Formal verification of the functionnal layer of robotic and autonomous systems

Mohammed Foughali

▶ **To cite this version:**

Mohammed Foughali. Formal verification of the functionnal layer of robotic and autonomous systems. Logic [math.LO]. INSA de Toulouse, 2018. English. NNT : 2018ISAT0033 . tel-02080063v2

HAL Id: tel-02080063

https://laas.hal.science/tel-02080063v2

Submitted on 26 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Fédérale
Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut National des Sciences Appliquées de Toulouse

Présentée et soutenue par

Mohammed FOUGHALI

Le 17 décembre 2018

Vérification Formelle des Modules Fonctionnels de Systèmes Robotiques et Autonomes

Ecole doctorale : **SYSTEMES**

Spécialité : **Robotique et Informatique**

Unité de recherche :
LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes

Thèse dirigée par
Malik GHALLAB et François-Felix INGRAND

Jury

M. Charles PêCHEUR, Rapporteur
M. Herman BRUYNINCKX, Rapporteur
Mme Jeanette CARDOSO, Examinateur
M. Jacques COMBAZ, Examinateur
M. Silvano DAL ZILIO, Examinateur
M. Jacques MALENFANT, Examinateur
M. Malik GHALLAB, Directeur de thèse
M. Félix INGRAND, Co-directeur de thèse

Formal Verification of the Functional Layer of Robotic and Autonomous Systems

Mohammed Foughali

Abstract

The goal of this thesis is to add to the efforts toward the long-sought objective of secure and safe robots with predictable and *a priori* known behavior. For the reasons given above, formal methods are used to model and verify crucial properties, with a focus on the functional level of robotic systems. The approach relies on automatic generation of formal models targeting several frameworks. For this, we give operational semantics to a robotic framework, then several mathematically proven translations are derived from such semantics. These translations are then automatized so any robotic functional layer specification can be translated automatically and promptly to various frameworks/languages. Thus, we provide a mathematically correct mapping from functional components to verifiable models. The obtained models are used to formulate and verify crucial properties on real-world complex robotic and autonomous systems.

This thesis provides also a valuable feedback on the applicability of formal frameworks on real-world, complex systems and experience-based guidelines on the efficient use of formal-model automatic generators. In this context, efficiency relates to, for instance, how to use the different *model checking* tools optimally depending on the properties to verify, what to do when the models do not scale with model checking (e.g. the advantages and drawbacks of *statistical model checking* and *runtime verification* and when to use the former or the latter depending on the type of properties and the order of magnitude of timing constraints).

Keywords: Robotics, computer science, software engineering, formal methods, verification, real-time.

Résumé

Les systèmes robotiques et autonomes ne cessent d'évoluer et deviennent de plus en plus impliqués dans les missions à coût considérable (e.g. exploration de l'espace) et/ou dans les milieux humains (e.g. chirurgie, assistance handicap). Cette implication remet en question les pratiques adoptées par les développeurs et ingénieurs pour donner un certain degré de confiance à ces systèmes. En effet, les simulations et campagnes de tests ne sont plus adaptées à la problématique de sûreté et fiabilité des systèmes robotiques et autonomes compte tenu (i) du caractère sérieux des défaillances éventuelles dans les contextes susmentionnés (un dommage à un robot très coûteux ou plus dramatiquement une atteinte aux vies humaines) et (ii) de la nature non exhaustive de ces techniques (les tests et simulations peuvent toujours passer à côté d'un scénario d'exécution catastrophique.

Les méthodes formelles, bien qu'elles offrent une solution mathématique élégante aux problèmes de sûreté de fonctionnement et de fiabilité, peinent à s'imposer, de leur côté, dans le domaine de la robotique autonome. Cette limitation devient encore plus visible au niveau fonctionnel des robots, i.e. les composants logiciels interagissant directement avec les capteurs et les actionneurs. Elle est due à plusieurs facteurs. D'abord, les composants fonctionnels reflètent un degré de complexité conséquent, ce qui mène souvent à une explosion combinatoire de l'espace d'états atteignables (comme l'exploration se veut exhaustive). Ce problème force les spécialistes soit à se limiter à des applications très simples, soit à recourir à des abstractions qui s'avèrent fréquemment exagérées, ce qui nuit à la véracité des résultats de la vérification (e.g. l'oubli des contraintes temporelles, la non inclusion des spécificités du hardware). En outre, les composants fonctionnels sont décrits à travers des languages et frameworks informels (ROS, GenoM, etc.). Leurs spécifications doivent alors être traduites en des modèles formels avant de pouvoir y appliquer les méthodes formelles associées. Cette opération, nommée formalisation, est souvent pénible, lente, et exposée à des erreurs vu la complexité des comportements que représentent les composants

fonctionnels des robots. La formalisation fait face également à un autre problème également pesant, à savoir le manque de portabilité. Cela se résume au fait que chaque traduction doit être refaite dès qu'un composant change ou évolue, sans parler des nouvelles applications faites de nouveaux composants, ce qui implique un investissement dans le temps aux limites de la rentabilité. A noter que cette thèse ne s'intéresse pas aux composants du haut niveau dits "décisionnels" des systèmes robotiques et autonomes. En effet, ces composants sont souvent basés sur des modèles bien définis, même formels, ce qui facilite leur connexion aux méthodes formelles. Le lecteur intéressé peut trouver dans lalittéature de nombreuses contributions y étant pertinentes. Aux limitations décrites précédemment, s'ajoute le problème de l'indécidabilité vis-à-vis les formalismes et les techniques de vérification.

Par example, les travaux comparant les Réseaux de Petri Temporels "à la , Merlin" et les Automates Temporisés, deux formalismes phares de modélisation des systèmes concurrents, demeurent trop formels pour les communautés autres que celle des méthodes formelles. Il existe néanmoins des travaux qui présentent des techniques qui permettent de bénéficier des deux formalismes, bien qu'elles ne soient (i) appliquées qu'à des exemples académiques classiques, loin de la complexité des composants fonctionnels robotiques et autonomes et (ii) restreintes aux classes des réseaux non-interprétés (pas de possibilité d'avoir des données/variables partagées).

Nous proposons, dans ce travail de recherche, de connecter GenoM3, un framework de développement et déploiement de composants fonctionnels robotiques, à des langages formels et leurs outils de vérification respectifs. Cette connexion se veut automatique pour pallier aux problème de non portabilité, décrit au paragraphe précédent. GenoM3 offre un mécanisme de synthèse automatique pour assurer l'indépendance des composants du middleware. Nous exploitons ce mécanisme pour développer des templates en mesure de traduire n'importe quelle spécification de GenoM3 en langages formels. Ceci passe par une formalisation de GenoM3: une sémantique formelle opérationnelle est donnée au langage. Une traduction à partir de cette

sémantique est réalisée vers des langages formels et prouvée correcte par bisimulation. Nous comparons de différents langages cibles, formalismes et techniques et tirerons les conclusions de cette comparaison. La modélisation se veut aussi, et surtout, efficace. Un modèle correct n’est pas forcément utile. En effet, le passage à l’échelle est particulièrement important.

Cette thèse porte donc sur l'applicabilité des méthodes formelles aux composants fonctionnels des systèmes robotiques et autonomes. Le but est d'aller vers des robots autonomes plus sûrs avec un comportement plus connu et prévisible. Cela passe par la mise en place d'un mécanisme de génération automatique de modèles formels à partir de modules fonctionnels de systèmes robotiques et autonomes. Ces modèles sont exploités pour vérifier des propriétés qualitatives ou temps-réel, souvent critiques pour les systèmes robotiques et autonomes considérés. Parmi ces propriétés, on peut citer, à titre d'exemple, l'ordonnançabilité des tâches périodiques, la réactivité des tâches sporadiques, l'absence d’interblocages, la vivacité conditionnée (un évènement toujours finit par suivre un autre), la vivacité conditionnée bornée (un évènement toujours suit un autre dans un intervalle de temps borné), l'accessibilité (des états “indésirables” ne sont jamais atteints), etc. Parmi les défis majeurs freinant l'atteinte de tels objectifs, on cite notamment:

- Contrairement aux spécifications décisionnelles, les modules fonctionnels sont décrits dans de langages informels. La formalisation est dure, inévidente, et sujette à des erreurs compte tenu des comportements atypiques qui peuvent se présenter à ce niveau. Cette formalisation est aussi non réutilisable (besoin de re-formaliser pour chaque nouvelle application). Il existe une multitude de techniques de vérification et de formalismes mathématiques pour la modélisation. Le choix n'est pas évident, chaque formalisme et chaque technique présentant des avantages et des inconvénients. La complexité des modules fonctionnels (nombre de composants, mécanismes de communication et d'exécution, contraintes temporelles, etc.) mène à des problèmes sérieux de passage à l'échelle (explosion de l'espace d'états atteignables).

- Il existe une déconnexion importante entre les deux communautés (de robotique et de vérification formelle). D'une part, les roboticiens n'ont ni la connaissance ni les moyens (en terme de temps surtout mais aussi de background) de s'investir dans les méthodes formelles, qui sortent de leur domaine. D'autre part, les spécialistes des méthodes formelles restent loin de s'attaquer à des problématiques si complexes faute de connaissances en robotique. Cette thèse tacle la totalité de ces problèmes en proposant une approche de traduction prouvée mathématiquement et automatisée de GenoM vers:
- Fiacre/TINA (model checking)
- UPPAAL (model checking)
- UPPAAL-SMC (statistical model checking)
- BIP/RTD-Finder (SAT solving)
- BIP/Engine (enforcement de propriétés en ligne)

La thèse propose également une analyse du feedback expérimental afin de guider les ingénieurs à exploiter ces méthodes et techniques de vérification efficacement sur les modèles automatiquement générés.

Mots-clés: Robotique, informatique, méthodes formelles, vérification, temps-réel

Contents

Chapter 1

Introduction

The last few decades are characterized by a fast growth in hardware technology, such as smart sensors and on-board electronics. As a result, the capabilities of robotic and autonomous systems have increased, which motivated their deployment over a large spectrum of domains. This large deployment often involves contact with humans or critical missions (*e.g.* home assistants, rescue robots, deep space, self-driving cars, cyber-physical systems). However, robotic and autonomous systems need a software to fulfill their missions.

Software governs the evolutions of robotic and autonomous systems. It implements sensory-motor functions and transforms the inner capabilities of the hardware elements into tangible actions. This major role of software puts it at the heart of missions success, but also makes it a major source of failures. This last statement is supported by a number of incidents that maculate the actual deployment of software autonomy. Among these, we cite the *Uber* self-driving car accident last March in Tempe, Arizona. Elaine Herzberg, a 49-year-old pedestrian was fatally hit by an autonomous Volvo in an incident widely covered by the media. Experts reports seem to agree on the fact that the car sensors did actually detect the victim, but software decided not to swerve, possibly considering the sensor feedback to be a false positive[1]. This is an example that shows that envisaging a wider involvement of robotic and autonomous systems in our daily life requires to achieve a higher level of trust in their software.

We point out that, at this level, we need to distinguish between (1) failure at the specification level of sensory-motor functions and (2) failure of software. The former (1) is due to the functions being faulty at the algorithmic/mathematical level, so even if the software implements them correctly, the system would still fail with regard to what we expect it to do. The latter (2) results from an erroneous implementation of the sensory-motor functions, so even if the algorithmic specification is correct, the system would still fail with regard to what such a specification entails. In this thesis, we focus on the pure software failures (category (2)), even though diagnosing a failure in this category may reveal an algorithmic aberrance (category 1). In the example given above, it is hard to categorize the failure (decision not to swerve) exclusively in category (1) or category (2), although the current investigation course seems to point to the direction of a pure software failure. Indeed, the laser reflection was detected, but the interpretation of it was "wrong" and the reactions were therefore unadapted, with regard to the specification.

[1] https://www.theguardian.com/technology/2018/may/08/ubers-self-driving-car-saw-the-pedestrian-but-didnt-swerve-report

The addressed problem at a glance In the current practice, robotic software trustworthiness relies on testing campaigns, best coding practices, and the choice of sound architecture principles. While such methods are helpful, they unfortunately do not provide guarantees on crucial *properties* such as *schedulability* of tasks, absence of *deadlocks*, *leads to* (an event b always follows an event a in the future), and bounded response (an event b always follows an event a within a bounded amount of time). Such properties often reflect the requirements on the safety and predictability of the system. For instance, a deadlock in a part of the system means that some execution scenario ends up in a state where no further evolution is possible, in that part of the system. Such a scenario is undesirable while an autonomous system is undergoing a critical or dangerous mission, the success of which requires a correct coordination of all the software pieces in the system. Another example is an autonomous system (*e.g.* a drone) taking part in a *hard real-time application*, where all of its tasks need to finish processing before a statically assigned *deadline*. Failing to satisfy this property, known as schedulability, may induce *e.g.* an erroneous servoing that would lead a drone to crash, and possibly injure humans. The bounded response is crucial in *e.g.* self-driving cars, where we want to know that for all possible execution scenarios, the maximum time difference between requesting a brake action and the actual braking is small enough so the vehicle stops before colliding with an obstacle (which might be a pedestrian). Verifying this type of properties is thus necessary to obtain a high level of trust in the robotic software, yet the routinely employed methods fall short of giving the desired answers. Indeed, scenario-based testing for instance, widely used in robotics, is non exhaustive and thus cannot verify a property with a known level of certainty. Consequently, the reliability of robotic software does not rise to the level found in many regulated domains, such as the aeronautic or nuclear industries, where *formal methods* are used to check the most vital parts of systems [Woodcock et al., 2009].

The question that arises is why formal methods are not systematically employed to verify robotic software? There exist many reasons, emerging from the specificities of the robotics domain, such as the unstructured nature of environments, compared to other domains like aeronautics. But, more generally, to answer this question, we need to differentiate the levels of robotic software, mostly viewed as *functional* (tightly coupled with the sensors and actuators) and *decisional* (in charge of *deliberative* functions). In contrast to most of the decisional ones, functional specifications are written in informal languages. Thus, in order to apply formal methods to the functional level, we need first to *formalize* its specifications. The formalization is hard, error-prone and non automatic (it needs to be re-done from scratch for each new application). Additionally, there is a large number of existing formalisms/tools that can be employed in modeling and verification. The mutual advantages and drawbacks of such formalisms/tools depend on the applications/properties to verify and cannot thus be known beforehand. In practice, the high cost of modeling often limits the choice to only one formalism/tool, which makes it impossible to know whether verification might be improved with other formalisms/tools. Moreover, the complexity of the functional level (e.g. number of *components*, timing constraints, communication mechanisms) often leads to scalability issues. Overall, there is a visible gap between the robotic and formal methods communities. On one hand, robotic programmers have neither the knowledge nor the time to invest in applying formal methods to their applications. On the other hand, formal methods specialists are often far from dealing with systems as complex as the robotic ones.

Proposition The goal of this thesis is to add to the efforts toward the long-sought objective of secure and safe robots with predictable and *a priori* known behavior. For the reasons given above, formal methods are used to model and verify crucial properties, with a focus on the functional level of robotic systems. The approach relies on automatic generation of formal models targeting several frameworks. For this, we give operational semantics to a robotic framework, then several mathematically proven translations are derived from such semantics. These translations are then automatized so any robotic functional layer specification can be translated automatically and promptly to various frameworks/languages. Thus, we provide a mathematically correct mapping from functional components to verifiable models. The obtained models are used to formulate and verify crucial properties (see examples above) on real-world complex robotic and autonomous systems.

This thesis provides also a valuable feedback on the applicability of formal frameworks on real-world, complex systems and experience-based guidelines on the efficient use of formal-model automatic generators. In this context, efficiency relates to, for instance, how to use the different *model checking* tools optimally depending on the properties to verify, what to do when the models do not scale with model checking (e.g. the advantages and drawbacks of *statistical model checking* and *runtime verification* and when to use the former or the latter depending on the type of properties and the order of magnitude of timing constraints).

Outline of the chapter The rest of this chapter is organized as follows. First, we overview one of the popular hierarchical architectures in robotic software and review the existing component-based frameworks for the functional level. This helps us to argue in favor of the framework of our choice. Second, the robotic software reliability problem is introduced. We give notable examples on software failures in real-world robots and their direct influence on the deployability of those robots. Formal verification is then overviewed at the decisional level and the difference of its applicability to the different architectural levels is discussed. This motivates our choice to focus on the functional level. Third, we overview the state of the art of formal verification of functional components. We identify the main problems hindering a systematic verification of functional components using formal methods. Finally, we present our contributions as viable solutions to the identified problems and outline the plan of the thesis.

1.1 Software in robotics

Software engineers and developers work permanently on providing robust frameworks to specify and execute robotic applications. These efforts are confronted with a raising complexity of robotic and autonomous systems, which makes software development, use and maintenance costly and challenging. Indeed, the simplest applications nowadays involve several sensors/actuators and thousands of lines of code. Furthermore, autonomous systems are highly heterogeneous. Timing constraints, for instance, differ greatly from a function to another both range-wise (from hundreds of nanoseconds to several seconds) and urgency-wise (*e.g.* hard or weakly hard real-time [Bernat et al., 2001]).

In order to tackle this complexity efficiently, robotic software is often broken according to the role of its parts and the degree of their autonomy and direct involvement with the hardware. The first direction of dissociating these parts is hierarchical, separating those that directly interact with sensors and actuators from the decision-making,

deliberative ones; such a separation produces levels, also known as *layers*. The second direction consists in partitioning a layer into different reusable *components* according to the functionalities they are in charge of.

1.1.1 Layers

Layering separates software pieces with a high level of autonomy from those that process sensor inputs and send outputs to actuators. This hierarchical splitting produces three layers [Gat and Bonnasso, 1998; Alami et al., 1998]:

Decisional layer In charge of high-level deliberative functions pertaining to decision making. Such functions may be *e.g.* planning, acting and learning [Ingrand and Ghallab, 2017] and usually require a certain knowledge of the system and its environment with some abstract representation. The decisional layer outputs high-level plans resulting from applying its computations, often relying on *heuristics*, performed over some data from a lower layer.

Functional layer Tightly coupled to the hardware in charge of perception and action, that is sensors and actuators. It is in charge of control loops that deal with elementary robot actions. It implements functions that manage such low-level actions, including *e.g.* localization, vision and motion planning. The functional layer also feeds the higher layers with inputs when deliberation is needed.

Executive layer Plays the role of a middleman between the highest layer, the decisional, and the lowest one, the functional. It selects, according to the actions received from the decisional layer, the operations to perform at the functional one, with proper parameters and ordering. Subsequently, it returns reports on functions execution to the decisional layer so the latter may properly supervise the plans and select the next actions.

Note that despite its popularity, this is not the only hierarchical architecture that we encounter in robotic systems. Indeed, the specificities of a given application might influence the criteria according to which the system is layered. For instance, some architectures rely on temporal characteristics as a layering principle, such that high-level tasks operate at lower frequencies than low-level ones (example in [Albus, 1995]). A broad view on robotic architectures may be found in [Kortenkamp and Simmons, 2008].

The three-layer architecture presented here does not draw any borders between layers, which results in systems where one layer is dominant. Moreover, the loose definition of the executive layer leaves blurry spots when trying to specify an autonomous system as, often, the borders between it and the other layers are hard to localize (example in [Knight et al., 2000]). Additionally, the executive layer presence prevents often access to the functional layer by the decisional one, which may result in inconsistencies. These problems led to the adoption of a more compact two-layer representation where the executive layer is absorbed by one of the other layers or both (example in [Volpe et al., 2001]). In this thesis, we focus on the functional layer as this will be motivated in Sect. 1.2.

1.1.2 Component-based software

Within each layer, software is still breakable into different units, called *components*, following their functionality. In this section, we focus on the functional layer component-based design as we overview a broad range of its existing frameworks. The focus on the functional layer here pertains to the main goals and contributions of this thesis (Sect. 1.5) and exposes the *middleware dependency* recurrent problem in robotics (Sect. 1.1.2.2).

1.1.2.1 Motivation

Component-based design is particularly convenient for robotic systems, where software, inherently complex, is required to be equally reusable (an overview is given below). Typically, each functional component is in charge of a robotic functionality of which it implements the algorithms. A robotic application is then built by combining a number of components that communicate in order to fulfill the application requirements. Component-based design is therefore powerful due to the possibility to reuse components for different applications and to implement the same component differently according to the application (*e.g.* same functionality but different algorithms), which produces a broad range of systems resulting from existing components. This compositionality is the spirit of widely used robotic frameworks, such as ROS [Quigley et al., 2009] and Orocos [Bruyninckx, 2001]. In this thesis, it is important to rely on a robotic framework that features reusability and compositionality. Indeed, in order to make our work the most accessible to robotic engineers, we need to comply with the current trends in robotics including component-based design. The use of a component-based framework to specify the robotic applications is therefore important.

1.1.2.2 Middleware dependency

Robotic software components are highly dynamic. There are therefore preponderant needs to handle their mutual interaction and their communication with the operating system.This is done by the *Middleware*. In order to meet the components communication and synchronization rigorous requirements, robotic middleware evolve continuously [Kramer and Scheutz, 2007; Elkady and Sobh, 2012]. This important role of middleware makes it tightly coupled to the component-based framework, so it is considered often as a part of the latter. This is the case of ROS (respect. Orocos), providing a communication layer called ROS-Comm (respect. Orocos Real-Time Toolkit RTT[2]). Due to the specificities of each middleware [Mohamed et al., 2008], it is rather common among robotic programmers to design components for a particular implementation. One may even end up with components using different middleware within the same application.

This foggy line between component-based software and middleware questions the reusability of robotic components, which is the main advantage of component-based design (Sect. 1.1.2.1). It is even argued in [Smart, 2007] that the current components-middleware tight coupling practice in robotics constitutes a main speed bump in the path of robotic research as "we spend our time reimplementing known algorithms and techniques, rather than discovering new ones". This problem is known as the *middleware dependency*. In this thesis, it is important to use an approach that efficiently solves this problem. Indeed, this will minimize the cost of verifying different implementations

[2] http://www.orocos.org/node/26

of the components.

1.1.2.3 Overview of existing frameworks

Software and robotic engineers and researchers propose a various range of specification frameworks that are also solutions to the middleware dependency problem. The common factor to the majority of these solutions is mainly attempting to dissociate component-based frameworks from middleware. In many cases, the propositions consist in enhancing general-purpose software paradigms to increase their suitability for robotic applications, including adding layers to enable their connection to the middleware. This explains the striking resemblance between a decent number of proposed approaches (see examples below) and the widely used Unified Modeling Language UML [3].

UML is a general-purpose graphical modeling language, popular among software engineers especially in object-oriented programming [Coad and Nicola, 1993]. In its latest stable release, UML 2.5 features an architectural design with *connectors* for a natural communication between components [Clements et al., 2003], which apparently suits the robotic software needs at the functional layer. Nevertheless, UML diagrams are unable to capture a range of information, inherently important and growingly required in robotic applications, such as timing constraints and thread allocation [Brugali, 2015]. The literature is rich with efforts attempting to bridge the gap between UML and the needs in robotic software, starting with the emergence of the UML profile Modeling and Analysis of Real-Time Embedded Systems (MARTE) [Faugere et al., 2007; Demathieu et al., 2008]. Mainly, MARTE allows annotating architectural elements with real-time features and is therefore suitable for timed analysis, hence the development of automatic generators from MARTE to schedulability analysis tools in [Medina and Cuesta, 2011]. However, MARTE models are still disconnected from a real-world deployment, which explains why the robotic case studies using this UML extension are fictive [Demathieu et al., 2008]. Robotic developers are discouraged to create practical implementations based on these frameworks, due to an insufficient flexibility for specification, reuse and deployment of robotic components. This led to the introduction of other approaches that, while still heavily inspired by UML, are more specific to robotics, which eases the implementations.

Among these approaches, we distinguish RobotML [Dhouib et al., 2012], a UML profile specific to robotic applications. It provides a graphical environment for developing the robotic components, referred to as *systems*, with some automatic generators to middleware (mainly Orocos-RTT). While a number of successfully deployed case studies exist in the frame of the Proteus [4] project, RobotML provides no connection with ROS-Comm, currently the middleware of the most used robotic framework.

Another equally mature approach is the model-driven SmartSoft [Schlegel et al., 2009], where components may communicate through a limited set of patterns typically used in robotics (*e.g.* clients/server and publisher/subscriber). The toolchain comprises a generator for platforms using the CORBA standard [Mowbray and Zahavi, 1995]. Despite fully deployed applications (*e.g.* the collaborative robot butler in [Dennis et al., 2016]), SmartSoft offers no bridging with middleware layers of popular robotic software (such as Orocos-RTT and ROS-Comm).

In contrast to techniques seemingly derived from existing general-purpose lan-

[3] http://www.uml.org

[4] Platform for RObotic modelling and Transformation for End-Users and Scientific communities project, http://www.anr-proteus.fr

guages, some solutions to middleware dependency rely on relatively novel suggestions, either reinventing a toolchain from scratch or proposing a looser definition of components. In [Jang et al., 2010], the Open Platform for Robotic Services OPRoS is presented. It is a hierarchical component-based framework where *atomic components* may be composed into *composite components* that form the robotic application. The framework has also an *executer* to run these components on a target platform. Once more, no attention is given to bridging this framework to existing popular middleware, which reduces the reusability of the components. Moreover, the potentiality of this bridging remains questionable due to the great complexity of the framework design, which questions equally the usability of the framework by robotic engineers. For instance, there are several types of communication *ports* (data, service, event) and several layers wrapped in the application (composer, executer), and the framework enforces some scheduling choices at the design level.

A quite different approach is presented in [Adam et al., 2017]. Instead of developing the application in a precise manner, only its architecture is designed (the components, as empty boxes, and their interactions). Then, a *model-to-model M2M* transformation is performed to *e.g.* remove hierarchies. Finally, a *model-to-target M2T* transformation is realized to obtain an "empty" executable model. The direct advantage of this approach is the possibility to reuse components that are already developed in the target (*e.g.* a ROS component for a given robot functionality) rather than worrying about the component algorithms at the design level. This comes, however, at the obvious expense of a higher cost induced by the two-layer transformation chain, requiring different levels and kinds of knowledge.

The chosen component-based framework in this thesis is GenoM3 [Mallet et al., 2010], due to its following advantageous characteristics. First, besides similarities with UML 2.5 (*e.g.* components and the connection between ports), GenoM3 is well suited (and was developed) for robotic applications. For instance, it allows specifying periodic and aperiodic behaviors and finite-state-machine services (Sect. 2.2.2, Sect. 2.2.3). Second, it adopts a level of specification that is convenient for robotic programmers, as it does not burden them with *e.g.* enforced schedulers. Third, learning GenoM3 is time- and cost-efficient for robotic engineers, as only a basic knowledge in component-based design and robotic software is required. Finally, and not to omit the main issue overviewed in this section, GenoM3 specifications are independent from the implementation (the development of the component is decoupled from the middleware GenoM3 currently supports, more in Sect. 2.3). It provides automatic generation to PocoLibs [5] and ROS-Comm middleware.

1.2 Reliability

1.2.1 Safety issues

The convenience of component-based design, coupled with a loose connection to middleware (Sect. 1.1.2), is quite promising toward a large deployment of robotic applications in our daily life. However, an easy-to-reuse, easy-to-deploy software needs also, more importantly, to be safe. Indeed, serious doubts arise on the safety of robotic software, especially when involved in costly missions (*e.g.* space exploration) and direct contact with humans (*e.g.* home and surgery assistants). These doubts are well justified, as many studies confirm. For instance, the deployment of the museum guide

[5] https://git.openrobots.org/projects/pocolibs

robot RoboX9 is assessed over a period of five months in [Tomatis et al., 2003]. Among the over four thousand failures recorded, the overwhelming majority are software related (96%), including over two hundred deemed "critical". Moreover, the failure of robotic and autonomous systems is likely to have an impact that is less tolerated by the humans than the failures of the humans themselves. For instance, while human-caused road accidents are usually non-news, the injury of a human by a self-driving car makes it easily to the headlines. This philosophical issue is debated in [Shalev-Shwartz et al., 2017].

Still, the lack of assurance that characterizes robotic software today is not caused by a lack of awareness in the community. Robotic components are systematically tested, both on the field and by the intermediary of sophisticated simulators such as Gazebo [Koenig and Howard, 2004] and Morse [Echeverria et al., 2011] (a study on the ability of simulation to reveal bugs is given in [Sotiropoulos et al., 2017]). The problem resides rather in the inability of testing methods to rise to the required level of guarantees. The scenario-based conventional methods of testing may demonstrate a severe inefficiency faced with the complexity of robotic and autonomous systems. For instance, we find in [Pecheur, 2000] a noteworthy, practical example that exposes the non adequacy of scenario-based testing to autonomous missions. It is the case of the Remote Agent Experiment RAX [Nayak et al., 1999], where even a thorough, long-term (over a year) test failed to detect software bugs beforehand. Indeed, RAX had to be stopped only a few hours after assigning it the control of a NASA's Deep Space mission in 1999. This emergency measure was due to the occurrence of a dormant deadlock scenario. The non exhaustive nature of testing prevented shedding the light on this failure as the very scenario that led to the deadlock was never explored. The urban challenge organized by the Defence Advanced Research Projects Agency DARPA provides another valuable example in 2007. It involves the autonomous vehicle *Alice*, developed at the California Institute of Technology. Alice was a successful participant of earlier versions of the challenge, *e.g.* in 2005 [Murray et al., 2005], and underwent a strict set of tests using simulators (thousands of hours) and on the field (over 450km). Unexpectedly, Alice was disqualified from the 2007 competition due to a serious software bug related to the implementation of handling intersections when nearby objects are detected, a scenario that never occurred during the testing campaigns [Kress-Gazit et al., 2011]. These cases are merely examples among many that expose the non suitability of scenario-based testing, either on the field or using simulators, for verifying complex robotic and autonomous systems. This emphasizes the urgent need for more accurate techniques, that are up to the challenge of a safer and larger involvement of autonomous systems in various domains.

1.2.2 Formal verification, a promising alternative

These observations motivate the attempts, since a few decades, to support the robotic software with more sophisticated, mathematically founded methods. That is, to gradually replace scenario-based testing with *formal validation and verification (V&V)*, widely adopted in other domains such as aeronautics and nuclear industries [Bowen and Stavridou, 1993; Andersen and Romanski, 2011]. Contrary to testing, formal V&V uses mathematically based analysis methods, *i.e. formal methods* [Bjørner and Havelund, 2014] to assert whether a *property* is satisfied by a system. We distinguish verification from validation as follows. Validation tries to answer the question "are we doing the right thing?", that is, whether the specification coincides with the functional and non-functional requirements. Verification, on the other hand, checks the

correctness of the implementation with regard to the specification in order to answer the question "are we doing it (what we want to do) right?". In this thesis, we focus on formal verification as we verify the properties desired by the robotic programmer on deployed systems.

Due to its mathematical foundation, formal verification provides an elegant and sound solution. Among existing formal verification techniques, we briefly describe the following (more in Chapt. 3):

- Model Checking [Clarke et al., 1999]: relies on automata-theoretic methods to check the validity of properties expressed as *temporal-logic formulae* (*e.g.* LTL [Vardi and Wolper, 1986] and CTL [Emerson and Srinivasan, 1988]) on the model of a system. Model checking is automatic and exhaustive but can be unfeasible because of the combinatory explosion when exploring all the possible states. Statistical Model Checking SMC [Legay et al., 2010] is sometimes evoked to reduce the cost of model checking. Among SMC techniques, we mention simulating the system for finitely many executions in order to evaluate the properties with some probability.

- Deductive Verification: consists in deducing, from a specification of the system, possibly annotated with *e.g.* pre- and post-conditions, a set of statements to prove in order to check the validity of the system with regard to its specification. The correctness of these statements is verified using *e.g.* classical theorem proving [Green, 1981] or Hoare logic [Hoare, 1969]. Deductive techniques can reason on infinite systems due to the power of induction but deriving the proof requires both expertise and costly efforts, which makes them mainly suitable for small programs rather than whole systems [Pecheur, 2000].

- Runtime Verification [Leucker and Schallhart, 2009]: the property is checked or enforced at runtime. Runtime verification poses the problem of the scope of its suitability. Indeed, it is quite hard to decide whether one can rely on a monitor at runtime and how to deal with the property violation remains an open issue. The enforcement of properties at runtime is a widely explored solution [Ligatti and Reddy, 2010; Gabel and Su, 2010].

1.2.2.1 Formal verification at the decisional layer

At the decisional layer, the use of formal methods to reason about software becomes more and more common. Indeed, decisional-layer models are often formal. For instance, most of the planning existing models (*e.g.* PDDL [McDermott et al., 1998] and ANML [Smith et al., 2008]) are formally defined with complete semantics. This alleviates the task of applying formal methods to decisional components as their *formalization*, the most time-consuming and error-prone step in verification [Pecheur, 2000], is not needed. This simplification may be one of the reasons why we find a large corpus of quality works in the literature that apply formal methods to the decisional layer. Some examples of these works are given in the next paragraph. It is however worth emphasizing that the *learning* functions are the exception to the convenient connection between decisional components and formal methods. Indeed, despite some verification works on mathematically well-founded models such as neural networks (see [Huang et al., 2017; Katz et al., 2017] for latest results), applying formal methods to learning algorithms is still a major issue (*e.g.* formally specifying such systems is an open challenge [Seshia et al., 2016]).

In [Hähnel et al., 1998], the authors propose GOLEX, a safe and robust executer/-monitor for the formal acting language GOLOG [Levesque et al., 1997], based on the situation calculus [McCarthy, 1968]. A real-world application shows the capabilities of GOLEX to successfully monitor a mobile robot in unstructured environments. The approach proves also to be efficient in other applications, such as a tour-guide in a museum of Bonn given by the mobile robot RHINO. Bounded response properties are enforced online in a coffee delivery application. If the time bound to serve the next customer cannot be respected, GOLEX ignores the current sub-plan and produces a new plan after it removes the next customer from the waiting list.

Symbolic model checking [Clarke et al., 1996] is used in [Cimatti et al., 2004] to achieve *Conformant Planning*, that is finding a sequence of actions to achieve a goal even in the presence of uncertainty and non-determinism. The formal model of planning *domains* is encoded as Binary Decision Diagrams BDDs [Bryant, 1992] and a Conformant Planning algorithm is consequently developed. The efficiency of the approach is shown through experiments with a number of planning domains. One particularity of this work is the fact that model checking is used as a reliable technique for planning and not for mere verification, which is rendered possible thanks to the formal nature of the underlying planning model.

The formal model of the temporal planner IxTeT [Abdeddaim et al., 2007] is translated into UPPAAL-TIGA [Behrmann et al., 2007] timed-game-automata-based models. The authors evoke the direct advantage of this translation, allowing the verification of the obtained models with UPPAAL-TIGA.

In [Pecheur and Simmons, 2000], autonomous controllers based on Livingstone [Williams and Nayak, 1996], a NASA model-based health monitoring system, are considered. Livingstone specifications are automatically translated into SMV [Burch et al., 1992], a symbolic model checker. The translation is applied *e.g.* to the Livingstone model for the In-Situ Propellant Production ISPP and important properties, such as recoverability from failures, are verified. Both SMV and Livingstone relied on synchronous models, which reduced the difficulty of the translation into "the discrepancies in variable naming conventions between the Lisp-like syntax of Livingstone and the Pascal-like syntax of SMV" [Pecheur, 2000].

1.2.2.2 High-level abstractions

A popular domain for formal verification of high-level robotic software is the *controller synthesis*. From high-level models of the robot and a set of desired properties, both expressed in Linear-time Temporal Logic LTL, a high-level, reactive controller that guarantees such properties is synthesized. This is the driving idea of *e.g.* [Kress-Gazit et al., 2008; Raman et al., 2013].

There are also several works involving cooperating robots or human-robot interactions. For instance, in [Stocker et al., 2012], a multi-agent model involving a robotic assistant, a human carer, a person and an intelligent house is developed in Brahms [Sierhuis and Clancey, 2002]. The models are translated to the SPIN model checker [Holzmann, 1997] and high-level properties such as the bounded response property "if the person requests food then the robot will eventually deliver it within an hour" are verified.

Another example is given in [Gjondrekaj et al., 2012] where the high-level behavior of three robots cooperating to transport an object to a goal zone is modeled in the formal agent-based language KLAIM [De Nicola et al., 1998]. The probability of reaching the goal without collision by one of the robots is then estimated.

Other works abstract the robot behaviors to a its high-level model (the functional layer is considered correct) in order to verify relevant properties in uncertain environments. This is the case of *e.g.* [Aniculaesei et al., 2016] where the *passive safety property* (that is, no collision occurs while the robot is moving [Macek et al., 2008]) is verified using the state-of-the-art model checker UPPAAL (Sect. 3.4).

1.2.2.3 Issues at the functional layer

Contrary to most of the decisional and high-level specifications, functional components are neither written in formal languages nor amenable to heavy abstractions. Indeed, popular component-based frameworks nowadays such as ROS (Sect. 1.1.2) are not defined formally. Furthermore, abstractions at this level may quickly lead to models that do not reflect the real behavior induced by the underlying components (*e.g.* ignoring some timing constraints or interleavings). Therefore, the formalization, inevitable at this level, is particularly challenging and costly and the formalized models are not guaranteed to scale. This explains the fact that the level of integration of formal methods at the functional layer is severely behind the actual needs, as will be explained in the next section.

1.3 Formal verification of functional components

In this section, we overview the state of the art of formal verification at the functional layer. We categorize the contributions into three main verification approaches and give examples in each category.

Deductive verification In [Täubig et al., 2012], the authors report on their experiences in verifying the implementation of a collision avoidance algorithm. Each function is annotated with pre- and post-conditions as well as a *memory layout* and a *modification frame* that limits the effects of the function on memory. Then, it is checked if whenever a function is called such that the call satisfies the pre-conditions and memory layout, the function will terminate, and the state corresponding to the termination satisfies the post-conditions and the modification frame. Important properties like the correct implementation of the braking model are verified using a combination of pen-and-paper and computer-aided (using ISABELLE/HOL [Nipkow et al., 2002]) proofs. The work resulted in certification for use of the algorithm implementation up to SIL3 of IEC 61508-3. The approach requires a heavy human intervention and a very good knowledge of the proof systems and the tools, not to mention the reverse engineering of the algorithms as to properly write the pre-/post-conditions and proof systems.

The authors of [Kouskoulas et al., 2013] verify a control function of a surgical robot using the KeYmaeraD theorem prover for differential-dynamic logic [Platzer, 2008]. The (safety) property of interest is that for all configurations, all possible uses of the robot and at any time, “if the surgeon starts the tool at a safe place, the tool remains in a safe place”. The dimensions of a “safe place” area are computed so that the patient is not harmed when the tool is within that area. This property verifies thus that the “free movement” of the surgeon hand happens always within a safe area, not in direct contact with the patient. Counterexamples are generated and the function is proven thus unsafe. A formally proven safe alternative is proposed. The major part of the work is manual (*e.g.* modeling the system in differential-dynamic logic) and a profound knowledge of both continuous models and proofs in differential logic is needed.

