared to the execution time before reduction (278.313 s), the execution time is reduced to 2.484 s.

**Table 8.5.1:** Real-Time Property Verification Results

| Property | | Property Value (ms) | State/Transition Number | | Execution Time (s) | |
|---|---|---|---|---|---|---|
| | | | Before Reduc. | After Reduc. | Before Reduc. | After Reduc. |
| **Latency** | System | N/A | 9378/23250 | N/A | N/A | N/A |
| | WCT | 450.4 | 67105/145024 | 9/10 | 278.313 | 2.484 |
| | BCT | 75.2 | 11162/28922 | 8/9 | 43.781 | 3.719 |
| **Freshness** | System | N/A | 53/85 | N/A | N/A | N/A |
| | WCT | 316429 | 259/446 | 34/44 | 7.578 | 3.688 |
| | BCT | 1012 | 125/202 | 54/79 | 7.360 | 2.125 |

## 8.6 Scalability Tests

The proposed methods target large scale systems. We implement a series of experiments to assess whether the cost of verification is acceptable for large scale systems by extending the case study. We use the same parameters as the work of [Lau12]: $N$ and $P$ (Fig. 8.6.1). The depth of the case study is extended by increasing $P$. The width of the case study is extended by increasing $N$. We test the scalability of the proposed methods by increasing the size of each functional chain using parameters $P$ and $N$.

### 8.6.1 Experiments on the Latency Functional Chain

The latency functional chain is enlarged by increasing the number of *NDB*. Each latency functional chain traverses $P$ *NDB*, i.e. $2P + 3$ functions.

$$\begin{aligned} L_1 = &\xrightarrow{req_1} KU_1 \xrightarrow{wpId_1} FM_1 \xrightarrow{query_1} NDB_1 \xrightarrow{query_2} \dots \xrightarrow{query_{P-1}} NDB_{P-1} \xrightarrow{query_P} NDB_P \\ &\xrightarrow{answer_P} NDB_{P-1} \xrightarrow{answer_{P-1}} \dots \xrightarrow{answer_2} NDB_1 \xrightarrow{answer_1} FM_1 \xrightarrow{wpInfo_1} MFD_1 \xrightarrow{disp_1} \end{aligned} \tag{8.1}$$

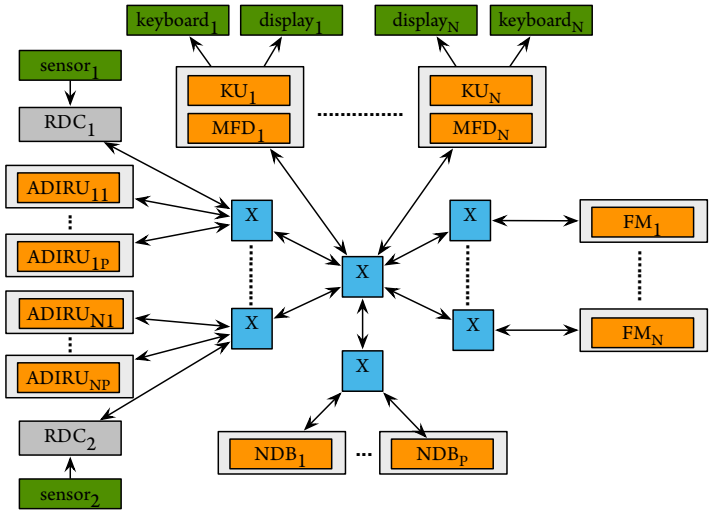



**Figure 8.6.1:** Architecture with Scalability Parameters

By increasing $P$ from 1 to 11, we give out the property values and solving time by using the proposed verification and reduction approaches in Table 8.6.1. Most of the time is consumed in the reduction phase. Once the reduction is finished, the analysis time is faster, even when the binary search method is used to compute the exact property value. Fig. 8.6.2 shows that the solving time of the case study is almost linear with respect to the *NDB* number.

**Table 8.6.1:** Scalability Test for Latency Property

| *NDB*/Fun. | Prop. Val. (ms) | | S/T (after R.) | | Reduction Time (s) | Analysis Time (s) | | Solving Time (s) | |
|---|---|---|---|---|---|---|---|---|---|
| | WCT | BCT | WCT | BCT | | WCT | BCT | WCT | BCT |
| 1/7 | 75.2 | 450.4 | 9/10 | 8/9 | 38.049 | 2.484 | 1.860 | 40.533 | 39,909 |
| 2/8 | 125.2 | 750.4 | 9/10 | 8/9 | 57.876 | 2.656 | 1.883 | 60.532 | 59,759 |
| 3/9 | 275.2 | 1050.4 | 9/10 | 6/5 | 79.813 | 2.812 | 2.079 | 82.625 | 81,892 |
| 4/10 | 375.2 | 1350.4 | 9/10 | 6/5 | 102.500 | 2.906 | 2.079 | 105.406 | 104,579 |
| 5/11 | 425.2 | 1650.4 | 9/10 | 6/5 | 124.987 | 3.015 | 2.102 | 128.002 | 127,089 |
| 6/12 | 575.2 | 1950.4 | 9/10 | 6/5 | 149.359 | 2.891 | 2.196 | 152.250 | 151,555 |
| 7/13 | 675.2 | 2250.4 | 9/10 | 6/5 | 169.607 | 2.953 | 2.227 | 172.560 | 171,834 |
| 8/14 | 725.2 | 2550.4 | 9/10 | 6/5 | 193.329 | 3.031 | 2.250 | 196.360 | 195,579 |
| 9/15 | 875.2 | 2850.4 | 9/10 | 6/5 | 216.239 | 3.000 | 2.211 | 219.239 | 218,45 |
| 10/16 | 975.2 | 3150.4 | 9/10 | 6/5 | 239.953 | 3.047 | 2.195 | 243.000 | 242,148 |
| 11/17 | 1025.2 | 3450.4 | 9/10 | 6/5 | 263.049 | 3.188 | 2.195 | 266.237 | 265,244 |

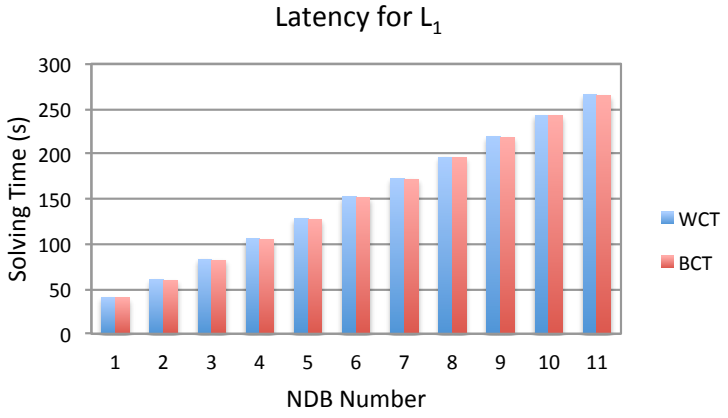



**Figure 8.6.2:** Solving Time of Scalable Latency Property

### 8.6.2 Experiments on the Freshness Functional Chain

The freshness functional chain is enlarged by increasing the number of $ADIRU_1$. Each functional chain traverses $P$ $ADIRU_1$, e.g. $P + 2$ functions.

$$F_1 = \xrightarrow{pres_1} ADIRU_{1,1} \xrightarrow{pres_2} ADIRU_{1,2} \xrightarrow{pres_3} \ldots \xrightarrow{pres_P} ADIRU_{1,P} \xrightarrow{speed_1} FM_1 \xrightarrow{ETA_1} MFD_1 \xrightarrow{disp_1} \quad (8.2)$$

By increasing the number of $ADIRU_1$ from 1 to 31, we give out the property values and solving time in Table 8.6.2. The reduction consumes almost no time in this case. The solving time is almost linear to the $ADIRU_1$ number, as shown in Fig. 8.6.3.