The correctness of a mutual exclusion function over shared memory is analyzed in [Kazanzides et al., 2012]. The function is implemented using the *cisst* software package [Kapoor et al., 2006], a collection of component-based libraries for robotic surgical systems, linked with ROS-Comm. The function considers a circular buffer with an array of state vectors. The authors rely on the History for Local Rely/Guarantee HLRG logic [Fu et al., 2010] to develop paper-and-pen inference rules that led to the detection and fixing of a data corruption bug. The proofs are not automatized and the function verified is very specific.

In [Meng et al., 2015], the correctness of ROSGen, a code generator for ROS components, is verified. The proof assistant Coq [Huet et al., 1997] is used interactively with the user to reason on the proof systems. The *Data Delivery* property, that is the data sent by a sensor is correctly handled by the controller and delivered to the actuator(s), is also proven correct for any targeted platform. In order to guarantee the correctness of ROSGen, a formal sub-version of ROS called *ROS nodes* is proposed, but no proofs are given on its correctness with regards to ROS. Moreover, the approach is hard to generalize and the verification of *e.g.* timed properties is unfeasible.

Overall, as said in the last section, deductive verification is costly and requires considerable human intervention. Furthermore, robotic programmers do not have the required knowledge and expertise to efficiently use theorem provers and proof assistants. Also, the type of properties is restricted such that *e.g.* timed properties (like bounded response) cannot be verified. This explains why the works applying theorem proving to robotics are often done by formal methods experts and mostly focus on a small piece of the specification (a function) rather than the system as a whole.

Model checking The synchronous language ESTEREL [Boussinot and de Simone, 1991] (see [Benveniste and Berry, 1991] for synchronous languages) is used in [Simon et al., 2006] to verify important properties ranging from safety to liveness. The Orccad environment [Simon et al., 1997] is used to build KeepStable, a set of stabilisation procedures of an underwater vehicle. Procedures are hierarchically built from tasks, whose coordination is automatically translated into ESTEREL. The safety property consists in a correct handling of *exceptions*, whereas liveness corresponds to a procedure eventually reaching its goal. Other properties related to the conformance between the procedure requirements and behavior are checked visually. The authors invoke the threat of combinatory explosion for larger applications, which would render the "visual" verification impossible. ESTEREL is used in other model-checking-based verification works such as [Sowmya et al., 2002; Kim and Kang, 2005], where the formalization is manual as robotic specifications are either translated by hand to, or hard-coded in ESTEREL.

RoboChart [Miyazawa et al., 2016] is used in several verification efforts such as [Miyazawa et al., 2017]. RoboChart models are automatically translated into Communicating Sequential Processes (CSP) [Roscoe, 2010] in order to verify behavioral and timed properties using the FDR model checker [Gibson-Robinson et al., 2014]. RoboChart is, however, not a robotic framework (its models are not executable on robotic platforms). That is, each robotic application, initially specified in a robotic framework, needs to be modeled first in RoboChart before it can be translated into CSP.

Model checking techniques are also applied to an Autonomous Underwater Vehicle AUV in [Molnar and Veres, 2009]. A series of manual transformations is performed to bridge the robotic specification with the multi-agent model checker MCMAS [Lo-

muscio et al., 2009]. Collision-avoidance properties are checked. The approach is tedious and requires many transformations which reduces its reproducibility and raises the risks of errors.

The PRISM probabilistic model checker [Kwiatkowska et al., 2011] is used in [Hazim et al., 2016] to verify bounded response properties. A case study involving a ground autonomous vehicle is given, where PRISM estimates the probability of finding an object by the vehicle in a bounded amount of time. Despite an attempt to formalize ROS graphs, no operational semantics is given which makes the formalization both manual and ad-hoc.

Another attempt to formalize ROS components is developed in [Halder et al., 2017] where UPPAAL is used to verify buffer-related properties (no overflow). ROS components are not formalized and only the message passing part (publisher/subscriber) is modeled, manually. Furthermore, there is no attempt to verify bounded response properties, crucial and challenging in message-sending contexts (*e.g.* a message will be always delivered within a known bounded amount of time).

In [Gobillot et al., 2014], specifications written in the Modeling Autonomous VEhicles framework MAUVE are verified. Although MAUVE is oriented toward schedulability analysis[6], the authors evoke the verification of behavioral properties with the model checker TINA (Chapt. 3) after a translation into the RT-Fiacre formal language [Abid and Dal Zilio, 2010]. No further details are given on the translation or the verification results.

Globally, despite the fact that model checking is automatic, its application to robotics is no less problematic than that of deductive verification. Indeed, the formalization of robotic specifications, written in non-formal languages, is quite challenging and error prone and needs a kind of knowledge that is often out of the competence scope of robotic programmers. Furthermore, works on model checking in robotics suffer from the state-space explosion problem (Sect. 1.2) and alternatives are rarely proposed.

Runtime verification The Java PathExplorer tool is presented in [Havelund and Rosu, 2001]. It allows checking, at runtime, the system against temporal logic formulae and classically undesired properties in concurrent execution such as deadlocks. A case study with the NASA robot Rover K9 is presented. The developed monitors are in Java while the logic engine (to check the properties) is in Maude [Clavel et al., 2002], which causes a perceptible slowing down of the original programs by an order of magnitude. Moreover, the approach is not generalized to component-based robotic frameworks.

The Request and Resource Checker R^2C is proposed in [Py and Ingrand, 2004a]. It is an execution controller plugged in the executive layer to prevent faulty behaviors. The latter are described as deliberative commands that could lead to inconsistencies at the functional layer. For this, R^2C has a set of constraints against which it continuously checks the global state of the system and prevents it from transiting into a faulty state. R^2C is successfully deployed on an All-Terrain Robotic Vehicle (ATRV) where it correctly reacts to injected faults by rejecting services requests or terminating running services. It also helped to locate a bug at the decisional level. The approach is solid and automated but does not support the enforcement of timed properties due to the untimed (yet temporal) nature of its underlying formal model (*e.g.* consider a state faulty if some function is waiting for resources for more than some amount of time).

BIP [Basu et al., 2011], a modeling and verification framework based on automata,

[6] Hence its connection with the Orocos-RTT middleware.

is used in the joint verification effort presented in [Abdellatif et al., 2012]. The functional components, written in GenoM (version 2), of an outdoor robot with two navigation modes, are modeled in BIP. Safety constraints, such as "the robot must not communicate and move at the same time" are automatically translated from logical formulae into BIP then added to the model. The latter is run within the BIP-Engine on DALA, an iRobot ATRV, and the constraints are consequently enforced at runtime. Due to the untimed nature of BIP and the lack of some time information (*e.g.* execution times of code) back then, only periods are considered through logical ticks. Furthermore, it is not possible to verify the soundness of the translation from GenoM to BIP due to the absence of operational semantics of the former.

In [Huang et al., 2014], the authors present ROSRV, a runtime verification environment for ROS-based robotic systems. A monitoring layer is added on top of the components to intercept the different messages and commands. The generated monitors are successfully implemented on a simulated LandShark military robot. The monitors restrict the execution to scenarios satisfying security and safety properties. The approach is not generalized to other robotic frameworks. Furthermore, it is hard to verify the generated monitors due to the absence of a formal model of the ROS components.

Performance Level Profiles PLPs are proposed in [Brafman et al., 2016] to describe the components desired performance in a robotic application. The approach helped to reveal a bug in the path planning component of a Compact Track Loader CTL. PLPs are defined semi-formally and their development is quite costly. Indeed, only the generation of their objects is automatic, since the user needs to fill variable updaters and condition validation functions. Additionally, there is no support for timed properties. PLPs require also information, such as expected execution time and runtime distribution, the gathering of which is challenging and not covered by the contribution.

In sum, works on runtime verification (through monitoring or enforcement) in robotics face generalizability issues and their automation is limited. Moreover, these methods are usually complementary to offline verification (*e.g.* via model checking and/or theorem proving) as it is often risky to deploy unverified components, even when a monitoring layer is added.

1.4 Identifying the problems

Clearly, the application of formal verification to functional components remains an open challenge. We try thus to define the major problems causing this, which helps us draw the main axes of our contributions. All the works cited in Sect. 1.3 suffer of at least one of these problems.

Feasibility and accuracy of formal modeling One inevitable step toward the formal verification of functional components is their formal modeling. As shown throughout this chapter, this phase is particularly difficult and error prone, which differentiates the applicability of formal methods to the functional layer as opposed to the decisional one, where specifications are already formal (Sect. 1.2). This is due, mainly, to the fact that component-based frameworks in robotics are not formal (Sect. 1.1.2). This makes the derived formal models also questionable, in terms of their correctness vis-à-vis their robotic counterparts. We refer to this problem as *Problem 1*. The proposed solution of using formal languages directly to encode robotic specifications (such as ESTEREL in [Kim and Kang, 2005]) is not suitable for the robotic community, rather familiar with robotics specific frameworks (Sect. 1.1.2). We need thus to strengthen

these frameworks with clear semantics to facilitate the formal modeling of their specifications.

Automation of formal modeling Another problem with formal modeling is the lack of automation. That is, the modeling is often application dependent (one needs to go through this tedious phase again for each new application). This problem, that we refer to as *Problem 2*, is partially induced by the lack of semantics in robotic component-based frameworks (*Problem 1*). We need an approach for a full automatization of formal modeling of functional components.

Scalability The threat of combinatory explosion leads to heavy abstractions that often reduce the coverability of the formal model. For instance, the formal models in [Desai et al., 2017] represent only some execution scenarios from their underlying robotic specifications, which made the authors undergo a tedious combination between model checking and runtime verification. Other abstractions consist in ignoring timing constraints, which is no longer acceptable in today's systems real-time requirements[7]. This problem, referred to as *Problem 3*, restricts the majority of works to simple applications that are often not deployed on real robots. We need to face the real complexity by applying formal methods to real-world robots, and propose formal solutions in case of scalability issues.

Finding the right method/formalism/tool This is one of the major, yet less treated, problems of formal verification of real-world systems. The multitude of available tools/formalisms/techniques often overwhelm engineers, as most of their mutual advantages and drawbacks depend on the applications/properties to verify and cannot thus be known beforehand. For instance, one tool may perform better than another for a liveness property, while it is the other way around for a reachability property. The choice of the most suitable formalism and associated verification technique is therefore not obvious. The literature contains formal works that compare the expressiveness of some formalisms, *e.g.* [Bérard et al., 2005; Berthomieu et al., 2006]. There is, however, an important disconnection between such works and the real-world, complex applications proposed by the robotics community. This gap widens as formal methods are out of a roboticist field of expertise. Some efforts propose general-rule translations between prominent formalisms, such as time Petri nets (Sect. 3.3.1) and timed automata (Sect. 3.4.1) in [Berard et al., 2013]. These translations are however structural as they do not support extensions with e.g. shared variables, crucial to the modeling convenience of complex robotic systems and their connection to state-of-the-art verification tools. This problem, that we refer to as *Problem 4*, is exacerbated by the fact that for each formalism, various tools are proposed. Even when making their choice, practitioners have to face the problem of opting for a tool almost blindly. *Problem 4* is further worsened by the lack of automation (*Problem 2*), since the high cost of modeling limits the choice often to only one formalism/tool, which makes it impossible to know whether verification might be improved with other formalisms/tools. Valuable examples on the painful experience of engineers exploring the use of verification frameworks for embedded software are given in [Todorov et al., 2018]. *Problem 4* may have great consequences on both the feasability of the modeling (*Problem 1*) and scalability of the models (*Problem 3*), and must be tackled by proposing clear guidelines

[7] We note that, seen at a "mission" level, time is not necessarily crucial as compared to fulfilling the mission correctly. Still, it is at the functional level, on which we focus here.

to robotic programmers based on real-world experiences. Overall, there is a large gap between the robotics and formal methods communities, and further efforts are needed to narrow it.

Obviously, this list of problems is not exhaustive, but represents the issues the work of this thesis tries to overcome. Indeed, one may define other equally important challenges. For instance, popular approaches nowadays rely on compromises between execution time and quality of code (see examples for stereo vision algorithms in [Veksler, 2003; Yu et al., 2010]), which cannot be considered by models where timing constraints are known beforehand. Another example is guaranteeing the robustness of the behavior, defined at the software level, vis-à-vis open environments, which cannot be determined at the functional layer. Therefore, this kind of problems is out of the focus of this thesis but the work presented here constitutes an important step toward solving them.

1.5 Contributions

At this stage, we have presented some of the main formal verification efforts in robotics. Our focus on the related work applying formal methods to the functional level (Sect. 1.3) is justified, at the end of Sect. 1.2, by highlighting the difference with the decisional layer in this regard. Analyzing the state of the art and current practice helped us clearly define a set of problems with which formal verification of functional components is confronted. In this section, we describe our contributions as proposed remedies to the identified problems in a structured manner.

Contribution 1 We tackle *Problem 1* by proposing formal semantics for GenoM3, our chosen robotic component-based framework for this thesis. Components have thus formal definitions and their operational semantics is developed formally (Chapt. 4) in a suitable formalism (Chapt. 3).

Contribution 2 We tackle *Problem 2* by developing automatic generators (*aka* templates) to state-of-the-art formal languages and verification tools (Chapt. 5, Chapt. 6), namely Fiacre/TINA (Sect. 3.3), UPPAAL (Sect. 3.4), UPPAAL-SMC (Sect. 3.5) and BIP (Sect. 3.6). The output of these templates is proven faithful to their input as GenoM3 components (Chapt. 5), such a proof being feasible thanks to *Contribution 1*. *Contribution 2* allows thus the automatic generation of any GenoM3 specification into a number of formal targets with no effort from the robotic engineer.

Contribution 3 We tackle *Problem 3* by (i) striving to avoid all non-realistic abstractions, *e.g.* all timing constraints are taken into account (Chapt. 5, Chapt. 6), including code Worst Case Execution times and tasks periods and (ii) applying our approach to real-world application, deployed on actual robots (Sect. 2.4). We propose templates for formal statistical (UPPAAL-SMC) and runtime (BIP) models to cope with scalability issues, when necessary.

Contribution 4 We tackle *Problem 4* by giving experience-based advices on when to use which tool according to the properties to verify (Chapt. 7). We also assist robotic programmers on which method to use (UPPAAL-SMC or BIP) when exhaustive methods do not scale.

It is worth to point out that the focus on the functional layer does not reduce in any case the importance of verifying the decisional layer components. On the contrary, it contributes to the verification of a robotic system as a whole. For instance, the environment in which a robotic system evolves needs to be taken into consideration, and this is practically done at higher abstraction levels (Sect. 1.2.2.1, Sect.1.2.2.2). Consequently, more general properties such as the success of a robotic "mission", or the data consistency in multi-robot collaboration, can be reasoned upon. Similarly, the results of the verification in this thesis are to be complemented with those of verifying the algorithms using *e.g.* theorem proving techniques, which is also out of the scope of this thesis. Such verification will allow to conclude on properties equally important as the ones presented in this thesis, such as the robustness of algorithms against disturbances (*e.g.* sensor noise).

1.6 Outline

The remainder of this thesis is organized as follows:

- Robotic framework and case studies (Chapt. 2):
  We present our chosen robotic framework GenoM3: the requirements leading to its design, the implementation of its components and their behavior. We show then two real-world case studies deployed using this framework.

- Semantics formalism and formal frameworks (Chapt. 3):
  We present timed transition systems TTS, our chosen formalism for giving operational semantics to GenoM3. We give the semantics of this formalism and examples on why it is suitable for formalizing GenoM3 components. Then, we present the frameworks targeted by our translations (Fiacre/TINA, UPPAAL, UPPAAL-SMC and BIP) and their underlying formalisms (time Petri nets, timed automata and some of their flavors).

- Formalization of GenoM3 components (Chapt. 4):
  GenoM3 components are formalized as TTS systems. We develop the rules of deducing the TTS from the specification of each of GenoM3 entities within a component. At the end of the chapter, we have an unambiguous operational semantics of GenoM3 translatable to other formal targets.

- Correct translation of GenoM3 semantics (Chapt. 5):
  The semantics is translated to timed automata. We give general rules on how to translate GenoM3 semantics to timed automata and outline the difficulties of the process. We prove using bisimulation that the TTS semantics and their timed automata translation are equivalent.

- Mapping to formal frameworks (Chapt. 6):
  We use the TTS semantics and the timed automata translation to map GenoM3 components into the targeted frameworks in a generic way. We show an example of the mapping using a real-world component. Finally, examples on the process of automatizing the mapping are given.

- Verification results (Chapt. 7):
  We automatically generate formal models for our real-world case studies. We use the generated models to verify crucial properties. We show how the different

models can be used complementarily and optimally according to the application, the properties to verify, the size of the model and its timing constraints.

- Conclusion: We conclude by summarizing the advances to the state of the art provided by this thesis and drawing the major axes of future work.

Chapter 2

GenoM3

2.1 Introduction

In this chapter, we present the GenoM3 framework. We first introduce the different architectural elements of a GenoM3 component, the building unit of robotic applications specified and verified in this thesis. Then, we go through the automatic generation feature of GenoM3, the template mechanism, and show how it is a key aspect for developing a reproducible, automated approach to formally verify robotic applications. Finally, we present the two real-world robotic specifications used as case studies in this thesis, namely the quadcopter flight controller and the Osmosis terrestrial navigation showcase.

2.2 Overview

GenoM3 [Mallet et al., 2010] is a tool to specify and implement robotic functional components. The LAAS architecture [Ingrand et al., 2007] proposes a modular approach where each functional component acts as a "server" in charge of a given functionality. The latter may range from simple low-level driver control (e.g. the velocity control of the propellers of a drone, camera, etc) to more integrated computations (e.g. Simultaneous Localization And Mapping (SLAM), Potential-Field navigation, Rapidly-exploring Random Tree (RRT) motion planning, etc.).

2.2.1 Requirements

We consider that a typical component is a *program* which needs to handle and manage the following aspects:

Inputs and Outputs : a component interacts with external clients and other components. For the former, the control flow, it must handle *requests* from client(s) and send back *reports* to the client which issued the request, to act on the result. For the latter, the data flow, it must provide a mechanism to share data with other components and read data from other components. Data flow and control flow are semantically different and correspond to two different ways a component can interact with, respectively, other components and external clients.

Algorithms : the core algorithms needed to implement the functionality the component is in charge of must be appropriately organized within threads as to preserve the reactivity of the component and the schedulability of the various possibly concurrent algorithms. A component may have just one service to provide, but most of the time, there are a number of such services associated to the considered robotic functionality. The way algorithms are specified and organized in a component is a tradeoff. One can let the programmer organize their code with no design requirements or provide structure guidelines that must be followed. The latter case enables automated verification of properties on the code, given that the set of guidelines is well known. However, these organization rules must remain simple and easily understandable for robotic programmers.

Data sharing : the various algorithms, possibly concurrent, running in the component, may have to share data that represent the internal state of the component. These data need to be handled correctly respecting *e.g.* mutual exclusion conditions.

2.2.2 Implementation

To achieve such requirements of a functional component, we propose to organize each one along the structure shown in Fig. 2.1. The implementation choices will be explained while presented.

Specifying components in GenoM3 is the programmer's design choice. Thus, there are a number of considerations, depending on various factors such as hardware constraints and algorithms complexity, that they have to take into account. Here, we describe in more details the different elements of a GenoM3 component, how they interact, and how they are specified, in a generic manner. That is, the description given here is not specific to any middleware or component.

To ease the comprehension of the different elements, a support example is given in listing 2.1. It shows the *dotgen* (extension *.gen*) specification of a simple GenoM3 component called DEMO developed for illustration purposes. DEMO is a simple one-dimensional motion component of a mobile robot. Its main functionalities are to move the mobile for a relative distance within the interval $[-1, 1]$, to monitor its position, to read its speed, and to change it. The elements in charge of these operations are given within the description of the component constituents hereafter.

Control Task : A component always has a *control task* that manages the control flow by processing *requests* and sending *reports* (from/to external clients). The control task must be highly reactive and is only assigned quick computations. It also manages interruption and activation of longer computations (see more in Sect. 2.2.3). The control task is implicitly comprised within a component and the user does not need to (and should not) specify it, hence its absence from listing 2.1.

Execution Task(s) : Aside from the *control task*, one may need one or more *execution tasks*, aperiodic or periodic, in charge of longer computations. The component DEMO has one execution task called motion (periodic at 400 ms, lines 17-19).

Services : The core algorithms needed to implement the functionality of the component are encapsulated within *services*. Each *service* is associated to a *request* (with the same name). One may also define a *permanent service* (running without being requested) attached to each *execution task*. In the DEMO component, services are *MoveDistance* (move the mobile for a relative distance within $[-1, 1]$, lines 34-45),

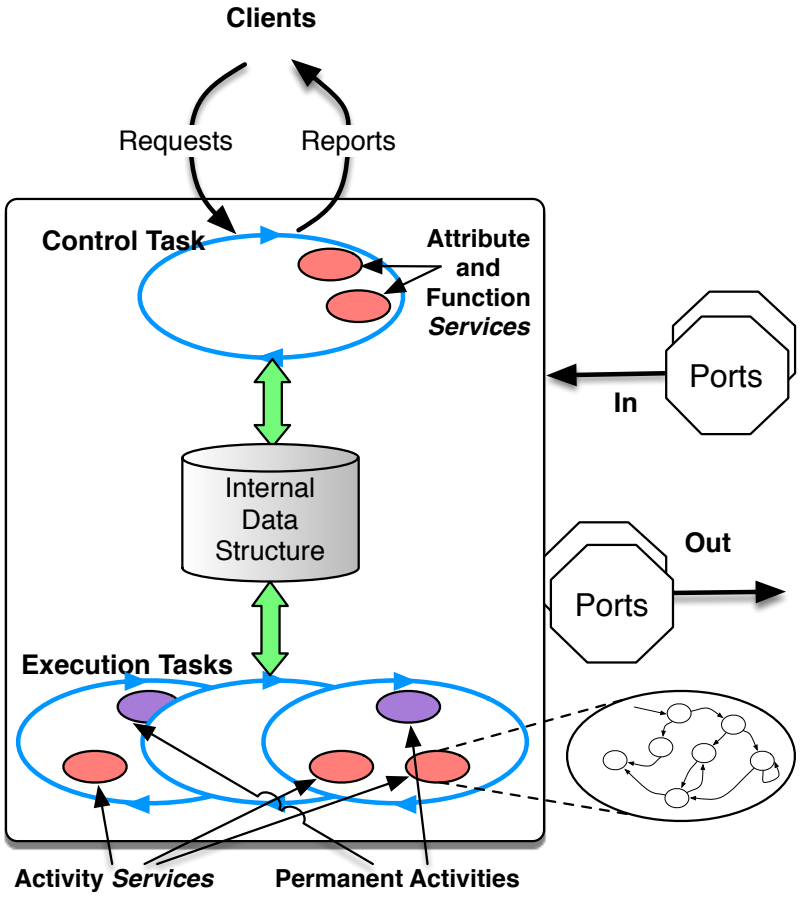


Figure 2.1: A generic GenoM3 component.

Monitor (monitor the position, lines 46-53), *GetSpeed* (get the current speed, lines 26-27), *SetSpeed* (change the current speed, lines 22-25), *Finish* (stop moving, lines 29-32), and the permanent service of motion (initialization, line 19).

IDS : A local *internal data structure* is provided for all the services to share parameters, computed values or state variables of the component. It is appropriately accessed (i.e. with proper locking) by the services when they need to read or write one or more of its fields (lines 4-7). For instance, the arguments of *GetSpeed* specify that it reads the current speed from the IDS (line 26).

Ports : They specify the shared data and the access direction to them (read "in" or write "out"), the component needs or produces from/for other components. The component DEMO provides one port that it writes (*out* mode, line 10).

Exceptions : One may specify *exceptions*, which can be returned by services to *report* on execution errors (lines 13-14).

```
/* ---- component declaration ---- */
component demo {
/* ---- Data structures and IDS ---- */
ids {
       demo::state state;  /* Current state */
       demo::speed speedRef;  /* Speed reference */
       double posRef;};

/* ports declaration: direction type name */
port out demo::state Mobile;

/* exception declaration */
exception TOO_FAR_AWAY {double overshoot;};
exception INVALID_SPEED;

/* execution tasks declaration */
task motion {
       period 400 ms;
       codel <start> InitDemoSDI(out ::ids, port out Mobile) yield ether;};
/* services declaration */
/* atributes */
attribute SetSpeed(in speedRef :"Mobile speed") {
       doc  "To change speed";
       validate controlSpeed (local in speedRef);
       throw INVALID_SPEED;};
attribute GetSpeed(out speedRef = :"Mobile speed") {
       doc  "To get current speed value";};
/* functions */
function Finish() {
       doc  "Stops motion and interrupts all motion requests";
       codel StopMotion();
       interrupts MoveDistance;};
/* activities */
activity MoveDistance(in double distRef :"Distance in m") {
       doc  "Move of the given distance";
       validate controlDistance(in distRef, in state.position);
       codel <start> mdStartEngine(in distRef, in state.position, out posRef)
              yield exec, ether;
       codel <exec> mdGotoPosition(in speedRef, in posRef, out state, port out
            Mobile)
              yield exec, end;
       codel <end> mdStopEngine() yield ether wcet 1 ms;
       codel <stop> mdStopEngine() yield ether;
       interrupts MoveDistance;
       task motion;
       throw TOO_FAR_AWAY;};
activity Monitor (in double monitor = 0 :"Monitored absolute position in m",
              out double position) {
       doc  "Monitor the passage on the given position";
       validate controlPosition (in monitor);
       codel <start> monitor(in monitor, in ::ids) yield pause::start, end wcet 2 ms;
       codel <end> monitorStop(in ::ids, out position) yield ether;
       codel <stop> monitorStop(in ::ids, out position) yield ether;
       task motion;
       throw TOO_FAR_AWAY;};
};
```

Listing 2.1: Excerpt from the GenoM3 specification of the DEMO component.

2.2.3 Behavior

We go in more details and see how these different elements interact and how the component internally runs.

Codels Code elements, or codels, are small chunks of C or C++ code (*e.g.* the codel `StopMotion` in line 31 matchs a C function whose body is defined in a separate file). When defined within *activities*, codels are associated with *states* in a finite-state machine (see *activities* and *FSM* below). For instance, the codel `mdGotoPosition` (line 39) is associated with the state `exec` (more details below).

Services Services hold the specifications of the algorithms handled by the component. Services can take arguments (*e.g. SetSpeed* takes a *SpeedRef*, line 22), and return values (*e.g. GetSpeed* outputs a *SpeedRef*, line 26). A service may have a `validate` codel (*e.g. Monitor*, line 48). When the control task receives a service request, it runs the service `validate` codel, if any, to check whether the service arguments are valid (it reports an error to the client that requested it if they are not). A service may also specify other services it interrupts (*e.g. Finish* interrupts *MoveDistance*, line 32). Aside from control services (see below), a service may not run unless all the services it interrupts are terminated. A service that is ready to run is called an *activated* service. There are two types of services:

Control Services, are only for quick computations which should not delay the control task (that executes them). A control service may be an *attribute* (setter or getter of fields of the IDS, *e.g. GetSpeed*), or a *function* (for quick and simple computations, *e.g. Finish* . A GenoM3 component offers four predefined functions, namely: *Kill* (stop the component), *Abort* (interrupt an *activity*, see activities below), *Connect Port* to connect a local *in* port to a distant *out* port and *Connect Service* to connect a service of another component.

Activities, are executed by the execution task specified in their declaration (*e.g.* line 44, the activity *MoveDistance* is executed by the task motion). Activities are *finite-state machines*, each state associated with a codel.

FSM define the behavior of the activity through states, codels and *transitions*. A codel specifies the state it is associated to and the C or C++ function it will call, with the arguments (taken from the activity arguments, the IDS and the ports of the component) they need for their execution (*e.g.* `mdGotoPosition` is associated with the state `exec` of *MoveDistance*, it reads the fields *speedRef* and *posRef* of the IDS and writes the field *state* of the IDS and the port **Mobile**, line 39). A codel specifies also the possible *transitions* subsequent to its execution (*e.g.* the execution `mdStartEngine`, associated with the state `start` of *MoveDistance*, returns the state `exec` or the state `ether`, line 38). The non-determinism is resolved at runtime when executing the codel, which returns upon completion the next state to transit to. Taking a transition labeled *pause* stops the execution of the activity until the next cycle of its execution task (see execution tasks below), the activity is thus *paused* (*e.g.* if the execution of `monitor` in activity *Monitor* returns `pause::start`, *Monitor* is paused at state `start` until the next cycle of the task motion). Each codel may specify a WCET, namely the worst case execution time of the codel on a given platform (*e.g.* `mdStopEngine`, associated with the state `end` of *MoveDistance*, has a WCET of 1 ms, line 41). More about how to collect WCET in Sect. 2.3.2. Any activity FSM has the states `start` (entry point) and `ether` (end point). When the latter is reached, the activity is *terminated* and reported to the client. The state `stop`, if exists, is associated with the codel to execute when the

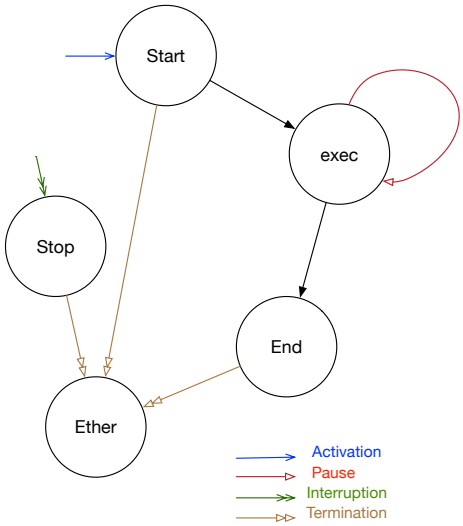


Figure 2.2: FSM of *MoveDistance* (lines 34-45 of listing 2.1).

activity is interrupted (*e.g.* line 51). If an activity with no `stop` codel is interrupted, it transits directly to `ether`. Fig. 2.2 is a visual illustration of the FSM behavior of activity *MoveDistance* (WCETs are omitted).

The organisation of activities along FSMs may be seen wrongfully as an unnecessary burden for programmers. Indeed, nothing prevents the programmer to have one `start` codel that does it all. Yet, breaking code along an FSM brings a number of advantages as it *e.g.* improves code execution interleaving and provides a finer model of data sharing and code interlocking (several shorts computations using each a fragment of resources brings a better concurrency level and allows shorter task periods than a single long computation that uses all resources). Furthermore, FSMs are amenable to translation into formal languages (Chapt. 4).

Control task The control task has a cyclic behavior that consists in managing the requests and reports of the component, executing control services and activating and interrupting activities. It runs the `validate` codels for services which specify one. If there exist activities that are incompatible with the requested service, the control task instructs the execution tasks in charge of such activities to interrupt them. If the request is for a control service (attribute or function), the control task executes it immediately. Otherwise, the requested activity is put on hold until all the incompatible instances are correctly interrupted and terminated. The requested activity is then activated. The control task instructs thus the execution task in charge of such activity to run it, and sends an *intermediate reply* to the client to inform it that processing has started. Upon completion of any service, the control task sends a *final reply* to the corresponding client (service ended nominally, interrupted, or failed by throwing an exception).

Execution tasks Execution tasks are cyclic tasks that can be periodic or aperiodic (*e.g.* the period of motion is 400 ms, line 18). With each cycle (triggered by a period signal or event occurrence), the execution task runs, sequentially, its permanent activity (if any) and all the instances of the activities it is in charge of, previously activated by

the control task. The execution of an activated instance ends when the instance is paused or terminated. In the former case, the instance will be resumed at the next cycle.

Internal Data Structure The IDS stores data that represent the internal state of the component, shared among tasks and services. For instance, the IDS of DEMO (lines 4-7) stores the current current position and current speed (in the field *state*), the speed reference and the position reference of the mobile. Access to the IDS is mutually exclusive. One can see that the proper specifications (enforced by GenoM3) of the codel arguments allows for a fine grain locking of the IDS and thus a high level of concurrency (only the needed field(s) by a codel are locked when it executes and simultaneous readings are allowed).

Concurrency The control task and the executions task(s) are run as concurrent threads on the hardware. That is, they are implemented as parallel tasks, with concurrent access to the IDS fragments. Multi-core executions are thus supported. We note that we do not, however, consider *distributed* applications, that is over networks of computers. Indeed, actively researched phenomena related to distribution such as clock drift [Giridhar and Kumar, 2006] and implications of CAP theorem [Brewer, 2012] are out of the scope of this thesis. We thus focus on a fine-grain locking model that allows parallel execution over a finite number of cores, but running on a single computer. This is the case for both of our case studies (Sect. 2.4, more details about the quadcopter hardware in Sect. 7.2.1.6).

Ports Data flow between components is made through ports (line 10). As seen above, ports usage (in or out) is also declared in codels arguments (*e.g.* line 39). Consequently, over a large set of components composing a robotic functional layer, we have a clear model of which codels use a particular port. When formalized, this model will enable verifying important properties such as not reading a port that has not been written at least once before (Chapt. 7).

Note that, time-wise, operations other than executing codels/services (activities and control services) are considered to take a negligible amount of time. For instance, the algorithm of mutual exclusion is supposed to detect that a resource is free "as soon as" it is available, that is, a negligible amount of time elapses between releasing a resource by a codel and detecting such release by the system. This approximation is backed up by the efficient implementations available today for such "atomic" operations as opposed to time-consuming code at the services level.

Several components and communication Functional-layer components need to exchange data. In a sensor-based navigation application, for instance, the collaboration of several components is indispensable (*e.g.* the information collected by a component from a sensor is a necessary input for another component handling a controller). GenoM3 offers ports in order to enable interaction between several components. If a port **P** is declared **out** (respect. **in**) within a component C, any codel of C may be entitled to write (respect. read) **P**. It is up to the programmer to decide which codels have access to the port. The predefined control service *Connect Port* allows connecting an **in** port of one component to an **out** port of another component. For instance, let C be a component featuring an **in** port **P** of the same type as the port **Mobile** of DEMO (line 10 of listing 2.1). Connecting **P** (component C) to **Mobile** (component DEMO),

made through requesting *Connect Port*, allows the codels of **C** to read data written on **Mobile** (by the codels of DEMO). More examples are given in Sect. 2.4.

Clients GenoM3 components are usually unable to evolve unless controlled by external clients. Indeed, apart from permanent activities, services need to be requested in order to be served by the component (see the semantics of the control task in Chapt. 4). For instance, after being implemented and run, the component DEMO, whose specification is shown in listing 2.1, does not execute any service. Indeed, the control task, in charge of the component, needs to receive requests in order to run control services and activate activities. For this, clients, which are external entities to the component, send the requests for the services they want to run, together with the arguments, if any. For example, the following line in a *Tcl* client requests the activity *MoveDistance* with the argument 0.5, that is requests moving the mobile for 0.5m:

```
demo::MoveDistance(0.5)
```

GenoM3 components can be controlled by various types of clients such as C, Tcl, and OpenPRS [1]. In each case, the libraries are automatically generated through templates (see next section).

2.3 Templates

2.3.1 Overview

GenoM3 specifications, *i.e.* dotgen files, do not enforce any specific implementation. Besides specifying the algorithms executed by the codels, one needs to generate all the files for the implementation, that varies from a middleware to another, and still must induce a behavior that agrees with the one presented in the last section. Furthermore, the interaction with the clients needs also to be made possible though generating the necessary libraries. One may also want to generate additional files such as documentation and comments. The *template* mechanism [Mallet et al., 2010] aims initially at generating, from *dotgen* specifications, the implementation, clients libraries, and additional files as explained above.

The template mechanism workflow is summarized as follows. First, GenoM3 parses the *dotgen* files and, if they are syntactically correct, builds an internal representation of the specified components in terms of Tcl structures. This representation is fed to the tcl interpreter together with the template (which is a tcl program) to generate the wanted files in the needed format, which may be seen as the template instance of the input components (Fig. 2.3). Since the files may be of different extensions, templates support generating unrestricted-format text files. This versatility of the mechanism enables the possibility to write several useful templates beyond middleware implementations and communication with various types of clients (Sect. 2.3.4).

2.3.2 Middleware and implementation

Specifying a GenoM3 component is thus decoupled from the implementation. The programmer specifies the component constituents (except the control task) as well as the services and codels parameters (*e.g.* ports, IDS fields) and their read/write access in the *dotgen* file. They also specify the algorithmic core of their codels without

[1] https://git.openrobots.org/projects/openprs

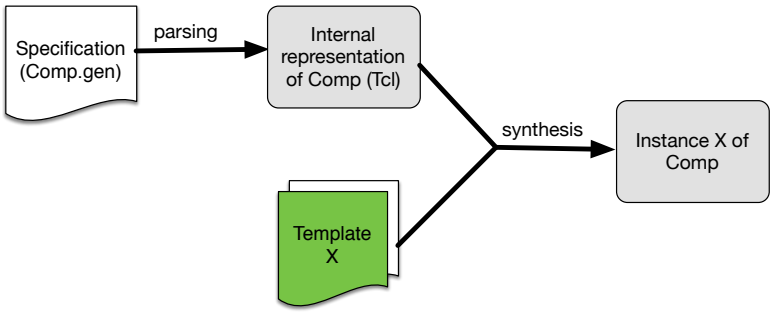


Figure 2.3: GenoM3 templates (generic).

making any implementation-specific call (involving the implementation middleware). Codels only depend on the objects passed as arguments, that are found in the *dotgen* specification, *i.e.* in the IDS fields and/or ports and do not therefore depend on the implementation environment. Similarly, activity codels can only return the next state in the activity FSM or an exception, also found in the specification. Middleware templates generate an implementation of the generic behavior of tasks and services in accordance with the description given in Sect. 2.2.2 and Sect. 2.2.3. Moreover, they automatically synthesize the *glue code* in charge of making calls to the middleware. This ensures a complete independency from the implementation. Fig. 2.4 shows an overview of GenoM3 workflow for implementation independence (*aka* middleware independence). Two main middleware templates are available for GenoM3 components (for PocoLibs[2] and ROS-Comm [Quigley et al., 2009]).

At the behavioral level, the middleware templates handle two different aspects of the component implementation:

- The component: englobes the mechanisms of evolution of tasks and services while properly implementing the mutual exclusion over shared memory and the control flow with regards to external clients. This implementation must be in accordance with the expected behavior as described in Sect. 2.2.3. Behaviorally, middleware templates agree on this aspect (more details in Sect. 6.3). For the control flow, both templates use mailbox mechanisms where the control task is notified whenever a new request is received.

- The data flow: a specific part which deals with the communication between components through ports. This part reflects the main divergence between the available middleware templates. Indeed, the Pocolibs template implements ports as shared memory whereas the ROS-Comm one uses *topics*. This means that ports in Pocolibs need to be protected from simultaneous access from different codels, which is not required in ROS-Comm. We still, because of real-time requirements, favor the Pocolibs implementation in this thesis as explained hereafter.

Pocolibs vs. ROS-Comm A ROS topic has one or more *publishers* and *subscribers*. A publisher (respect. subscriber) writes (respect. reads) the topic. With the ROS-Comm implementation, a publisher message is buffered then pushed into each subscriber queue by an internal ROS thread, additional to those created for each GenoM3

[2] https://git.openrobots.org/projects/pocolibs

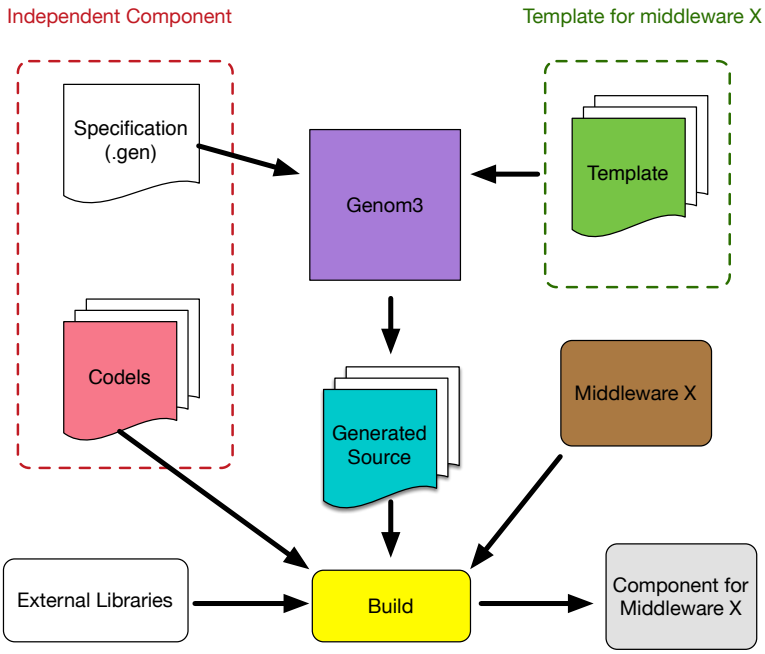


Figure 2.4: Generating a GenoM3 component for a middleware X

task in the application. Similarly, a ROS internal thread pulls the message for each subscriber. This boils down to writing for each subscriber, which creates a load proportional to the number of subscribers. On the other hand, each port in Pocolibs is a data field properly protected to prevent simultaneous access. A codel that reads/writes a port may thus do so whenever it is executed by its task in its own thread. Despite delays in codel execution due to mutual exclusion over ports, the Pocolibs implementation is the better choice as (i) no additional threads are involved and (ii) the number of readers/writers of a port has no side effect on jobs loads. This removes unpredictable and uncontrollable behaviors and adds thus to the accuracy of the generated formal models (Sect. 6.3). The ROS-Comm implementation is still more practical for distributed applications deployed over more than one platform, which is not the case for our applications (Osmosis and quadcopter, Sect. 2.4).

WCET Middleware templates are enriched to provide runtime execution times for each codel. The template synthesizes code which, when the application ends, outputs for each codel the number of times it was executed and the minimum, the average and the maximum execution time. For instance, below is the output for the codel `mdGotoPosition` (activity *MoveDistance* of component DEMO, listing 2.1).

```
Demo: mdGotoPosition called: 2567 times, min: 0.0007, max: 0.015, average:
    0.001
```

Specifying WCET is optional. The reason for this choice is that WCET is related to the hardware, so the developer is not required to provide such information at the design level. However, for verification purposes, WCET information are necessary to reason on timed properties in highly complex concurrent applications. Programmers are therefore able to enrich their specifications with WCET information at a later stage of the

```
The component <"[$component name]"> has <"[llength [$component tasks]]"> execution
     task(s):
<' foreach t [$component tasks] { '>
       Task <"[$t name]"> periodic at <"[$t period]"> s in charge of
       <"[llength [$t services]]"> activities:
       <'foreach s [$t services] {'>
              <"[$s name]">
       <'}'>
<' } '>
```

Listing 2.2: A simple template code for illustration.

deployment process. A "cleaner" possible alternative is to embed WCET information in a separate file from the *dotgen* specification.

WCET computation is a hard research problem in reality [Wilhelm et al., 2008]. Still, the information provided here may be exploited in the future to develop more precise models of WCET. So far, we gather these information on several runs and estimate the WCET to be a larger value than the maximum execution time obtained for each codel.

2.3.3 Client libraries

Besides the implementation of the component itself, also known as the *server*, by the middleware templates, there is a template that synthesizes C client libraries to control the component. It can be implemented by both PocoLibs and ROS-Comm. The interface of the library allows vital operations such as requesting services (see example in Sect. 2.2.3) and reading data on ports. The interface supports also the display of intermediate and final reports, as well as errors and exceptions.

The C client libraries provide also JSON where client interface is generic (not generated for each component). The code is dynamically loaded which allows controlling any component generically. Services, ports and datatypes are described with JSON dictionaries. OpenPRS clients use also the C libraries. The OpenPRS template generates the necessary code to control any component(s) with OpenPRS, where the supervisor is produced using *Transgen3*[3].

2.3.4 Mechanism

A template accesses the Tcl internal representation produced by GenoM3 (Fig. 2.3). This representation contains all the component(s) information (*e.g.* tasks and their periods, activities and their codels). Templates have no restriction on what they can synthesize from internal representations produced by GenoM3. For instance, the template code in listing 2.2 generates the output shown in listing 2.3 when called with the component DEMO (listing 2.1). The interpreter outputs everything without change except what is enclosed in *markers* <' '> that it evaluates in Tcl without output, and in <" "> that it evaluates and outputs the result. Output is written to a text file whose extension is the programmer's choice. This gives developers the freedom to write templates that output files in any desired language. More practical examples are given in Sect. 6.4.

[3] https://www.openrobots.org/wiki/transgen3

```
The component demo has 1 execution task(s):
       Task motion periodic at 0.4 s in charge of 2 activities
              MoveDistance
              Monitor
```

Listing 2.3: Output of Listing 2.2 when called with DEMO (listing 2.1).

This powerful mechanism paves the way toward an automated modeling and verification of robotic applications specified in GenoM3. Indeed, since the template feature comes with no output restrictions, one may, in theory, develop templates that produce the equivalent formal models, in any given language, of any GenoM3 specification. This is the basis of the automatic translations from GenoM3 to several formal frameworks developed and used in this thesis.

2.4 Examples

There are several applications developed and deployed using GenoM3, both academic (*e.g.* the RobNav navigation[4]) and involving industrial partners (*e.g.* the autonomous driving in the SafeNav H2020 CPSE-Labs project[5]). Among these, we focus on two real-world (yet academic) applications to use as case studies in this thesis, namely *Osmosis* [6] and *quadcopter*. These applications present a high level of complexity with a broad range of timing requirements that varies from a few microseconds (*quadcopter*) to hundreds of milliseconds (*OSMOSIS*). We introduce the components of each application with fair details. In each figure (*e.g.* Fig. 2.7), each box corresponds to a GenoM3 component, and octagons are *out* ports written by the components they are attached to and read by other components through an arrow (thus *in* ports are abstracted). Inside each box, we list the execution tasks, their periods (if any), and a partial list of the services provided by this component. The *perm* keyword refers to permanent activities. As said in Sect. 2.2, designing GenoM3 components is the programmer's choice, so these applications could have been built differently. This presentation aims thus at explaining the functionalities of the components in our design rather than justifying the latter's choices.

2.4.1 Osmosis

Osmosis is an H2020 CPSE-Labs project that aims at assessing safety in autonomous and intelligent systems. The case study involves a robot equipped with a Laser Range Finder LRF for laser-based potential field navigation. The application presented here is inspired from experiments where the robot inspects lights on airport landing strips[7]. The GenoM3 specification of Osmosis includes 10 components[8] (Fig. 2.5) that are integrated with the real *Robotnik* robot and work also in simulation using the *Gazebo* simulator. In the latter case, *Gazebo* replaces the lower level four components (Fig. 2.6)

[4] Specifications available at https://redmine.laas.fr/projects/robnav/repository

[5] http://www.cpse-labs.eu/experiment.php?id=c3_fr_safenav

[6] Open-Source Material fOr Safety assessment of Intelligent Systems https://osmosis.gitlab.io/

[7] See case study at http://138.100.58.3/web/marketplace/producto.html

[8] Specifications available at https://redmine.laas.fr/projects/osmosis (in the corresponding sub-projects).

and provides odometric positions and laser ranges in ROS topics.

- LASERDRIVER is in charge of the LRF. It has a scan task (periodic at 100 ms) in charge of the *StartScan* activity. The latter produces, on the port **Laser**, the laser's ranges in front of the robot. The port **Laser** is updated at every period with new data from the sensor. This component is absent in the simulation.

- ROBOTDRIVER handles the communication between the GenoM3 ports, **Odometry** and **Cmd**, and the Robotnik ROS topics. It has a genomTOros task (period 40ms) that reads the speed on the **Cmd** port (SAFETYPILOT) and writes publishes its content to a ROS topic for speed command. In parallel, the rosTOgenom task (aperiodic) subscribes to the ROS topic containing the current odometric speed and writes it to the **Odometry** port. This component is absent in simulation mode.

- IMUDRIVER handles the Inertial Measurement Unit IMU. It has one task Update that reads periodically (at 100Hz) the IMU driver through its activity *Measure*. The position orientation, the angular velocity and the linear acceleration together with their covariances, are written on the port **IMU**. This component is absent in simulation.

- GPSDRIVER handles the Global Positioning System GPS. It has one task Update that reads periodically (at 10Hz) the GPS driver through its activity *Measure*. It then writes the coordinates x (longitude), y (latitude) and z (altitude) with covariances to the port **GPS**. This component is absent in simulation.

- POM implements an Unscented Kalman Filter UKF. It has two execution tasks: (i) io reads **Odometry**, **IMU** and **GPS** measurements from their respective components and inserts them together with their timestamps in a buffer (ii) filter applies a UKF to the buffered timestamped measurements to produce an estimated state with new covariances. The UKF-estimated state gives the coordinates of the robot together with the orientation, the linear velocity, the angular velocity and the linear acceleration. This filtered state, that we refer to from now on as the robot *position* is written to the port **Pose**. Both tasks are periodic at 10ms.

- NAVIGATION computes intermediary positions toward a goal position. It reads **Pose** from POM and the file *Nav Graph*,which gives the navigation nodes on the airfield, and produces the intermediary positions in **Target**. It has one task navigate in charge of two activities *GotoPosition* and *GotoNode*. The former takes the goal position as an argument and, given the current position from **Pose**, updates **Target** with intermediary positions that it reads from *Nav Graph*. The latter does a similar job but the position given as an argument has to correspond to a node on the graph read from *Nav Graph*.

- POTENTIALFIELD performs a potential-field-based navigation. It has one task plan (period 100ms) with one activity *TrackTarget* which applies the potential field navigation algorithm. Given a goal in **Target** (from NAVIGATION) and the robot position **Pose** (from POM), the distance to the goal position is computed. Moreover, *TrackTarget* reads **Scan** (from LASERDRIVER) and computes distances to obstacles from the laser ranges. Computed distances to goal position and obstacles are used then to compute the attractive (to the goal) and repulsive (from the obstacles) forces which are used to compute the velocity (angular and linear) to reach the goal. Such velocity is written to the port **PFCmd**.

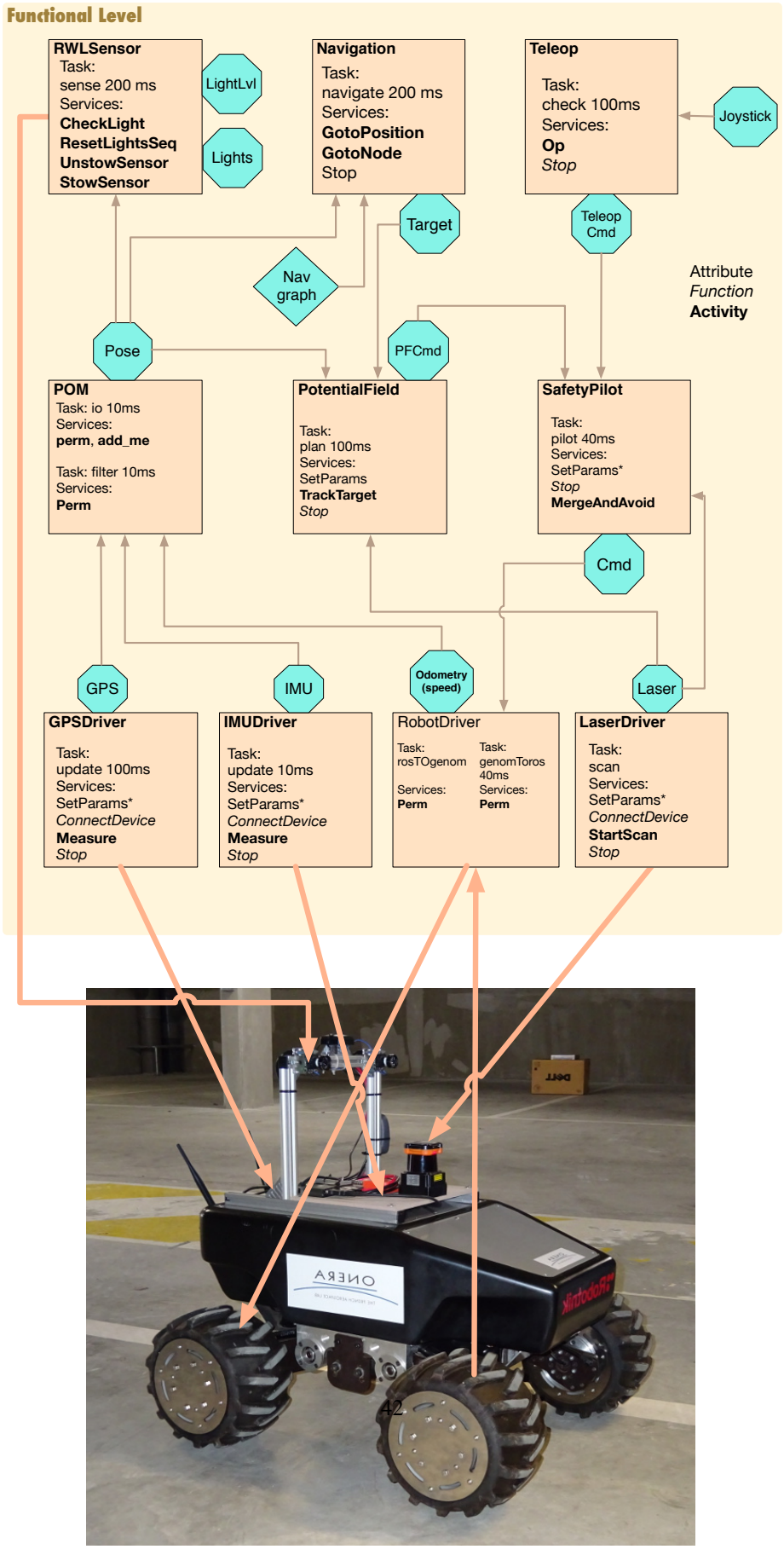


Figure 2.5: Osmosis functional level.

- TELEOP is for manual navigation using a joystick. The joystick signal, consisting in a direction and a speed (through a speed button) is stored on the port **Joystick**. The task check, periodic at 100ms, has one activity *Op* that reads the signal stored on **Joystick** and converts it into a velocity that it writes to **Teleop Cmd**. This component is not available in simulation.

- SAFETYPILOT deals with obstacle avoidance and joystick commands. It has one task pilot, periodic at 40ms. This component is fed with a velocity either from potential field navigation (in **PFCmd** from POTENTIALFIELD) or from joystick commands (in **Teleop Cmd** from TELEOP). In the first case, the activity *MergeAndAvoid* computes distances to obstacles from **Laser** ranges (from LASERDRIVER) to ensure a safe command, that it writes to **Cmd**, in case dynamic obstacles appear abruptly. In the second case, the velocity on **Cmd Teleop** is directly copied to **Cmd**, ignoring that on **PFCmd** (from POTENTIALFIELD), which guarantees prioritizing commands from the human operator.

- RWLSENSOR is the component that manages the *RunWay Light sensor* of the robot. It has one task sense running at 5Hz. The activity *CheckLight* measures a light intensity and stores the value in the port **LightLvl** then, using **Pose** (from POM), updates the checked light level on **Lights** which contains an array of all the lights where each is defined by its position and latest measured intensity.

In both modes, simulation and real robot, important properties that neither the specification nor the implementation can guarantee emerge. For instance, the failure to update **Laser** by LASERDRIVER within a set time bound must be detected and an emergency routine must be followed consequently. This is a practical example that reflects the need of connecting robotics to formal methods, which is a main motivation of this thesis. We recall that properties at the algorithmic level are not in the scope of this thesis (Chapt. 1), as we focus on verification rather than validation. For instance, our approach does not deal with problems such as the local minima in potential-field navigation algorithms [Guerra et al., 2016] that cannot be analyzed in an automatic manner.

2.4.2 Quadcopter

The quadcopter flying at LAAS runs 5 functional components[9] (Fig. 2.7). This application is also available in simulation where the component MRSIM simulates the the effect of the propeller velocity on the quadcopter, and produces the current propeller velocity, IMU and GPS. In the following, we describe each of the components design and functionality:

- MIKROKOPTER is the component in charge of the quadcopter low-level hardware. The quadcopter is controlled by applying a velocity to each propeller, and produces the current velocities, as well as its current *IMU* values. The component has two execution tasks i) comm, aperiodic, in charge of polling, parsing and storing data from the hardware (to get the current propellers velocity and IMU) and ii) main, periodic at 1KHz, which reads the **cmd velocity** port and writes the two ports **IMU** and the propellers **actual velocity**.

[9] Specification available at `https://git.openrobots.org/projects/telekyb3` (in the corresponding sub-projects).

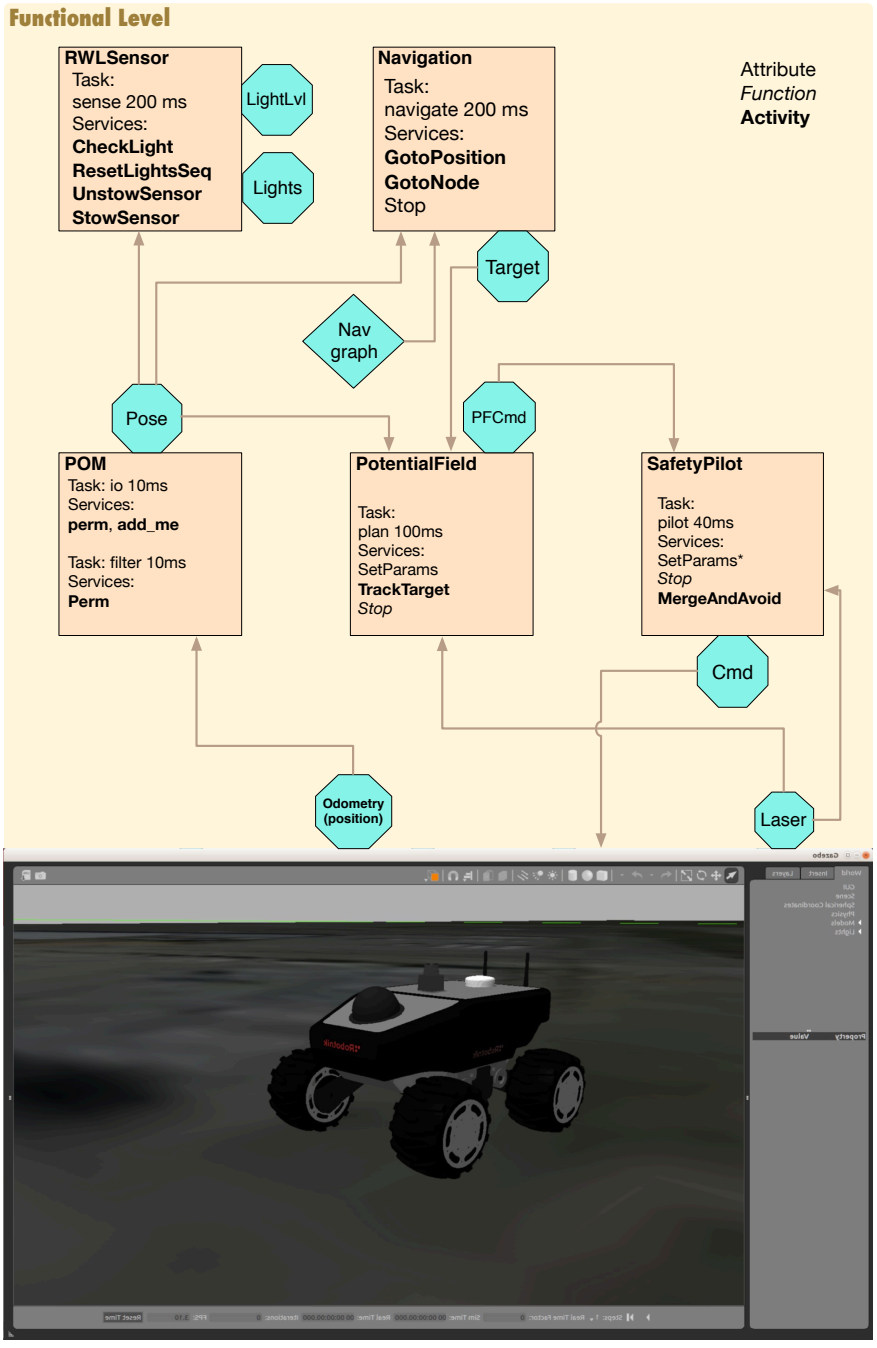


Figure 2.6: Osmosis functional level (simulation).

- OPTITRACK is the component handling the current position of the quadcopter as perceived by our "OptiTrack" motion capture system. It has one execution task publish that provides the current position of the quadcopter in the **mocap pose** port. Its period is 4ms. This component is absent in simulation.

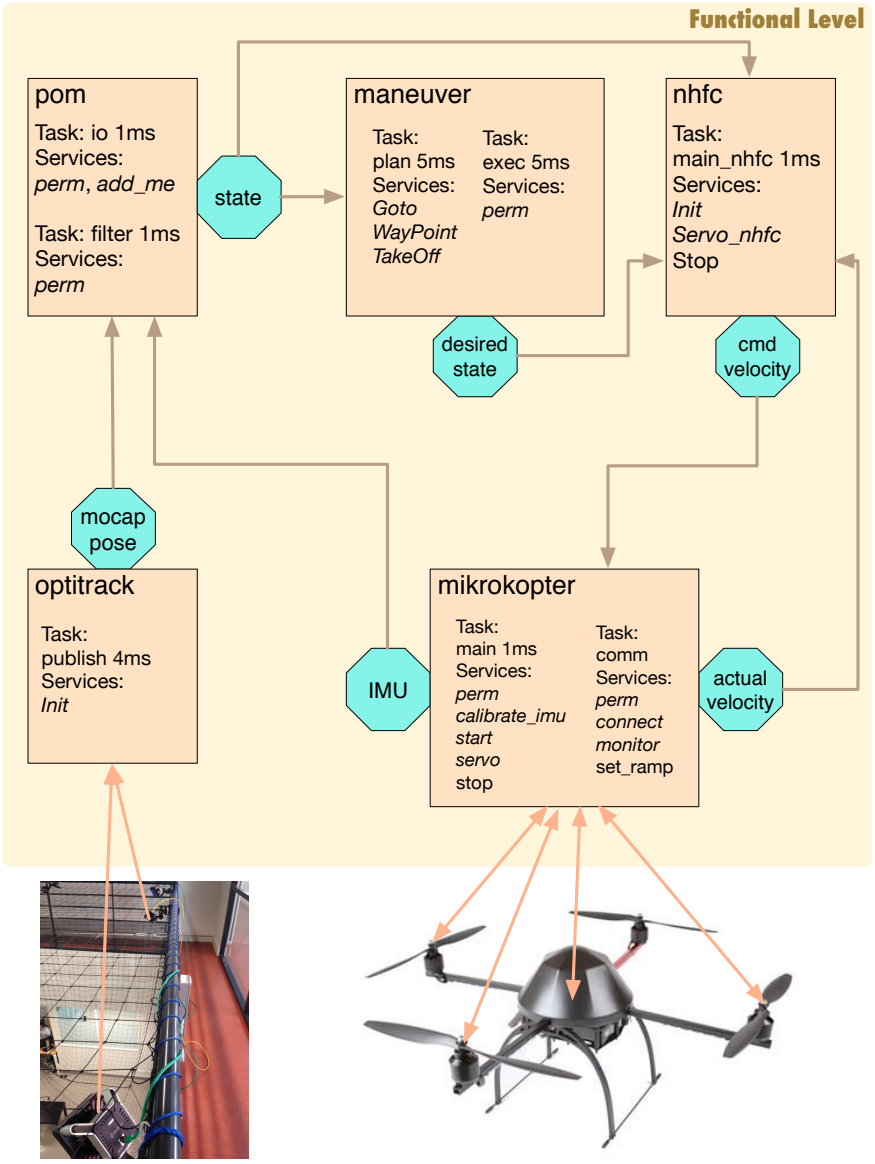


Figure 2.7: The quadcopter functional level. Activities are in *Italic* font.

- POM is the same component POM in Osmosis (Sect. 2.4.1) but with less input ports and with a higher frequency. It merges the **mocap pose** position produced by OPTITRACK and the **IMU** from MIKROKOPTER and produces a UK-filtered position in port **state**. Its two periodic execution tasks io and filter run at 1KHz.
- MANEUVER is the navigation component, it has two execution tasks exec and plan both periodic at 5ms. Given a position or waypoints to navigate to, it reads the **state** and computes the intermediate positions to fly to that it copies to **desired state**.
- NHFC (Near Hovering Flight Controller) is the core of the flight controller. Run-

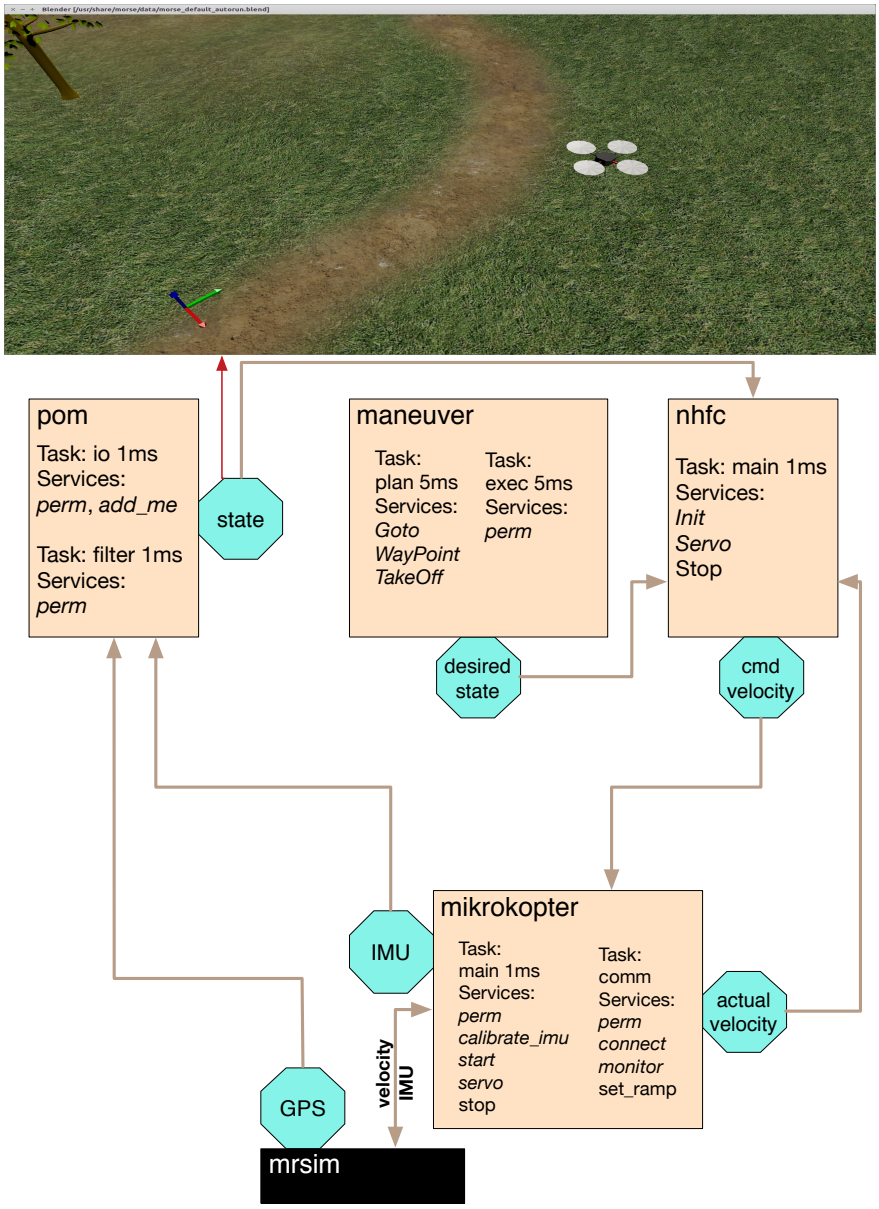


Figure 2.8: The quadcopter functional level (simulation).

ning one task main_nhfc at 1KHz, it reads the **actual velocity** port of the propellers, the current position in the **state** port of POM, and the desired position (port **desired state**) of MANEUVER and produces the proper **cmd velocity** port containing the desired velocity of the propellers (which is then read by MIKROKOPTER) to reach and hover near this position.

- MRSIM for simulation only. It simulates the hardware and provides data on propellers velocity and IMU, fed to MIKROKOPTER. It has a **GPS** port on which it produces the simulated position, used by POM to produce **state**.

In simulation, the filtered position in **state** (POM) is fed to the *Morse* visualizer. The latter will therefore display the quadcopter simulated motion using the successive poses provided by POM.

Note the high frequency at which most of the components perform. Indeed, flight controllers are usually critical and their tasks need to evolve at a high periodicity rate. To deploy such systems in human environments, it is very important to verify properties that the sole specification cannot guarantee, no matter what the chosen middleware and OS are. For instance, violating the period of any periodic task within the most critical components MIKROKOPTER, NHFC and POM, might result in a catastrophic behavior. Indeed, when we provoke period violations in *e.g.* filter (POM), using basic temporary suspension functions such as *sleep()*, we may visualise a drone crash after a few deadline misses on the simulator. The urge of formally verifying such crucial properties consolidates the motivation of the work presented in this thesis.

2.5 Conclusion

We advocate in this chapter the use of GenoM3 as our robotic framework of choice. This is justified along the chapter through the presentation of GenoM3 model-based approach, which makes it amenable to formalization, but also through its powerful template mechanism that will enable a fully automatic connection with formal frameworks. Also, the two real-world examples given at the end of the chapter, that are currently deployed on real robots, show the maturity of the framework and give valuable insights on the type of crucial properties that roboticists are interested in and that cannot be verified at the robotic programming level. This presentation exposes thus a non negligible part of the motivation of this thesis. This consolidates the conclusions drawn in Chapt. 1 on the urge of bridging robotic applications with formal methods and the convenience of using GenoM3 to this end.

Chapter 3

Semantics Formalism and Formal Frameworks

3.1 Introduction

This work relies on building automatic generators from GenMe to several formal languages and tools. This requires giving formal semantics to GenMe and therefore choosing a convenient formalism. In this chapter, we present our choice for formalizing GenMe, that is a more general version of the *Timed Transition Systems* (TTS) presented in [Henzinger et al., 1991]. We will also justify why we chose TTS over other possible formalisms. We start by presenting formal definitions of TTS and their semantics. We present then a graphical version of TTS "components" known as *Timed Transition Diagrams* and show how TTS can be built from the composition of these diagrams.

In a second part of this chapter, we introduce the formal languages/tools to which GenMe specifications will be automatically translated. We briefly present their underlying formalisms and their modeling and verification features. Simple examples are given through each section to clarify the formal and informal definitions. At the end of each section, the choice of the formal framework is motivated with regard to our verification needs.

3.2 Timed Transition Systems (TTS)

The chosen formalism is a variation of the Timed Transition Systems (TTS) presented in [Henzinger et al., 1991]. TTS provide a high level formalism suitable for giving operational semantics that are both independent from the implementation and understandable. There are several arguments that justify this choice and that will be given at the end of this section.

One difference between our definition of TTS and the one proposed in [Henzinger et al., 1991] is that we consider a dense-time model (durations and time constraints have values in $\mathbb{R}_{\geq 0}$ with interval bounds in $\mathbb{Q}_{\geq 0} \cup \infty$) whereas the original presentation relies on a discrete-time model (durations have values in $\mathbb{N}$). We also accept more general timing constraints, using time intervals with possibly left-open and right-open bounds. This extension of TTS makes them closer to the time model used in Timed Automata

and Time Petri Nets, which are two of the target formalisms used in this thesis. Finally, we consider a much simpler composition mechanism, as we will show in the following definitions.

3.2.1 Notations

We start this section by defining some notations that will be useful in the remainder of this chapter. We use $\mathbb{I}$ to denote the set of well-formed (time) intervals over positive reals, with rational lower bounds and rational or infinite upper bounds. An element i of $\mathbb{I}$ can be of one of four types: (where $a \in \mathbb{Q}_{\geq 0}$ and b can be either a rational number or the infinity symbol, ∞, meaning an infinite bound).

- $[a, b]$ (with $a \leqslant b$),
- $]a, b]$ (with $a < b$),
- $[a, b[$ (with $a < b$),
- $]a, b[$ (with $a < b$).

We say that $\downarrow i = a$ is the lower bound of the interval i and $\uparrow i = b$ is its upper bound. We also say that the interval $[a, b]$ is punctual when $a = b$.

In the following, we will often use the notation $\sqsubset a, b \sqsupset$ for time intervals, where $\sqsubset$ and $\sqsupset$ are the left and right bounds of i. Therefore we have $\sqsubset = [$ for a closed interval on the left and $\sqsupset = [$ for an open (strict) interval on the right. Likewise we use ']' for an open interval on the left and closed interval on the right. By an abuse of notation, we will also conflate bounds with comparison operators between reals. We say that $\sqsubset$ is the strict comparison operator $<$ when the left bound is open ($\sqsubset =]$) and that $\sqsubset$ is the operator $\leqslant$ when the bound is closed ($\sqsubset = [$). Likewise, we say that $\sqsupset$ is the operator $\leqslant$ when the right bound is closed and $<$ otherwise. With this choice of notation, an interval $i = \sqsubset a, b \sqsupset$ is exactly the set of real values $x \in \mathbb{R}_{\geq 0}$ such that $a \sqsubset x$ and $x \sqsupset b$.

For any date δ in $\mathbb{Q}_{\geq 0}$ and interval $i \in \mathbb{I}$, we denote $i - \delta$ the time interval obtained by shifting i (to the left) by an amount of δ. The operation is defined only if $\delta < \uparrow i$ (or if $\delta \leq \uparrow i$ and the right bound of i is closed), which we call the *upper bound condition*. We consider four different cases depending on the "shape" of interval i. Assume $a' = \max(0, a - \delta)$:

- if $i = [a, b]$ and $\delta \leqslant b$ then $i - \delta = [a', b - \delta]$,
- if $i =]a, b]$ and $\delta \leqslant b$ then $i - \delta =]a - \delta, b - \delta]$ if $\delta \leqslant a$ and $[0, b - \delta]$ otherwise,
- if $i = [a, b[$ and $\delta < b$ then $i - \delta = [a', b - \delta[$,
- if $i =]a, b[$ and $\delta < b$ then $i - \delta =]a - \delta, b - \delta[$ if $\delta \leqslant a$ and $[0, b - \delta[$ otherwise (With the convention that $\infty - \delta = \infty$).

3.2.2 Syntax of TTS

A Timed Transition System TTS is a tuple $\langle U, S, s_0, \tau, I \rangle$ where:

- U is a finite set of variables. Each variable is implicitly typed. We use $\mathrm{dom}(u)$ to denote the domain of variable u;

- S is a set of states. Each state of S is an interpretation of variables in U, that is a mapping from variables $u \in U$ to values in $\mathrm{dom}(u)$;
- s_0 is the initial state ($s_0 \in S$) that maps each variable to its initial value;
- τ is a set of transitions. Each transition $t \in \tau$ defines a partial mapping over states in S, that is, for every $t \in \tau$ and for every state $s \in S$ either: (a) there is a unique successor state $s' \in S$ such that $s \xrightarrow{t} s'$ (we write $succ(s,t) = \{s'\}$); or (b) there is no such successor $succ(s,t) = \varnothing$ (in which case we use the notation $s \xrightarrow{t} \varnothing$);
- $I : \tau \mapsto \mathbb{I}$ maps each transition $t \in \tau$ to a *static (time) interval* $I(t) \in \mathbb{I}$.

We denote $succ(s,t)$ the set of successors of state s by a transition $t \in \tau$. A transition $t \in \tau$ is said *enabled* at s if and only if s is a source state of t, that is $s \xrightarrow{t} s'$ (or equivalently $succ(s,t) \neq \varnothing$). We denote $\mathcal{E}(s)$ the set of transitions enabled at s.

From our definition of TTS, the set $succ(s,t)$ has cardinality at most one. This allows us to simplify the presentation of the semantics (especially when defining the notion of persistent transitions later in this section) without loosing any expressiveness (a state s may still have many successors over transitions with different names).

3.2.3 Semantics of TTS

In a TTS $\langle U, S, s_0, \tau, I\rangle$, each (enabled) transition is associated with a timing constraint, that is an interval $I(t) = \sqsubset a, b \sqsupset \in \mathbb{I}$. The semantics of time depends on the dates at which the transition becomes enabled. Informally, if we are in state s since the date Δ and if transition t can occur, then we can "take" the transition starting at a date $\Delta + \downarrow I(t) \sqsubset d$ and no later than a date $d' \sqsupset \Delta + \uparrow I(t)$, unless t is disabled in between by taking another transition.

The semantics of a TTS is therefore given over pairs (s, ϕ) where $s \in S$ is a state and $\phi : \tau \to \mathbb{I}$ is a mapping from transitions to time intervals. Intuitively, if t is enabled at s, then $\phi(t)$ contains the dates at which t can be possibly taken in the future. Hence, a transition t can be taken (immediately) only when 0 is in $\phi(t)$. Likewise, a transition t cannot remain enabled for more than its timespan, that is the value $\uparrow\phi(t)$.

We use $\phi \dot{-} \delta$ for the partial function that associates, at a state s, each transition $t \in \mathcal{E}(s)$ to the value $\phi(t) - \delta$ (the interval $\phi(t)$ shifted by δ). This function is useful to model the effect of time progress on the enabled transitions of a TTS. (Note that $\phi \dot{-} \delta$ is defined only when $\phi(t) - \delta$ is defined for all $t \in \mathcal{E}(s)$, that is δ satisfies the upper bound condition for all $\phi(t)$).