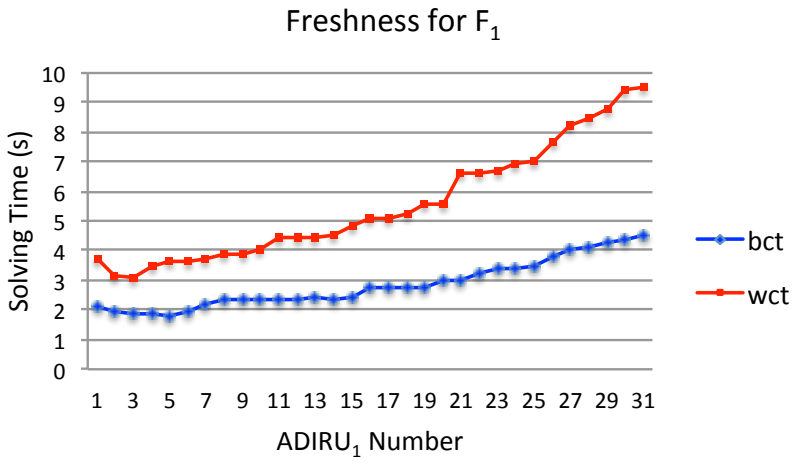



**Figure 8.6.3:** Solving Time of Freshness Property

## 8.7 Comparison to the Results in the Work of Lauer

The work of Lauer proposed a verification method for end-to-end real-time properties on IMA systems. The verification method is based on the formal modeling using the tagged signal model, which is then transformed into an Integer Linear Programming (ILP) problem. Upper-bounds of end-to-end properties are

**Table 8.6.2:** Scalability Test of Freshness Property

| $ADIRU_1$/Fun. | Prop. Val. (ms) | | S/T (after R.) | | Reduction Time (s) | Solving Time (s) | |
|---|---|---|---|---|---|---|---|
| | WCT | BCT | WCT | BCT | | WCT | BCT |
| 1 / 4 | 316429 | 1012 | 34/44 | 54/79 | 0 | 2,125 | 3,688 |
| 2 / 5 | 406585 | 1162 | 67/99 | 74/113 | 0 | 1,968 | 3,187 |
| 3 / 6 | 496741 | 1312 | 123/194 | 97/153 | 0 | 1,844 | 3,094 |
| 4 / 7 | 586897 | 1462 | 187/306 | 123/199 | 0 | 1,906 | 3,484 |
| 5 / 8 | 677053 | 1612 | 253/424 | 152/251 | 0 | 1,797 | 3,625 |
| 6 / 9 | 767209 | 1762 | 326/555 | 184/309 | 0 | 1,953 | 3,64 |
| 7 / 10 | 857365 | 1912 | 403/694 | 219/373 | 0 | 2,172 | 3,75 |
| 8 / 11 | 947521 | 2062 | 484/841 | 257/443 | 0 | 2,344 | 3,844 |
| 9 / 12 | 1037677 | 2212 | 569/996 | 298/519 | 0 | 2,343 | 3,875 |
| 10 / 13 | 1127833 | 2362 | 658/1159 | 342/601 | 0 | 2,39 | 4,031 |
| 11 / 14 | 1217989 | 2512 | 751/1330 | 389/689 | 0 | 2,344 | 4,406 |
| 12 / 15 | 1308145 | 2662 | 848/1509 | 439/783 | 0 | 2,344 | 4,438 |
| 13 / 16 | 1398301 | 2812 | 949/1696 | 492/883 | 0 | 2,406 | 4,438 |
| 14 / 17 | 1488457 | 2962 | 1054/1891 | 548/989 | 0 | 2,343 | 4,516 |
| 15 / 18 | 1578613 | 3112 | 1163/2094 | 607/1101 | 0 | 2,422 | 4,812 |
| 16 / 19 | 1668769 | 3262 | 1276/2305 | 669/1219 | 0 | 2,781 | 5,109 |
| 17 / 20 | 1758925 | 3412 | 1393/2524 | 734/1343 | 0 | 2,734 | 5,109 |
| 18 / 21 | 1849081 | 3562 | 1514/2751 | 802/1473 | 0 | 2,734 | 5,281 |
| 19 / 22 | 1939237 | 3712 | 1639/2986 | 873/1609 | 0 | 2,75 | 5,563 |
| 20 / 23 | 2029393 | 3862 | 1768/3229 | 947/1751 | 0 | 3,016 | 5,578 |
| 21 / 24 | 2119549 | 4012 | 1901/3480 | 1024/1899 | 0 | 3.000 | 6,609 |
| 22 / 25 | 2209705 | 4162 | 2038/3739 | 1104/2053 | 0 | 3,266 | 6,594 |
| 23 / 26 | 2299861 | 4312 | 2179/4006 | 1187/2213 | 0 | 3,39 | 6,672 |
| 24 / 27 | 2390017 | 4462 | 2324/4281 | 1273/2379 | 0 | 3,391 | 6,953 |
| 25 / 28 | 2480173 | 4612 | 2473/4564 | 1362/2551 | 0 | 3,453 | 7,047 |
| 26 / 29 | 2570329 | 4762 | 2626/4855 | 1454/2729 | 0 | 3,765 | 7,687 |
| 27 / 30 | 2660485 | 4912 | 2783/5154 | 1549/2913 | 0 | 4,062 | 8,25 |
| 28 / 31 | 2750641 | 5062 | 2944/5461 | 1647/3103 | 0 | 4,125 | 8,422 |
| 29 / 32 | 2840797 | 5212 | 3109/5776 | 1748/3299 | 0 | 4,25 | 8,796 |
| 30 / 33 | 2930953 | 5362 | 3278/6099 | 1852/3501 | 0 | 4,375 | 9,406 |
| 31 / 34 | 3021109 | 5512 | 3451/6430 | 1959/3709 | 0 | 4,532 | 9,484 |

computed as optimal solutions of the ILP. They showed on the case study that we reused that the approach is scalable.

According to the experimental results in the scalability tests by Lauer, for the latency requirement, when increasing the number of *NDB*, the solving time slowly grows when the *NDB* number is less than 7. After that, the solving time rapidly grows. In our test, the solving time grows is almost linear with respect to the *NDB* number. For the freshness requirement, when increasing the number of $ADIRU_1$, the solving time in both methods is almost linear with respect to the function number.

## 8.8 Conclusion

In order to test the whole property-specific proposal, we use an FMS avionic case study investigated by M. Lauer et al.. We rely on the descriptions provided by Lauer et al.. The latency and freshness real-time requirements are assessed in the case study. We model the property-specific architecture of the case study using UML-MARTE composite structure diagram, and model the property-specific behavior using activity and state machine diagrams. The UML-MARTE model is then mapped to property-specific TPN model using the mapping semantics. The real-time requirements are specified using the property patterns. After performing the property-specific reduction for TPN presented, the marking abstraction state-class graph is generated to assess the real-time properties using the observer-based model checking. We increase the length of each functional chain for both properties. The experiment results show that our approach is able to analyze large scale systems more complex than the current real systems implemented in the Airbus A380 FMS.

# 9

# Conclusion

## Résumé

Cette thèse a proposé et expérimenté des méthodes dirigées par les propriétés et un ensemble d'outils associés pour la vérification des exigences temps réels pour les systèmes critiques industriels à l'aide de techniques de vérification de modèles. Elle s'appuie sur le langage UML et son profil MARTE en tant que langage utilisateur, et sur les réseaux de Petri temporisés en tant que langage de vérification. Les propositions ont été validées à l'aide d'un cas d'étude avionique, une partie du système de gestion de vol (FMS) définie par M. Lauer en 2013, dans lequel les exigences de latence et fraîcheur des données sont évaluées. L'approche complète comporte les cinq parties suivantes : la définition de la sémantique d'exécution spécifiques aux propriétés temps réel pour les modèles d'architecture et de comportement spécifiés en UML/MARTE; la spécification des exigences temps réel en s'appuyant sur un ensemble de patrons de vérification atomiques dédiés aux propriété temps réel; une méthode itérative d'analyse à base d'observateurs pour des réseaux de Petri temporisés; des techniques de réduction de l'espace d'états spécifiques aux pro-

priétés temps réel pour des réseaux de Petri temporisés; une approche pour l'analyse des erreurs détectées par « vérification des modèles » en s'appuyant sur des idées inspirées de la « fouille de données » (« data mining »). Ces propositions ont conduit au développement d'un prototype d'outil qui exploite le langage JAVA et Eclipse Modeling Framework (EMF).

Les aspects suivants pourront étendre les travaux effectués dans cette thèse :

- Les activités appliquées suivantes pourraient être réalisées à court terme : compléter et améliorer le prototype d'outil par l'exploitation de CCSL comme langage de spécification de propriétés, expérimenter les propositions dans d'autres cas d'étude, comparer l'approche d'analyse des erreurs avec les méthodes existantes sur d'autres catégories de modèles.

- A moyen terme, les approches proposées pourraient être appliquées pour la même famille de propriétés à d'autres langages de modélisation utilisateur, par exemple AADL et EAST-ADL. ou à des langages intermédiaires comme FIACRE.

- Egalement à moyen terme, la sémantique d'exécution de l'ordonnancement de ressource pourrait être spécialisée aux différentes stratégies habituelles. Nous avons proposé un algorithme d'ordonnancement générique avec prise en compte de la préemption. Celui-ci décide à qui seront allouées les ressources et permet d'estimer des pires et meilleurs cas car il autorise toute forme d'allocation. La prise en compte de la sémantique précise des politiques d'ordonnancement classique permettra d'améliorer la précision des valeurs estimées.

- A plus long terme, l'approche de réduction spécifique aux propriétés doit être développée à la fois dans des directions fondamentales et appliquées. Dans cette thèse, nous avons éliminé les structures sans rapport avec la propriété attendue et combiné les états et transitions de sous-réseaux pour obtenir une abstraction spécifique à la propriété. La fonction de réduction est utilisée pour identifier la régularité des comportements et construire un remplaçant par le sous-réseau sélectionné. La fonction de raffinement est utilisée pour vérifier la correction du remplaçant par rapport au réseau original selon les propriétés considérées. Au moment de rédaction de cette thèse, les fonctions de réduction et de raffinement reposent sur l'approche de vérification des modèles exploitant les observateurs. Cette approche peut encore être améliorée en diminuant le temps utilisé pour la réduction et le raffinement. Il semble possible de construire les remplaçant de sous-réseau en générant le graphe d'état qu'une seule fois et puis en analysant sa structure. Il serait également souhaitable de prouver

une fois pour toute la correction de ces substitutions pour éviter d'effectuer ces vérifications à chaque exploitation de la méthode.

- A plus long terme également, la vérification des transformation de modèle exploitées devrait être réalisée. La méthode proposée dans cette thèse repose sur de nombreuses transformations de modèles qui doivent préserver la sémantique des modèles UML-MARTE et réseau de Petri temporisé. Il est donc nécessaire de vérifier cette préservation. De nombreuses techniques sont disponibles pour vérifier ce genre de propriétés pour des langages dont la sémantique est formellement définie. En ce qui concerne des langages dont il n'existe pas de formalisation de référence tels UML-MARTE, de nouvelles approches doivent être explorées. Une solution est de traduire le même modèle UML-MARTE vers différents langages de vérification et de vérifier si celles-ci convergent vers les mêmes sémantiques formelles. Ceci permet d'obtenir une meilleure confiance dans les différentes interprétations mais ne remplace pas une sémantique formelle de référence. En effet, si toutes les interprétations sont erronées de manière cohérente, rien ne sera mis en évidence même si la diversité des formalismes permet de réduire le risque. Seuls les tests et la relecture humaine par rapport à la spécification semi-formelle permettent actuellement de pallier à ce soucis mais ces activités ne sont pas exhaustives. Une variante consiste à exploiter d'autres formes de sémantique en définissant UML-MARTE sous la forme de propriétés comportementales que doivent satisfaire l'exécution des modèles. Cette approche peut être mise en oeuvre simplement en s'appuyant sur la vérification de modèle à chaque traduction et peut être également prouvée une fois pour toute. L'intérêt de cette forme de sémantique axiomatique est qu'elle est plus proche de la forme des spécifications en langue naturel et peut s'appuyer sur les travaux existant en ingénierie des exigences pour formaliser des exigences en langue naturelle.

This thesis designed and experimented a property-driven methodology and an associated toolset for the verification of real-time requirements for industrial scale safety critical systems based on model checking tools. It relies on the UML and its MARTE profile as end-user modeling languages, and on the Time Petri Nets (TPN) as verification language. It was validated using an avionic use case: a subset of a Flight Management System focusing on latency and freshness requirements. The whole methodology and the prototype includes five parts: real-time property specific system model mapping, real-time property specification, observer-based real-time property verification, real-time property specific state space reduction, and automatic fault localization.

As a final conclusion, we summarize the fulfillment of objectives in Section 9.1, and the application of research results in Section 9.2. We also outline some directions for the future research in Section 9.3.

## 9.1 Fulfillment of Objectives

**Fulfillment of challenge 1: Specification, implementation and validation of a real-time property specific execution semantics for UML-MARTE models that allows scalable verification.** With respect to the expected real-time requirement, we have defined a real-time property specific execution semantics for UML-MARTE architecture models (composite structure diagram) and behavior models (activity and state machine diagrams). The definition of this execution semantics follows the property-driven approach proposed by Combemale et al. It allows to map UML-MARTE entities to TPN models, which makes UML model executable and analyzable by the TINA model checking toolset. This mapping conforms to the UML specification 2.4.1. It abstracts the system in order to provide more scalable verification for a specific family of properties. The full mapping library is given in Appendix A. A generic scheduling algorithm including a preemption option is also defined. This scheduling algorithm is used to decide for the given time $T$, which resource instance(s) will be allocated to which requester(s).

**Fulfillment of challenge 2: Practical real-time requirement specification method for verification purpose.** From the viewpoint of requirement assessment, we advocate that the qualitative property patterns proposed by Dwyer and the quantitative property patterns proposed by Konrad are not semantically atomic. These property specifications need to be decomposed into atomic elements to improve the verification efficiency. A property specification method that can ease the verification is needed to bridge this gap. We have defined a set of atomic real-time property patterns. These property patterns can be directly

used to specify real-time requirements. The properties expressed using Dwyer/Konrad's patterns and CCSL languages can also be automatically translated to the verification targeted atomic property elements, which will then be assessed using the observer-based verification approach.

**Fulfillment of challenge 3: Scalable model checking support for the verification of real-time properties in TPN model.** The model checking toolset TINA that our work relies on can express qualitative properties using LTL and CTL logic formulae, but not quantitative properties. To assess the real-time properties in an efficient manner, we have defined a set of event-based TPN observers and state-based tts observers, that will be associated to the TPN system under observation. These observers express the same semantics as the atomic elements defined in the real-time property patterns. The proposed observer-based approach allows to generate the high abstraction state class graph that only preserves marking information using the tina state space generation tool from the TINA toolset. It relies on the accessibility assertions in the modal $\mu$-calculus (MMC) and the muse model checker from the TINA toolset.

**Fulfillment of challenge 4: Property-specific state space reduction method.** We propose a property specific reduction tool to eliminate the property-irrelevant TPN structures and to build an equivalent of the property-relevant TPN structures in the TPN system model. The reduction tool exploits the commutativity of TPN sub-nets which results in the same property-specific behavior before expanding the state class graph. The equivalent has less states and transitions, and thus directly reduces the scale of computation.

**Fulfillment of challenge 5: Failure analysis approach to locate the origin of fault.** We propose an automated fault localization approach based on model checking to ease and accelerate debugging by locating and ranking the suspicious elements in a model when a safety property is unsatisfied. Inspired by the TF-IDF (term frequency-inverse document frequency) measure and the Kullback–Leibler Divergence theory, we propose a suspiciousness factor to rank the potentially faulty transitions. We apply this approach to property specific TPN model relying on observers-based verification approach to provide all the faulty execution traces and the violation states in the state class graph preserving markings. Based on the mapping semantics from UML to TPN, the faulty transitions can be back-traced from TPN to UML.