Let t be enabled at s with $s \xrightarrow{t} s'$. We say that a transition k is *persistent* (with $k \neq t$) if it is also enabled at s'. The transitions that are enabled at s' and not at s are called *newly enabled*. We define the predicates $pers(s,t)$ and $nenabl(s,t)$ that describe, respectively, the sets of persistent and newly enabled transitions after t is taken from s. We see that if t is still enabled in s' then it is necessarily newly enabled.

$$\begin{array}{lcl} pers(s,t) & = & \{k \in \tau \mid k \in \mathcal{E}(s) \wedge k \in (\mathcal{E}(s') \setminus \{t\}) \wedge s \xrightarrow{t} s'\} \\ nenabl(s,t) & = & \{k \in \tau \mid k \notin (\mathcal{E}(s) \setminus \{t\}) \wedge k \in \mathcal{E}(s') \wedge s \xrightarrow{t} s'\} \end{array}$$

With all these notations, we can define the semantics of a TTS as a Kripke structure (a rooted, *state graph*) such that:

$$\text{(discrete)}\ \frac{t \in \mathcal{E}(s) \qquad 0 \in \phi(t) \qquad s \xrightarrow{t} s' \in \tau \qquad \forall k \in \mathcal{E}(s') : \phi'(k) = \phi(k) \text{ if } k \in pers(s,t) \text{ and } I(k) \text{ otherwise}}{(s,\phi) \xrightarrow{t} (s',\phi')}$$

$$\text{(continuous)}\ \frac{\delta \in \mathbb{Q}_{\geq 0} \qquad \phi \dot{-} \delta \text{ defined}}{(s,\phi) \xrightarrow{\delta} (s, \phi \dot{-} \delta)}$$

Table 3.1: Operational Semantics of TTS.

- states in the graph are pairs (s, ϕ) where $s \in S$ is a state and ϕ is a map from $t \in \mathcal{E}(s)$ to $\mathbb{I}$,
- the initial state is (s_0, ϕ_0) where ϕ_0 is such that $\phi_0(t) = I(t)$ for each transition $t \in \mathcal{E}(s_0)$ (all transitions possible from s_0 are newly enabled),
- *discrete transitions*: from every reachable state (s, ϕ) and every transition $t \in \mathcal{E}(s)$, we have a (discrete) transition $(s, \phi) \xrightarrow{t} (s', \phi')$ when $0 \in \phi(t)$ and $s \xrightarrow{t} s'$. In this case ϕ' is the unique mapping such that: $\phi'(k) = \phi(k)$ for all transitions $k \in pers(m, t)$ and $\phi'(k) = I(k)$ otherwise,
- *continuous transitions*: for every delay $\delta \in \mathbb{Q}_{\geq 0}$ such that $\phi \dot{-} \delta$ is defined, we have a (continuous) transition $(s, \phi) \xrightarrow{\delta} (s, \phi \dot{-} \delta)$.

From this definition, we see that time progress does not change the set of enabled transitions; but it may change the set of transitions that may be taken immediately (the set of transitions such that $0 \in \phi(t)$). We can also see that the state graph of a TTS is generally infinite. Indeed, in most cases, we can choose between an infinite number of continuous transitions. This is, for instance, the case when there are no transitions in τ enabled at s (in which case we can let time elapse by an unbounded amount).

We give, In Table 3.1, an alternate definition of the reduction relation using notations borrowed from structural operational semantics, where the relation $\rightarrow$ is defined by a set of inference rules. We use this notation later in order to simplify the presentation of the semantics of GeneralizedTimedMaude.

In this work, we have chosen a *strong time semantics*, meaning that we must always take an enabled transition from a state (s, ϕ) if there is no delay $\delta > 0$ such that $\phi \dot{-} \delta$ is defined (that is when time cannot elapse). Since a transition cannot become disabled from a continuous transitions, it follows from this choice that a TTS cannot have a "timelock", that is a situation in which a system is blocked because every timing constraints is indefinitely false.

3.2.4 Timed Transition Diagrams

In this section, we define a graphical notation for TTS (called Time Transition Diagrams, or TTD for short) as well as a composition operation between TTD that is also inspired from the work in [Henzinger et al., 1991]. Basically, we can see every atomic TTD as a component and composition as a way to build more complex systems through the synchronization and interactions of simpler systems. In this approach, the composition of multiple TTD (viewed as *components*) results in a TTS (viewed as the

system).

A timed transition diagram (TTD) P is a finite directed graph where each *edge* e is labeled with: an interval $I(e)$; a guard g_e; and operations op_e. Each TTD operates on a finite set of variables, Y. We use V to denote the set of vertices of P and v_0 to denote its unique initial vertex. In the remainder of this thesis, guards that are always true, operations with no effect on variables and $[0, \infty[$ intervals will not be represented.

As with TTS, we say that a state s of a TTD is a mapping from variables in Y to values. We consider a distinguished variable, or *control vertex*, denoted π, whose value gives the "current vertex" of the TTD. Hence $\mathrm{dom}(\pi) = V$ and the initial value of π is v_0.

Informally, a guard is a boolean expression over Y that defines when an edge can be taken, whereas an operation is a sequence of instructions that can modify the values stored in these variables (all the updates declared in an operation are processed atomically). We use the expression $g(s)$ to denote the "truth" value of a guard g on the state s. likewise, we use the notation $s' = op(s)$ when the results of op on Y from s agree with the interpretation of Y at s'.

We show in Fig. 3.1 a simple generic TTD example with two vertices, v_0 and v_1, and one edge e. The initial vertex, in this case v_0, is denoted with an incoming edge without source vertex.

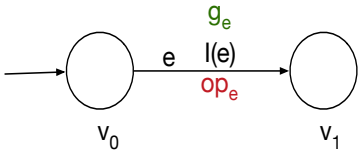


Figure 3.1: A generic TTD example

Given a TTD P, we can associate its meaning, $[\![P]\!]$, that is a TTS that describes the semantics of P. The meaning of P is the TTS $\langle U = Y \cup \{\pi\}, S, s_0, \tau, I\rangle$ such that:

- S is the set of states, where each state is an interpretation of the π and each variable in Y.
- the initial state s_0 is the mapping associating π to v_0 (initially the control vertex is at v_0) and all the variables in Y to their default value,
- τ is the set of edges of P and $s \xrightarrow{e} s'$ if and only if there is an edge e in P from v_i to v_j such that: $s(\pi) = v_i$ and $s'(\pi) = v_j$; and $g_e(s)$ is true; and $s' = op_e(s)$.
- The function I maps every edge e in P to the interval $I(e)$.

3.2.5 Composition of TTDs

The parallel composition of a finite number of TTDs, $P_1, \ldots, P_n$, over a set of shared variables, U_s, results in a TTS denoted:

$$\{\Theta\}[\,\|_{i\in 1..n}\, P_i\,]$$

A TTS also defines an initial valuation, Θ, that gives the initial assignation of variables in U_s to values.

We assume that the edge (identifiers) of different components are always distinct: if e is an edge in P_i then it cannot be an edge in P_j with $i \neq j$. We also assume that each component (TTD) P_i can have access to a set of local variables, denoted U_i, besides the variables in U_s. (We assume that $U_i \cap U_s = \varnothing$ and $U_i \cap U_j = \varnothing$ for all indexes $i, j \in 1..n$ with $i \neq j$). We also consider one distinguished control vertex, π_i, for each component P_i, whose role is to store the current vertex of the TTD. Hence the set of variables declared in a TTS is:

$$U = U_s \cup \left(\bigcup_{i \in 1..n} U_i \right) \cup \left(\bigcup_{i \in 1..n} \{\pi_i\} \right)$$

Given the parallel composition $\{\Theta\}[\, \|_{i \in 1..n}\, P_i\,]$, we can easily define a TTS with the set of variables U that will give the "semantics of the system". Assume that $\langle U_i, S_i, s_i^0, \tau_i, I_i \rangle$ is the meaning of P_i for all $i \in 1..n$. Then the meaning of $\{\Theta\}[\, \|_{i \in 1..n}\, P_i\,]$ is the TTS $\langle U, S, s_0, \tau, I \rangle$ such that:

- the set S lists all the possible interpretation of U,
- the initial state $s_0 \in S$ is the only interpretation such that $s_0(x) = s_i^0(x)$ if x $x \in U_i$ (in particular $s_0(\pi_i) = v_0^i$ for all $i \in 1..n$, where v_0^i is the initial vertex of the component P_i); and $s_0(x) = \Theta(x)$ for all $x \in U_s$,
- τ is the union of all edges found in the components $P_1, \ldots, P_n$, that is $\tau = \bigcup_{i \in 1..n} \tau_i$. The result of taking transition e in the TTS boils down to taking e in the corresponding component. Let $s_{|U}$ denote the restriction of a state (mapping) s to the variables in U. Then, if e connects v_j^i to v_k^i in P_i and $U_{s_{op}} \subseteq U_s$ is the set of shared variables affected by op_e, we have $s \xrightarrow{e} s' \in \tau$ if and only if $s_{|U_i \cup U_{s_{op}}} \xrightarrow{e} s'_{|U_i \cup U_{s_{op}}} \in \tau_i$ and $s'(x) = s(x)$ for all variables $x \in U \setminus (U_i \cup U_{s_{op}})$,
- the static time interval function I maps every transition $e \in \tau$ to the time interval $I_i(e)$, where P_i is the component containing the edge e.

The notion of a TTS defines a composition operator over TTDs and their compositions. This is basically the same operation as the one in [Henzinger et al., 1991] with the simplification that all the components must start in their initial state. That is we only consider a "synchronous start" of TTDs.

3.2.6 Sequential behavior

The fact that TTS support the use of variables eases building several classes of systems by simply composing TTDs. In this section, we show how to use TTS in order to build a system from the "sequential composition" of components. Sequential composition will be a useful operation when defining the behavior of execution tasks in Chapter 4.

We only describe the construction for a specific example. Let us consider the parallel composition of the TTDs in Fig. 3.2. We assume that the set of shared variables U_s contains a variable Π that will denote the "identity" of the only currently executing component; that is $\mathrm{dom}(\Pi) = \{P_0, P_1, P_2\}$ with $\Theta(\Pi) = P_0$.

The sequential composition of the three components is the TTS $\{\Theta\}[\, \|_{i \in 0..2}\, P_i\,]$ where the guard of each edge in the TTD P_i include the test $\Pi = P_i$. With this

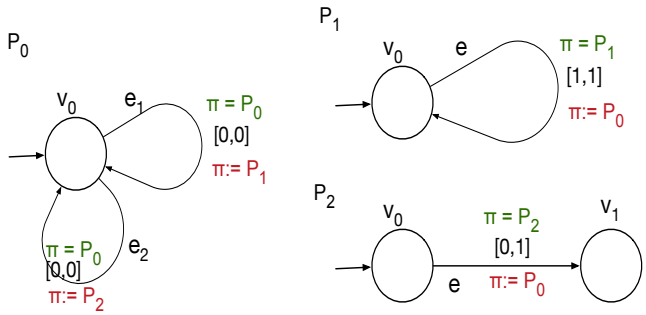


Figure 3.2: TTDs of a sequential system

constraint, it is only possible to take a transition from the component whose identity is the current value of Π. Therefore, at most one component can execute at a time.

In this particular example, component P_0 plays the role of a scheduler that gives the control randomly to either P_1 or P_2. Giving the control more than once to P_1 leads a deadlock (no discrete transitions possible in the resulting TTS).

3.2.7 Suitability

We discuss the rationale for the choice of TTS for formalizing GenoM3, as opposed to *e.g* other formalisms based on clocks, such as *Timed Automata* (Sect. 3.4.1). There are several arguments that favor such a choice among which we emphasize the following.

Variables and compositionality: As seen in Chapter 2, GenoM3 relies on a compositional approach where robotic applications contain several components communicating together. Moreover, components themselves are built from entities that interact in order to ensure a correct behavior with regards to the requirements. For instance, the control task interacts with the execution tasks to instruct them on which activities to run or interrupt (Sect. 2.2.2 and Sect. 2.2.3). The power of TTS through the composition of TTDs is very useful in such circumstances. Indeed, shared variables and the parallel operator allow to ease the modeling of the complex asynchronous communication within and between GenoM3 components (more in Chapt. 4). Also, the sequential behavior within execution tasks can be conveniently modeled using shared variables and parallel operators as seen in Sect. 3.2.6 (more in Chapt. 4).

Variables, guards and time intervals: The possibility to have guards over variables and to use time intervals makes TTDs suitable for modeling the entities of a GenoM3 component. For instance, one may, within a TTD model of an activity, condition through the guards the execution of a codel by the availability of resources, and use WCET as upper bounds (we give examples about this in Chapter 4).

Urgencies: many of the behaviors in GenoM3 are subject to *global urgency constraints* rather than local ones. For instance, executing a codel happens *as soon as* it has secured the needed resources within the IDS, shared between all tasks. These aspects are modeled easily in TTS as opposed to clock-based transition systems such as those based on *e.g.* classical *timed automata* where urgencies can be expressed only locally

using *invariants*. The confrontation between the two models in terms of expressing urgencies is explained in details in Sect. 6.3.3.2 and Sect. 7.2.1.4.

The remaining sections in this chapter are devoted to a high-level description of the other formalisms and tools used in this thesis.

3.3 Fiacre and TINA

We present in this section the formal language Fiacre and the model-checking toolbox TINA, developed at *LAAS-CNRS*. Both Fiacre and TINA are freely available with some introductory material at, respectively, `http://www.laas.fr/fiacre` and `http://www.laas.fr/tina`. The model checkers and state abstractions available in TINA can be used to explore and analyze Fiacre descriptions translated into extended *time Petri nets*, introduced hereafter.

3.3.1 Time Petri Nets

Introduction Petri nets [Petri, 1962] are a prominent model for the analysis of concurrent, distributed and discrete-event systems (examples in [Ramamoorthy and Ho, 1980; Genc and Lafortune, 2003; Leveson and Stolzy, 1987]). There exist different flavors of Petri nets extended with *time* to enable modeling and analyzing timed and real-time systems. Time Petri nets [Merlin and Farber, 1976] is one of these extensions. A time Petri net (TPN for short) enriches a Petri net with time intervals associated with the transitions of the net and specifies thus the possible time delays between the last enabledness of these transitions and their activation (or firing in Petri net terminology).

In a TPN, each transition t is associated with a time interval $I(t) \in \mathbb{I}$, in much the same way a transition t in a TTS has a timing constraint. We use the same notations from Sect. 3.2, namely:

- $\downarrow I(t)$ is the lower bound of interval $I(t)$, also called the *earliest firing deadline* of t,
- $\uparrow I(t)$ is the upper bound of $I(t)$ (it can be equal to ∞), also called the *latest firing deadline* of t.

Firing a transition in a TPN is constrained by the same *Strong Time Semantics* condition than the TTS defined in Sect. 3.2. Actually, we will encounter similar notions of enabled, newly enabled and persistent transitions, but given in terms of Petri net markings rather than TTS states.

More formally, a TPN N is a tuple $\langle P, T, Pre, Post, m_0, I\rangle$ such that:

- $\langle P, T, Pre, Post, m_0\rangle$ is a Petri net, with P the set of places, T the set of transitions, $m_0 : P \to \mathbb{N}$ the initial marking, and $Pre,\ Post : T \to (P \to \mathbb{N})$ the precondition and postcondition functions,
- I is the *static interval function*, that associates a time interval in $\mathbb{I}$ to every transition in T.

In the following, $\mathcal{E}(m)$ is the set of enabled transitions at marking m, that is the set of transitions $t \in T$ such that $m \geq Pre(t)$ (we use the pointwise comparison between functions). The shifting function $\phi - \theta$ and the partial function $\phi \dot{-} \theta$ that associates,

at a marking m, each transition $t \in \mathcal{E}(m)$ to the value $\phi(t) - \theta$ are the same as defined in Sect. 3.2.

At a marking m, we say that a transition k is *persistent* (with $k \neq t$) if it is also enabled in the marking $m - Pre(t)$, that is if $m - Pre(t) \geq Pre(k)$. The transitions that are enabled at m' and not at m are called *newly enabled*. We define the predicates $pers(m, t)$ and $nenabl(m, t)$ that describe, respectively, the sets of persistent and newly enabled transitions after t fires from m. We see that if t is still enabled in m' then it is necessarily newly enabled.

$$pers(m, t) = \{k \in \mathcal{E}(m) \backslash \{t\} \mid m - Pre(t) \geq Pre(k)\}$$
$$nenabl(m, t) = \{k \in (T \backslash \mathcal{E}(m)) \cup \{t\} \mid m - Pre(t) + Post(t) \geq Pre(k)\}$$

The semantics of a TPN is then given over marking-interval pairs (m, ϕ) where m is a marking and ϕ contains the dates at which each $t \in \mathcal{E}(m)$ can be fired in the future. In particular, t may be fired immediately if $0 \in \phi(t)$. The initial marking-interval pair is (m_0, ϕ_0) where m_0 is the initial marking and $\phi_0(t) = I(t)$ for each enabled transition $t \in \mathcal{E}(m_0)$. Starting from (m_0, ϕ_0), only two types of transitions are allowed:

discrete transition: $(m, \phi) \xrightarrow{t \in \mathcal{E}(m)} (m', \phi')$ given that $0 \in \phi(t)$, where $m' = m - Pre(t) + Post(t)$ and ϕ' is defined as follows: $\phi'(k) = \phi(k)$ for all persistent transition $k \in pers(m, t)$ and $\phi'(k) = I(k)$ otherwise.

continuous transition: $(m, \phi) \xrightarrow{\theta} (m, \phi')$ given that θ satisfies the upper bound condition of $\phi \dot{-} \theta$ for each $t \in \mathcal{E}(m)$ (see Sect. 3.2), where ϕ' is defined over the set of enabled transitions at m, $\mathcal{E}(m)$, as follows: $\forall t \in \mathcal{E}(m) : \phi'(t) = \phi(t) \dot{-} \theta$. In particular, if no transition is enabled at m ,that is $\mathcal{E}(m) = \varnothing$, then θ may be any arbitrary value in $\mathbb{Q}_{>0}$ which allows time to diverge unboundedly.

Example Figure 3.3 shows a TPN with three places and four transitions. The transition t_{init} is initially fireable and may fire at any moment in the future. Let τ be the value of the global time at the moment t_{init} fires (if it does). Transitions t_0 and t_1 are both enabled (p_0 is marked) but none is already fireable. When τ evolves by one time unit, t_0 becomes fireable as $\downarrow I(t_0) = 1$. If it is fired within the interval $[\tau + 1, \tau + 2[$, p_0 is unmarked, p_2 is marked and both t_0 and t_1 are no longer enabled. However, if t_0 is not fired and τ evolves by one time unit further, t_1 becomes also fireable. Within the interval $[\tau + 2, \tau + 3[$, either t_1 or t_0 *may* fire. At $\tau + 3$, $\uparrow I(t_0)$ is reached and thus t_0 *must* fire or become disabled (that is either t_0 or t_1 *must* fire, which will disable both of them).

Let τ' be the value of the global time at the moment either t_0 or t_1 fires. Firing t_0 will enable t_2 which must fire at exactly $\tau' + 1$. Firing t_1, on the other hand, will enable t_3, which will fire in the interval $[\tau' + 3, \tau' + 5]$.

These temporal constraints allow reasoning on timed properties of the net. For instance, we may prove that the minimum (respect. maximum) amount of time p_0 is marked continuously is 1 (respect. 3) time units. Similarly, the minimum (respect. maximum) amount of time p_0 remains unmarked (after being already marked) is 1 (respect. 5) time units.

Enriching TPN TPN can be conveniently enriched by a number of features enhancing their expressiveness like *priorities*, expressing that some transitions should be favored over others when fireable at the same instant or *data-processing*, consisting of

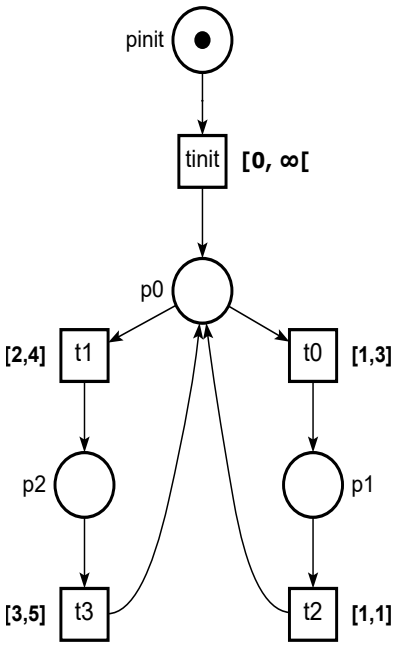


Figure 3.3: Time Petri net example

synchronizing the evolution of the TPN with computations on a set of variables in some programming notation. In this thesis, only the data-processing extension is needed. We will refer from now on to TPN enriched with data as D-TPN.

3.3.2 Fiacre

Introduction Fiacre (for Format Intermédiaire pour les Architectures de Composants Répartis Embarqués, Intermediate Format for the Architectures of Embedded Distributed Components in French) is a formal specification language for describing compositionally both the behavioral and timing aspects of embedded and distributed systems [Berthomieu et al., 2008]. Fiacre is based on communicating state machines with a rich notion of transitions, i.e transitions might embed large sequences of code easing the model mapping from complex applications (*e.g.* a Fiacre single transition may embed different control structures such as conditionals and loops). Timing aspects and firing semantics in Fiacre are identical to the strong time semantics of TPN introduced in Sect. 3.3.1. Fiacre specifications can be used as an input format for formal verification tools (mainly real-time model-checkers) as well as for simulation purposes. Fiacre stems from several projects in different applicative domains like telecoms and avionics [Berthomieu et al., 2014; Bourdil et al., 2014; Rangra and Gaudin, 2014].

Apart from its ability to model priorities and timing constraints (using a dense time model), a distinctive feature of Fiacre is to include a rich set of datatypes: booleans, integers and integer ranges, records, tagged unions, arrays and queues. The language is statically typed, with depth subtyping to handle integer ranges. In terms of process interactions, Fiacre supports both the classical paradigms of shared variables and synchronous message passing à la process calculi. Finally, Fiacre provides functions, native or imported.

```
1   process example is
2   states s0, s1, s2
3
4   from s0
5   select
6          wait [1,3]; to s2
7          [] wait [2,2]; to s1
8   end
9
10  from s1
11  wait [1,5];
12  to s0
13
14  from s2
15  wait [0,1];
16  to s0
```

Listing 3.1: A simple Fiacre process

Fiacre descriptions are made of processes and components, both parametrizable by values, value locations (shared variables) and interaction labels (for communication or synchronization)

Fiacre processes A process describes a sequential behavior; it specifies a set of control states and a set of transitions, each expressing a state change by a statement built from deterministic constructs (assignments, conditionals, loops, and sequential composition), nondeterministic constructs (nondeterministic choice and assignments), interaction statements and jump statements. Listing 3.1 shows an example of a simple Fiacre *process*.
Line 2 defines the states of the Fiacre process. The first state is by default the initial one. Transitions are described afterwards within **blocks**. Each block defines the possible transitions from a given state. For instance, the very first block (lines 4 to 8) describes the existence of two possible transitions (to state **s2** and to state **s1**) from state **s0**. Nondeterminism is expressed via the statement **select** (lines 5 to 8) and timing using the keyword **wait** preceding the time interval associated with each transition. A process can be parameterized by variables and ports as shown in the sequel.

Components Components describe in a hierarchical manner the architecture of the system as the parallel composition of process or component instances. Components also specify the interactions between the constituting processes or components, and possibly constrain these interactions with timing and/or priority requirements. The next paragraphs detail the composition into components via examples.

Example: communication through shared variables Listings 3.2 and 3.3 show two simple Fiacre processes sharing a variable x. **x** is of type **0..3** (an integer ranging from 0 to 3). The blocking statement **on** is used to express guards on transitions (*e.g.* line 5 of Listing 3.2). As long as the guard is false, the guarded transition is disabled. The value of x is updated when firing transitions (*e.g.* line 7 of Listing 3.2).

Now listing 3.4 defines the Fiacre **component** that encapsulates *proc1* and *proc2* and allows them to communicate through x. It declares and initializes the shared variables (line 2) and defines the parallel composition of the processes (lines 4 to 7).

```
process proc1 (&x: 0..3) is
states s0, s1

from s0
on x=1;
wait [2,2];
x:=2;
to s1

from s1
on x=3;
wait [0,1];
x:=0;
to s1
```

Listing 3.2: A simple Fiacre process *proc1*

```
process proc2 (&x: 0..3) is
states s0, s1

from s0
on x=0;
wait [0,0];
x:=1;
to s1

from s1
on x=2;
wait [0,1];
x:=3;
to s0
```

Listing 3.3: A simple Fiacre process *proc2*

```
component CMP is
var x: 0..3:= 0

par
       proc1 (&x)
       || proc2 (&x)
end
```

Listing 3.4: Fiacre component encapsulating *proc1* (listing 3.2) and *proc2* (listing 3.3)

```
1   process sync1 [R: sync] is
2   states s0, s1
3
4   from s0
5   R;
6   to s1
7
8   from s1
9   wait [0,1];
10  to s0
```

Listing 3.5: A simple Fiacre process *sync1*

```
1   process sync2 [R: sync] is
2   states s2
3
4   from s2
5   R;
6   to s2
```

Listing 3.6: A simple Fiacre process *sync2*

Example: communication through ports Listings 3.5 and 3.6 describe two processes *sync1* and *sync2* communicating through the port R. A port has a *profile* which determines the type of messages that can be passed through it. The profile of R is **sync** which is a pure synchronization profile, i.e no data flow is allowed through **R**. The line 5 of both listings is blocking: this means that, when communicating through R with process *sync1*, the self-loop transition of process *sync2* from *s2* to *s2* is not enabled unless process *sync1* is in state *s0*.

Finally, listing 3.7 shows the component *SYNCMP* encapsulating *sync1* and *sync2* and allowing them to interact through R. It defines R, its profile **sync** and a time interval $[1, 1]$ (line 2). This means that the transitions synchronized over R are associated with the interval $[1, 1]$. If the interval is not defined, it defaults to $[0, \infty[$.

Verification Fiacre descriptions can be complemented by declarations of properties. Atomic properties include the states of process instances, predicates on the values of variables and Fiacre events (interactions). The Fiacre observables are boolean combinations of atomic properties. They can be combined to form property patterns in the style of [Dwyer et al., 1999]. For checking real-time properties, these patterns are enriched with time constraints [Abid et al., 2014]. For verification, the real-time patterns are translated by the Fiacre compiler into LTL properties on the Fiacre description instrumented with observers.

As an illustration, we present the timed property "*source leadsto target within* $[d1, d2]$" and show how it is handled in Fiacre. This property asserts that along each path some state obeying *target* occurs within a delay in interval $[d1, d2]$ after each state obeying *source*, where *source* and *target* are some observables and $[d1, d2]$ is a time interval. This property is encoded using a Fiacre process (an observer) given in listing 3.8; the process is automatically generated from the property formula (*source* **leadsto** *target* **within** $[d1, d2]$) and connected with the main Fiacre program through two transition guards on the *source* and *target* observables. With this observer, the property is to show that the state `error` of the observer is unreachable.

```
component SYNCMP is
port R: sync in [1,1]

par * in // consider all interactions
        sync1 [R]
       || sync2 [R]
end
```

Listing 3.7: Fiacre component encapsulating *sync1* (listing 3.5) and *sync2* (listing3.6)

```
process LeadsToWithin is
states idle, start, watch, error
from idle
on source; to start
from start
       wait [d1,d1]; to watch
from watch
select
       on target; wait [0,0]; to idle
unless
       wait ]Δ,...[; to error /* where Δ = d2 − d1 */
end
```

Listing 3.8: Fiacre process for the *leadsto within* property

3.3.3 TINA Toolbox

Introduction TINA [Berthomieu et al., 2004], the TIme Petri Net Analyzer, is a toolbox for the analysis and verification of TPN and D-TPN possibly enriched with priorities and stopwatches. TPN state spaces are infinite due to the dense nature of time. To sidestep this problem, finite abstractions known as *State Classes* have been defined since the 1980's, see for instance [Berthomieu and Menasche, 1983]. The State classes construction, known as the *State Class Graph* (*SCG*), is suitable for LTL model-checking as it preserves markings and traces. A simple variation of the construction (reducing classes by inclusion) only preserves markings and is typically coarser; it is the method of choice for reachability analysis.

The TINA toolbox provides state space generators and offline model-checkers for LTL and modal μ-calculus. The generators can produce compressed representations of SCGs into files. Some classes of properties can also be checked on the fly when building SCGs. When a property is not satisfied, a counterexample is generated. A counterexample can be turned into a timed trace and replayed in a simulator.

Verification of Fiacre models For their verification, Fiacre descriptions are translated into D-TPN by an optimized compiler. The latter, `frac`, performs syntax analysis and type checking, then encodes the description into a D-TPN for TINA preserving its semantics. The compilation process includes a model optimization pass that simplifies redundant transitions, removes dead code and abstracts some variables, retaining only those contributing to the state (unlike e.g. those only used as temporaries). This optimization pass helps reduce the size of the SCG.

The `frac` compiler also translates the properties declared in the description into properties in the format supported by the TINA model checkers. Verifications of Fiacre

properties are then carried out exactly like verification of TINA models properties; in case of failure, a timed scenario can be computed, corresponding to a Fiacre scenario.

3.3.4 Conclusion

Fiacre and TINA provide a modeling and verification workflow based on TPN and their extensions. This makes Fiacre a natural choice as a target formal language for our robotic specifications. Indeed, TPN are a prominent model for the modeling of concurrent and real-time systems and are expressive enough to model the aspects present in such systems. For instance, it is quite simple to model urgencies using the fact that firing intervals depend on the enabledness of the transitions. Furthermore, the different optimized SCG constructions offered by TINA makes it suitable for verifying complex real-time properties of robotic applications. In particular, in our examples, we will often use an optimized state class reduction approach, based on checking the inclusion of classes, that can drastically reduce the size of the generated SCGs and provides an efficient method for checking safety properties.

3.4 UPPAAL

UPPAAL [Behrmann et al., 2004] is a model checker based on *Timed (Safety) Automata* (Sect. 3.4.1) extended with data, user-defined functions and *urgent channels*. An UPPAAL system is made of one or several *processes*, composed using the parallel operator $\|$. To formulate properties, UPPAAL supports a *query language* (Sect. 3.4.3) based on a fragment of the branching logic TCTL. It also features a graphical simulator where counterexamples may be replayed and analyzed. UPPAAL has been developed in a close collaboration between the University of Uppsala (Sweden) and the University of Aalborg (Denmark) and is available, together with its extensions on `http://www.uppaal.org`. UPPAAL is widely used in verification of real-time systems and implements efficient optimization techniques such as minimal cost reachability [Larsen et al., 2001] and symmetry reduction [Hendriks et al., 2004].

3.4.1 Timed Automata

Introduction Timed Automata is a theory for modeling and verification of timed systems. In the original version of the theory [Alur and Dill, 1994], Timed Automata extend *finite-state Büchi automata* with real-valued clocks. The behavior of such automata is therefore restricted by defining constraints on the clock variables and a set of accepting states. A simpler version allowing local invariant conditions and known as *Timed Safety Automata* is introduced in [Henzinger et al., 1994]. In this thesis, we focus on Timed Safety Automata and refer to them as Timed Automata or TA for short.

Formally A timed automaton TA is a tuple

$$TA = \langle L, l_0, X, \Sigma, E, I \rangle \tag{3.1}$$

where:

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,

- X is a finite set of continuous variables called clocks,
- $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions partitioned into inputs (Σ_i) and outputs (Σ_0),
- E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, action label $a \in \Sigma$, and φ is a binary relation on $\mathbb{R}^X$,
- I assigns an invariant predicate $I(l)$ to any location l.

Semantics The semantics of TA is defined over a Kripke structure, whose states are pairs $s = (l, v) \in L \times \mathbb{R}^X$, with $v \models I(l)$, and transitions defined as:

- delay transitions: $((l, v) \xrightarrow{d} (l, v')$ with $d \in \mathbb{R}_{\geq 0}$ and $v' = v + d)$, and
- discrete transitions: $((l, v) \xrightarrow{a} (l', v')$ if there is an edge (l, g, a, Y, l') such that $v \models g$ and $v' = v[Y]$, where $Y \subseteq X$, and $v[Y]$ is the valuation assigning 0 when $x \in Y$ and $v(x)$ otherwise).

Example Fig. 3.4 shows a simple TA with three locations l_0 (initial), l_1 and l_2 and a clock c. With locations l_1 and l_2 are associated the invariants (in purple) $c \leq 2$ and $c \leq 1$, respectively. This means that whenever l_1 (respect. l_2) is reached, it must be left at most when the valuation of c is equal to 2 (respect. 1). The *reset* actions (in blue) assign the valuation 0 to c when the edges they are associated to are taken (edges from l_0 to l_1 and from l_1 to l_2). The guards (in green) must be satisfied when an edge is taken.

The absence of an invariant on location l_0 makes taking its outgoing edge possible no matter what the valuation of c is. Let τ be the value of the global time at the moment the outgoing edge of l_0 is taken. This means that l_1 is reached at τ and must be left within the interval $]\tau, \tau + 2]$. This interval is left-open because each outgoing edge of l_1 is guarded with the strict inequality $c > 0$ (l_1 cannot be left at τ). Let τ' be the value of the global time at the moment the edge from l_1 to l_2 is taken. Location l_2 will be left within $]\tau', \tau' + 1]$ and the initial location is reached.

We may use these temporal constraints (invariants and guards) to assert timed properties of the automaton. For example, we may prove that the maximum amount of time separating two successive visits of location l_0 is 3.

3.4.2 Extending TA

Urgencies TA urgencies may be expressed only locally through invariants. To deal with urgencies expressed globally, *e.g.* involving different TA components, TA are extended with urgencies in [Bornot et al., 1998]. We refer to such formalism as *Urgency Timed Automata UTA*. When an edge in a UTA is *eager*, that we note $\wr$, it *must* be taken (or disabled by taking another edge) as soon as enabled. That is, when an *eager* edge is enabled, time is not allowed to progress until this very edge is taken or disabled.

Data variables To ease the modeling of real-world systems, often communicating through shared variables, TA may be extended with data variables. In such a case, guards and assignments, originally allowed only on clocks (equality/inequality for guards and reset for assignments), become possible on variables as well. We refer to this extension as DTA. UTA extended with data are referred to as DUTA.

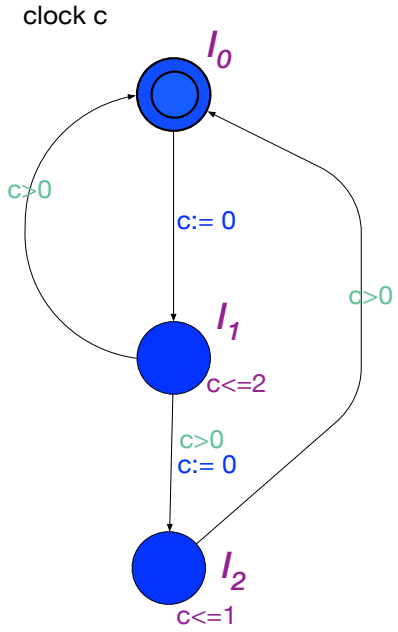


Figure 3.4: Timed automaton example

3.4.2.1 Composition of DUTA

The parallel composition of n DUTA is the system $\{Init\}[\,\|_{i\in 1..n}\, A_i\,]$, where each A_i is a DUTA and $Init$ defines the initial valuations of shared variables.

The semantics of a DUTA composition is thus given over a Kripke structure with states of the form of pairs $s = (l, v)$. The difference with the states given in the semantics of a single (non-extended) TA in Sect. 3.4.1 is that now (i) l stores the current location for each DUTA A_i and the valuation of each non-clock variable in the system, (ii) v stores the valuation of all clocks in the composition (in each A_i) and $v \models I_i$ for all A_i.

The transitions are then defined as in Sect. 3.4.1: discrete and delay. Here, the discrete transitions may contain a set of $\wr$ transitions, such that an $\wr$ transition corresponds to an $\wr$ edge. When enabled, an $\wr$ transition deactivates all delay transitions until it is taken (or disabled by taking another concurrent discrete transition). A large example over DUTA compositions in terms of GenoM3 applications is given in Sect. 5.3.

3.4.2.2 In UPPAAL

UPPAAL offers several extensions of TA to make modeling real-world timed systems easier and more practical. It supports DTA over booleans and integers and a restricted class of UTA (see below). We refer to a (possibly extended) UPPAAL TA as a *process*.

Broadcast channels Multiparty synchronizations are allowed in UPPAAL through (non-blocking) broadcasts. A sender may synchronize with several receivers. A receiver that can synchronize in the current state of the system must synchronize. A sender, on the other hand, that can synchoronize in the current state of the system does

so with the maximum number of receives that can synchronize in the same state. If the sender can synchronize in the current state where no receiver can synchronize, the sender may still execute the sending action, *i.e.* broadcast is never blocking.

Urgent channels UPPAAL implements a restricted class of UTA, where urgencies are allowed only over channels, on which timing constraints are forbidden. That is, one may define an *eager* transition, involving the synchronization of at least two edges in two different processes, but may not define an urgent edge. Moreover, time constraints are not allowed on edges contributing to urgent channels. Despite such restrictions, urgent channel allow implementing urgencies expressed globally, *i.e.* depending on the behavior of more than one processes in the system.

Data and user functions Besides clocks, data variables are also allowed to take parts in guards and actions. Supported data types are booleans and integers. User functions are also supported in a C-like syntax with no support for pointers.

3.4.3 UPPAAL query language

UPPAAL features a query language to express the properties the user wants to verify. The query language is based on a small fragment of the branching timed logic TCTL with path and state formulae.

State formulae A state formula is any expression than can be evaluated to *true* or *false* on a global state of the system. This may involve locations, clocks and data variables, *e.g.* $a == 2$ where a is a global variable, $p.x < 1$ where x is a clock in process p and $p.l_0$ where l_0 is a location of process p. The first formula evaluates to true in all states of the system where a is equal to 2, the second for all states where the valuation of x is inferior to 1, and the third for all states where the current location of p is l_0.

Path formulae Path formulae enable quantifying over traces of the system evolution. Path formulae in UPPAAL use the operators (i) A (along all paths, inevitably), (ii) E (there exists a path, possibly) and (iii) $\rightsquigarrow$ (leads to, inevitably). In the following, we give the five path formulae supported by UPPAAL (ϕ and ψ are state formulae). Note that nesting path formulae is not allowed in UPPAAL.

- $A\Box\phi$: ϕ holds in all states of the system.
- $E\Box\phi$: there exists a maximal path where ϕ is true in all states. The last two formulae are suitable for *safety* properties.
- $E \diamond \phi$: there exists a reachable state that satisfies ϕ. This formula is suitable for *reachability* properties.
- $A \diamond \phi$: ϕ is eventually satisfied.
- $\phi \rightsquigarrow \psi$: whenever ϕ is satisfied, the satisfaction of ψ eventually follows. The last two formulae are suitable for *liveness* properties.

3.4.4 Verification in UPPAAL

Models can be drawn in the graphical editor or encoded in the *.xta* format. Listing 3.9 shows how the TA in Fig. 3.4 is encoded in the *.xta* format. This gives an organized and clear view of the processes when models are too complex to be readable in the graphical format. Therefore, the *.xta* is the format of choice in this thesis.

```
process example () {
clock c;
state l0, l1{x≤2}, l2{x≤1};
init l0;

trans  l0 →l1 {assign x:=0; },
       l1 →l0 { guard x>0; },
       l1 →l2 { guard x>0; assign x:=0; },
       l2 →l0 { guard x>0; };
}
```

Listing 3.9: .xta encoding of the TA in Fig. 3.4

In the *verifier*, one can insert the properties using the allowed path formulae shown in Sect. 3.4.3. The properties are verified *on the fly* and a counterexample is produced if the property is violated. Counterexamples can be replayed in the graphical simulator for diagnosis purposes.

3.4.5 Conclusion

UPPAAL is a state-of-the-art TA model checker that is well known for its user-friendliness and performance. As Fiacre/TINA for TPN, UPPAAL is the model checker of choice for TA. Besides their convenience for real-time and concurrent systems, TA semantics enable a particular ease for modeling and verification of bounded response properties thanks to the clocks. UPPAAL graphical interface allows visualizing the models and especially analyzing counterexamples which eases diagnosis in case of non satisfaction of properties of interest by the model.

3.5 UPPAAL-SMC

UPPAAL-SMC is an extension of UPPAAL based on *stochastic timed automata* (see below). In addition to the classical TA models, UPPAAL-SMC supports modeling and verifying systems the bevahior of which depend on stochastic and non-linear features such as cyber-physical systems with complex dynamics and uncertainties. Moreover, UPPAAL-SMC is a tradeoff approach that does not require the exploration of the whole state space and is thus a promising alternative for large models that do not scale with regular UPPAAL. This comes however at the expense of precision as the truth value of properties is no longer given with absolute certainty.

3.5.1 Stochastic Timed Automata

A stochastic timed automaton (STA) is a tuple

$$STA = \langle TA, \mu, \gamma \rangle \tag{3.2}$$

where:

- TA is a timed automaton (Def. 3.1),
- μ is the set of all density delay functions $\mu_s \in L \times \mathbb{R}^X$, which can be either uniform or exponential distribution,
- γ is the set of all output probability functions γ_s over the Σ_o output edges of the automaton.

The delay density function μ_s over delays in $\mathbb{R}_{\geq 0}$ for each state a is either uniform or exponential distribution depending on the invariant of l of the state s. Let E_l denote the disjunction of guards such that $(l, g, o, -, -) \in E$ for some output o. With $D(l, v) = \sup\{d \in \mathbb{R}_{\geq 0} : v + d \models I(l)\}$ we denote the supremum delay (the least of maximal delays), whereas with $d(l, v) = \inf\{d \in \mathbb{R}_{\geq 0} : v + d \models E_l\}$ we denote the infimum delay (the greatest of minimal delays) before enabling an output. If $D(l, v) < \infty$ then the delay density function μ_s for a given state s is a uniform distribution over the interval $[d(l, v), D(l, v)]$, otherwise it is an exponential distribution with a rate $P(l)$, given by the density function $P(l).exp(-P(l).t)$ for $t \geq 0$ (0 otherwise) where t is the time relative to the output enabledness and exp the exponential function. For every state s, the output probability function γ_s over Σ_o is the uniform distribution over the set $\{o : (l, g, o, -, -) \in E \wedge v \models g\}$ whenever the set is non empty. The stochastic semantics of the distributions and the delay intervals defined by the classic semantics of the underlying TA agree. It is thus easy to show that an STA semantics is defined over a Kripke structure with discrete and continuous transitions (as it is the case for TA semantics).

Example Fig. 3.5 shows an STA that extends the TA in Fig. 3.4. On invariant-free locations (only l_0 here), exponential rates are provided (10 on l_0, in purple). Note that, given the density function, a larger exponential rate implies a higher (respect. lower) probability to leave the location at smaller (respect. larger) time values, which justifies the choice of large exponential rates in our case studies on invariant-free locations (Sect. 7.2.2). We may also define discrete probabilities on the outgoing edges of a given location. Here, the outgoing edges of l_1 have the values $1/3$ (from l_1 to l_0) and $2/3$ (from l_1 to l_2). This means that the probability to take the first (respect. second) edge is $1/3$ (respect. $2/3$).

NSTA and UPPAAL-SMC Under the assumption of input-enabledness, disjointedness of clock sets and output actions, a parallel composition of (composable) STA defines a network of STA (NSTA), as follows: $A1 \parallel A2 \parallel ... \parallel An$. The states of the NSTA are defined as a tuple $s = \langle s_1, ..., s_n \rangle$, where s_j is a state of A_j of the form (l, v), where $l \in L^j$ and $v \in \mathbb{R}^{X^j}$. The probabilistic semantics of NSTA in UPPAAL-SMC is based on the principle of independence between components, where different automata synchronize based on standard broadcast channels. Each component decides on its own (that is, based on a given delay density function and output probability function) how much to delay before outputting and what output to broadcast at that moment. Therefore, only broadcast channels are allowed.

3.5.2 Verification in UPPAAL-SMC

Extension of the *.xta* format The *.xta* format was extended to support probabilities on edges and exponential rates on invariant-free locations. Listing 3.10 shows how the

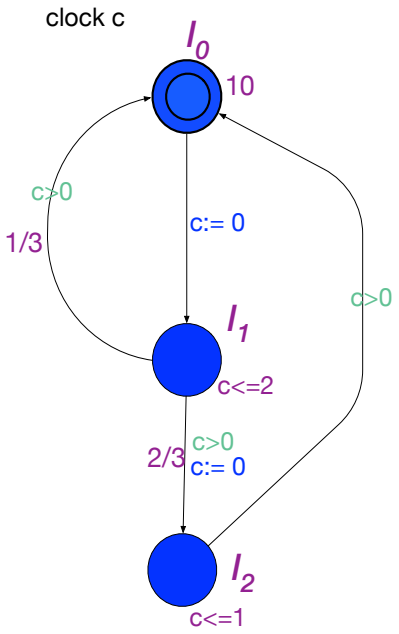


Figure 3.5: Stochastic extension of the TA in Fig. 3.4

STA in Fig. 3.5 is encoded in *.xta* for UPPAAL-SMC. In line 3, the exponential rate for location l_0 is specified. Line 4 introduces a *branchpoint* from which probabilistic edges emerge. That is, for each location sl that is a source of edges with probabilities with target locations tl_0 ... tl_n, we need one branch point sl_b such that non-deterministic edges from sl to tl_0 ... tl_n are replaced with:

- An edge from sl to sl_b, guarded with the common guard to non-deterministic edges (line 8 in our example in listing 3.10),
- and non-deterministic edges from sl_b to tl_0 ... tl_n with the same operations and probabilities of the replaced edges from sl to tl_0 ... tl_n (lines 9-10 in our example in listing 3.10).

The graphical UPPAAL-SMC model of listing 3.10 is shown in Fig. 3.6. Note how UPPAAL-SMC abuses the term "probability" to denote the number of occurrences for each edge, as it computes the probability automatically by dividing that number on the sum of occurrences on all edges, which explains the values 1 and 2 in Fig. 3.6 (instead of $1/3$ and $2/3$ in Fig. 3.5)

Query language UPPAAL-SMC uses a weighted extension of the MITL logic (WMITL) [Bulychev et al., 2012]. It allows to express properties over runs and has the following grammar:

$$\phi ::= s|\neg\phi|O\phi|\phi' U_{x\leq d}\phi'' \tag{3.3}$$

where s is a state formula (Sect. 3.4.3), O (respect. U) is the *next* (respect. *until*) operator and d a bound on the valuation of the clock x. One may notice immediately that "leads" to and bounded response properties cannot be expressed and thus cannot be verified with UPPAAL-SMC.

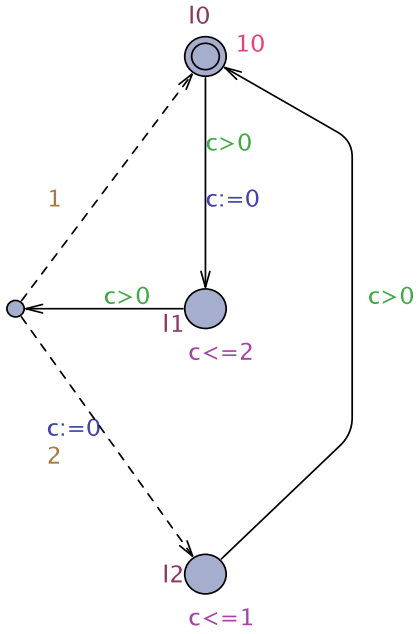


Figure 3.6: UPPAAL-SMC model of the STA in Fig. 3.5

```
1   process example () {
2   clock c;
3   state l0 {;10}, l1{x≤2}, l2{x≤1};
4   branchpoint l1_b
5   init l0;
6
7   trans  l0 →l1 {assign x:=0; },
8          l1 →l1_b {guard x>0; }
9          l1_b →l0 { probability 1; },
10         l1_b →l2 {assign x:=0; probability 2; },
11         l2 →l0 { guard x>0; };
12  }
```

Listing 3.10: .xta encoding of the TA in Fig. 3.5

Probabilistic verification UPPAAL-SMC uses simulation-based algorithms to give and approximate answer to one of the following questions:

- *Hypothesis testing*: is the probability of satisfying ϕ within time $x \leq d$ is greater or equal to a certain threshold p? ($Pr(Op_{x \leq C}\phi) \geq p$), where Op is either $\Diamond$, or $\Box$,
- *Probability evaluation*: what is the probability $Pr(Op_{x \leq C}\phi)$ for some NSTA?
- *Probability comparison*: is $P(Op_{x \leq C}\phi_1) > P(Op_{y \leq D}\phi_2)$?

3.5.3 Conclusion

The cost of SMC algorithms (*e.g.* time and memory consumption) is particularly low compared to classical model-checking algorithms. SMC is thus a technique of choice when models are too large to be handled by model checkers. In contrast, some types of properties are not supported (*e.g.* bounded response) and the results of the verification are given in terms of probabilities rather than boolean answers. SMC might be seen then as a fair compromise between simulation and exhaustive verification.

3.6 BIP

BIP (Behavior, Interaction, Priority) is a component-based language for modeling, executing and analyzing real-time systems developed at *Verimag*[1]. A system is represented by a set of components (behavior) that interact through *connectors* which define weak and/or strong synchronizations (interaction). The conflicts between connectors are possibly resolved using priorities. Complex systems can thus be built hierarchically using *compound* components, that encapsulate sub-systems made of components constrained with connectors and priorities. In this thesis, we use the RT (Real-Time) version of BIP that supports timing constraints over clocks and refer to it as simply BIP. The underlying formalism of BIP is timed automata, already introduced in Sect. 3.4.1.

The building unit of a BIP model is the simplest type of components, *i.e.* with no hierarchies, and is knows as an *atom*. An atom is a DUTA (urgencies and data variables are allowed). *Ports* are used to interact with other atoms and are thus the building blocks of connectors (similar to channels in UPPAAL). Connectors may be *rendezvous* or *broadcasts*. Rendezvous are strong synchronizations (similar to handshake synchronization channels in UPPAAL with the advantage of supporting multiparty interactions). Broadcasts are weak synchronizations similar to broadcast channels in UPPAAL. Connectors may be exported to build other connectors hierarchically. BIP supports the use of external data types and functions for simulation and execution purposes.

Example Fig. 3.7 shows a simple example of three BIP components interacting through connectors with priority rules. Each component has ports that label each edge (*e.g.* the edge from location *idle* to location *busy* in component A is labeled with the port *start*). The little red circle denote that the port is *exported* for strong synchronization (*e.g.* the port *sync* of component $B1$). The black links are the connectors (*e.g.* $s1$ is a *rendezvous* connector that involves the ports *sync1* of A and *sync* of $B1$. The priority pr denotes that the interactions through $s1$ have a higher priority than those possible through $s2$. Note that, in contrast to UPPAAL, BIP offers no support for graphical representation.

[1] BIP documentation and downloads are available at http://www-verimag.imag.fr/BIP-Tools-93.html

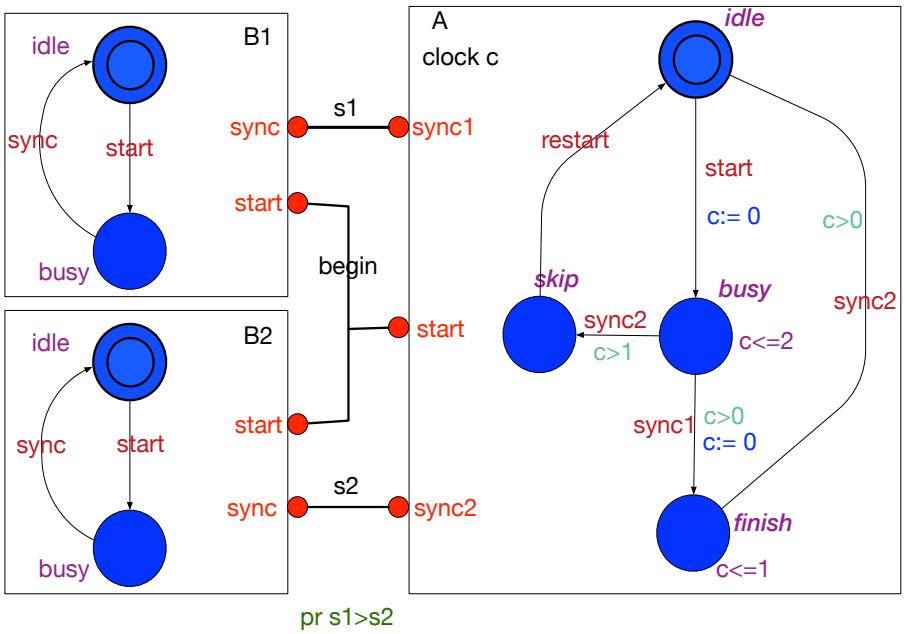


Figure 3.7: A BIP example (graphical)

In the textual description, we need first to define the ports and connectors types. Here the ports are basic without data flow, hence the empty arguments (line 2). The *define* keyword is used to specify the possible interactions through the connector (broadcast or rendezvous, line 5). For instance, if *c_sync2* was a broadcast connector type, p' would be the way to define p as the sender:

```
1   /* port types */
2   port type Basic()
3   /* connector types */
4   connector type c_sync (Basic p, Basic q)
5   define p q
6   end
7   connector type c_sync2 (Basic p, Basic q, Basic r)
8   define p q r
9   end
```

Then, the atom types are defined. We show only the type a for component A. The keyword *provided* (*e.g.* line 19) is for guards and *reset* (*e.g.* line 20) for resetting clocks. After describing the behavior, invariants are defined (lines 34-35):

```
1   /* atoms types */
2   atom type A()
3   clock c
4   port Basic restart()
5   export port Basic sync1()
6   export port Basic sync2()
7   export port Basic start()
8
9   state idle, busy, finish, skip
10
11  initial to idle
12
13  on start
```

```
14              from idle to busy
15              reset c
16
17              on sync1
18              from busy to finish
19              provided c>0
20              reset c
21
22              on sync2
23              from busy to skip
24              provided c>1
25              reset c
26
27              on restart
28              from skip to idle
29
30              on sync2
31              from finish to idle
32              provided c>0
33
34              invariant inv1 at busy when (x ≤2)
35              invariant inv2 at finish when (x ≤1)
36              end
```

Finally, we build the compound (line 2) instantiating the components (lines 3-4) and the connectors (lines 6 to 8) and defining the priorities (line 10). Note that the components $B1$ and $B2$ have the same behavior and that they are instantiated from the same atom declaration (line 4). The : $*$ given after the name of the connectors *s1* and *s2* (line 10) denote all the possible interactions. Thus, the priority *pr* states that any possible interaction through *s1* have a higher priority than any possible interaction through *s2*.

```
1               /* compound definition */
2               compound type example()
3               component a A()
4               component b B1(), B2()
5               /* connectors */
6               connector c_sync s1(A1.sync1, B1.sync)
7               connector c_sync s2(A1.sync2, B2.sync)
8               connector c_sync2 begin(A1.start, B1.start, B2.start)
9               /* priorities */
10              priority pr s1:*>s2:*
```

Now if we analyze the behavior of this model, we may see that, for instance, there is no reachable global state where the current location of A is *busy* and the current location of *B1* (or *B2*) is *idle*. This is due to the *rendezvous* connector *begin*. More interestingly, we may prove that the location *skip* of A is never reached because of the application of priority *pr*.

3.6.1 RTD-Finder

RTD-Finder [Aştefănoaei et al., 2014] is a deductive, compositional verification tool that overapproximates the reachable state space using invariants. It aims to overcome the state space explosion problem often encountered in complex real-time systems. RTD-Finder extends its untimed version D-Finder [Bensalem et al., 2009] with the possibility to reason over time by considering *history clocks* in the generation of the components local invariants. Only TA are supported by RTD-Finder (data variables and urgencies are ignored by the tool).

The compositional verification spirit of RTD-Finder relies on deducing the proper-

ties of a system from the properties of its individual components and their interactions. The history clocks track the time elapsed since a given action has been last performed. History clocks are formulated as constraints and are taken into account when computing the invariants of the components. This permits to, additionally, derive relations between the clocks of different components. The invariants of the components are then complemented with the interactions invariants, computed as in D-Finder, to obtain an overapproximation of the reachable state space of the system.

RTD-Finder supports only safety properties, expressed as invariants themselves. For instance, the safety property *location skip of A is never reached* from the example in Fig 3.7 is expressed as follows: $\neg A1.skip$. The evaluation of a safety property is conducted on the global invariant of the system using SAT solving techniques. For this purpose, the latest version of RTD-Finder features an interface with the state-of-the-art SMT solver Z3 [De Moura and Bjørner, 2008].

3.6.2 The BIP Engine

The back-end compiler of BIP generates source code in C++ for execution purposes. The *engine* ensures a correct execution of the generated source code following the semantics of BIP. It computes execution sequences of the underlying model. In general, an executable model is created by linking a C++ representation to the runtime of the engine. The engine computes execution sequences with respect to the target platform (concrete execution) or the host machine (simulation). In the first case, the engine ensures the correct connection between the model and the hardware with respect to the latter's inputs/outputs and time constraints. In the second case, the engine interprets time logically.

Since the underlying BIP models are formally specified and follow DUTA semantics, and that the actual code of the real system can be executed by the engine, the engine may replace the program of the real system. This may be any system that runs C or C++ code. Using the engine makes it possible to (i) monitor the execution of the system online with regard to *e.g.* its timing constraints and (ii) augment the model with properties that the engine will enforce online. The BIP engine is thus the framework of choice for runtime monitoring and verification. However, these advantages come at the cost of relatively average performance that hinders using the engine for applications running at high frequencies, such as the quadcopter.

3.6.3 Conclusion

The verification technique implemented in RTD-Finder, based on an overapproximation rather than an exact exploration of the reachable state space, makes it a valid candidate for the verification of our models whenever they do not scale or induce a high cost with model checking. In contrast, RTD-Finder supports only safety properties and handles neither urgencies nor data variables, which reduces, respectively, the verification feasibility and the convenience of modeling.

The BIP engine offers an efficient environment for runtime monitoring and enforcement of properties online. In particular, the latter is advantageous when the properties of interest cannot be verified offline because *e.g.* they involve data that the offline model, at its abstraction level, cannot capture. In addition, runtime enforcement of properties does not require exploring the whole state space at once and is thus generally scalable. In contrast, the complex computations that need to be performed continuously by the BIP engine may have a non negligible side effect on the resource consumption

in the robotic platform.

3.7 Conclusion

In this chapter, we introduced the formal frameworks that will be used for the automated modeling and verification of our robotic specifications. These frameworks use different techniques and come with various features. We rely therefore on their respective advantages to use them in an efficient and complementary manner in order to respond to our verification needs. In sum, four major methods are available:

- Model checking: with UPPAAL and Fiacre/TINA. The respective strengths of these tools (reduction-by-inclusion technique and automated observers for Fiacre/TINA, overall performance and user-friendly diagnosis for UPPAAL) offer guidance on using them efficiently depending on the properties and the complexity of the robotic application.
- Invariant-based verification: with RTD-Finder. The main advantage of this method (theoretically more scalable than model checking) calls for experimenting RTD-Finder with safety properties when models do not scale or entail significant cost with model checking.
- Statistical model checking: with UPPAAL-SMC. The support for stochastic behavior motivates the extension of GenoM3 with probabilities over non-deterministic transitions. Moreover, the low cost of SMC algorithms makes of UPPAAL-SMC an alternative to consider when models scale with neither model checking nor invariant-based verification.
- Runtime enforcement of properties: with the BIP-engine. The (online) partial exploration of the state space allows a scalable enforcement of desired properties online. Also, the fact that the engine runs the actual code on the robotic platform makes it possible to express properties that depend on low-level data (absent in the offline model).

Chapter 4

Formalizing GenoM3

4.1 Introduction

In this chapter, we propose a formalization of GenoM3 components. We first motivate the need of such formalization and its feasibility. We give afterwards the formal definitions of a simplified version of a GenoM3 component where the most important constituents and mechanisms are taken into account. Finally, we derive from these definitions the operational semantics, given in TTS (Sect. 3.2), based on the behavior described informally in Sect. 2.2.3.

4.2 Importance and feasibility

As seen in Chapt. 1, the absence of formal semantics in low-level robotic frameworks is quite problematic. Indeed, it is mostly cumbersome and error-prone to try to model robotic specifications, specified within informal frameworks such as ROS, in formal languages and frameworks. Furthermore, computer science is a mathematically founded discipline where, for instance, formal semantics is at the heart of programming languages. Formal semantics gives a clear, unambiguous definition to the language/software contrary to informal descriptions that might be interpreted differently by different readers. In the case of robotics, such semantics would make it possible to soundly translate robotic specifications into various formal frameworks. Indeed, since the translated specifications obey some formal semantics, it is possible to construct a proof of soundness between the semantics and the translation (Chapt. 5).

GenoM3 is amenable to formalization due, mainly, to its model-based nature. Indeed, the definition of the entities a GenoM3 component may have is clear and finite. For instance, we know that each GenoM3 component has one control task, and we know how it evolves. We also know that a GenoM3 component may have a finite number of activities, and that each activity has a finite number of codels, and the evolution rules of an activity within its execution task are well defined. Overall, there is a finite set of rules on what the programmer may define (definitions) and how the component evolves (operational semantics) in GenoM3. This makes the formalization of GenoM3 possible by carefully mapping each entity and rule into TTS.

We give formal definitions and operational semantics of a *lightweight* version of GenoM3. This version preserves the most important mechanisms including concurrency, mutual exclusion, activation and interruption of activities. Validate codels, con-

trol services and aperiodic execution tasks are excluded. These choices permit giving in-depth insights on semanticizing ambiguous, yet crucial, software aspects of GenoM3 (such as interruptions). At the same time, the formalization is not overloaded with simpler and clearer mechanisms such as the execution of control services. For simplicity, we abuse notation to make the term *codel* refer, from now on, interchangeably to the codel or the state it is associated to, and bears always the name of the state rather than the function it calls upon execution.

4.3 Syntax and syntactical restrictions

Let $Comp$ be a GenoM3 component. We define hierarchically the constituents of $Comp$:

4.3.1 Activities

An activity A is a tuple

$$\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$$

where:

- ID_A is the unique activity name,
- C_A is a set of codels with at least two codels (the entry codel $start_A$ and the final codel $ether_A$):
$$\{start_A, ether_A\} \subseteq C_A.$$
An activity may also have a "stop codel", $stop_A$, that defines the code to be executed when the activity is interrupted,
- $W_A : C_A \backslash \{ether_A\} \mapsto \mathbb{Q}_{>0}$ is a function that associates to every codel its WCET (Sect. 2.2.3). We do not define a WCET for the codel $ether$, reserved for termination only (there is no code attached to it),
- T_A is a set of transitions of the form $c \to c'$ where c and c' are codels in C_A. We denote such a transition by simply $c \to$ (or $\to c'$) when the identity of codel c (or c') is unimportant,
- $T_A^P \subseteq T_A$ is the set of *pause* transitions.

4.3.2 Execution task

An execution task ET is a tuple

$$\langle Per, \mathcal{A}, Inc, V \rangle$$

where:

- $\mathcal{A}$ is the non-empty set of activities ET is in charge of. We use the notation $ID_{\mathcal{A}}$ to refer to the set $\bigcup_{A \in \mathcal{A}} ID_A$ of all IDs of activities in $\mathcal{A}$,
- $Per \in \mathbb{Q}_{>0}$ is the period,

- Inc is the *incompatibility function* that maps the ID of each activity in $\mathcal{A}$, say ID_A, to the set of activities in $\mathcal{A}$ that are incompatible with A, that is the activities that must be interrupted before A is launched. Therefore:
  $Inc : ID_{\mathcal{A}} \mapsto P(ID_{\mathcal{A}})$, where $\mathcal{P}(S)$ denotes the powerset (the set of all subsets) of S,
- V is a set of variables.

4.3.3 Control task

A control task CT is a specific task dedicated to the interaction between a component and its surrounding. It is also responsible for "marking" an activity as ready for execution or for interruption, and reports on the termination of activities. In GenoM3, the user does not specify the control task whose behavior is defined implicitly. Therefore, a control task is only defined at this level by a set of local variables that we call V.

4.3.4 Component

A component $Comp$ is a tuple $\langle CT, E, V, \mu \rangle$ where:

- CT is a control task,
- E is a set of execution tasks,
- V is a set of variables (shared between CT and each ET in E).
- $\mu : C \mapsto \mathcal{P}(C)$ is the mutual exclusion function, where C is the union of all the codels in all activities of all execution tasks in E. Informally, the set $\mu(c)$ lists the set of codels that cannot simultaneously execute with c. In the remainder of this document, codels c such that $\mu(c) = \varnothing$ are referred to as *thread safe*. Otherwise we say that c is *non thread safe*.

4.3.5 Application and well-formed specifications

An application, denoted App in the rest of the text, is simply a set of components.

We will only consider *well-formed* applications, that are defined by syntactic restrictions on the activities and execution tasks that they include.

First, we require that each codel in an activity A, excluding $ether_A$, must have at least one successor in the relation defined by the set of transitions T_A. More formally, for any activity A and codel c in $C_A \setminus \{ether_A\}$, there must be a transition of the form $c \rightarrow c'$ in T_A, with $c' \in C_A$.

Second, we require that a transition in T_A must not involve a *stop* codel as a target. Indeed stop codels are reserved for interruptions. Similarly, it cannot involve an *ether* codel as its source, since *ether* is reserved for termination. Also, an *ether* codel cannot be the target of a *pause* transition because the latter is for suspension until the next period, while the former is for termination.

All the previous requirements can be expressed more succinctly with the following constraints:

$$\forall c \in C_A \backslash \{ether_A\}\, \exists c' \in C_A : c \rightarrow c' \in T_A$$
$$\forall c, c' \in C_A : (c \rightarrow c' \in T_A) \Rightarrow (c \neq ether_A \wedge c' \neq stop_A)$$
$$\forall c, c' \in C_A : (c \rightarrow c' \in T_A^P) \Rightarrow (c' \neq ether_A)$$

Finally, *ether* codels are always thread safe (there is no code attached to them, Sect. 4.3.1). Also, there must be no mutual exclusion between codels of activities that belong to the same execution task. Indeed, any two activities A and B in the same execution task are executed sequentially "by construction" (no activities in the same task can run concurrently). Therefore we require that $\mu(c) \cap C_B = \mu(c') \cap C_A = \varnothing$ for all c in C_A and c' in C_B.

4.4 Semantics of lightweight GenoM3

The operational semantics of GenoM3 entities is given in terms of TTDs that are composed together to build components and applications. Then we can derive a notion of reduction on GenoM3 by lifting the corresponding relation at the TTS level (Sect. 3.2.3). As a consequence, we can define the behavior of GenoM3 components independently from the implementation (in accordance with the informal description given in Sect. 2.2.3). In the next section, we refine the operational semantics by defining a more precise notion of actions.

Here, we need to distinguish between what the programmer specifies (which is reflected at the syntactical level, for instance in transitions between codels declared in activities, Sect. 4.3), and what is implicitly specified, that is, enforced at execution to produce the expected behavior, like for instance interruption transitions (to codel `stop` if it exists). Indeed, the programmer does not specify transitions to the `stop` codel, if it exists (Sect. 4.3.5), as such transitions are defined by default and automatically executed when applicable (Sect. 2.2.3). We define the semantics of a GenoM3 component gradually through three levels:

- Mono-task component: the component contains only one execution task (no control task),
- Multi-task component: the component contains a finite number of execution tasks (no control task),
- All-task component: the component contains a finite number of execution tasks and a control task.

This layering will help us present the semantics progressively, in an understandable way, but also select the right level according to the objective (presentation, translation, proof) such as both readability and convenience are preserved (more in next section).

4.4.1 Level 1: mono-task component

This is the lowest level in complexity (and highest abstraction). In this context, the component contains only one execution task, which means that all the codels are thread safe (see the property of $\mu()$ in Sect. 4.3.5).

For the sake of simplicity, we stop referring to the names of edges in TTDs (Sect. 3.2.4). That is, an edge e that connects vertex v to v', denoted also $v \xrightarrow{e} v'$ in Sect. 3.2.4 will be referred to, from now on, as simply $v \rightarrow v'$, $v \rightarrow$ (when the identity of v' is uninmportant), or $\rightarrow v'$ (when the identity of v is unimportant). This will alleviate the notations but still permits to define edges uniquely through their source and target vertices and the set they belong to as we will see hereafter. It will also ease loading edges with *actions* in Sect. 5.3.

Definition 1 ***Activities semantics.***
The operational semantics of an activity $\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$ *(Sect. 4.3.1) is given by a TTD (Sect.3.2.4) such that:*

- *Vertices* V*: each* $c \in C_A$ *is mapped to one vertex with the same name* $c \in V$*. The initial vertex* v_0 *is* $ether_A$*.*
- *Edges* E *are partitioned into a set of* nominal edges*,* E^N*, and additional edges,* E^A*. That is* $E = E^N \cup E^A$ *where:*
  - *Nominal edges: each transition* $c \rightarrow c'$ *in* T_A *is mapped to an edge* $c \rightarrow c'$ *in* E^N*. We distinguish three disjoint sets of nominal edges:* $E^N = E^P \cup E^T \cup E^X$*.* E^P *is the (possibly empty) set of pause edges that maps the set of pause transitions* T^P*;* E^T *is the (possibly empty) set of termination edges of the form* $\rightarrow ether$ *and* E^X *the (possibly empty) set of the remaining (execution) edges.*
  - *Additional edges: We distinguish two disjoint sets of additional edges:* $E^A = E^S \cup E^I$*.* E^S *contains the additional edge for starting* $ether \rightarrow start$*.* E^I *is the set of additional edges for interruption: (i) from vertex* $c = ether$ *and (ii) from each vertex* c *such that there is an edge* $\rightarrow c$ *in* E^P *to vertex* $stop_A$ *if* $stop_A \in C_A$ *(to vertex* $ether_A$ *otherwise).*
- *Time intervals I:* $I =]0, W(c)]$ *for each edge in* E^N *and* $I = [0, 0]$ *for each edge in* E^A*.*

Consequently, the set of nominal edges maps the transitions that the programmer specifies, while the set of additional edges reflects internal actions enforced by GenoM3 to handle starting and interruption of activities. The additional edges for interruption E^I ensure that an activity that is interrupted before starting or after a pause will execute the interruption routine: transit to `stop` (if it exists) or terminate by transiting to `ether` (otherwise).

Edges uniqueness For activities, due to the restrictions defined in Sect. 4.3.5, the sets E^N and E^A are necessarily disjoint, and thus all subsets of E^N and E^A are mutually disjoint. Moreover, it is not possible to have two different edges with exactly the same source and target codel, so specifying the source and target of an edge defines it uniquely. The only exception is for codels c that are both the target of a pause transition ($\exists \rightarrow c \in T_A^P$) and the source of a transition to *ether* ($c \rightarrow ether_A \in T_A$) in an activity that does not have a *stop* codel. In this case, we end up with two edges connecting c to $ether_A$: one nominal for termination (in the set E^T) and one additional for interruption (in the set E^I). Here, it is sufficient to mention also to which set the edge belongs to define it uniquely. For TTDs of other entities excluding the control task (such as the task *manager*, see below), edges are uniquely defined through their source and target vertices. This remains true at *level 2* and *level 3* (next sections).

Example This example shows the definition of an activity A and its operational semantics.
Syntactic definition (from Sect. 4.3.1)

- $C_A = \{start_A, main_A, ether_A\}$,
- $W_A(start_A) = 1$, $W_A(main_A) = 2$,

- $T_A = \{start_A \rightarrow main_A, main_A \rightarrow ether_A\}$,
- $T_A^P = \varnothing$.

Semantics We apply **Definition 1** to A to get the TTD of A in Fig. 4.1. Note the edge from $ether_A$ to $ether_A$ that represents interruption (absence of codel $stop_A$ here).

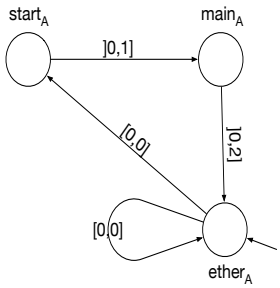


Figure 4.1: Activity TTD example (mono-task context)

Definition 2 ***Execution task semantics.***
The semantics of an execution task $ET = \langle Per, \mathcal{A}, Inc, V \rangle$ *is a TTS (parallel composition, Sect.3.2.5)*

$$ET = \{\Theta\}[Tim||Ex]$$

where Θ *gives the initial values of the shared variables (given below) and* Tim *is the timer.*

Ex is a TTS (sequential composition, Sect.3.2.6)

$$\{\Theta\}[M||(\underset{A\in\mathcal{A}}{||} A)]$$

where M *is the task manager and* $\underset{A\in\mathcal{A}}{||} A$ *is the sequential composition (Sect.3.2.6) of all activities* A *in* $\mathcal{A}$ *(Sect. 4.3.2).*

The set of variables V *contains:* N*, the set of names of activities to be executed nominally,* R*, the set of names of activities to be interrupted (both* N *and* R *are defined over* $ID_{\mathcal{A}}$*, Sect. 4.3.2), sig, the period signal (boolean), and* Π*, the control passing variable (of type* $ID_{\mathcal{A}} \cup M$*, the same idea as in Sect.3.2.6). The initial values are* $\Theta(N) = \Theta(R) = \varnothing$*,* $\Theta(sig) = False$*, and* $\Theta(\Pi) = M$*.*

Π is initialized to M to ensure that the manager has the control when the system starts (the global control is held by the *manager* M at the initial state of the underlying TTS). Both Tim and M are TTDs whose behavior is defined in the sequel.

Definition 3 ***Timer semantics.***
The timer has one vertex and one edge. The latter is associated with the interval $[Per, Per]$ *and the operation* $sig := true$ *(Fig. 4.2).*

Changing the value of *sig* to *true* corresponds to transmitting a signal asynchronously to the *manager* (see below). The time interval $[Per, Per]$ ensures that this signal is transmitted at exactly each period (each Per time units).

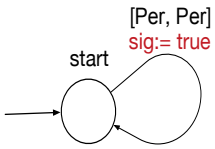


Figure 4.2: Timer TTD

Definition 4 ***Manager semantics.***
The manager is a TTD with two vertices: wait and manage. The edges, guards, operations and time intervals are shown in Fig. 4.3.

The location *wait* denotes waiting for the next period signal and *manage* is to execute activities, if any. The union $N \cup R$ defines the set of activities to execute through their IDs. The operation $\Pi := rand(N \cup R)$ gives the global control randomly to one of the activities whose ID is in $N \cup R$ (by assigning randomly an element from $N \cup R$ to Π). The manager transits back to *wait* as soon as the set defined by this union is empty.

Since $\Theta(N) = \Theta(R) = \varnothing$, no activity would ever be executed by the manager. This is normal because fulfilling activities requests is the role of the control task that we do not have at this level. The manager performs the operation $rrand(N, R)$ to solve this problem. It initializes randomly N and R, over the set of IDs of the activities ET is in charge of; while respecting the disjointness condition $N \cap R = \varnothing$ and the uniqueness condition $(ID_A \in S \wedge ID_B \in S) \Rightarrow (A \neq B)$ with S is either N or R. Note how the guard on the edge from *wait* to *manage* does not contain the clause

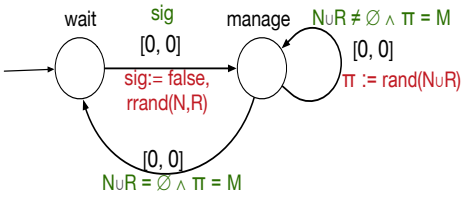


Figure 4.3: Manager TTD

$\Pi = M$. This is indeed not necessary as we may easily prove that if the *manager* is at vertex *wait*, then $\Pi = M$ (this kind of invariants will be useful when we prove the soundness of our translation in the next section):

- First visit: $\Theta(\Pi) = M$,
- Subsequent visits: each subsequent visit results from taking the edge from *manage* to *wait*, which is itself guarded by $\Pi = M$ and does not modify Π,
- Time progress: all operations that change Π from M to something else are on the edges whose source vertex is *manage*, which means that the value of Π when reaching *wait*, proven above to be M, will remain so as long as the current vertex of *manager* is *wait*. Activities have also access to Π but never change it to something else than M (see below).

Now we see how the TTD of an activity A given in **Definition 1** is enriched with guards and operations when involved in the execution task.

Definition 5 ***Activities semantics (enriched).***
Each incoming edge to $ether$ (each element of E^T if a codel stop exists, of $E^T \cup E^I$ otherwise) and each pause edge (each element of E^P) is augmented with the operation $\Pi := M$ and the operation $UP(ID, N, R)$ that removes ID from N or R, whichever set it belongs to. Additional edges for interruption (E^I) are guarded with $\Pi = ID \wedge ID \in R$. The starting edge (the only element of E^S) and each edge $c \rightarrow$ in E^N such that there exists an edge $\rightarrow c$ in E^P are guarded with $\Pi = ID \wedge ID \in N$.

Let us illustrate with an example how this augmentation with guards and operations coincides with the behavior given in Sect. 2.2.3. We consider the same activity A (Fig. 4.1) and a second activity B defined as follows:

- $C_B = \{start_B, main_B, stop_B, ether_B\}$,
- $W_B(start_B) = 1$, $W_B(main_B) = 2$, $W_B(stop_B) = 1$,
- $T_B = \{start_B \rightarrow main_B, main_B \rightarrow main_B, stop_B \rightarrow ether_B\}$,
- $T_B^P = \{main_B \rightarrow main_B\}$.

We apply **Definition 5** to get the TTDs of A and B in Fig. 4.4 when evolving within the execution task whose manager and timer are represented in Fig. 4.3 and Fig. 4.2 (**Definition 4** and **Definition 3**), respectively. The non-determinism on whether to execute nominally or interrupt at the beginning of the execution (from $ether$ or wherever the activity was paused) is resolved by finding to which set the activity ID belongs (*e.g.* edges from $main_B$ to $stop_B$ and from $main_B$ to $main_B$). At the end of the execution, either by taking a $pause$ edge (*e.g.* edge from $main_B$ to $main_B$) or reaching $ether$ (*e.g.* edge from $main_A$ to $ether_A$), the control is given back to the $manager$ through the operation $\Pi := M$. Together with such operation, the activity updates the set N or R by removing its ID from the set it belongs to through the operation $UP(ID, N, R)$. This is to denote that there is no further execution required for this activity in the current cycle. Note that checking whether the activity has the control is necessary only on starting and interrupting edges and when resuming after a pause (**Definition 5**) as we may easily prove that when activating any of the remaining edges, Π is always equal to the activity ID.

Component At this level, the component is simply the execution task ET. It is thus derived from **Definition 2**.

4.4.2 Level 2: multi-task component

At this level, the component may contain several execution tasks, which means that some codels may be non thread safe. Only the operational semantics of activities change.