## 9.2 Application of Research Results

We have implemented the approaches presented in this thesis as the prototype toolset UMLMMC (UML-MARTE Model Checker). It includes the following tools:

- (RTM) **R**eal-**T**ime property specific system model **M**apping tool,
- (RTS) **R**eal-**T**ime property **S**pecification tool,
- (RTV) observer-based **R**eal-**T**ime property **V**erification tool
- (RTR) **R**eal-**T**ime property specific state space **R**eduction tool,
- (FLMC) automatic **F**ault **L**ocalization tool in **M**odel **C**hecking.

**Supported tools.**

At the very beginning of this work, we relied on the Topcased[1] (version 5.1.0 under MacOS X 64 bit) environment for purpose of modeling and development. Our toolset is now compatible with the Papyrus[2] (version 0.10.0) and the PolarSys IDE [3] (version 0.7).

For model checking, we rely on the TINA toolset 3.2.0 for Intel Macs under MacOS X 64 bit.

**Implementation using JAVA and EMF.**

This toolset has been developed using the JAVA language and Eclipse Modeling Framework (EMF) [4], which is a modeling framework and code generation facility for building tools and other applications based on a structured data model. On the one hand, the JAVA language can ease the integration of tools; on the other hand, JAVA provides many sophisticated tools for the debugging feature, which is mandatory to rapidly test and improve the algorithms involved in our approaches. The prototype toolset includes 30264 lines of JAVA code.

[1] http://www.topcased.org/
[2] https://www.eclipse.org/papyrus/
[3] http://polarsys.org/
[4] https://www.eclipse.org/modeling/emf/

## 9.3 Future Research Directions

Both practical and fundamental aspects can follow the work conducted in this thesis.

### 9.3.1 Short Term Activities

Before discussing the future research directions, we first summarize the following short term future works:

- The development of the prototype toolset can be further completed and improved. We have presented that the real-time requirements based on Dwyer's and Konrad's works and CCSL constraints can be automatically transformed to our real-time property patterns. For now, the prototype of this tool does not cover all the property mapping. It will be completed in the near future.
- In this thesis, we have experimented our approaches on the FMS case study for the end-to-end real-time requirements. Other industrial case studies should be experimented and used to further validate our proposal. We plan to conduct such activities on railroad signaling system models in the Open ETCS project [5] and on other aeronautic use cases in the P project.
- The fault localization approach can be further experimented and compared with the existing approaches. Other test beds involving different kinds of errors should be developed. This feedback analysis approach can be integrated in other model checking tools to help the designer in the failure analysis.

### 9.3.2 Resource scheduling semantics mapping

In the UML-MARTE models, behaviors (activity and state machine) consumes resources such as CPU, memory, communication resources etc. The scheduling policy applied by the scheduler will impact the real-time requirements. Thus, if the target system relies on some external resources, the real-time behavior for the resources scheduling needs to be explicitly specified in the TPN model.

The MARTE profile MARTE::MARTE_Foundations::GRM::Scheduler:schedPolicy provides some typical scheduling policies for real-time embedded systems, such as Earliest Deadline First, FIFO, Fixed Priority, Least Laxity First, Round Robin, Time Table Driven. It also allows users to define their own scheduling policy. Mapping semantics for these well-known scheduling policies to TPN model could introduce some

[5] http://openetcs.org/

semantic ambiguities. For example, when using Fixed Priority scheduling policy, there is no explicit indication in the UML-MARTE level to specify what should be the scheduler behavior if two requests have the same priority; but as this information is mandatory for the TPN modeling, then a semantic gap is potentially created. If we do not have full determinism all the time in the UML-MARTE models, it is possible to introduce fairness properties to handle the conflicts in TPN models.

Besides, the exact behavior of some dynamic scheduling policy could not be mapped to TPN in a trivial way. For example the EDF/preemptive policy always need to compute for each reassignment cycle the process which is the closest to its deadline. This requires a dynamic comparison between the amount clock/time state of each transition and the given reference, which is unfeasible neither in classical TPN nor in TPN with data extension.

In this thesis, we have proposed a generic scheduling algorithm with preemption option. This scheduling algorithm is used to decide for the given time $T$, which resource requester(s) will be allocated by the resource instance(s). The mapping semantics for specific scheduling policy can be a direction for future research. Gherbi and Khendek discussed in [GK09] a model transformation enabling the schedulability analysis in UML-SPT (the predecessor of MARTE) real-time systems and implemented a prototype using ATL. They also defined a metamodel encapsulating the main information required for the schedulability analysis. The UML-SPT is then transformed to the target analysis model using this metamodel. The analysis metamodel and the transformation method could be used by our future work on schedulability analysis.

### 9.3.3 Future Research Direction for Property Specific Reduction Approach

The property-relevant structure is reduced using the commutativity of TPN sub-nets which result in the same property-specific behavior. The property-specific reduction method relies on similar ideas to partial order reductions. It can be used for asynchronous concurrent systems, in which most of the activities in different processes are performed independently, without a globally synchronization.

We first identify and extract the reducible sub-blocks from the whole system using an *Identification* function. Then the state space of the reducible sub-blocks are reduced using a *Reduction* function. The reduced sub-blocks are derived, and are then used to replace the original sub-blocks after their soundness is assessed using a *Refinement* function.

At the time of writing this thesis, the *Reduction* and *Refinement* functions rely on the real-time property specification and observer-based verification approaches in our verification toolset. Suppose a reducible

TPN sub-net is $N_s$ and its reduced sub-net is $N'_s$. We use the *Reduction* function to search for the sequential and loop sections that are used as the behavioral pattern of $N_s$, and then verify if this pattern behaves as the same as the system's real behavior using the *Refinement* function. If verified, an $N'_s$ conforming to this pattern will replace $N_s$. The reduction and refinement functions can be formally specified and proved. This should be further studied in the near future.

On the other hand, once an $N_s$ is identified, in order to compute the associated $N'_s$, some related TPN observers need to be associated to $N_s$, and the corresponding state class graphs are then generated $m$ times ($m$ depends on the behavior of $N_s$). Indeed, this approach has reduced the state space explosion problem in asynchronous systems using a time–memory tradeoff. But this approach can still be improved by decreasing the time used for the reduction and refinement. It is possible to build the sequential and loop sections for $N'_s$ by generating the state class graph only once and then analyzing its topology structure. This will be an interesting future research direction.

Boucheneb and Barkaoui proposed in [BB13] an effective method for reducing interleaving semantics redundancy in the reachability analysis of Time Petri Net. Their work showed that the union of state zones reached by different interleavings of the same set of transitions is not necessarily a state zone. They established sufficient conditions which ensure that this union is a state zone and showed how to compute this state zone without computing intermediate ones. It is possible to draw lessons from this work and propose more efficient property specific reduction methods for TPN models.

### 9.3.4 Verification of Model Transformation

The automatic model transformation referred to in this work is in fact a semantic mapping, which preserves all the property-related semantics of the source UML-MARTE model. A concern with this method is whether the model transformation (semantic mapping) is correct. In other words, how to verify this model transformation (semantic mapping). Indeed, this is a crucial question. Some surveys of the state-of-the-art about the verification of code generation [Dav03, Nec11] and the verification of model transformation [CS13, PSS98] summarized some expected properties to be verified and possible verification techniques, as discussed in Section 3.8 of Chapter 3.

The verification of model transformation for the UML-MARTE model is not trivial. Generally, the best way to verify if the model transformation preserves the intended semantics is to compare the state space graph of the source and target formal models. A formal semantics must then be defined for UML models as

a reference semantics. The execution semantics is then compared with this reference semantics. However, since UML is semi-formal,a formal definition is needed to establish the reference, which is one of the work in this thesis. Our proposal relies on a translation to a formal model instead of a direct formal specification of an operational semantics that would allow to build the state space at the UML level. This does not change the fundamental issue: how to validate this formal specification?

A solution may mitigate the problem by mapping the UML-MARTE model to different formal models and verifying if they converge into the same formal semantics. Nevertheless, whether the semantics is lost between a semi-formal model and a formal one can only be assessed using testing and human proof reading.