Definition 6 ***Activities semantics (level 2).***
The operational semantics of an activity $\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$ (Sect. 4.3.1) is given by a TTD such that:

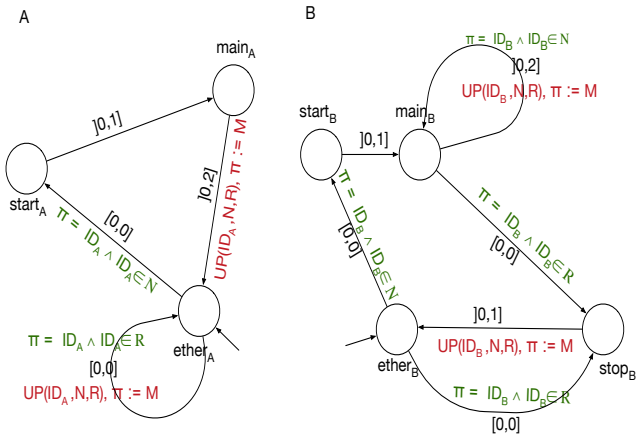


Figure 4.4: Activities A and B in task ET (level 1)

- *Vertices V: each $c \in C_A$ s.t. $\mu(c) \neq \varnothing$ (non thread safe) is mapped to two vertices c and c_{exec}.* ***Definition 1*** *applies otherwise.*
- *Edges E: partitioned into nominal edges E^N and additional edges E^A:*
  - *Nominal edges E^N: partitioned into E^P (pause edges), E^T (termination edges) and E^X (execution edges). Each transition in $T_A \backslash T_A^P$ from a non-thread-safe codel c to c' is mapped to an edge $c_{exec} \rightarrow c'$ in E^X (in E^T if $c' = ether$). Each transition in T_A^P from a non-thread-safe codel c to c' is mapped to an edge $c_{exec} \rightarrow c'$ in E^P. For the remaining transitions in T_A,* ***Definition 1*** *applies to get their mapping in E^N,*
  - *Additional edges E^A: partitioned into E^I (interruption edges), E^S (starting edges) and E^M (mutual exclusion edges). E^M is the set of edges $c \rightarrow c_{exec}$ for all non-thread-safe codels c.* ***Definition 1*** *applies to get E^S and E^I.*
- *Time intervals:* ***Definition 1*** *applies on all edges.*

The manager and the timer remain unchanged (**Definition 3** and **Definition 4**). Now we see how the activities at this level are enriched when involved in ET.

Definition 7 ***Activities semantics (enriched, level 2).***
Definition 5 *applies. Then, each additional edge $c \rightarrow c_{exec}$ in E^M is guarded with $Fr(c) \wedge \Pi = ID \wedge ID \in N$ if there exists an edge $\rightarrow c$ in E^P ($Fr(c)$ otherwise), such that $Fr(c)$ is true if and only if c'_{exec} is not the current vertex of its activity (in the global state of the underlying TTS) for all c' in $\mu(c)$.*

The guard $Fr(c)$ is to ensure no two codels sharing some resources run simultaneously. It is implementable through *e.g.* shared variables (see example in [Foughali et al., 2016], section 6.1).

Example Let us consider the same activities A and B from the previous level semantics (Sect. 4.4.1). The behavior is the same, but some codels become non thread safe due to the existence of other execution tasks:

- Activity A: The codel $main_A$ becomes non thread safe ($\mu(main_A) \neq \varnothing$).
- Activity B: The codel $main_B$ becomes non thread safe ($\mu(main_B) \neq \varnothing$).

Applying **Definition 6** then **Definition 7** to A and B give the TTDs in Fig. 4.5.

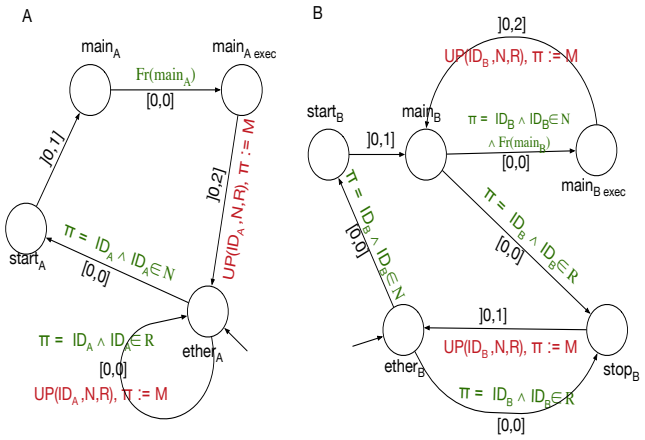


Figure 4.5: Activities A and B in task ET (level 2)

Component At this level, the component is the TTS

$$Comp = [E]$$

where $E = \|_{i \in 1..n}\ ET_i$ is the parallel composition of all execution tasks in the component $Comp$.

4.4.3 Level 3: all-task component

Definition 8 ***Control task semantics.***
The semantics of a control task (Sect. 4.3.3) is given over the TTD in Fig. 4.6 where: $rec(ID)$ evaluates to true when an activity ID is received, $Insert(ID, Wa)$ inserts the received ID in the local variable (a set) Wa, and report is the operation of reporting to external entities.

Requesting an activity, denoted by the guard $rec(ID)$ (ID received), triggers an urgent edge from the initial vertex $idle$ to the vertex $busy$. The received activity name is inserted in the set Wa, which is an initially empty local variable denoting the names of the activities waiting for activation (it is the only element of V given in Sect. 4.3.3). Another possible edge with the same source and target vertices is triggered when an activity finishes its execution (the guard will be formalized later). The edge from $busy$ to end includes the operations of *e.g.* interruption and activation, which will be given later. The edge from end to $idle$ corresponds to sending replies through the operation *report* to external entities.

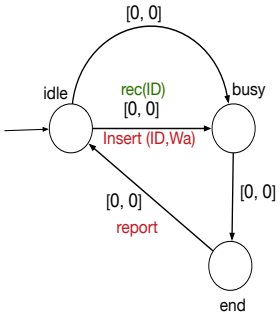


Figure 4.6: Control task TTD

Definition 9 ***Component semantics.***
a GenoM3 *component is a TTS*

$$Comp = \{\Theta\}[CT \parallel E]$$

where CT is a control task, and $E = \|_{i \in 1..n} ET_i$ is the TTS resulting from the parallel composition of all the n execution tasks ET in component $Comp$ (CT and E are the operational counterparts of CT and E in Sect. 4.3.4), and Θ gives the initial values of the shared variables (from V in Sect. 4.3.4). These shared variables are: Act the set of activated activities, In the set of interrupted activities and Fi the set of finished activities. $\Theta(Act) = \Theta(In) = \Theta(Fi) = \varnothing$.

Act and In are modifiable only by CT that determines who is activated and who is interrupted (read-only for E) and Fi is modifiable by everyone (both activities and control task need to update it).

Now, the execution tasks and control task diagrams will be enriched with operations over shared variables to ensure a correct behavior within $Comp$ (with regard to that given informally in Sect. 2.2.3). Within an execution task, the manager (**Definition 4**) and the activities (**Definition 7**) will be enriched as follows:

Definition 10 ***Manager semantics (level 3).***
*On the edge from $wait$ to $manage$ (**Definition 4**), N and R are copied from Act and In, respectively (instead of randomization). Only the names of the activities that this execution task is in charge of (i.e. activities members of $\mathcal{A}$, Sect. 4.3.3) are copied, excluding those in Fi. That is, for task ET, the restricted copy of Act into N results in the set $N = \{ID_A | A \in \mathcal{A} \wedge A \in Act \wedge A \notin Fi\}$ (and similarly when copying In into R). We denote this operation by $rcopy$ (restricted copy), see Fig. 4.7.*

The restricted copy eliminates possible infinite execution scenarios (the execution task makes the copy once, the activities activated afterwards will be processed at the next period). Excluding the elements in Fi when copying ensures that already terminated activities will not be re-executed (unless requested again in the future).

Definition 11 ***Activities semantics (level 3)***
*The enriched TTD is obtained from **Definition 7**. Then, on each incoming edge to $ether$ (each element of E^T if a codel $stop$ exists, of $E^T \cup E^I$ otherwise), a new operation that inserts the activity ID in the set Fi is added.*

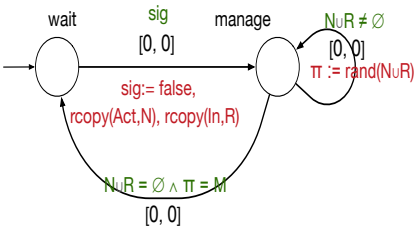


Figure 4.7: Manager TTD (level 3)

This new operation will notify the control task to act accordingly on the termination of the activity (see below). Applying **Definition 11** to activities A and B (Fig. 4.5) gives the TTDs in Fig. 4.8.

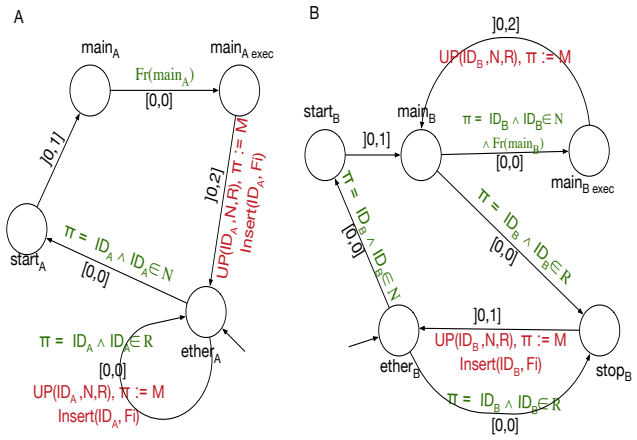


Figure 4.8: Activities A and B in task ET (level 3)

Definition 12 ***Control task semantics (enriched).***
*The control task (**Definition 8**) is enriched as follows: the edge $idle \rightarrow busy$ (not guarded with $rec(ID)$) is guarded with a non-emptiness condition on Fi (Fig. 4.9). The edge $busy \rightarrow end$ is associated with the following operations (in this order):*

- *update Act and In by removing the IDs in Fi:*
  $Act := Act \backslash (Act \cap Fi)$ (and same for In)
  We refer to this operation as $U()$,
- *empty Fi,*
- *Activate and interrupt: move elements of Wa to Act if possible (and from Act to In if necessary):*
  $\forall id \in Wa$:
  ***if** $Inc(id) \cap (Act \cup In) = \varnothing$ then*
  $Wa := Wa \backslash \{id\}$ and $Act := Act \cup \{id\}$ (activation)
  ***else if** $Inc(id) \cap Act \neq \varnothing$ then*

$In := In \cup (Act \cap Inc(id))$ *and* $Act := Act \backslash (Act \cap Inc(id))$ *(interruption).*
We refer to this operation as $A\_I()$.

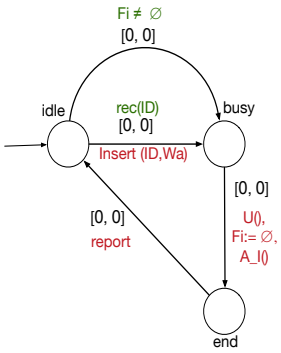


Figure 4.9: Control task TTD (enriched)

The guard $Fi \neq \varnothing$, combined with the urgency interval $[0, 0]$ (the edge $idle \rightarrow busy$ not guarded with $rec(ID)$), allows the control task to update the sets Act and In as soon as an activity ends. The operation $U()$ on the edge $busy \rightarrow end$ ensures that this update is correct by removing the ended activities from the sets of activities to be executed ($Act \cup In$). The operation $A\_I()$ activates the waiting activities if possible. That is, for each waiting activity A (in Wa), it checks if there is at least an activity incompatible with it that is still not terminated (in $Act \cup In$). If it is the case, then A needs to wait further (remains in Wa) and the incompatible activities with A that are not interrupted (in Act) need to be moved to In. Otherwise, A is activated (moved from Wa to Act). After these operations, the control task reports to the external entity that requested the finished activities (if any, edge $end \rightarrow idle$).

4.4.4 Application

A robotic specification written in GenoM3 contains usually several components. We give thus the definition of a robotic application in terms of operational semantics. We can apply this at any level, which gives us different views of an application at different levels of abstraction. Note that the data flow through *ports* is not specified at this level as its mechanisms depend on the implementation [Foughali et al., 2018].

Definition 13 ***Application semantics.***
An m*-component specification is the TTS resulting from the parallel composition of all components*

$$app = [||_{i \in 1..m}\ Comp_i]$$

.

4.5 Conclusion

In this chapter, we formalize a lightweight version of GenoM3 in TTS. The formal definitions give an unambiguous characterization of the most complex GenoM3

constituents, namely activities, execution and control tasks. We thus tackle the most delicate mechanisms such as interruption and communication between tasks while abstracting away less delicate aspects like the execution of control services. This makes our semantics both understandable and extendable. Indeed, this semantics is amenable to enriching with the discarded, less complex entities/aspects with a minimal effort. We provide thus a sort of abstract syntax that, despite helping practitioners grasp the notion of components and their ingredients, defines the attributes on which operational semantics are built. That is, each abstract element in a tuple has an operational meaning that helps define the behavior of the global system. The work on semanticizing GenoM3 allowed to clarify several ambiguous notions such as the incompatibility between activities and the behavior of pause transitions. Additionally, it allowed, using the full power of TTS, GenoM3 to evolve from a single-threaded version, where tasks executed sequentially using a global lock, to a multithreaded one where tasks run in parallel following a fine-grain mutual exclusion model.

In contrast to the descriptions given in Sect.2.2.3, the operational semantics favors unambiguity and gives a clear view on the behavior of GenoM3 components. Indeed, the semantics given in this section in terms of TTDs composed in parallel would always give the same TTS for the same GenoM3 specifications, while informal descriptions might be interpreted in different ways. Also, besides the choice of TTS, only elementary operations over sets and booleans are used which abstracts away from more tedious structures and complex operators and contributes thus to the comprehension of the formalization. This will smooth translating the semantics to other formalisms and proving the soundness of such translations as we will see in the next chapter.

Chapter 5

Translation of GenoM3 Semantics

5.1 Introduction

In this chapter, we translate the high-level semantics of lightweight GenoM3 (Chapt. 4) to DUTA (Sect. 3.4.2). We then prove the correctness of the translation using bisimulation. This argues in favor of the soundness of the approach but also the possibility to reproduce similar proofs for the implementation-level mappings (Chapt. 6). It thus paves the way to a generic mapping of GenoM3 components into the targeted formal frameworks, namely Fiacre, UPPAAL and BIP.

5.2 Translation to DUTA

DUTA use clocks which evolve monotonically with time and do not depend on edges enabledness. It is thus important to translate while preserving a semantically equivalent behavior under clocks. This equivalence will be proven using bisimulation (Sect. 5.3). From the previous section, we easily notice that the main source of complexity in GenoM3 resides at the execution tasks level. Thus, for readability and convenience, we restrict our translation to the first two levels of operational semantics (Sect. 4.4.1 and Sect. 4.4.2). At these levels, we use the $rrand(N, R)$ initialization (Sect. 4.4.1) which covers all the possible evolutions of execution tasks as N and R would contain at least all possible IDs if the control task was involved. That is, the set of all the possible configurations of N and R resulting from the application of $rrand(N, R)$ is a superset of that resulting from applying the restricted copy $rcopy(N, R)$ (Sect. 4.4.3).

5.2.1 Mono-task component

The objective is now to define the DUTA equivalent to the TTS of ET (Sect. 4.4.1):

$$\{\Theta\}[Tim \parallel M \parallel (\underset{A \in \mathcal{A}}{\parallel} A)]$$

where Tim, M and A are, respectively, the DUTA translations of the timer, the manager and each activity in $\mathcal{A}$. Θ will define the initial values of shared variables in the

DUTA of ET that will have the same names as in in the TTS, *i.e.* N, R, Π and sig. We give hereafter the definitions of the elements of the DUTA of ET.

Definition 14 ***Timer Tim (DUTA).***
The DUTA translation of the $timer$ is given by the following rules:

- *clocks: The timer has one clock xt, whose initial valuation is zero,*
- *locations: The timer has one location $start$ that maps the vertex $start$ of its TTD counterpart (**Definition 3**). It is associated with the invariant $xt \leq Per$,*
- *edges: The timer has one edge from $start$ to $start$ that maps its TTD counterpart. With this edge, a guard $xt = Per$ and an operation that resets xt to zero are associated. The $sig := true$ operation is also associated with the same edge.*

The invariant on location $start$ is to enforce its unique outgoing edge to be taken at Per time units at most. The guard on the latter ($xt = Per$) is to ensure taking it at exactly each period, and the reset operation $xt := 0$ to recount the period from zero each time. Consequently, the period signal through sig is sent periodically.

Fig. 5.1 shows the $timer$ TTD given in **Definition 3** and its DUTA counterpart, resulting from applying **Definition 14**.

(a) Timer TTD (b) Timer DUTA

Figure 5.1: Timer TTD to DUTA (**Definition 14**)

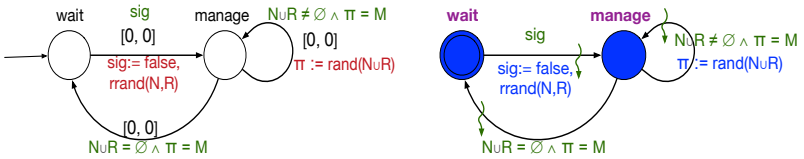
Definition 15 ***Manager M (DUTA).***
The DUTA translation of the $manager$ is given by the following rules:

- *locations: The manager has two locations $wait$ and $manage$ that map their TTD counterparts (**Definition 4**),*
- *edges: The manager has three edges that map their TTD counterpart. Guards and operations are the same as in the TTD version. Now the urgency on each TTD edge, ensured with $[0, 0]$ intervals, is enforced by making each edge in the DUTA counterpart $eager$.*

Fig. 5.2 shows the $manager$ TTD given in **Definition 4** and its DUTA counterpart, resulting from applying **Definition 15**.

We define now translation rules for activities. Due to the special $pause$ statements, one needs to be particularly careful with the translation of activities. For starts, let us consider activity A with the restriction $T_A^P = \varnothing$. We will clarify later with an example why $pause$ behaviors at this level are more delicate to translate to DUTA and propose a solution as a general rule (see **Definition 17** below).

Definition 16 ***Activities A (DUTA, restricted).***
The DUTA translation of an activity A such that $T_A^P = \varnothing$ is given by the following rules:



(a) Manager TTD (b) Manager DUTA

Figure 5.2: Manager TTD to DUTA (**Definition 15**)

- *clocks: An activity A has one clock x_A, whose initial valuation is zero,*
- *locations: Each vertex in the underlying TTD (**Definition 5**) is mapped to a location with the same name in the DUTA. Each location $c \neq ether$ is associated with an invariant $x_A \leq \uparrow I(c \rightarrow c')$ with c' any vertex in the TTD s.t. $c \rightarrow c'$ in E ($\uparrow I$ of any outgoing edge of c is equal to $W(c)$ of the underlying codel, **Definition 1**[1]),*
- *edges: (1) Each edge of the underlying TTD is mapped into an edge in the target DUTA with the same source and target. (2) Urgency intervals $[0, 0]$ are mapped into $\wr$ tags ($eager$ edges). (3) Each outgoing edge of a location that is associated with an invariant $x_A \leq W(c)$ is guarded with $x_A > 0$. (4) Each incoming edge to a location with an invariant $x_A \leq W(c)$ is associated with the reset operation over x_A. (5) Guards (respect. operations) associated with each edge in the DUTA result from the conjunction (respect. sequencing) of guards (respect. operations) of its TTD counterpart and the guards (respect. resets) of clocks as defined in (3) and (4).*

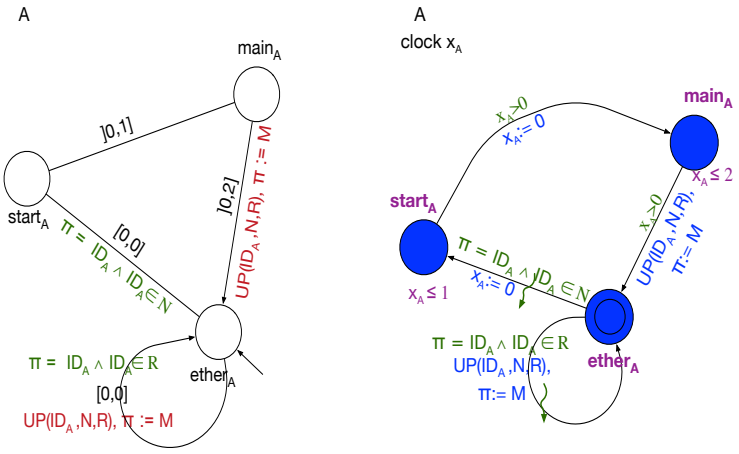
The invariants ensure that the execution of each codel takes between zero and $W(c)$ units of time. For clock x, the guards $x > 0$ are to eliminate 0 as a possible execution time and the reset operations are to ensure counting $W(c)$ starting at zero. Consequently, each codel c is executed in a non-zero amount of time inferior or equal to its WCET $W(c)$.

As an example, Fig. 5.3 shows the TTD of activity A (Sect. 4.4.1, Fig. 4.4 left) and its DUTA counterpart, resulting from applying **Definition 16**.

Let us now focus on activity B at the same level (Sect. 4.4.1, Fig. 4.4 right). We note immediately that B violates the restriction in **Definition 16** since $T^P \neq \varnothing$. The activity B is a good practical example to show why **Definition 16** may lead to incorrect translations in some cases due to the nature of clocks in DUTA.

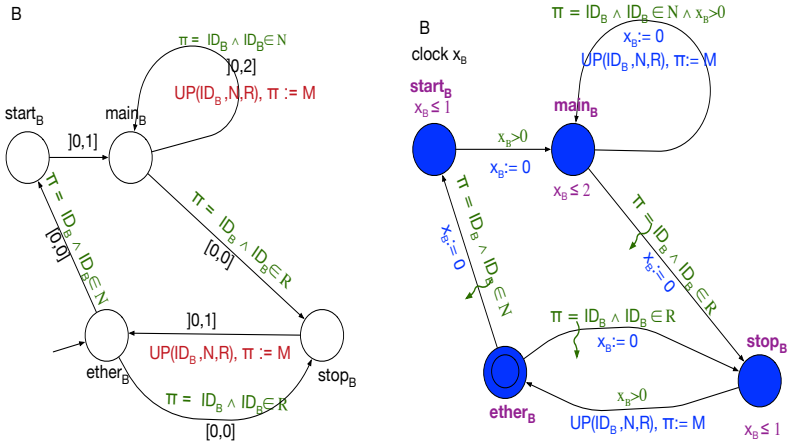
Fig. 5.4 shows the TTD of activity B (Sect. 4.4.1, Fig. 4.4 right) and its DUTA counterpart, resulting from applying **Definition 16**. This translation is incorrect. Indeed, if B passes the control back to the $manager$ after a $pause$ transition (taking the edge from $main_B$ to $main_B$ in the DUTA in Fig. 5.4), the clock x_B will continue evolving monotonically and the DUTA will timelock after 2 time units unless it resumes the control before then (all outgoing edges from location $main_B$ are disabled). This problem is due in part to the fact that clocks evolve independently from edges enabledness in DUTA (in contrast to TTDs where time intervals are relative to the date

[1]This is true for all outgoing edges of c here because no pause transition exists in the underlying activity, and thus no interruption is possible from any $c \neq ether$



(a) TTD of Activity A task ET (level 1) (b) DUTA of Activity A in task ET (level 1)

Figure 5.3: Activities TTD to DUTA (activity A, level 1, **Definition 16**)



(a) TTD of Activity B task ET (level 1) (b) Incorrect DUTA of Activity B (level 1)

Figure 5.4: Incorrect TTD to DUTA translation (activity B, level 1, **Definition 16**)

their edge was last enabled). We propose thus a new generic translation that is valid for all activities at this level without restrictions.

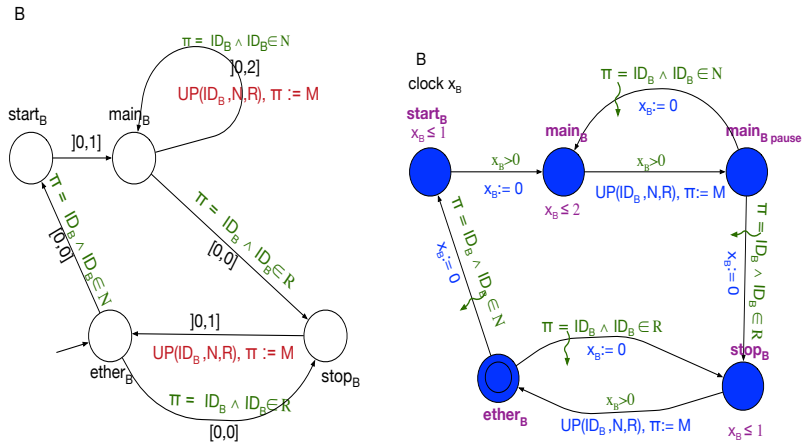
Definition 17 ***Activities*** A ***(DUTA, level 1).***
The DUTA of an activity A *is defined using the following translation rules:*

- *clocks: Same as in* ***Definition 16****,*
- *locations: Each vertex* c *of a codel* c *s.t. there exists* $\rightarrow c$ *in* T^P *is mapped to, besides the location* c *(**Definition 16**), another location* c_{pause}*. The rules on translating vertices in **Definition 16** apply on the remaining vertices to obtain the remaining locations,*

- *edges: Obtained through two steps:*
  - (a) Each edge $c \xrightarrow{g,op} c'$ *in* E^P *(**Definition 1**) is mapped to an edge* $c \xrightarrow{x_A>0,op} c'_{pause}$ *in the DUTA, and an eager edge* $c'_{pause} \xrightarrow{g,x_A:=0} c'$ *is added.*
  - (b) Each interruption edge (in E^I*) in the TTD from* $c \neq ether$ *to stop (respect. to* $ether$*, **Definition 1**) is mapped to an edge from location* c_{pause} *to stop (respect. to* $ether$*)*[2]*. Then, Rule (1) of **Definition 16** is applied on the remaining edges of the TTD to obtain their counterpart in the DUTA. Finally, rules (2) to (5) in **Definition 16** are subsequently applied to all edges obtained at step (b).*

These additional rules will allow time on clocks to evolve unboundedly at locations c_{pause}, that is when the activity is paused. Resuming the activity nominally is then equivalent to taking the eager edge $c_{pause} \rightarrow c$ and the clock will be reset at this very edge to count the WCET of c starting from 0.

Now, applying **Definition 17** to activity A will give exactly the same outcome as when applying **Definition 16** (Fig. 5.3). Let us apply **Definition 17** to activity B for which **Definition 16** is not valid as shown in Fig. 5.4. The new translation is given in Fig. 5.5. Here we know that $main_B$ is reached only when B has the control and with a prior clock reset, which eliminates the potential timelock seen in Fig. 5.4.



(a) TTD of Activity B task ET (level 1) (b) DUTA of Activity B in task ET (level 1)

Figure 5.5: TTD to DUTA translation (activity B, level 1, **Definition 17**)

5.2.2 Multi-task component

The DUTA translation rules remain unchanged for the timer Tim' and manager M'. We extend now translation rules for activities to take into account non-thread-safe codels.

Definition 18 ***Activities*** A' ***(DUTA, level 2).***
The DUTA of an activity A *is defined using the following translation rules:*

- *clocks: Same as in **Definition 16**,*

[2]To ensure interruption of a paused activity occurs as soon as the latter is resumed.

- *locations: Each vertex c in the underlying TTD (**Definition 7**) of a thread-safe codel c s.t. there exists $\rightarrow c$ in T^P is mapped to, besides the location c , another location c_{pause}. Each remaining vertex in the underlying TTD (**Definition 7**) is mapped to a location with the same name in the DUTA. Each location c that maps a vertex c of a thread-safe codel $c \neq ether$ is associated with an invariant $x_A \leq \uparrow I(c \rightarrow c')$ with c' any vertex in the TTD s.t. $c \rightarrow c'$ in E^N. The same invariant rule is applied to each location c_{exec},*

- *edges: Obtained through two steps:*
  - (a) Each edge $c \xrightarrow{g,op} c'$ in E^P s.t. c' is thread safe is mapped to an edge $c \xrightarrow{x_A>0,op} c'_{pause}$ in the DUTA, and an eager edge $c'_{pause} \xrightarrow{g,x_A:=0} c'$ is added.
  *- (b) Each interruption edge (in E^I) in the TTD from $c \neq ether$ to stop (respect. to ether, **Definition 1**) is mapped to an edge from location c_{pause} to stop (respect. to ether)[3]. Then, Rule (1) of **Definition 16** is applied on the remaining edges of the TTD to obtain their counterpart in the DUTA. Finally, rules (2) to (5) in **Definition 16** are subsequently applied to all edges obtained at step (b).*

We note immediately the resemblance between this translation and that given for *level 1*. Indeed, only thread-safe codels targeted by pause transitions induce a non-direct mapping of vertices and edges, and this aspect is already covered at *level 1*. For instance, applying **Definition 18** to activities A and B at level 2 (Sect. 4.4.2, Fig. 4.5) gives the models in Fig. 5.6. Notice how, in the absence of thread-safe codels targeted by pause transitions, the translation is rather a one-to-one mapping (besides clock-related constraints).

5.3 Translation soundness

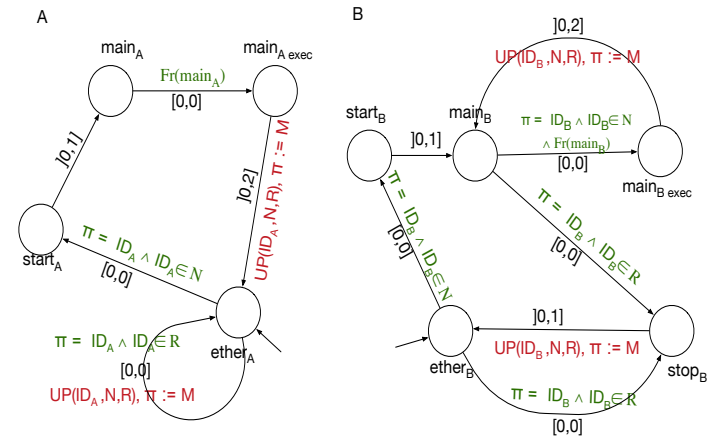
In this section, we use weak timed bisimulation (**Definition 20** below) to prove that our translation from TTS to DUTA is correct. To make the proof readable and the definitions minimal, we restrict it at *level 1*. This choice is both convenient and representative since it shows the most delicate aspect of the translation, related to thread-safe codels targeted by *pause* transitions. Indeed, we saw in the previous section how, except this aspect, the translation is rather straightforward.

5.3.1 Execution actions

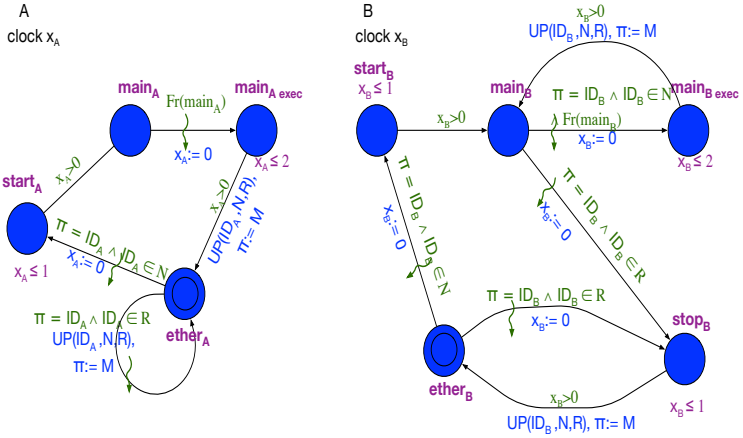
To ease following the events within a GenoM3 execution task, we define a set of possible *actions*. Each action represents a category of similar events that obey the same guards and have similar side effects on global variables. This will also ease reasoning on the soundness of the translation to DUTA. We first define the actions for the original system (in TTS) then the translation (in DUTA).

Nominal execution Nominal edges E^N are the ones explicitly specified in the GenoM3 specification (Sect. 4.3.1 and **Definition 1**). In order to partition these edges according to the actions they pertain to, we need to have a similar precondition for them. The issue here is that nominal edges (members of E^N) do not necessarily obey the property $ID_A \in N$. Indeed, in activity B for instance (Sect. 4.4.1, Fig. 4.4 right), the edge from $stop_B$ to $ether_B$ is nominal, yet it is taken when $ID_A \in R$. This will make it hard to

[3]To ensure interruption of a paused activity occurs as soon as the latter is resumed.



(a) TTDs of Activities A and B in task ET (level 2)



(b) DUTA of Activities A and B in task ET (level 2)

Figure 5.6: Activities TTD to DUTA (A and B, **Definition 18**)

express nominal actions distinguishably from interruption ones. We propose thus the following.

Definition 19 ***Augmenting interruption edges.***
*Enriching an activity A TTD is given by **Definition 5**, then each interruption edge $c \rightarrow stop$ (in E^I) is augmented with the operation $R := R \backslash \{ID_A\}$ (remove ID_A from R)[4]. The DUTA of A is then obtained from **Definition 17**.*

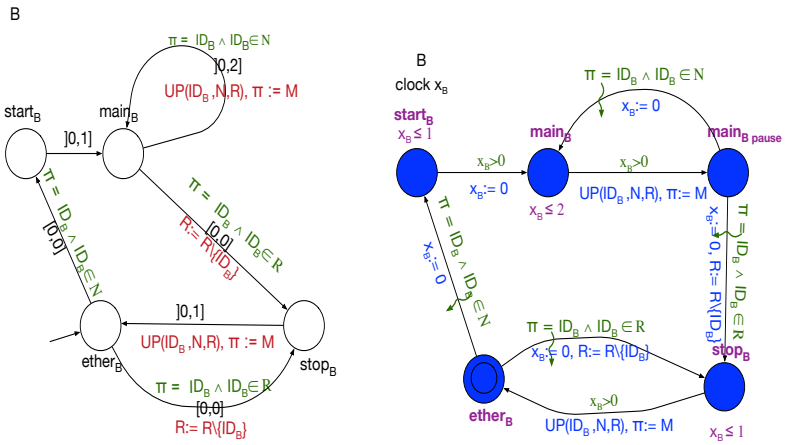
Lemma 1 ***Correctness of Definition 19.***
*Activities TTDs and DUTA obtained from **Definition 19** induce the same behavior as the ones obtained from **Definition 5** and **Definition 17**. That is, augmenting interruption edges $c \rightarrow stop$ with the operation $R := R \backslash \{ID_A\}$ does not alter the behavior of the execution task.*

[4]if $ether$ is the target codel of the interruption edge, then this is not needed.

Proof 1 *Removing ID_A from R at the beginning of the interruption (when taking the interruption edge) is equivalent to removing ID_A from R at the end of the interruption (with a termination or a pause edge). Indeed, between these two events, A has the control, that is $\Pi = A$, which means that all edges in the manager and other activities are disabled (the composition of the activities and the manager is sequential, **Definition 2**). It follows that no edge depending on R is enabled, and thus the behavior remains unchanged.*

Additionally, when performing $R := R \backslash \{ID_A\}$ (when taking the interruption edge to *stop*) is followed by performing $UP(ID_A, N, R)$ (when taking a termination edge), removing ID_A from R is redundant, that is the operation $UP(ID_A, N, R)$ is side-effect free (since ID_A has been already removed from R).

Definition 19 makes it easier to differ between interruption edges and nominal edges. Simply, a nominal edge must satisfy $ID_A \notin R$ while an interruption edge must satisfy $ID_A \in R$. We will use thus this definition for our proof. Fig. 5.7 shows the TTD and DUTA of activity B (Fig. 5.5) when applying **Definition 19**.



(a) TTD of activity B (**Definition 19**) (b) DUTA of activity B (**Definition 19**)

Figure 5.7: TTD and DUTA of activity B (**Definition 19**)

TTS

First, we partition the edges within an activity as follows:

- Interrupt activity A (ia): This action contains all additional edges for interruption (**Definition 1**), that is all edges in E^I,
- Finish activity A (fa): This action contains all nominal termination and pause edges (**Definition 1**), that is all edges in $E^P \cup E^T$,
- Execute activity A (ea): This action contains all nominal non-pause, non-termination edges plus the additional edge (for starting) $ether \rightarrow start$ (**Definition 1**), that is all edges in $E^X \cup E^S$.

Second, each edge in the manager and the timer corresponds to a distinguished action:

- Start timer (st): corresponds to taking the only possible edge in the *timer* (**Definition 3**),

$$st: \frac{\begin{array}{c} 0 \in \phi(st) \\ s'(sig) = true \qquad \phi'(st) = [Per, Per]) \end{array}}{(s, \phi) \xrightarrow{st} (s', \phi')}$$

Table 5.1: Action st.

- Start manager (sm): corresponds taking the edge from vertex *wait* to vertex *manage* (**Definition 4**),
- Launch manager (lm): matches taking the edge from vertex *manage* to itself (**Definition 4**),
- Finish manager (fm): matches taking the edge from vertex *manage* to vertex *wait* (**Definition 4**).

It is intuitive to say that these actions are (i) disjoint and (ii) cover all the edges in the execution task. Indeed, from the partitioning of the actions over edges above and from **Definition 1**, **Definition 3** and **Definition 4**, it follows that the actions cover all the possible edges (no edge remains untied to an action). Additionally, from the definition of the actions above and the mutual disjointness of all the subsets of nominal and interruption edges given in **Definition 1** (Sect. 4.4.1), it follows that the sets of actions are disjoint.

Now, we give for each action some inference rules in terms of TTS semantics: the properties that must be satisfied before taking the action and the side effects of taking it on state variables (and on future dates of taking edges, when uniquely defined). We recall that M and Tim are, respectively, the manager and timer TTDs. By abuse of notation, we refer to an edge by the action it is associated with. For instance, $c \xrightarrow{fa} c'$ is an edge associated with fa (by abuse of notation, an edge fa) from c to c', that is an edge $c \rightarrow c'$ that belongs to $E^P \cup E^T$ (see partitioning of actions above). The edges preserve thus their uniqueness according to their source and target vertex, and the set of edges they belong to (that we can retrieve from the action on the edge). This simplification helps writing the inference rules without loading the notations further.

Discrete actions (TTS): In the following, s' agrees with s on all state variables unless indicated otherwise. The formula $\exists c \xrightarrow{act} c'$ means there is an edge act from c to c' in the TTD of activity A (even if not enabled). R' and N' are the results of applying $rrand()$ to R and N, respectively.

Action st Taking this action requires satisfying the timing constraints at the timer edge. That is, st is taken at state s in the underlying TTS *iff* the Kripke state (s, ϕ) (see TTS semantics in Sect. 3.2.3) satisfies $0 \in \phi(st)$. Similarly, the state s' satisfies $sig = true$ (table 5.1).

Action sm To take this action, the manager must be at vertex $wait$ and must have the period signal ($sig = true$). After taking this action, the manager is at location $manage$, sig becomes false and N and R are randomly initialized (table 5.2).

Action lm To take this action, the manager must have the control ($\Pi = M$) and there must be activities to execute ($(N \cup R) \neq \varnothing$). According to the target Kripke state of

$$sm: \frac{s(sig) = true \quad s(\pi_M) = wait \quad s'(sig) = false \quad s'(\pi_M) = manage \quad s'(N) = N' \quad s'(R) = R'}{(s, \phi) \xrightarrow{sm} (s', \phi')}$$

Table 5.2: Action sm.

$$lm.1: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = ether \quad ID_A \in s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.2: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = c \neq ether \quad ID_A \in s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.3: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = ether \quad ID_A \notin s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.4: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = c \neq ether \quad ID_A \notin s'(R) \quad (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa})}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

Table 5.3: Action lm.

$$fm: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) = \varnothing \quad s(\pi_M) = manage \quad s'(\pi_M) = wait}{(s, \phi) \xrightarrow{fm} (s', \phi')}$$

Table 5.4: Action fm.

this action, we distinguish four cases (table 5.3): the activity that will take the control is to interrupt from $ether$ (rule $lm.1$), the activity that will take the control is to interrupt after a pause (rule $lm.2$), the activity that will take the control is to execute nominally from $ether$ (rule $lm.3$), or the activity that will take the control is to execute nominally after a pause (rule $lm.4$).

Action fm To take this action, the manager must have the control ($\Pi = M$), must be at vertex $manage$ and there must be no remaining activities to execute ($(N \cup R) = \varnothing$). Taking this action switches the manager vertex to $wait$ (table 5.4).

Action ia We distinguish four cases (table 5.5): the source vertex is $ether$ and the target vertex is $stop$ (rule $ia.1$),the source vertex is not $ether$ and the target vertex is

$$ia.1: \frac{\exists ether \xrightarrow{ia} stop \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = ether \quad ID_A \in s(R) \quad s'(\pi_A) = stop \quad ID_A \notin s'(R) \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$ia.2: \frac{\exists c \neq ether \xrightarrow{ia} stop \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \in s(R) \quad s'(\pi_A) = stop \quad ID_A \notin s'(R) \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$ia.3: \frac{\exists ether \xrightarrow{ia} ether \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = ether \quad ID_A \in s(R) \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$ia.4: \frac{\exists c \neq ether \xrightarrow{ia} ether \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \in s(R) \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

Table 5.5: Action ia.

$$fa.1: \frac{\exists c \xrightarrow{fa} c' \neq ether \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \notin s(R) \quad \phi(fa) = I_{fa} - \theta \quad \theta > 0 \quad s'(\pi_A) = c' \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{fa} (s', \phi')}$$

$$fa.2: \frac{\exists c \xrightarrow{fa} ether \quad s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \notin s(R) \quad \phi(fa) = I_{fa} - \theta \quad \theta > 0 \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{fa} (s', \phi')}$$

Table 5.6: Action fa.

$stop$ (rule $ia.2$), the source vertex is $ether$ and the target vertex is $ether$ (rule $ia.3$), the source vertex is not $ether$ and the target vertex is $ether$ (rule $ia.4$).

Action fa We distinguish two cases (table 5.6): taking a pause edge (in E^P, rule $fa.1$) or taking a termination edge (in E^T, rule $fa.2$).

$$ea.1: \frac{\begin{array}{c} s(\Pi) = ID_{A\in\mathcal{A}} \quad ID_A \notin s(R) \quad s(\pi_A) = ether \\ s'(\pi_A) = start \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa} \end{array}}{(s,\phi) \xrightarrow{ea} (s',\phi')}$$

$$ea.2: \frac{\begin{array}{c} \exists c \neq ether \xrightarrow{ea} c' \\ s(\Pi) = ID_{A\in\mathcal{A}} \quad ID_A \notin s(R) \quad s(\pi_A) = c \quad \phi(ea) = I_{ea} - \theta \quad \theta > 0 \\ s'(\pi_A) = c' \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa} \end{array}}{(s,\phi) \xrightarrow{ea} (s',\phi')}$$

Table 5.7: Action ea.

Action ea We distinguish two cases (table 5.7): taking an additional (starting) edge (in E^S, rule $ea.1$) or taking a nominal edge (in E^X rule $ea.2$).

Note that the definitions of discrete actions preconditions are both necessary and sufficient, and also expressed minimally. For instance, the condition to take the action lm (table 5.3) seems to lack the clause ($\pi_M = manage$). This clause is, however, not necessary because we may easily prove that if $(\Pi = M \wedge (N \cup R) \neq \varnothing)$ then $\pi_M = manage$.

Proof 2 *If* $s \models (\Pi = M \wedge \pi_M \neq manage)$ *then:*
either $s = s_0$*, which means that* $(N \cup R) = \varnothing$)
or s *is reached by taking the edge* $manage \xrightarrow{g=(\Pi=M\wedge N\cup R=\varnothing), op=null)} wait$*, which means also that* $(N \cup R) = \varnothing$)
It follows that the only vertex where $(\Pi = M \wedge (N \cup R) \neq \varnothing)$ *may evaluate to true is* $manage$

Another example is the lack of the clause $\exists ether \xrightarrow{ea} start$. This is a rule optimization since, by definition (i) any activity has the codels $start$ and $ether$ (Sect. 4.3.1), (ii) there is always an additional edge $ether \rightarrow start$ (in E^S, **Definition 1**) and (iii) this edge is necessarily an ea edge (see the partitioning of actions above).

Similarly, side effects are expressed minimally and effects on future times to take transitions are mentioned only when certain. For instance, we may easily prove that each execution action ea results in a state where only ea or fa are possible (table 5.7).

Proof 3 *From the partitioning, action* ea *is taken either on a starting edge (*$ether \rightarrow start$*) or on a nominal edge that is neither a pause nor a termination edge (*$c \rightarrow c' \in E^X$*). It follows that* ea *is operation free (no side effects on shared variables (**Definition 1** and **Definition 5**). Now, since* $s \models (ID_A \notin s(R))$ *(table 5.7), then* $s' \models (ID_A \notin s'(R))$*, which means that* ia *is not possible from* s'*. Additionally, since each vertex has at least one successor (**Definition 1** and Sect. 4.3.5), then either* ea *or* fa *are possible at* s'*.*

Example: In activity B (Fig. 5.5a), the edge from $main_B$ to $main_B$ is a *pause* edge, it corresponds thus to the *finish B* action fb. The same action is associated with taking the termination edge from $stop_B$ to $ether_B$. The edges from $main_B$ to $stop_B$ and $ether_B$ to $stop_B$ are interruption edges, and therefore correspond to the *interrupt B* action ib. Finally, the remaining edges are the starting edge and nominal edges that are neither for termination nor for pause, that is ea edges. Fig. 5.8 shows the TTD of B where edges are tagged with their corresponding actions (guards and operations are omitted for readability).

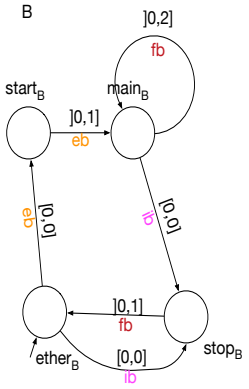


Figure 5.8: Actions in activity B

$$d.1: \frac{\phi(st) = I_{st} - a \quad a < Per \quad s(sig) = false \quad s(\pi_M) = wait \quad d \in]0, Per - a] \quad \phi'(st) = \phi(st) - d}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

$$d.2: \frac{\phi(st) = I_{st} - a \quad a < Per \quad s(\Pi) = ID_A \quad s(\pi_A) = c \neq ether \quad ID_A \notin s(R) \quad \phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \quad b < W(c) \quad d \in]0, min(Per - a, W(c) - b)] \quad \phi'(st) = \phi(st) - d \quad \phi'(ea) = \phi(ea) - d \vee \phi'(fa) = \phi(fa) - d}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

Table 5.8: Time actions d.

Time actions (TTS) We define the inference rules of taking a time action in table 5.8.

Informally, to let time evolve for a strictly positive amount d (non-trivial time step), we must have: (i) A timer period still has not elapsed since the last tick, that is $\phi(st) = I_{st} - a \neq [0, 0]$. Additionally, there must be no urgent edges possible either in the manager M or in an activity A, whichever has the control. If M has the control, that is $\Pi = M$, then the only case where no urgent edge is enabled is when M is at location $wait$[5] given that sig evaluates to false (see the manager model in **Definition 3**). If A has the control, that is $\Pi = ID_A$, it must be at a vertex c different than $ether$ because all possible edges at $ether$ are urgent (see the activities model in **Definition 4**), the urgent edge ia, if exists, must be deactivated at c, that is $ID_A \notin R$ (table 5.5) and the time elapsed since visiting c must be inferior than $W(c)$ of the underlying codel c (that is $\phi(ea) = I_{ea} - b \neq [0, 0] \vee \phi(fa) = I_{fa} - b \neq [0, 0]$). (ii) The time amount to let elapse must be superior to zero (non trivial) and must not violate any timing constraint, that is it must be at most equal to $\uparrow\phi(st)$ (if the manager is at $wait$) or the supremum of $\uparrow\phi(st)$, $\uparrow\phi(ea)$, and $\uparrow\phi(fa)(otherwise)$.

[5]Here $\Pi = M$ is redundant since the manager is at location $wait$, hence the absence of the precondition $\Pi = M$ from table 5.8.

$$st: \frac{v(xt) = Per \quad l'(sig) = true \quad v'(xt) = 0}{(l, v) \xrightarrow{st} (l', v')}$$

Table 5.9: Action st (DUTA).

$$sm: \frac{l(sig) = true \quad l(\pi_M) = wait \quad l'(sig) = false \quad l'(\pi_M) = manage \quad l'(N) = N' \quad l'(R) = R'}{(l, v) \xrightarrow{sm} (l', v')}$$

Table 5.10: Action sm (DUTA).

$$lm.1: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = ether \quad ID_A \in l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.2: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = c_{pause} \quad ID_A \in l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.3: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = ether \quad ID_A \notin l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.4: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = c_{pause} \quad ID_A \notin l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

Table 5.11: Actions lm (DUTA).

DUTA The composition of the DUTA of the *manager* M, *timer* Tim and activities (**Definition 14**, **Definition 15** and **Definition 18**) results in a Kripke structure with pairs (l, v) as states (Sect. 3.4.2.1). We may thus define the conditions and side effects for each action, defined in the original TTS, in the DUTA system.

Actions (DUTA translation). In the following, l' agrees with l on all state variables unless indicated otherwise. R' and N' are the results of applying $rrand()$ to R and N, respectively. We keep the notation $\pi(P)$, used in TTS, to denote the control location of DUTA P. The inference rules for actions st, sm, lm, fm, ia, fa and ea are given, respectively, in table 5.9, table 5.10, table 5.11, table 5.12, table 5.13, table 5.14 and table 5.15.

Example: In activity B (Fig. 5.5b), the edge from $main_B$ to $main_{B\,pause}$ maps a *pause* edge in its TTD counterpart, it corresponds thus to the *finish B* action fb. The interruption action ib is associated with taking any of the edges $main_{B\,pause}$ to $stop_B$ or $ether_B$ to $stop_B$, as both map interruption edges in the TTD counterpart. The edge

$$fm: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) = \varnothing \quad l(\pi_M) = manage \quad l'(\pi_M) = wait}{(l, v) \xrightarrow{fm} (l', v')}$$

Table 5.12: Action fm (DUTA).

$$ia.1: \frac{\exists ether \xrightarrow{ia} stop \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) = ether \quad ID_A \in l(R) \quad l'(\pi_A) = stop \quad ID_A \notin l'(R) \quad v'(x_A) = 0}{(l, v) \xrightarrow{ia} (l', v')}$$

$$ia.2: \frac{\exists c_{pause} \xrightarrow{ia} stop \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) = c_{pause} \quad ID_A \in l(R) \quad l'(\pi_A) = stop \quad ID_A \notin l'(R) \quad v'(x_A) = 0}{(l, v) \xrightarrow{ia} (l', v')}$$

$$ia.3: \frac{\exists ether \xrightarrow{ia} ether \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) = ether \quad ID_A \in l(R) \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l, v) \xrightarrow{ia} (l', v')}$$

$$ia.4: \frac{\exists c_{pause} \xrightarrow{ia} ether \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) = c_{pause} \quad ID_A \in l(R) \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l, v) \xrightarrow{ia} (l', v')}$$

Table 5.13: Actions ia (DUTA).

$$fa.1: \frac{\exists c \xrightarrow{fa} c'_{pause} \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = c \quad v(x_A) > 0 \quad l'(\pi_A) = c'_{pause} \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l, v) \xrightarrow{fa} (l', v')}$$

$$fa.2: \frac{\exists c \xrightarrow{fa} ether \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) = c \quad ID_A \notin l(R) \quad v(x_A) > 0 \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l, v) \xrightarrow{fa} (l', v')}$$

Table 5.14: Actions fa (DUTA).

$$ea.1: \frac{l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = ether \quad l'(\pi_A) = start \quad v'(x_A) = 0}{(l, v) \xrightarrow{ea} (l', v')}$$

$$ea.2: \frac{\exists c \neq ether \xrightarrow{ea} c' \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = c \quad v(x_A) > 0 \quad l'(\pi_A) = c' \quad v(x_A) = 0}{(l, v) \xrightarrow{ea} (l', v')}$$

Table 5.15: Actions ea (DUTA).

$$\tau: \frac{l(\pi_A) = c_{pause} \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l'(\pi_A) = c \quad v'(x_A) = 0}{(l, v) \xrightarrow{\tau} (l', v')}$$

Table 5.16: Internal action tau (DUTA).

$ether_B$ to $start_B$ and the edge $start_B$ to $main_B$ map, respectively, the starting edge and the only nominal edge that is neither a termination nor a pause edge in the TTD counterpart, they are therefore *execute B* actions eb. Now, the edge from $main_{B\,pause}$ to $main_B$ does not match the definition of any action and will be thus treated as an internal action τ. Fig. 5.9 shows the DUTA of activity B where edges are tagged with their corresponding actions (non-clock guards and operations are omitted for readability).

The internal discrete action τ The internal action τ is possible only when the activity has the control, its current location is c_{pause} and it is not interrupted. Taking τ changes the current location to c and resets the clock of the activity (**Definition 17**). Formally, the inference rules are given in table 5.16.

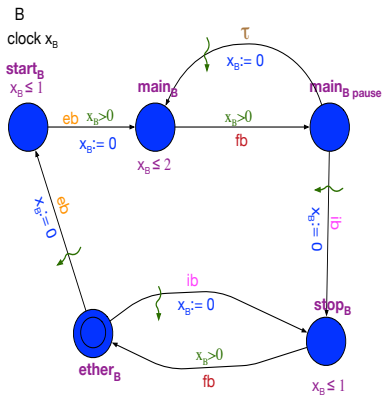


Figure 5.9: Actions in DUTA of activity B

$$d.1: \frac{\begin{array}{c} v(xt) < Per \quad l(sig) = false \quad l(\pi_M) = wait \\ d \in]0, Per - v(xt)] \\ \forall x \in X : v'(x) = v(x) + d \end{array}}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

$$d.2: \frac{\begin{array}{c} v(xt) < Per \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) \neq ether \\ ID_A \notin l(R) \qquad\qquad v(x_A) < W(c) \\ d \in]0, min(Per - v(xt), W(c) - v(x_A))] \\ \forall x \in X : v'(x) = v(x) + d \end{array}}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

Table 5.17: Time action d (DUTA).

Time actions (DUTA) The preconditions and effects of time actions are given by the inference rules in table 5.17 (X is the set of clocks in the DUTA system).

5.3.2 Absence of st effect on activities

The action st has no effect on the enabledness or timing constraints of activities actions.

Proof 4 *The action st changes only the variable sig, that is not involved in any guard $g(act) \mid act \in \{ea, fa, ia\}$. It follows that if act is enabled (or disabled) before taking st, it will remain so after taking it, both in the TTDs and DUTA. As for timing constraints, Since the enabledness is not affected in the TTD then we may write:* $(s, \phi) \xrightarrow{st} (s', \phi') \Leftrightarrow \phi'(act) = \phi(act) \forall act \in \{ea, fa, ia\}$. *In the DUTA, the clocks valuations are trivially unaffected because the timer has no access to the activities clocks, intrinsically local to their components.*

5.3.3 Absence of external actions effects on timer

Any action that is external to the timer has no effect on the enabledness or timing constraints of the timer action.

Proof 5 *The action st is guard free, which means that it is always enabled, and thus no other action can affect its enabledness or timing constraints. In the DUTA, the clock valuations is trivially unaffected because external actions have no access to the timer clock, intrinsically local to it.*

5.3.4 Edges equivalence

Let At and Ad be, respectively, the TTD and DUTA of some activity A. (i) there is an action edge ea between vertices c and c' in At iff there is an identical action edge between locations c and c' in Ad. (i) there is an action edge ia between vertices $ether$ and c' in At iff there is an identical action edge between locations $ether$ and c' in Ad. (iii) there is an action edge ia (respect. fa) between vertices c and c' in At iff there is an identical action edge between locations c_{pause} and c' (respect. c and c'_{pause}) in Ad, given that there is an edge $\rightarrow c \in E^P$ in At (respect. given that c$\rightarrow c'$ is in E^P). (iv)

there is an action edge fa between vertices c and $ether$ in At iff there is an identical action edge between locations c and $ether$ in Ad. Formally:
(i) $\exists c_{At} \xrightarrow{ea} c'_{At} \Leftrightarrow \exists c_{Ad} \xrightarrow{ea} c'_{Ad}$
(ii) $\exists ether_{At} \xrightarrow{ia} c'_{At} \Leftrightarrow \exists ether_{Ad} \xrightarrow{ia} c'_{Ad}$
(iii) (action *ia*) $(\exists c_{At} \xrightarrow{ia} c'_{At} \wedge \exists \rightarrow c \in E^P) \Leftrightarrow \exists c_{Ad\ pause} \xrightarrow{ia} c'_{Ad}$
(iii) (action *fa*) $(\exists c_{At} \xrightarrow{fa} c'_{At} \in E^P) \Leftrightarrow \exists c \xrightarrow{fa} c'_{Ad\ pause}$
(iv) $\exists c_{At} \xrightarrow{fa} ether_{At} \Leftrightarrow \exists c_{Ad} \xrightarrow{fa} ether_{Ad}$

Proof 6 *From **Definition 17**, each edge $c \rightarrow c'$ in the TTD is mapped to $c \rightarrow c'$ in the DUTA, except for interruption edges $c \rightarrow c'$ mapped to $c_{pause} \rightarrow c'$ iff c satisfies $\exists \rightarrow c \in E^P$ (respect. pause edges $c \rightarrow c'$ mapped to $c \rightarrow c'_{pause}$).*

5.3.5 Bisimilarity between TTS and DUTA systems

Let Ψ and Γ be the Kripke structures over which the semantics of an execution task TTS and DUTA, respectively, is defined. Each state in Ψ is a pair (s, ϕ) where s is the TTS state and ϕ the future dates for taking transitions (Sect. 3.2.3). Each state in Γ is a pair (l, v) where l is the interpretation of all variables excluding clocks and v the valuation of each clock x in the DUTA composition (Sect. 3.4.2.1). The objective is to prove that Ψ and Γ are timed bisimilar.

Definition 20 ***Timed bisimilarity.***
We say that Ψ and Γ are timed bisimilar iff for some binary relation $\mathcal{R}$ and discrete or time-progress action α, the initial states of Ψ and Γ are in $\mathcal{R}$, that is $\psi_0 R \gamma_0$, and:

- *Γ simulates Ψ: if $(\psi \in \Psi) R (\gamma \in \Gamma)$ and $\psi \xrightarrow{\alpha} \psi'$ then $\exists \gamma' \in \Gamma$ s.t. $\gamma \xrightarrow{\alpha} \gamma'$ and $\psi' R \gamma'$,*

- *Ψ simulates Γ: if $(\psi \in \Psi) R (\gamma \in \Gamma)$ and $\gamma \xrightarrow{\alpha} \gamma'$ then $\exists \psi' \in \Psi$ s.t. $\psi \xrightarrow{\alpha} \psi'$ and $\psi' R \gamma'$.*

In our proof, we use the *weak* version of timed bisimilarity: some internal actions τ may be involved in $\xrightarrow{\alpha}$. That is, $x(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* x'$ is observed simply as $x \xrightarrow{\alpha} x'$. For a stronger equivalence between the models, we require τ to be a discrete action. That is, time is not allowed to progress when taking τ. We begin by defining the relation $\mathcal{R}$:

Definition 21 ***The binary relation $\mathcal{R}$:***

$$(\psi = (s,\phi))\mathcal{R}(\gamma = (l,v))\ iff \begin{cases} (1)\ (\forall u \in \{\Pi, N, R, sig\} : s(u) = l(u))\ \wedge \\ (2)\ (s(\pi_{Tim}) = l(\pi_{Tim}) \wedge \phi(st) = I_{st} - \theta \wedge v(xt) = \theta)\ \wedge \\ (3)\ (s(\pi_M) = l(\pi_M))\ \wedge \\ (4)\ (\forall A \in \mathcal{A} \mid s(\Pi) = l(\Pi) \neq ID_A : s(\pi_A) = l(\pi_A) \vee \\ (s(\pi_A) = c \wedge l(\pi_A) = c_{pause}))\ \wedge \\ (5)(\exists A \models (s(\Pi) = l(\Pi) = ID_A) \Rightarrow \\ (5.1)(s(\pi_A) = l(\pi_A) = ether) \vee \\ (5.2)(s(\pi_A) = l(\pi_A) = c \neq ether \wedge ((\phi(ea) = I_{ea} - \theta) \vee \\ \quad (\phi(fa) = I_{fa} - \theta)) \wedge v(x_A) = \theta) \vee \\ (5.3)(s(\pi_A) = c \wedge l(\pi_A) = c_{pause} \wedge \\ \quad (ID_A \in s(R) \vee ((\phi(ea) = I_{ea} \vee \\ \quad \phi(fa) = I_{fa}))))) \end{cases}$$

Informally, **Definition 21** of the relation between states ψ and γ says the following. Rules (1) to (4) stipulate that ψ and γ need to agree on all state variables, at the exception of the locations of idle activities (not being executed) that can be c_{pause} instead of c in γ. Additionally, the property $\phi(st) = I_{st} - \theta \wedge v(xt) = \theta$ (Rule (2)) reflects that time in both timers progresses at the same rate (there is no guard on st which means $\phi(st)$ is always defined, and taking st in the timer DUTA resets xt). Rule (5) is only for the activity A currently executing (if any). Roughly, it says that the TTD vertex and the DUTA location of A need to be identical, and at which time must progress similarly. The location and vertex of the activity DUTA and TTD, respectively, must be identical if at $ether$ (5.1). The location of the DUTA of A must match the vertex of its TTD counterpart when executing a codel, where time must also progress identically (5.2). Finally, if the vertex of the TTD is c whereas the location of the DUTA is c_{pause}, then time is not allowed to progress and only instantaneous actions (mainly interruption actions) are possible (5.3).

Initial states We start with checking whether the initial states $\psi_0 = (s_0, \phi_0)$ and $\gamma_0 = (l_0, v_0)$ are in $\mathcal{R}$ (**Definition 20**). By definition, we know that initially:
$(s_0(N) = s_0(R) = l_0(N) = l_0(R) = \varnothing) \wedge (s_0(sig) = l_0(sig) = false) \wedge$
$(s_0(\Pi) = l_0(\Pi) = M)$,
$l_0(\pi_{Tim}) = s_0(\pi_{Tim}) = start \wedge \phi(st) = I_{st} \wedge v(xt) = 0$,
$l_0(\pi_M) = s_0(\pi_M) = wait$,
$\forall A \in \mathcal{A} : l_0(\pi_A) = s_0(\pi_A) = ether$,
$\nexists A \in \mathcal{A} \mid l_0(\Pi) = s_0(\Pi) = ID_A$.

It follows that ψ_0 and γ_0 satisfy all the rules in **Definition 21**, that is $\psi_0 \mathcal{R} \gamma_0$.
Now, we prove that Γ (weakly) time simulates Ψ (**Definition 20**). Let $\psi \in \Psi$ and $\gamma \in \Gamma$ be some states satisfying $\psi \mathcal{R} \gamma$.

Discrete actions

Action st**:** From inference rules in table 5.1, to take st from ψ, we must have:
$\psi = (s, \phi) \models (0 \in \phi(st))$ (1.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (1.a) and **Definition 21** (Rule (2)) we have:

$v(xt) = \theta \wedge \phi(st) = I_{st} - \theta$, knowing that $I_{st} = [Per, Per]$ and from (1.a) $0 \in \phi(st)$.
It follows that $\theta = Per$ and thus $v(xt) = Per$ (1.b)
Now from inference rules in table 5.9, to take st we must have
$\gamma = (l, v) \models (v(xt) = Per)$ (1.c)
From (1.b) and (1.c) it follows that action st is possible at γ.
We take now the action st from ψ to reach the state ψ'. From table 5.1 we have:
$\psi' = (s', \phi') \models (s'(sig) = true \wedge \phi(st) = [Per, Per])$ and s' agrees with s otherwise (1.d)
We take the action st from γ to reach the state γ'. From table 5.9 we have:
$\gamma' = (l', v') \models (l'(sig) = true \wedge v'(xt) = 0)$ and l' agrees with l otherwise (1.e)
From (1.d) and (1.e) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ', and from Sect. 5.3.2 (absence of effects on activities) we conclude that the rule (5) in **Definition 21** is satisfied as well.
It follows that $\psi' \mathcal{R} \gamma'$.

Action sm**:** From inference rules in table 5.2, to take sm from ψ, we must have:
$\psi = (s, \phi) \models (s(sig) = true \wedge s(\pi_M) = wait)$ (2.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (2.a) and **Definition 21** (Rules (1),(3)) we have at γ:
$l(sig) = true \wedge l(\pi_M) = wait$ (2.b)
Now from inference rules in table 5.10, to take sm we must have
$\gamma = (l, v) \models (l(sig) = true \wedge l(\pi_M) = wait)$ (2.c)
From (2.b) and (2.c) it follows that action sm is possible at γ.
We take now the action sm from ψ to reach the state ψ'. From table 5.2 we have:
$\psi' = (s', \phi') \models (s'(sig) = false \wedge s'(\pi_M) = manage \wedge s'(N) = N' \wedge s'(R) = R')$ and s' agrees with s otherwise (2.d)
We take the action sm from γ to reach the state γ'. From table 5.10 we have:
$\gamma' = (l', v') \models (l'(sig) = false \wedge l'(\pi_M) = manage \wedge l'(N) = N' \wedge l'(R) = R')$ and l' agrees with l otherwise (2.e)
From (2.d) and (2.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (2.a) and (2.d) (respect. (2.b) and (2.e)) we have $\Pi = M$ at ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Note that N' and R', being the result of a random initialization, may be different at ψ' and γ'. However, since the operation $rrand(N, R)$ is the same in both systems, it is sufficient to match the states pairwise, that is ψ' and γ' where the result of applying $rrand(N, R)$ is the same. It is trivial to prove that mapping $\psi' \in \Psi$ to $\gamma' \in \Gamma$ s.t. $\xrightarrow{sm} \psi' \wedge \xrightarrow{sm} \gamma' \wedge \psi(N') = \gamma(N') \wedge \psi(R') = \gamma(R')$ is a one-to-one function defined over all $\psi' \models \xrightarrow{sm} \psi'$, and thus for each ψ' resulting from taking an action sm there is $\gamma' \in \Gamma$ s.t. $\psi' \mathcal{R} \gamma'$.

Action lm**:** From inference rules in table 5.3, to take lm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing)$ (3.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (3.a) and **Definition 21** (Rule (1)) we have at γ:
$l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing$ (3.b)
Now from inference rules in table 5.11, to take lm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing)$ (3.c)
From (3.b) and (3.c) it follows that action lm is possible at γ.
We take now the action lm (rule $lm.1$) from ψ to reach the state ψ'. From table 5.3 we

have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.d)
We take the action lm (rule $lm.1$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.e)
From (3.d) and (3.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.1$).

We take now the action lm (rule $lm.2$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.f)
We take the action lm (rule $lm.2$) from γ to reach the state γ'. From table 5.11 we have:
$(l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.g)
From (3.f) and (3.g) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.2$).

We take now the action lm (rule $lm.3$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \notin s'(R))$ and s' agrees with s otherwise (3.h)
We take the action lm (rule $lm.3$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (3.i)
From (3.h) and (3.i) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.3$).

We take now the action lm (rule $lm.4$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \notin s'(R) \wedge$
$(\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (3.j)
We take the action lm (rule $lm.4$) from γ to reach the state γ''. From table 5.11 we have:
$\gamma'' = (l'', v'') \models (l''(\Pi) = ID_{A \in \mathcal{A}} \wedge l''(\pi_A) = c_{pause} \wedge ID_A \notin l''(R))$ and l'' agrees with l otherwise.
We take now the internal urgent action τ from γ'' to reach the state γ'. From table 5.16 we have:
$\gamma' = (l', v') \models (l'(\Pi) = l'(\pi_A) = c \wedge v'(x_A) = 0)$ and l' agrees with l'' otherwise (3.k)
From (3.j) and (3.k) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2[6] with $\theta = 0$) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.4$).

Action fm**:** From inference rules in table 5.4, to take fm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage)$ (4.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (4.a) and **Definition 21** (Rules (1),(3)) we

[6]Note that we know that location c from (3.k) is different from $ether$ since c_{pause} exists and we know by definition (Sect. 4.3.5) that $ether$ cannot be the target of a pause.

have at γ:
$l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage$ (4.b)
Now from inference rules in table 5.12, to take fm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage)$ (4.c)
From (4.b) and (4.c) it follows that action fm is possible at γ.
We take now the action fm from ψ to reach the state ψ'. From table 5.4 we have:
$\psi' = (s', \phi') \models (s'(\pi_M) = wait)$ and s' agrees with s otherwise (4.d)
We take the action fm from γ to reach the state γ'. From table 5.12 we have:
$\gamma' = (l', v') \models (l'(\pi_M) = wait)$ and l' agrees with l otherwise (4.e)
From (4.d) and (4.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (4.a) and (4.d) (respect. (4.b) and (4.e)) we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ia**:** From inference rules in table 5.5, to take ia (rule $ia.1$ or $ia.3$) from ψ, we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R))$ (5.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.a) and **Definition 21** (Rules (1), (5.1)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R)$ (5.b)
Now from inference rules in table 5.13, to take ia (rule $ia.1$ or $ia.3$) we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R))$ (5.c)
From (5.a), (5.b) and (5.c) and edges equivalence (Sect. 5.3.4) it follows that action ia (rule $ia.1$ or $ia.3$) is possible at γ.
We take now the action ia (rule $ia.1$) from ψ to reach the state ψ'. From table 5.5 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = stop \wedge ID_A \notin s'(R) \wedge (\phi' ea = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (5.d).
We take the action ia (rule $ia.1$) from γ to reach the state γ'. From table 5.13 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = stop \wedge v'(x_A) = 0 \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (5.e)
From (5.d) and (5.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking ia (rule $ia.1$).

We take now the action ia (rule $ia.3$) from ψ to reach the state ψ'. From table 5.5 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (5.f)
We take the action ia (rule $ia.3$) from γ to reach the state γ'. From table 5.13 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (5.g)
From (5.f) and (5.g) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (5) are satisfied by ψ' and γ' ((5) is satisfied because $\Pi = M$ at both ψ' and γ' and thus $\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$). It follows that $\psi' \mathcal{R} \gamma'$ after taking $ia.1$ or $ia.3$.

From inference rules in table 5.5, to take ia (rule $ia.2$ or $ia.4$) from ψ, we must have, besides the existence of an outgoing ia edge from c (to $stop$ (rule $ia.2$) or to

$ether$ ($ia.4$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \neq ether \wedge ID_A \in s(R))$ (5.h)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.h) and **Definition 21** (Rules (1),(5.3)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.i)
Now from table 5.13, to take ia (rule $ia.2$ or $ia.4$) we must have, besides the existence of an outgoing ia edge from c_{pause} (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.j)
From (5.h), (5.i), (5.j) and edges equivalence (Sect. 5.3.4) it follows that action ia (rule $ia.2$ or $ia.4$) is possible at γ.
Now, proving that $\psi' \mathcal{R} \gamma'$ after applying rule $ia.2$ (respect. $ia.4$) is identical to proving $\psi' \mathcal{R} \gamma'$ after applying rule $ia.1$ (respect. $ia.3$).

Action fa**:** From inference rules in table 5.6, to take fa from ψ, we must have, besides the existence of an outgoing fa edge from c (to $c' \neq ether$ (rule $fa.1$) or to $ether$ ($fa.2$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0))$ (6.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (6.a) and **Definition 21** (Rules (1),(5.2)[7]) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0$ (6.b)
Now from From inference rules in table 5.14, to take fa we must have, besides the existence of an outgoing fa edge from c (to c'_{pause} (rule $fa.1$) or to $ether$ ($fa.2$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0)$ (6.c)
From (6.a), (6.b), (6.c) and edges equivalence (Sect. 5.3.4) it follows that action fa is possible at γ.
We take now the action fa (rule $fa.1$) from ψ to reach the state ψ'. From table 5.6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = c' \neq ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.d)
We take now the action fa (rule $fa.1$) from γ to reach the state γ'. From table 5.14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = c'_{pause}, l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.e)
From (6.d) and (6.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

We take now the action fa (rule $fa.2$) from ψ to reach the state ψ'. From table 5.6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.f)
We take now the action fa (rule $fa.2$) from γ to reach the state γ'. From table 5.14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.g)
From (6.f) and (6.g) and absence of external actions effect on the timer (Sect. 5.3.3) it

[7]Here also we know that c in (6.a) is different from $ether$ (Sect. 4.3.5 and **Definition 1**, there is no nominal edge outgoing $ether$).

follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (6.a) and (6.f) (respect. (6.b) and (6.g)) we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ea**:** From inference rules in table 5.7, to take ea (rule $ea.1$) from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether)$ (7.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.a) and **Definition 21** (Rules (1),(5.1)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether$ (7.b)
Now from inference rules in table 5.15, to take ea (rule $ea.1$) we must have:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether)$ (7.c)
From (7.b), (7.c) it follows that action ea is possible at γ.
We take now the action ea (rule $ea.1$) from ψ to reach the state ψ'. From table 5.7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = start \wedge (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (7.d)
We take the action ea (rule $ea.1$) from γ to reach the state γ'. From table 5.15 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = start \wedge v'(x_A) = 0)$ and l' agrees with l otherwise (7.e)
From (7.d) and (7.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by ψ' and γ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 5.7, to take ea (rule $ea.2$) from ψ, we must have, besides the existence of an outgoing ea edge from c:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge \phi(ea) = I_{ea} - \theta \mid \theta > 0)$ (7.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.f) and **Definition 21** (Rules (1),(5.2)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0$ (7.g)
Now from inference rules in table 5.15, to take ea (rule $ea.2$) we must have, besides the existence of an outgoing ea edge from c:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0)$ (7.h)
From (7.f), (7.g), (7.h) and edges equivalence (Sect. 5.3.4) it follows that action ea (rule $ea.2$) is possible at γ.
We apply now the rule $ea.2$ from ψ to reach the state ψ' (table 5.7) then from γ to reach the state γ' (table 5.15). The proof that $\psi' \mathcal{R} \gamma'$ is similar to that when taking $ea.1$ (with replacing $start$ by c'.

Time actions From inference rules in table 5.8, to take d (rule $d.1$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(sig) = false \wedge s(\pi_M) = wait)$ (8.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.a) and **Definition 21** (Rules (1),(2, $\theta = a$),(3)) we have at γ:
$v(xt) = a \wedge l(sig) = false \wedge l(\pi_M) = wait$ (8.b)
Now from inference rules in table 5.17, to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(sig) = false \wedge l(\pi_M) = wait)$ (8.c)
From (8.a), (8.b) and (8.c) it follows that action d ($d \in]0, Per - a]$) is possible at γ.
We take now the action d ($d \in]0, Per - a]$) from ψ to reach the state ψ'. From ta-

ble 5.8 (rule $d.1$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d)$ (8.d).
We take the action d ($d \in]0, Per - a]$) from γ to reach the state γ'. From table 5.17 (rule $d.1$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $v'(xt) = a + d$ (8.e)
From (8.d) and (8.e) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (the manager has the control because $s(\pi_M) = l(\pi_M) = wait$ and thus $\nexists A \in \mathcal{A} \mid s(A) = l(A) = ID_A$) are satisfied by ψ' and γ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 5.8, to take d (rule $d.2$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
(8.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.a) and **Definition 21** (Rules (1),(2, $\theta = a$),(5.2, $\theta = b$)) we have at γ:
$v(xt) = a \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \wedge ID_A \notin l(R) \wedge v(x_A) = b$ (8.g)
Now from inference rules in table 5.17 (rule $d.2$), to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \neq ether \wedge ID_A \notin R \wedge v(x_A) < W(c))$ (8.h)
From (8.f), (8.g) and (8.h) it follows that action d ($d \in]0, min(W(c) - b, Per - a)]$) is possible at γ.
We take now the action d ($d \in]0, min(W(c) - b, Per - a)]$) from ψ to reach the state ψ'. From table 5.8 (rule $d.2$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d \wedge (\phi'(ea) = I_{ea} - b - d \vee \phi'(fa) = I_{fa} - b - d))$
(8.i).
We take the action d ($d \in]0, min(W(c) - b, Per - a)]$) from γ to reach the state γ'.
From table 5.17 (rule $d.2$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $(v'(xt) = a + d \wedge v'(x_A) = b + d)$
(8.j)
From (8.i) and (8.j) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (through the clause 5.2 with $\theta = b + d$) are satisfied by ψ' and γ'.
It follows that $\psi' \mathcal{R} \gamma'$.

We have thus proven that for all discrete and time actions, if $\psi \mathcal{R} \gamma$ and action act is possible from ψ s.t. $\psi \xrightarrow{act} \psi'$, then the same action act is possible from γ s.t. $\gamma \xrightarrow{act} \gamma'$ and $\psi' \mathcal{R} \gamma'$. It follows that Γ **(weakly) time simulates** Ψ. Similarly, we prove that Ψ **simulates** Γ in Appendix. A. Γ and Ψ are thus weakly timed bisimilar which proves the soundness of our translation.

5.4 Conclusion

We develop a sound translation from the TTS semantics (Chapt. 4) to DUTA (Sect. 3.4.2). We thus have, at the end of this chapter, accurate semantics of GenoM3 in the underlying formalisms of the target formal frameworks. Indeed, as we will see in Chapt. 6, the mapping of TTS (as TTDs) into Fiacre processes is straightforward and both BIP and UPPAAL are based on subclasses of DUTA. Besides allowing a convenient mapping into these tools in Chapt. 6, the translation presented here may be the basis to map GenoM3 into other frameworks based on DUTA and their subclasses. This adds to the value and the usability of the work presented in this chapter.