Another possible solution is derived from translation validation and proof carrying code that have been experimented for the same purpose for AADL in the QUARTEFT project. This method allows to verify that some important intended behavioral properties conform to the execution semantics. For example, we can define TPN observers to assess the run-to-completion processing semantics. More precisely, when an event occurrence is being processed, the other occurrences of this event cannot be accepted. However, when the behavior property specification and the execution semantics are both wrong in the same way, this method does not work. Then some test cases must be used to validate the execution semantics.

As a future research direction, the expected behavior properties would be defined and used to verify the conformance between the execution semantics and the behavior specification. This can validate some key execution semantics in the UML models.

### 9.3.5 Application of the Approaches to Other Modeling Language

The property-driven approaches proposed in this thesis can be applied to other end-user modeling language, for example AADL, EAST-ADL, or to intermediate languages like Fiacre. We can assess real-time requirements expressed in these models by following the same approaches: mapping end-user models to property specific TPN models, expressing real-time requirements using the proposed real-time property patterns, reducing TPN state space using property specific reduction, assessing real-time properties relying on observer-based model checking, and deriving the ranked faulty elements from the feedback analysis.

# A

# Appendix A:

# Coverage Library: Mapping UML-MARTE to TPN

## A.1 Coverage Library of Composite Structure Diagram

**Table A.1.1:** Coverage Library of Composite Structure Diagram

| **Coverage Library: UML-MARTE Composite Structure Diagram** | | |
|---|---|---|
| **Node Group** | **Node Type** | **TPN Mapping Coverage** |
| Object | Part | √ |
| | Role | |
| | Interface | √ |
| | Port | √ |
| | CollaborationUse | |
| Connections | Connector | √ |
| | InterfaceRealization | |
| | Role Binding | |

## A.2 Coverage Library of Activity Diagram

**Table A.2.1:** Coverage Library of Activity Diagram

| Coverage Library: UML-MARTE Activity Diagram | | |
|---|---|---|
| **Node Group** | **Node Type** | **TPN Mapping Coverage** |
| Common | Activity Partition | |
| Control | Initial Node | √ |
| | Decision Node | √ |
| | Merge Node | √ |
| | Fork Node | √ |
| | Join Node | √ |
| | Activity Final | √ |
| | Flow Final | √ |
| | Expansion Region | |
| | Structured Activity Node | |
| | Conditional Node | |
| | Interruptible Activity Region | |
| | Loop Node | |
| | Sequence Node | |
| Actions | Action | √ |
| Object | Activity Parameter | |
| | Central Buffer | √ |
| | DataStore | √ |
| | Expansion | |
| | Input Pin | √ |
| | Output Pin | √ |
| Connections | Control Flow | √ |
| | Object Flow | √ |
| | Exception Handler | |

## A.3 Coverage Library of State Machine Diagram

**Table A.3.1:** Coverage Library of State Machine Diagram

| **Coverage Library: UML-MARTE State Machine Diagram** | | |
|---|---|---|
| **Node Group** | **Node Type** | **TPN Mapping Coverage** |
| Object | Region | √ |
| | State | √ |
| | Composite State | √ |
| | Submachine State | √ |
| | ConnectionPointReference | |
| | FinalState | √ |
| Pseudostates | Initial | √ |
| | Deep History | √ |
| | Shallow History | √ |
| | Join | √ |
| | Fork | √ |
| | Junction | √ |
| | Choice | √ |
| | Entry Point | √ |
| | Exit Point | √ |
| | Terminate | √ |
| Connections | External Transition | √ |
| | Local Transition | √ |

# B

# Appendix B:
# Mapping Library: Real-Time Property Pattern

## B.1 Pattern Mapping Library

**Note**: $E$ stands for event, $S$ for state, $S^s$ for entering transition of state, $S^e$ for exiting transition of state, and $T$ for time $T$ *t.u.*.

**Table B.1.1:** Pattern Mapping Library

| **Pattern Mapping Library** | | |
|---|---|---|
| **Pattern** | **Dwyer / Konrad Property Pattern** | **Mapping Result** |
| **Absence** | Absence of S | Absent S |
| | Absence of S at least t | Absent $D(S) \geq t$ |
| | Absence of S at most t | Absent $D(S) < t + 1$ |
| | Absence of E | Absent $O(E^1)$ |
| | Absence of E at least t | Always $T(E^{i+1}, E^i) > t$ |
| | Absence of E at most t | Always $T(E^{i+1}, E^i) < t$ |
| **Existence** | Existence of S | Exist S |
| | Existence of S at least t | Exist $D(S) \geq t$ |
| | Existence of S at most t | Exist $D(S) < t + 1$ |
| | Existence of E | Exist $O(E^1)$ |
| | Existence of E at least t | ~~Illegal semantics~~ [1] |
| | Existence of E at most t | ~~Illegal semantics~~ [1] |
| **Universality** | Universality of S | Always S |
| | Universality of S at least t | Always $D(S) \geq t$ |
| | Universality of S at most t | Always $D(S) < t + 1$ |
| | Universality of E | ~~Illegal semantic~~ [1] |
| | Universality of E at least t | ~~Illegal semantics~~ [1] |
| | Universality of E at most t | ~~Illegal semantics~~ [1] |
| **Bounded Existence** | K-Bounded Existence of S | ~~Illegal semantic~~ [2] |
| | K-Bounded Existence of S at least t | ~~Illegal semantics~~ [2] |

*Continued on next page*

[1] Illegal in Dwyer's and Konrad's pattern system. An event is instantaneous, without time duration.
[2] Illegal in system design. A state is not specified by the number of occurrences.

| Pattern | Konrad Property Pattern | Mapping Result |
|---|---|---|
| | K-Bounded Existence of S at most t | ~~Illegal semantics~~ [2] |
| | K-Bounded Existence of E | Exist $O(E^k)$ |
| | K-Bounded Existence of E at least t | ~~Illegal semantics~~ [1] |
| | K-Bounded Existence of E at most t | ~~Illegal semantics~~ [1] |
| **Precedence** | $S_1$ precedes $S_2$ | $T(S_1^E, S_2^S) > 0$ |
| | $S_1$ precedes $S_2$ at least t | $T(S_1^E, S_2^S) > t$ |
| | $S_1$ precedes $S_2$ at most t | $T(S_1^E, S_2^S) < t$ |
| | S precedes E | $T(S^e, E) > 0$ |
| | S precedes E at least t | $T(S^e, E) > t$ |
| | S precedes E at most t | $T(S^e, E) < t$ |
| | E precedes S | $T(E, S^s) > 0$ |
| | E precedes S at least t | $T(E, S^s) > t$ |
| | E precedes S at most t | $T(E, S^s) < t$ |
| | $E_1$ precedes $E_2$ | $T(E_1, E_2) > 0$ |
| | $E_1$ precedes $E_2$ at least t | $T(E_1, E_2) > t$ |
| | $E_1$ precedes $E_2$ at most t | $T(E_1, E_2) < t$ |
| **Response** | $S_1$ leads to $S_2$ | $T(S_1^E, S_2^S) > 0$ |
| | $S_1$ leads to $S_2$ at least t | $T(S_1^E, S_2^S) > t$ |
| | $S_1$ leads to $S_2$ at most t | $T(S_1^E, S_2^S) < t$ |
| | S leads to E | $T(S^e, E) > 0$ |
| | S leads to E at least t | $T(S^e, E) > t$ |
| | S leads to E at most t | $T(S^e, E) < t$ |
| | E leads to S | $T(E, S^s) > 0$ |
| | E leads to S at least T | $T(E, S^s) > t$ |
| | E leads to S at most T | $T(E, S^s) < t$ |
| | $E_1$ leads to $E_2$ | $T(E_1, E_2) > 0$ |
| | $E_1$ leads to $E_2$ at least t | $T(E_1, E_2) > t$ |
| | $E_1$ leads to $E_2$ at most t | $T(E_1, E_2) < t$ |

## B.2 Scope Mapping Library

**Note**: *E* stands for event, *S* for state, $S^s$ for entering transition of state, $S^e$ for exiting transition of state, and *I* for time interval.

**Table B.2.1:** Scope Mapping Library

| **Scope Mapping Library** | | |
|---|---|---|
| **Scope** | **Konrad Property Pattern** | **Mapping Result** |
| **Global** | Global | Global |
| **Before** | Before E<br>E' Before E within $I[t_{min}, t_{max}]$ | Before $E^1$<br>(E'+$t_{max}$) After E $\wedge$ (E'+$t_{min}$) Before E |
| **After** | After E<br>After E within $I[t_{min}, t_{max}]$ | After $E^1$<br>After $(E + t_{min}) \wedge$ Before $(E + t_{max})$ |
| **Between** | Between $E_1$ and $E_2$<br>Between $E_1$ and $E_2$ within $I[t_{min}, t_{max}]$ | Between $E_1$ and $E_2$<br>$E_2$ After $(E1 + t_{max}) \wedge$ Between $(E_1 + t_{min})$ and $(E_1 + t_{max}) \vee E_2$ Before $(E1 + t_{max}) \wedge$ Between $(E_1 + t_{min})$ and $E_2$ |
| **After-Until** | After $E_1$ Until $E_2$ | (Exist $E_2$ After $E_1 \wedge$ Between $E_1$ and $E_1$) $\vee$ (Absent $E_2$ After $E_1 \wedge$ After $E_1$) |
| **Periodically** | During each period of E | Between $E^{-1}$ and E |

# Bibliography


[AADW09] Shaimaa Ali, James H Andrews, Tamilselvi Dhandapani, and Wantao Wang. Evaluating the accuracy of fault localization techniques. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 76–87. IEEE Computer Society, 2009.

[ABB+08] A Albinet, S Begoc, JL Boulanger, O Casse, I Dal, H Dubois, F Lakhal, D Louar, MA Peraldi-Frati, Y Sorel, et al. The memvatex methodology: from requirements to models in automotive application design. In *4th European Congress ERTS (Embedded Real Time Software), Toulouse, France*, 2008.

[ABD+07] Arnaud ALBINET, Jean-Louis Boulanger, Hubert Dubois, Marie-Agnès Peraldi-Frati, Yves Sorel, Quang-Dao Van, et al. Model-based methodology for requirements traceability in embedded systems. In *Proceedings of 3rd European Conference on Model Driven Architecture® Foundations and Applications, ECMDA'07*, 2007.

[ABD+08] Francis Alizon, Mariano Belaunde, Gregoire Dupre, Bertrand Nicolas, Sebastien Poivre, and Jacques Simonin. Les modèles dans l'action à france télécom avec smartqvt. *Génie logiciel*, (85):35–42, 2008.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[ADZLB12] Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. Real-time specification patterns and tools. In *Formal Methods for Industrial Critical Systems*, pages 1–15. Springer, 2012.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[Ali12] Mohammad Amin Alipour. Automated fault localization techniques; a survey. Technical report, 2012.

[Alu91] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, stanford university, 1991.

[AM08] Charles André and Frédéric Mallet. Clock constraints in uml/marte ccsl. Research Report RR-6540, INRIA, 2008.

[AM09] Charles André and Frédéric Mallet. Specification and verification of time requirements with ccsl and esterel. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 167–176, New York, NY, USA, 2009. ACM.

[AMCN09] Ermeson Andrade, Paulo Maciel, Gustavo Callou, and Bruno Nogueira. A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In *Digital Society, 2009. ICDS'09. Third International Conference on*, pages 266–271. IEEE, 2009.

[AZB13] Nouha Abid, Silvano Dal Zilio, and Didier Le Botlan. A verified approach for checking real-time specification patterns. *arXiv preprint arXiv:1301.7531*, 2013.

[B+86] Gérard Berthelot et al. Checking properties of nets using transformations. In *Advances in Petri Nets 1985*, pages 19–40. Springer, 1986.

[BB13] Hanifa Boucheneb and Kamel Barkaoui. Reducing interleaving semantics redundancy in reachability analysis of time petri nets. *ACM Trans. Embedded Comput. Syst.*, 12(1):7, 2013.

[BBB+57] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, L Mitchell Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, H Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198. ACM, 1957.

[BBBB11] Sabine Boufenara, Kamel Barkaoui, Faiza Belala, and Hanifa Boucheneb. Mapping uml activity diagrams to analyzable petri net models. In *Fifth International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2011)*, September 2011.

[BBDC+09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Trefler. Explaining counterexamples using causality. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[BBF+07] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Hubert Garavel, Frédéric Lang, Florent Peres, Rodrigo Saad, Jan Stoecker, François Vernadat, P Gaufillet, et al. The syntax and semantics of fiacre. *rapport LAAS*, 7264, 2007.

[BCAA00] Jet Propulsion Laboratory (U.S.). Special Review Board, J. Casani, United States. National Aeronautics, and Space Administration. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*. Jet Propulsion Laboratory, California Institute of Technology, 2000.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and computation*, 98(2):142–170, 1992.

[BCN98] Stan Budkowski, Ana Cavalli, and Elie Najm. *Formal Description Techniques and Protocol Specification, Testing and Verification*, volume 6. Springer, 1998.

[BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, March 1991.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[Ber83] G. Berthelot. *Transformations et analyse de réseaux de Petri: application au protocoles*. Rapports de recherche / Université de Paris-Sud, Laboratoire de recherche en informatique. LRI, 1983.

[Ber01] Bernard Berthomieu. La méthode des classes d'états pour l'analyse des réseaux temporels. In *3e congrès Modélisation des Systèmes Réactifs (MSR'2001)*, pages 275–290, 2001.

[BM83] Bernard Berthomieu and Miguel Menasche. An enumerative approach for analyzing time petri nets. In *Proceedings IFIP*. Citeseer, 1983.

[BM98] L Douglas Baker and Andrew Kachites McCallum. Distributional clustering of words for text classification. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 96–103. ACM, 1998.

[BNR03] Thomas Ball, Mayur Naik, and Sriram K Rajamani. From symptom to cause: localizing errors in counterexample traces. *ACM SIGPLAN Notices*, 38(1):97–105, 2003.

[BR01] Thomas Ball and Sriram K Rajamani. The slam toolkit. In *Computer aided verification*, pages 260–264. Springer, 2001.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[BRV04] 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.

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[BSF09] Henri Bauer, J-L Scharbarg, and Christian Fraboul. Applying and optimizing trajectory approach for performance evaluation of afdx avionics network. In *Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on*, pages 1–8. IEEE, 2009.

[BV03] Bernard Berthomieu and François Vernadat. State class constructions for branching analysis of time petri nets. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–457. Springer, 2003.

[BV07] Bernard Berthomieu and Françcois Vernadat. State space abstractions for time petri nets. *Handbook of Real-Time and Embedded Systems, Crc Computer & Information Science Series. Chapman & Hall*, 2007.

[C+97] Airlines Electronic Engineering Committee et al. Arinc specification 653. *Aeronautical Radio Inc*, 2551, 1997.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCG+04] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *Software Engineering, IEEE Transactions on*, 30(6):388–402, 2004.

[CCG+07] Benoît Combemale, Xavier Crégut, Pierre-Loïc Garoche, Xavier Thirioux, and François Vernadat. A property-driven approach to formal verification of process models. In *ICEIS (Selected Papers)*, volume 12 of *Lecture Notes in Business Information Processing*, pages 286–300. Springer, 2007.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CEFJ96] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[CES71] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking*. MIT press, 1999.

[CGP03] Jamieson M Cobleigh, Dimitra Giannakopoulou, and Corina S Păsăreanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.

[CGS04] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 73–82. ACM, 2004.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003.

[Cla08] Edmund M Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[CMC+08] Ermeson Carneiro, Paulo Maciel, Gustavo Callou, Eduardo Tavares, and Bruno Nogueira. Mapping sysml state machine diagram to time petri net for analysis and verification of embedded real-time systems with energy constraints. In *Advances in Electronics and Microelectronics, 2008. ENICS'08. International Conference on*, pages 1–6. IEEE, 2008.

[CR06] Franck Cassez and Olivier H. Roux. Structural translation from time petri nets to timed automata. *JSS*, 79(10):1456–1468, October 2006.

[Cra05] Michelle L Crane. *On the Syntax and Semantics of State Machines*. PhD thesis, Citeseer, 2005.

[CS13] Daniel Calegari and Nora Szasz. Verification of model transformations: a survey of the state-of-the-art. *Electronic Notes In Theoretical Computer Science*, 292:5–25, 2013.

[CV03] Edmund M. Clarke and Helmut Veith. Counterexamples revisited: Principles, algorithms, applications. In *Verification: Theory and Practice*, pages 208–224, 2003.

[CZ05] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[DAC98] Matthew B Dwyer, George S Avrunin, and James C Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM, 1998.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420. ACM, 1999.

[Dav03] Maulik A Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.

[DBRL12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012:9, 2012.

[DFH+91] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, Christine Paulin-Mohring, et al. The coq proof assistant user's guide: Version 5.6. 1991.

[Din08] Steven X. Ding. Basic ideas, major issues and tools in the observer-based fdi framework. In *Model-based Fault Diagnosis Techniques*, pages 13–19. Springer Berlin Heidelberg, 2008.

[DPC+09] Philippe Dhaussy, Pierre Yves Pillain, Stephen Creff, Amine Raji, Yves Le Traon, and Benoit Baudry. Evaluating context descriptions and property definition patterns for software formal validation. In *MoDELS*, pages 438–452, 2009.

[DSLT+04] Vincent Debruyne, Françoise Simonot-Lion, Yvon Trinquet, et al. East-adl-an architecture description language-validation and verification aspects. *Architecture Description Language*, 2004.

[EC80] E.Allen Emerson and EdmundM. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco Bakker and Jan Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin Heidelberg, 1980.

[Eng05] Condor Enginering. Inc. afdx/arinc 664 tutorial (1500—049), 2005.

[Ern03] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[ES96] E Allen Emerson and A Prasad Sistla. Symmetry and model checking. *Formal methods in system design*, 9(1-2):105–131, 1996.

[ES97] E. Allen Emerson and A. Prasad Sistla. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):617–638, 1997.

[ES07] John Erickson and Keng Siau. Theoretical and practical complexity of modeling methods. *Commun. ACM*, 50(8):46–51, August 2007.

[EWDC10] W Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[FFN91] Gerard Florin, Céline Fraize, and Stéphane Natkin. Stochastic petri nets: Properties, applications and tools. *Microelectronics Reliability*, 31(4):669–697, 1991.

[FGH06] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl): An introduction. Technical report, DTIC Document, 2006.

[FHN+06] Jean-Rémy Falleri, Marianne Huchard, Clémentine Nebut, et al. Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW'06: ECMDA Traceability Workshop*, pages 31–40, 2006.

[FM95] Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle, 1995.

[GBF99] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. *SIGSOFT Softw. Eng. Notes*, 24(6):303–321, October 1999.

[GDRA+12] Karen Godary-Dejean, Romain Richard, Gregory Angles, et al. Lpt-a tool for parametric tpn validation. In *VECoS'2012: 6th International Workshop on Verification and Evaluation of Computer and Communication Systems*, 2012.

[GK09] Abdelouahed Gherbi and Ferhat Khendek. From uml/spt models to schedulability analysis: approach and a prototype implementation using atl. *Automated Software Engineering*, 16(3-4):387–414, 2009.

[GKL04a] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *CAV*, pages 453–456, 2004.

[GKL04b] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In *Computer Aided Verification*, pages 453–456. Springer, 2004.

[GL94] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

[GL06] Volker Gruhn and Ralf Laue. Patterns for timed property specifications. *Electronic Notes in Theoretical Computer Science*, 153(2):117–133, 2006.

[GLM+05] Guillaume Gardey, Didier Lime, Morgan Magnin, et al. Romeo: A tool for analyzing time petri nets. In *Computer Aided Verification*, pages 418–423. Springer, 2005.

[GNP13a] Ning Ge, Shin Nakajima, and Marc Pantel. Efficient online analysis of accidental fault localization for dynamic systems using hidden markov model. In *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*, page 16. Society for Computer Simulation International, 2013.

[GNP13b] Ning Ge, Shin Nakajima, and Marc Pantel. Hidden markov model based automated fault localization for integration testing. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, pages 184–187. IEEE, 2013.

[GNP15] Ning Ge, Shin Nakajima, and Marc Pantel. Online diagnosis of accidental faults for real-time embedded systems using a hidden markov model. *Simulation*, 91(10):851–868, 2015.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *Computer Aided Verification*, pages 438–449. Springer, 1993.

[GP12a] Ning Ge and Marc Pantel. Time properties verification framework for uml-marte safety critical real-time systems. In *European Conference on Modelling Foundations and Applications*, pages 352–367. Springer, 2012.

[GP12b] Ning Ge and Marc Pantel. Verification of synchronization-related properties for uml-marte rtes models with a set of time constraints dedicated formal semantic. 2012.

[GP14] Ning Ge and Marc Pantel. Real-time property specific reduction for time petri net. In *International Workshop on Petri Nets and Software Engineering (PNSE@PetriNets)*, pages 165–179, 2014.

[GPC12a] Ning Ge, Marc Pantel, and Xavier Crégut. Formal specification and verification of task time constraints for real-time systems. In *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 143–157. Springer, 2012.

[GPC12b] Ning Ge, Marc Pantel, and Xavier Crégut. Time properties dedicated transformation from uml-marte activity to time transition system. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012.

[GPC14a] Ning Ge, Marc Pantel, and Xavier Crégut. Automated failure analysis in model checking based on data mining. In *International Conference on Model and Data Engineering*, pages 13–28. Springer, 2014.

[GPC14b] Ning Ge, Marc Pantel, and Xavier Crégut. Probabilistic failure analysis in model validation &verification. In *International Conference on Embedded Real Time Software and Systems (ERTS)*, 2014.

[GPC14c] Ning Ge, Marc Pantel, and Xavier Crégut. A uml-marte temporal property verification tool based on model checking. In *International Conference on Embedded Real Time Software and Systems (ERTS)*, 2014.

[Gro04] Alex Groce. Error explanation with distance metrics. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, 2004.

[GS97] Viktor Gyuris and A Prasad Sistla. On-the-fly model checking under fairness that exploits symmetry. In *Computer Aided Verification*, pages 232–243. Springer, 1997.

[GV03] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Model Checking Software*, pages 121–136. Springer, 2003.

[GvLH+96] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[Had90] S. Haddad. A reduction theory for coloured nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1989*, volume 424 of *Lecture Notes in Computer Science*, pages 209–235. Springer Berlin Heidelberg, 1990.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[HC96] G George Edward Hughes and Maxwell John Cresswell. *A new introduction to modal logic*. Psychology Press, 1996.

[Hol90] Gerard J Holzmann. Design and validation of protocols. In *Tutorial Computer Networks and ISDN Systems*. Citeseer, 1990.

[Hol96] Gerard Holzmann. On-the-fly model checking. *ACM Computing Surveys (CSUR)*, 28(4es):120, 1996.

[Hol97a] C Michael Holloway. Why engineers should consider formal methods. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 1, pages 1–3. IEEE, 1997.

[Hol97b] Gerard J Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[HP94] Gerard J Holzmann and Doron Peled. An improvement in formal verification. In *FORTE*, volume 6, pages 197–211, 1994.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[ID96] C Norris Ip and David L Dill. Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75, 1996.

[ITI07] Jean-Bernard ITIER. A380 integrated modular avionics - the history, objectives and challenges of the deployment of ima on a380. Technical report, ARTIST2 meeting on Integrated Modular Avionics, 2007.

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[Jen96] Kurt Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*, volume 1. Springer, 1996.

[JH05] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[JHS02a] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[JHS02b] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[JM11a] Manu Jose and Rupak Majumdar. Bug-assist: assisting fault localization in ansi-c programs. In *Computer Aided Verification*, pages 504–509. Springer, 2011.

[JM11b] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *ACM SIGPLAN Notices*, volume 46, pages 437–446. ACM, 2011.

[Jon72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[Jon83] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[KC05] Sascha Konrad and Betty HC Cheng. Real-time specification patterns. In *Proceedings of the 27th international conference on Software engineering*, pages 372–381. ACM, 2005.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[KL51] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

[Kob99] Cris Kobryn. Uml 2001: A standardization odyssey. *Commun. ACM*, 42(10):29–37, 1999.

[Kop11] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[Lau12] Michaël Lauer. *Une méthode globale pour la vérification d'exigences temps réel - Application à l'Avionique Modulaire Intégrée*. PhD thesis, juin 2012.

[LEBP11a] Michael Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. Latency and freshness analysis on ima systems. In *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*, pages 1–8. IEEE, 2011.

[LEBP11b] Michaël Lauer, Jérôme Ermont, Frédéric Boniol, and Claire Pagetti. Worst case temporal consistency in integrated modular avionics systems. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 212–219. IEEE, 2011.

[LEPB10] Michaël Lauer, Jérôme Ermont, Claire Pagetti, and Frédéric Boniol. Analyzing end-to-end functional delays on an ima platform. In *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 243–257. Springer, 2010.

[LGMC04] Juan Pablo López-Grao, José Merseguer, and Javier Campos. From uml activity diagrams to stochastic petri nets: application to software performance engineering. In *4th international workshop on Software and performance*, WOSP '04, pages 25–36, New York, NY, USA, 2004. ACM.

[Lio96] J. L. Lions. Ariane 5 flight 501 failure. Technical report, ESA: Ariane 501 Inquiry Board, 1996.

[LSV97] Edward A Lee and Alberto Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 234–241. IEEE Computer Society, 1997.

[LYF+05] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, September 2005.

[MAL10] Frédéric MALLET. Logical time in model-driven engineering (habilitation à diriger des recherches). 2010.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *Software Engineering, IEEE Transactions on*, (4):417–426, 1981.

[MC11] Julio L. Medina and Álvaro García Cuesta. From composable design models to schedulability analysis with uml and the uml profile for marte. *SIGBED Rev.*, 8(1):64–68, mar 2011.

[MCBD02] José Merseguer, Javier Campos, Simona Bernardi, and Susanna Donatelli. A compositional semantics for uml state machines aimed at performance evaluation. In *Discrete Event Systems, 2002. Proceedings. Sixth International Workshop on*, pages 295–302. IEEE, 2002.

[McM93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[MDH01] JL Medina, JM Drake, and M González Harbour. Uml-mast: modeling and analysis methodology for real-time systems developed with uml case tools. In *Proceedings of the Fourth International Forum on Design Languages, FDL*, volume 1, 2001.

[Mea55] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mero1] Stephan Merz. Model checking: A tutorial overview. In *Modeling and verification of parallel processes*, pages 3–38. Springer, 2001.

[MF76] P. Merlin and D. Farber. Recoverability of communication protocols–implications of a theoretical study. *Communications, IEEE Transactions on*, 24(9):1036 – 1043, 1976.

[MMo6] Steven Martin and Pascale Minet. Worst case end-to-end response times of flows scheduled with fp/fifo. In *Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on*, pages 54–54. IEEE, 2006.

[Moo56] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[MPFAo6] Frédéric Mallet, M-A Peraldi-Frati, and Charles André. From uml to petri nets for non functional property verification. In *Industrial Embedded Systems, 2006. IES'06. International Symposium on*, pages 1–9. IEEE, 2006.

[MRK+97] Louise E Moser, YS Ramakrishna, George Kutty, P Michael Melliar-Smith, and Laura K Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(1):31–79, 1997.

[MRSo8] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[MT98] Christoph Meinel and Thorsten Theobald. Ordered binary decision diagrams and their significance in computer-aided design of vlsi circuits. 1998.

[Nec11] George Necula. *Proof-carrying code*. Springer, 2011.

[Neu95] Peter G. Neumann. *Computer related risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[NMAo8] Wonhong Nam, P Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Formal Methods in System Design*, 32(3):207–234, 2008.

[OMG01] OMG. Model Driven Architecture - A Technical Perspective. Technical report, July 2001.

[OMG03] OMG. *Common Warehouse Metamodel (CWM)*, March 2003.

[OMG05a] OMG. *UML Profile for Schedulability, Performance, and Time Specification*, 2005.

[OMG05b] OCL OMG. 2.0 specification. *Object Management Group, Final Adopted Specification*, 2005.

[OMG08] OMG. Meta object facility (mof) 2.0 query/view/transformation specification version 1.0, April 2008.

[OMG09] OMG. *UML profile for MARTE: modeling and analysis of real-time embedded systems version 1.0*, 2009.

[OMG11a] OMG. *Meta Object Facility (MOF) Core Specification Version 2.4.1*, August 2011.

[OMG11b] OMG. *MOF 2 XMI Mapping (XMI)*, August 2011.

[OMG11c] OMG. OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1). Technical report, Object Management Group, August 2011.

[OMG13] OMG. *Semantics of a Foundational Subset for Executable UML Models (fUML)*, August 2013.

[ORR+96] Sam Owre, Sreeranga Rajan, John M Rushby, Natarajan Shankar, and Mandayam Srivas. Pvs: Combining specification, proof checking, and model checking. In *Computer Aided Verification*, pages 411–414. Springer, 1996.

[Pau00] Lawrence C Paulson. Isabelle: The next 700 theorem provers. *arXiv preprint cs/9301106*, 2000.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pnu85] Amir Pnueli. *In transition from global to modular temporal reasoning about programs*. Springer, 1985.

[Pri57] A Prior. Time and modality. 1957. *My present modification of the position there stated owes much to PT Geach's criticism in the Cambridge Review*, page 543, 1957.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166. Springer, 1998.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[QVT09] OMG QVT. Qvto specification, 2009.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, November 1997.

[RH80] C.V. Ramamoorthy and G.S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *Software Engineering, IEEE Transactions on*, SE-6(5):440–449, 1980.

[RtPR03] Manos Renieris and teven P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[Rus01] John Rushby. Theorem proving for verification. In *Modeling and verification of parallel processes*, pages 39–57. Springer, 2001.

[Sal85] Arto Salomaa. *Computation and automata*, volume 25. Cambridge University Press, 1985.

[SB96] Robert H Sloan and Ugo Buy. Reduction rules for time petri nets. *Acta Informatica*, 33(7):687–706, 1996.

[SLNM04] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar: a flexible real time scheduling framework. *Ada Lett.*, XXIV(4):1–8, nov 2004.

[SN08] Konrad Slind and Michael Norrish. A brief overview of hol4. In *Theorem Proving in Higher Order Logics*, pages 28–32. Springer, 2008.

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010.

[Spe05] ARINC Specification. 664: Aircraft data network. Technical report, Parts 1, 2, 7. Technical report, Aeronotical Radio Inc., 2002-2005, 2005.

[TMH08] Yann Thierry-Mieg and Lom-Messan Hillah. Uml behavioral consistency checking using instantiable petri nets. *Innovations in Systems and Software Engineering*, 4(3):293–300, 2008.

[Val91] Antti Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1991.

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998.

[VHJG95] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.

[WD09] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WQ09] W Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):573–597, 2009.

[WSQG08] W Eric Wong, Yan Shi, Yu Qi, and Richard Golden. Using an rbf neural network to locate program bugs. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 27–36. IEEE, 2008.

[WWQZ08] W Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. A crosstab-based statistical method for effective fault localization. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 42–51. IEEE, 2008.

[YYSQ10] Nianhua Yang, Huiqun Yu, Hua Sun, and Zhilin Qian. Mapping uml activity diagrams to analyzable petri net models. In *10th International Conference on Quality Software (QSIC'2010)*, pages 369 –372, july 2010.

[ZCP12] Faiez Zalila, Xavier Crégut, and Marc Pantel. Leveraging formal verification tools for dsml users: A process modeling case study. In *ISoLA (2)*, pages 329–343, 2012.

[ZCP13] Faiez Zalila, Xavier Crégut, and Marc Pantel. A transformation-driven approach to automate feedback verification results. In *Model and Data Engineering*, pages 266–277. Springer, 2013.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

[Zim78] H-J Zimmermann. Fuzzy programming and linear programming with several objective functions. *Fuzzy sets and systems*, 1(1):45–55, 1978.

[Zub91] WM Zuberek. Timed petri nets definitions, properties, and applications. *Microelectronics Reliability*, 31(4):627–644, 1991.