Chapter 6

Mapping to Formal Frameworks

6.1 Introduction

In this chapter, we show how GenoM3 implementations are mapped into their formal models counterparts in each of the frameworks presented in Chapt. 3. Since we choose the Pocolibs middleware[1], the mappings shown here are supposed to follow the Pocolibs implementation. However, because we focus on execution tasks and their activities, presenting the major part of the lightweight GenoM3 version formalized in Chapt. 4 (*level 2*, Sect. 4.4.2), the mapping presented here is also valid for the ROS-Comm implementation. Indeed, as explained in Sect. 2.3.2, the behavioral aspects of the Pocolibs and ROS-Comm implementations diverge only at the data flow level (communication between components through ports), not considered here. We will thus refer to the Pocolibs and ROS-Comm models as *implementation models*.

We derive TTS and DUTA models from the high-level TTS semantics (respect. its DUTA translation) at *level 2* as presented in Sect. 4.4.2 (respect. Sect. 5.2.2). We explain step by step how the implementation restricts the high-level semantics by eliminating a number of order-insensitive-induced behaviors. We then use the TTS and DUTA models and propose their coding in Fiacre (for TTS) and UPPAAL and BIP (for DUTA). Finally, we give examples on how the templates are used to generate the Fiacre/UPPAAL/BIP models automatically from GenoM3 specifications.

6.2 The implementation models

We see in this section how the semantics of execution tasks in TTS and its translation in DUTA (at *level 2*, Sect. 4.4.2 and Sect. 5.2.2) are restricted in implementation models.The implementation restricts the set of possible behaviors in the independent semantics of an execution task as follows. For implementation purposes, the sets of activities IDs to execute nominally (N) and to interrupt (R) are substituted with an array run of size n (the number of activities in the task) of *records*. Each record is composed of two fields: an activity ID and its $status$. Different IDs in the array may refer to activities with the same behavior (equivalent to $instances$ of the same activ-

[1] The arguments for this choice are given in Sect. 2.3.2.

ity in Sect. 2.2.3). The $status$ of an activity A ranges over three values: $nominal$, $interrupted$ and $void$, equivalent, respectively to $A \in N$, $A \in R$ and $A \notin N \cup R$ in the TTS semantics in Sect. 4.4. The record type is thus defined in an execution task ET (**Definitions 3, 4 and 7**) in pseudo-code as follows:

$$\texttt{type info is record} : \{id \in ID_{\mathcal{A}},\ status \in \{nominal, interrupted, void\}\}.$$

Note that another global array of size n_global (number of activities in the component) is introduced to replace the sets Act, In, Fi and Wa (**Definition 12**) and is writable by the control task. Since we focus only on the execution tasks in this chapter for simplicity, this global array will be abstracted to the sole operation of copy explained below.

Now, back to the execution task array run. It is initialized as follows. The fields id receive the IDs of the activities in the order of their definition in the dotgen. The fields $status$ are initialized to $void$. Each new execution cycle, upon reception of a period signal, starts with copying the updated statuses from the global array (see above), which replaces the operation $rrand(N, R)$ in **Definition 4**. We call the copying function $update(run)$. The manager of the task will then go through each cell of run. The control is passed to an activity A when visiting cell $run[i]$ *iff* $run[i].id = ID_A$ **and** $run[i].status \neq void$. The main difference with the independent semantics is that activities are executed in a predefined order, *i.e.* that of browsing the array cells in the increasing order of their indices (from 1 to n). This eliminates a number of behaviors in the generic semantics where selecting the activity to which the control is passed next is random (see operation $rand()$ in Sect. 4.4).

6.2.1 Implementation semantics (TTS)

Let us now consider the execution task independent operational semantics in Sect. 4.4 (**Definitions 3, 4 and 7**) and derive the implementation semantics using the restrictions explained above. An implemented execution task MET is a TTS (parallel composition, Sect. 3.2.5)

$$MET = \{\Theta\}[Tim \parallel Ex]$$

where Θ gives the initial values of the shared variables (given below) and Tim is the $timer$.

Ex is a TTS (parallel composition, Sect.3.2.5)

$$\{\theta\}[M \parallel (\parallel_{A \in \mathcal{A}} A)]$$

where M is the task $manager$ and $\parallel_{A \in \mathcal{A}} A$ is the parallel composition of all activities A in $\mathcal{A}$ (Sect. 4.3.2).

The local variables of MET, shared between Tim and Ex are: the array run with $\Theta(run[i]) = \{A_i \in \mathcal{A}, void\}$ for each cell i, the control passing variable $\Pi \in M \cup ID_{\mathcal{A}}$ with $\Theta(\Pi) = M$, the browsing variable $ind \in [1, n+1]$ with $\Theta(ind) = 1$, and the boolean period signal variable sig with $\Theta(sig) = false$ (the same as in the independent semantics). Now, we will see how the guards and operations in the TTDs of the manager (**Definition 4**) and the activities (**Definition 7**) are updated with respect to the new implementation variables and restricted set of behaviors. The TTD of the timer remains unchanged (it has no operations or guards involving N and R, replaced by the array run in the implementation).

Definition 22 ***Manager (implementation).***
The manager TTD (implementation) is derived from the manager TTD (specification, ***Definition 4****) with the following changes:*

- *With the edge from* $wait$ *to* $manage$*, the operations* $update(run)$ *and* $ind := next(run, ind)$ *are associated. The former updates the statuses in* run *while the latter browses* run *from index* ind *and updates it with the first index* i *that satisfies* $run[i].status \neq void$ *(*$n + 1$ *if none or if* ind *is not a valid index in* run*),*
- *On the edge from* $manage$ *to* $manage$*, the guard is* $ind < n + 1 \wedge \Pi = M$ *(the manager has the control and there are still activities to execute in the cycle) and the operation is* $\pi := run[ind].id$ *(give the control to the next activity to execute),*
- *On the edge from* $manage$ *to* $wait$*, the guard is* $ind = n + 1 \wedge \Pi = M$ *(the manager has the control and there are no more activities to execute in the cycle) and the operation is* $ind := 1$ *(reset the browsing index).*

Consequently, on the edge from $wait$ to $manage$, after updating the statuses in run, the browsing variable ind is updated to the index of the next activity to execute, if any (Fig. 6.1). If the computed ind is valid ($ind < n + 1$), the edge from $manage$ to $manage$ is taken and the control variable Π is updated to the ID of the next activity to execute, *i.e.* $run[ind].id$. As soon as the manager resumes the control, it takes the same edge to pass the control to the next activity to execute (if ind, updated by the last executed activity, is valid) or the edge from $manage$ to $wait$ to wait for the next period (otherwise).

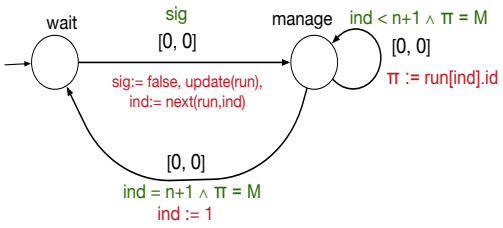


Figure 6.1: Manager TTD (implementation)

Now, we update the operations and guards on edges in the activities TTDs (**Definition 7**) using the new set of variables.

Definition 23 ***Activities (implementation):***
*In the specification TTD (****Definition 7****), Each expression of the type* $ID \in N$ *(respect.* $ID \in R$*) will become*
$run[ind].status = nominal$ *(respect.* $run[ind].status = interrupted$*). Each operation of the type* $UP(ID, N, R)$ *will be replaced by the index update operations* $ind := ind + 1$ *and* $ind := next(run, ind)$*.*

Note that since the execution is order-sensitive and the browsing ends when $ind = n + 1$, there is no need of updating the statuses of the activities in run (there is no risk of re-executing an activity within the same period and, at the next period, run will be updated from the global array anyway). For mutual exclusion, the operation $Fr(c)$ seen before is performed at runtime to prevent codels in conflict to execute at the same time.

As an example, let us apply these rules to activities A and B given in Fig. 4.5 to get the TTDs in Fig. 6.2.

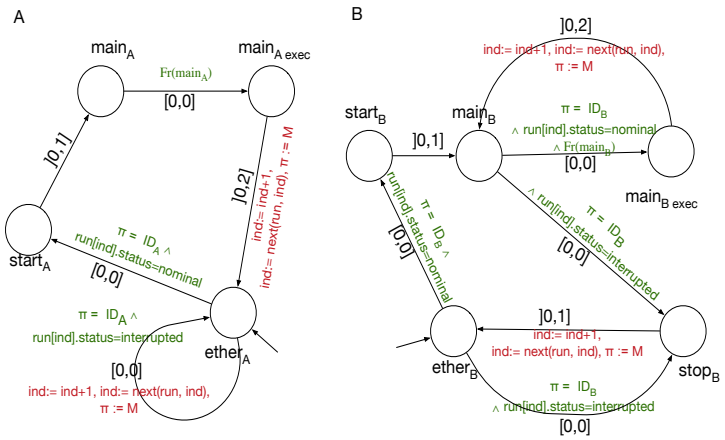


Figure 6.2: TTDs of activities A and B in ET (implementation)

6.2.2 Implementation semantics (DUTA)

It is trivial to derive the DUTA implementation semantics from the high-level TTS and DUTA operational semantics. Indeed, on the one hand, the implementation has no effect on timing constraints in the operational semantics. On the other hand, the TTS to DUTA translation is proven to be sound. It is thus sufficient to translate the TTS implementation semantics of execution tasks (**Definition 3**, **Definition 22** and **Definition 23**) to DUTA using the rules given in Sect. 5.2 (**Definition 14**, **Definition 15** and **Definition 18**). It follows that the timer implementation is identical to that in (**Definition 14**). The manager and activities are defined in the sequel.

Definition 24 ***Manager DUTA (implementation).***
The manager implementation DUTA results from applying ***Definition 15*** *to the TTD obtained from applying* ***Definition 22****.*

Applying **Definition 24** produces the DUTA in Fig. 6.3.

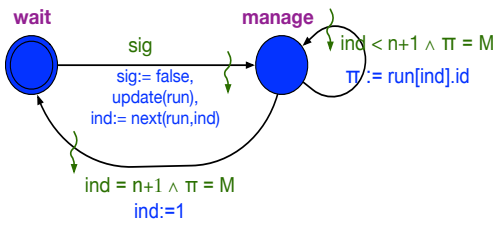


Figure 6.3: Manager DUTA (implementation)

Definition 25 ***Activities DUTA (implementation).***
The activities implementation DUTA results from applying ***Definition 18*** *to the TTD obtained from applying* ***Definition 23***

6.3 Mappings

Let us now show how the implementation of execution tasks is encoded in Fiacre, UPPAAL and BIP. We use a customized version of the component MANEUVER in the quadcopter case study (Sect. 2.4). To simplify the presentation, we consider only execution tasks and activities and convert permanent activities to plain activities that can be requested by clients (excerpt in Listing. 6.1).
From the dotgen, we can give the formal definition of each activity using Sect. 4.3.1. For simplicity, the base unit is the millisecond. The IDs, not shown here, are referred to as ID_A to differ between the activity and its identifier (name). For instance, the *ID* of *goto* is ID_{goto}.

Activity *SetState* (task plan)

- $C_{SetState} = \{start_{SetState}, ether_{SetState}\}$,
- $W_{SetState}(start_{SetState}) = 0.5$,

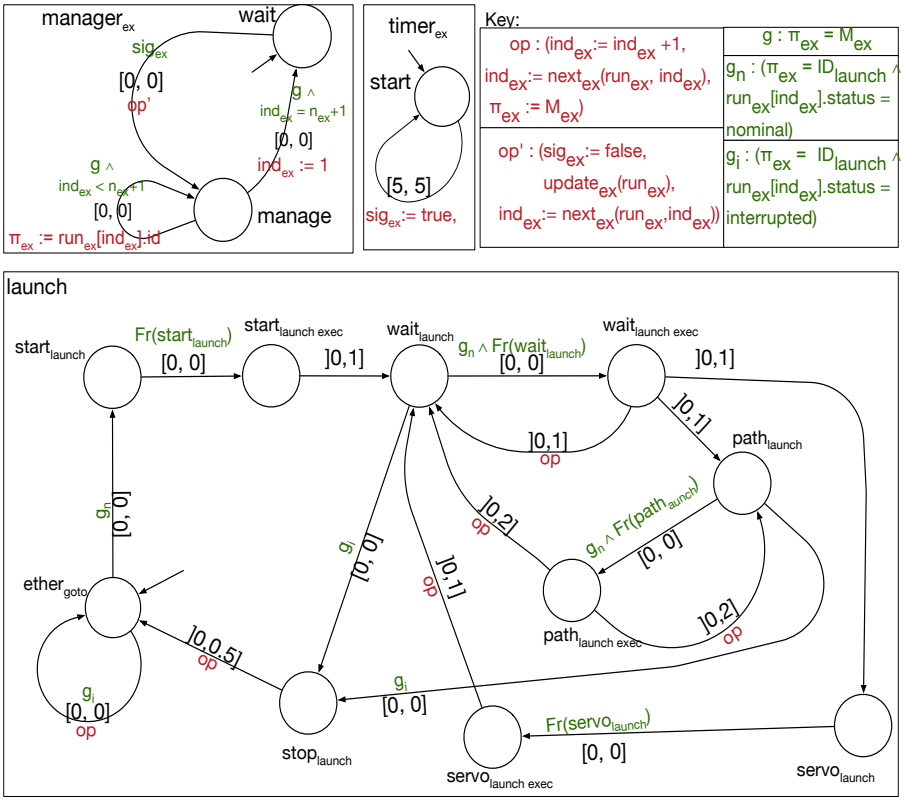


Figure 6.4: TTDs of exec

```
component maneuver {
...
/* IDS */
ids {
    planner_s planner;  /* trajectory planner */
    planner_s vplanner;  /* velocity planner */
    configuration_s reference;
    struct trajectory_t {
      sequence<configuration_s> t;
      unsigned long i;
    } trajectory;
    /* logging */
    log_s log;
  };
/* task "plan" */
task plan {
    period 5 ms;
  };
activity SetState() {
    doc  "Set initial planning position to current one";
    task plan;
    codel<start> mv_current_state_read(in state, out reference)
      yield ether wcet 0.5 ms;
  };
activity goto(in double x, in double y, in double z, in double yaw,
              in double duration) {
    doc  "Reach a given position from current state";
    task plan;
    local sequence<configuration_s> path;
    codel<start> mv_plan_goto(in planner, in reference,
                              in x, in y, in z, in yaw, in duration, out path)
      yield exec wcet 1 ms;
    codel<exec> mv_plan_exec(in path, out reference, inout trajectory)
      yield wait wcet 2 ms;
    codel<wait> mv_plan_wait(in trajectory)
      yield pause::wait, ether wcet 0.5 ms;
    interrupt goto;
  };
  /* task "exec" */
  task exec {
    period 5 ms;
}
activity launch {
    task exec;
    codel<start> mv_exec_start(out reference, out trajectory, out desired)
      yield wait wcet 1 ms;
    codel<wait> mv_exec_wait(in trajectory, in reference, out desired)
      yield pause::wait, path, servo wcet 1 ms;
    codel<path> mv_exec_path(inout trajectory,
                             in reference, out desired, inout log)
      yield pause::path, pause::wait wcet 2 ms;
    codel<servo> mv_exec_servo(inout reference, out desired, inout log)
      yield pause::wait wcet 1 ms;
    codel<stop> mv_exec_stop() yield ether wcet 0.5 ms;
  };
};
```

Listing 6.1: Excerpt of a custom version of MANEUVER component.

- $T_{SetState} = \{start_{SetState} \rightarrow ether_{SetState}\}$,
- $T^P_{SetState} = \varnothing$,
- $\mu(start_{SetState}) = \{start_{launch}, wait_{launch}, path_{launch}, servo_{launch}\}$.

Activity *goto* (task plan)

- $C_{goto} = \{start_{goto}, exec_{goto}, wait_{goto}, ether_{goto}\}$,
- $W_{goto}(start_{goto}) = 1$, $W_{goto}(exec_{goto}) = 2$, $W_{goto}(wait_{goto}) = 0.5$,
- $T_{goto} = \{start_{goto} \rightarrow exec_{goto}, exec_{goto} \rightarrow wait_{goto}, wait_{goto} \rightarrow wait_{goto}, wait_{goto} \rightarrow ether_{goto}\}$,
- $T^P_{goto} = \{wait_{goto} \rightarrow wait_{goto}\}$.

Activity *launch* (task exec)

- $C_{launch} = \{start_{launch}, wait_{launch}, path_{launch}, servo_{launch}, stop_{launch}, ether_{launch}\}$,
- $W_{launch}(start_{launch}) = 1$, $W_{launch}(wait_{launch}) = 1$, $W_{launch}(path_{launch}) = 2$, $W_{launch}(servo_{launch}) = 1$,
- $T_{launch} = \{start_{launch} \rightarrow wait_{launch}, wait_{launch} \rightarrow wait_{launch},$

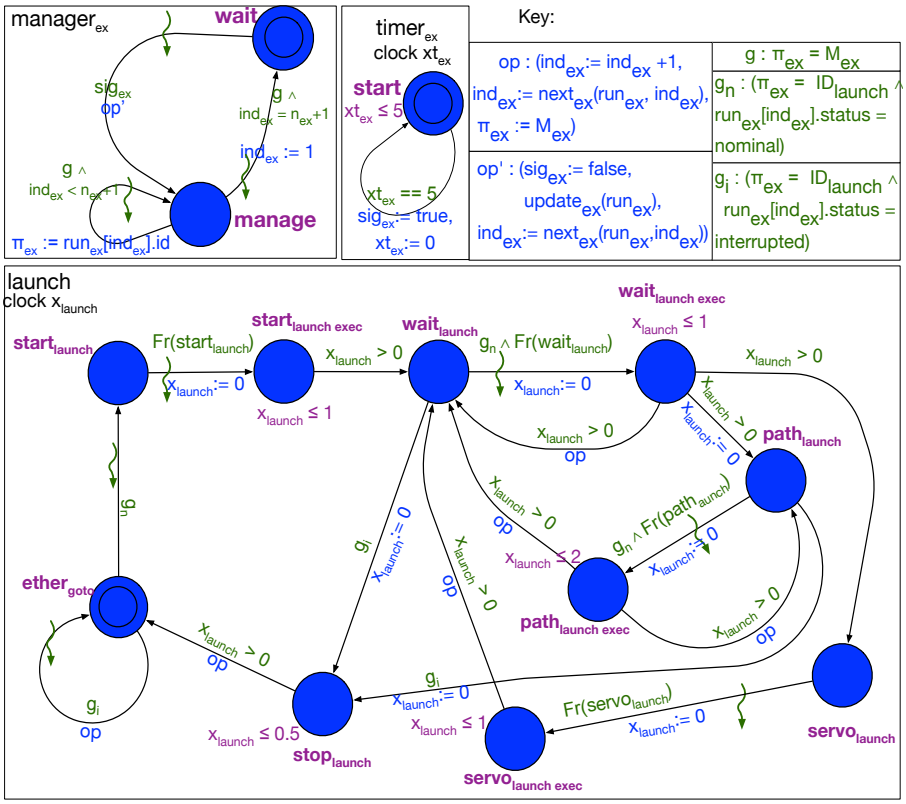


Figure 6.5: DUTAs of exec

$wait_{launch} \rightarrow path_{launch}, wait_{launch} \rightarrow servo_{launch}, path_{launch} \rightarrow path_{launch},$
$path_{launch} \rightarrow wait_{launch}, servo_{launch} \rightarrow wait_{launch}, stop_{launch} \rightarrow ether_{launch}\}$,

- $T^P_{launch} = \{wait_{launch} \rightarrow wait_{launch}, servo_{launch} \rightarrow wait_{launch},$
$path_{launch} \rightarrow path_{launch}, path_{launch} \rightarrow wait_{launch}\}$.

Mutual exclusion We compute now $\mu(c)$ for each codel c in each activity. In the implementation, $c' \in \mu(c)$ iff c' and c belong to activities in different tasks (see condition on mutual exclusion in Sect. 4.3.5) **and** c' writes (respect. reads) an IDS field that c reads or writes (respect. writes). That is, only simultaneous readings are allowed on an IDS field. For instance, we may conclude that codel `start` (activity *SetState*) and codel `start` (activity *launch*) are in conflict as they both write the field *reference* and their activities are run by different tasks, that is $start_{SetState} \in \mu(start_{launch})$ (and reciprocally). Middleware templates (Sect. 2.3.2) provide a function $mutex(c)$ that returns for any codel c the list of codels in $\mu(c)$ according to the explanation given above.
For activity *SetState*:
$\mu(start_{SetState}) = \{start_{launch}, wait_{launch}, path_{launch}, servo_{launch}\}$.
For activity *goto*:
$\mu(start_{goto}) = \{start_{launch}, servo_{launch}\}$,
$\mu(exec_{goto}) = \{start_{launch}, wait_{launch}, path_{launch}, servo_{launch}\}$,
$\mu(wait_{goto}) = \{start_{launch}, path_{launch}\}$.
For activity *launch*:
$\mu(start_{launch}) = \{start_{SetState}, start_{goto}, exec_{goto}, wait_{goto}\}$,
$\mu(wait_{launch}) = \{start_{SetState}, exec_{goto}\}$,
$\mu(path_{launch}) = \{start_{SetState}, exec_{goto}, wait_{goto}\}$,
$\mu(servo_{launch}) = \{start_{SetState}, start_{goto}, exec_{goto}\}$,
$\mu(stop_{launch}) = \varnothing$.

We can now use the inductive definitions **Definition 22** and **Definition 23** (respect. **Definition 24** and **Definition 25**) to get the implementation TTDs (respect. DUTAs) of the execution tasks plan and exec (the timers follow **Definition 3**, unchanged by the implementation). We use the subscripts p and ex to differ between variables and managers/timers TTDs/DUTAs names in, respectively, tasks plan and exec. For instance, run_{ex} is the implementation array run for task exec and $manager_p$ is the name of the TTD/DUTA of plan $manager$. The TTDs of exec (respect. plan) are given in Fig. 6.4 (respect. Fig. 6.6) and the DUTAs of exec (respect. plan) are given in Fig. 6.5 (respect. Fig. 6.7).

Note that knowing more about the exact value of $\mu(c)$ for each codel c allows us to explain easily using examples how guards $Fr(c)$ evaluate. For instance, in the TTD of *goto* (Fig. 6.6), taking the edge from $exec_{goto}$ to $exec_{goto\,exec}$ (which is equivalent to executing the codel `exec` of *goto*) is subject to satisfying the guard $Fr(exec_{goto})$. This guard evaluates to true only if the current state of the underlying TTS of the whole system, that is the parallel composition of all TTDs in both tasks, satisfies the priority $\forall c_a \in \mu(exec_{goto}) : \pi_a \neq c_{a\,exec}$, a being an activity TTD and $c_{a\,exec}$ one of its vertices (equivalent to: none of the codels in conflict with $exec_{goto}$ is currently executing). We can then use the definition of $\mu(exec_{goto})$ (see above) to easily formalize the evaluation of $Fr(exec_{goto})$ as *true* if and only if:
$(\pi_{launch} \neq start_{launch\,exec} \wedge \pi_{launch} \neq wait_{launch\,exec} \wedge \pi_{launch} \neq path_{launch\,exec}$
$\wedge \pi_{launch} \neq launch.servo_{launch\,exec})$

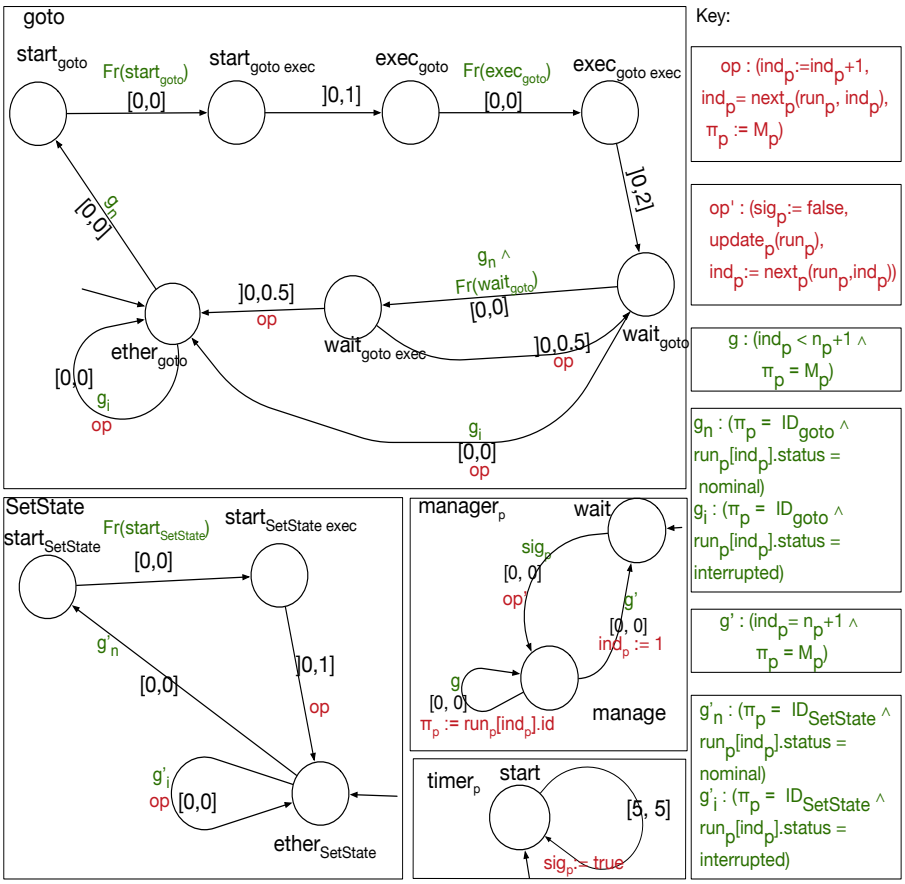


Figure 6.6: TTDs of plan

6.3.1 Mapping to Fiacre/TINA

As mentioned in Sect. 3.3, Fiacre supports a rich set of constructs such as unions and records. Let us specify the Fiacre component for the execution task exec. We start by defining the run_exec array type and size. First, we define the activity IDs and the statuses each as a union, then we build the record type from these types, and finally, we define the array run_exec type and size (listing 6.2).

```
/* Activity IDs */
type ID_exec is union ID_launch end
/* status */
type STATUS is nominal | interrupted | void end
/* array */
type CELL_exec is record id: ID_exec, status: STATUS end
const size_exec: nat is 1
type RUN_exec is array size_exec of CELL_exec end
```

Listing 6.2: Types I (Fiacre)

Now, we define the type of the browsing variable ind_exec as a natural ranging from 0 to n, and the type for the control passing variable Pi_exec (Π_{exec}) which is the union of all elements of type ID_exec and M_exec (listing 6.3).

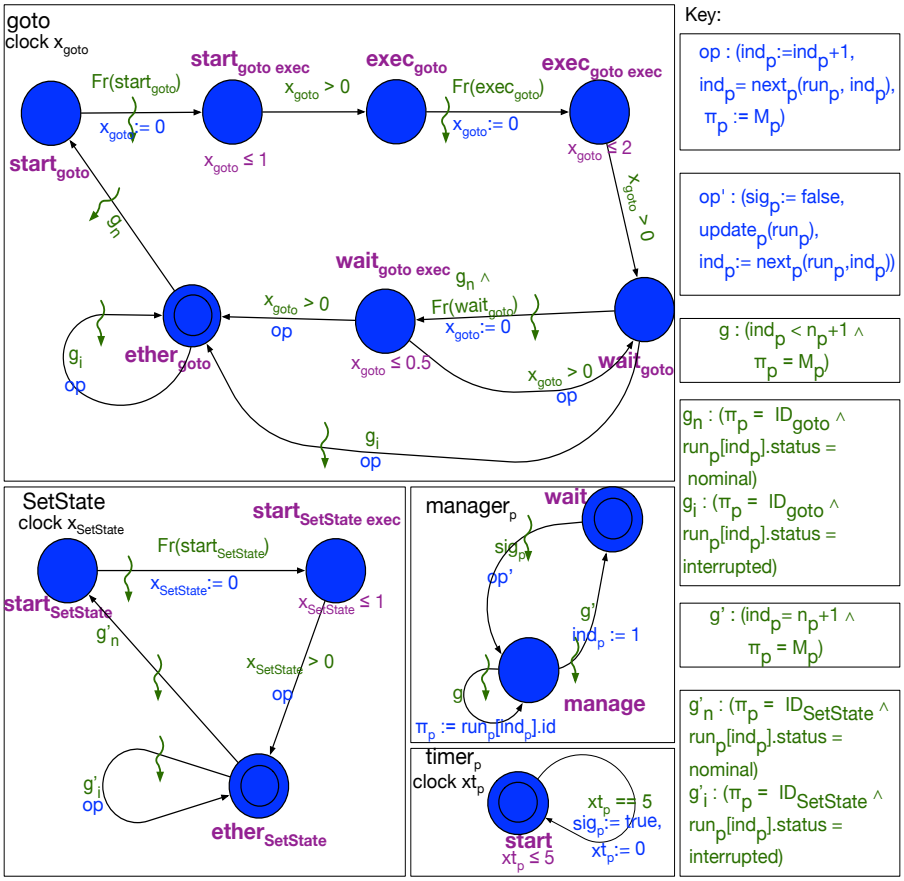


Figure 6.7: DUTAs of plan

We define then the type MUT for an array of booleans the size of which is the number of non-thread-safe codels (8 in this case, listing 6.4). An array mut of this type will be used to handle correctly the mutual exclusion aspect (see below).

```
const mut_nb: nat is 8 /* number of non-thread-safe codels in maneuver
    */
type MUT is array mut_nb of bool
```

Listing 6.4: Types III (mutual exclusion, Fiacre)

After type definitions, one may define the functions. A particularly interest is given to the $next\_exec()$ function given in the implementation semantics (the $update\_exec()$ function is abstracted). The : IND_exec after the arguments closing bracket means that the function return type is IND_exec (listing 6.5).

```
/* index */
type IND_exec is nat 0..size_exec end
/* control passing */
type PI_exec is union id: ID_exec | M_exec end
```

Listing 6.3: Types II (Fiacre)

```
function next_exec (run: RUN_exec, ind: IND_exec): IND_exec is
begin
while ind<size_exec do
       if (run[ind].status=nominal or run[ind].status=interrupted) then
              return ind
       end;
       ind:= ind+1
end;
return ind
end
```

Listing 6.5: The *next* function (task exec, Fiacre)

Now we encode the TTDs as Fiacre processes. As seen in Sect. 3.3, Fiacre processes are sequential with time semantics identical to that of TPN. Thus, in the absence of Fiacre ports, one may view a Fiacre process as a textual description of a TTD, restricted over the types and domains of variables supported by Fiacre. Let us start with implementing the TTD of the timer (Fig. 6.4). We give the name of the process and its arguments, enumerate the vertices, and describe the behavior using transition blocks as shown in Sect. 3.3 (Listing 6.6).

```
process timer_exec (&sig_exec: bool) is
state start
/* behavior */
from start
       wait [5,5];
       sig_exec := true;
to start
```

Listing 6.6: Timer I (task exec, Fiacre)

Now, we do the same for the $manager$ of exec[2] (Fig. 6.4) (listing 6.7).

```
process manager_exec (&sig_exec: bool, &run_exec: RUN_exec, &ind_exec:
     IND_exec, &pi_exec: PI_exec) is
state wait_, manage
from wait_
       on sig_exec;
       ... /* update the statuses in run_exec */
       sig_exec := false;
       ind_exec:= next_exec (run_exec, ind_exec);
to manage

from manage
       on pi_exec = M_exec;
       if ind_exec < size_exec then
              pi_exec:= run_exec[ind_exec].id;
              to manage
       else
              ind_exec:= 0;
              to wait_
       end
```

Listing 6.7: Manager (task exec, Fiacre)

Note that the $else$ clause includes values of ind_exec that can be both greater or equal to $size\_exec$. Actually, the value of ind_exec should never exceed $size\_exec$ and this

[2]Note that we replace the name of vertex $wait$ by $wait\_$ since the former is a reserved Fiacre keyword for timing constraints.

is verified at compilation by Fiacre thanks to upper-bounding the type of ind_exec by $size\_exec$ (listing 6.3).

Now we see how the activities are encoded, especially the mutual exclusion function $Fr(c)$. With each non-thread safe codel c, we associate the boolean $mut[r\_c]$ in the array mut (of type MUT, listing 6.4) whose truth value indicates whether the codel is currently executing. Thus, for the Fiacre transition from c to c_exec (equivalent to the edge from c to c_{exec} in the underlying TTD), $true$ is assigned to $mut[r\_c]$. Conversely, on each outgoing Fiacre transition from c_exec (equivalent to each outgoing edge from c_{exec} in the underlying TTD), $mut[r\_c]$ becomes false. The remaining mapping is straightforward. The Fiacre process for $launch$ (exec, Fig. 6.4) is shown in listing B.1 (Appendix. B).

We are now set to define the Fiacre component for task exec. This includes declaring and initializing the shared variables and defining the behavior of the component as the parallel composition of instances of the processes $timer\_exec$ (listings 6.6), $manager\_exec$ (listing 6.7) and $launch$ (listing B.1, Appendix. B) as shown in listing 6.8.

```
component exec_task(&mut: MUT) is
var run_exec: RUN_exec:= [{id=ID_launch, statusvoid}], pi_exec: PI_exec:= M_exec,
     ind_exec: IND_exec:= 0, sig_exec: bool:= false

par
        timer_exec(&sig_exec)
       || manager_exec (&sig_exec, &run_exec, &ind_exec, &pi_exec)
       || launch (&run_exec, &ind_exec, &pi_exec, &mut)
end
```

Listing 6.8: Component for task exec (Fiacre)

Following the same steps, we specify a component $plan\_task$ for the task plan as well. That is, we declare the necessary types, functions, processes and the component as shown for the task exec. At the end, we build the Fiacre component for MANEUVER by composing both $exec\_task$ and $plan\_task$, after declaring and initializing the mutual exclusion array mut, shared between both tasks (listing 6.9).

```
component maneuver is
var mut: MUT:= [false, false, false, false, false, false, false, false]

par
        exec_task(&mut)
       || plan_task(&mut)
end
```

Listing 6.9: Component for MANEUVER (Fiacre)

6.3.2 Mapping to UPPAAL

Before we show the different elements of the mapping, it is fair to point out the fact that UPPAAL does not support hierarchical composition. Indeed, contrary to Fiacre, models in UPPAAL must be flat. This means that we cannot compose the component MANEUVER hierarchically like we did in Fiacre. Though this is not a problem at the semantic level, it makes the models less readable, especially when modeling a robotic application with several components. Also, UPPAAL does not support doubles as time constraints. We have thus to change the base unit to a value small enough to represent all timing contraints as integers, that is 10^{-1}ms.

As specified in Sect. 3.4, *.xta* is our format of choice for automatic generation. We will show how we model the component MANEUVER, mainly the task plan (Fig. 6.7), in this format. Since one can also visualise *.xta* files in UPPAAL, we will also enrich the processes listings with DUTA figures exported from UPPAAL.

Let us start with defining the types. Types defined as unions in Fiacre, such as IDs, are integer constants here (not detailed). We define the type for each cell of the array run_plan using the $struct$ construct, similar to that in C, and declare and initialize run_plan:

```
typedef struct
{int [ID_SetState,ID_goto] id; int [void_,interrupted] status;}
CELL_plan;
const int size_plan:=2;
CELL_plan run_plan[size_plan] := { {ID_SetState,void_},
     {ID_goto,void_} };
```

Now, we define and initialize the remaining variables needed in the processes of plan:

```
/* mutual exclusion */
bool mut[mut_nb]:= {false, false, false, false, false, false, false,
     false};
/* other variables */
bool sig_plan:= false;
int[M_plan, ID_goto] pi_plan:= M_plan;
int[0, size_plan] ind_plan:= 0;
```

And the $next\_plan()$ function:

```
int[0, size_plan] next_plan (CELL_plan run[size_plan], int [0,
     size_plan] ind) {
while (ind < size_plan) {
if (run[ind].status != void_) {return ind;}
ind:= ind+1;}
return ind;}
```

Let us now deal with another issue, namely the urgencies. The implementation DUTA model of the task plan (Fig. 6.7) contains several $eager$ edges. In UPPAAL, the latter are not supported. Indeed, as mentioned in Sect. 3.4, only urgent *channels* are allowed in UPPAAL. To enforce urgency on urgent edges in Fig. 6.7, we add a process $urgency$ with only one location and one edge, and synchronize the latter, over a handshake urgent channel, with each urgent edge in the original implementation DUTA model (more explanation below). We define the urgent channel as follows:

```
urgent chan exe;
```

The nature of flat models in UPPAAL forces us to define the types, constants and variables for the task exec here as well, before starting to define the processes. It is sufficient to follow the same steps above. Once that is done, we are set to define the processes. Each DUTA in Fig. 6.7 will be mapped to an UPPAAL process, plus the urgency process. We will start with the latter (listing 6.10).

```
process Urgency(urgent chan &exe) {
state idle;
init idle;
trans
       idle →idle { sync exe?; };
}
```

Listing 6.10: Urgency process (UPPAAL)

```
process manager_plan(urgent chan &exe, int[M_plan, ID_goto] &pi_plan,
     int[0, size_plan] &ind_plan, CELL_plan &run_plan[size_plan], bool
     &sig_plan) {
state wait, manage;
init wait;
trans
 wait →manage { guard sig_plan; sync exe!;
                    assign .../* update statuses in run_plan */,
                         sig_plan := false, ind_plan:= next_plan
                         (run_plan, ind_plan); };
manage →manage { guard pi_plan == M_plan && ind_plan < size_plan;
     sync exe!;
                    assign pi_plan:= run_plan[ind_plan].id; };
manage →wait { guard pi_plan == M_plan && ind_plan == size_plan; sync
     exe!;
                    assign ind_plan:= 0; },
}
```

Listing 6.12: manager_plan process (UPPAAL)

This specification gives the DUTA in Fig. B.1 (Appendix. B). The ? means that the only edge in this process, from *idle* to *idle*, is the receiver on the urgent channel *exe*. Since this edge is always enabled, the urgent channel will be triggered (or deactivated) as soon as enabled, that is, as soon as any of the emitters (each of the *eager* edges in Fig. 6.7) is enabled.

Besides the implementation of urgencies, explained above, and the mutual exclusion, similar to the Fiacre mapping, the mapping from the DUTA in Fig. 6.7 to UPPAAL is straightforward. For instance, the timer process *timer_plan* is specified in listing 6.11 (and Fig. B.2, Appendix. B).

```
process timer_plan(bool &sig) {
clock x;
state start {x ≤50};
init start;
trans
start →start { guard x==50; assign sig_plan:= true, x:=0; },
}
```

Listing 6.11: timer_plan process (UPPAAL)

The manager process *manager_plan* in listing 6.12 (and Fig. B.3, Appendix. B).

The process *SetState* for the activity *SetState* in Fig. 6.8 (and listing B.2, Appendix. B).

And finally the process *goto* for the activity *goto* in Fig. 6.9 (and listing B.3, Appendix. B).

Similarly, we define the processes for the task exec. Once we finish with the processes, we declare the entities that will form our global system as process instances. For example, listing 6.13 shows the instantiations of all processes in the task plan:

```
tim_plan:= timer_plan(&sig_plan);
man_plan:= manager_plan(&exe, &pi_plan, &ind_plan, &run_plan,
     &sig_plan);
set_state:= SetState(&exe, &pi_plan, &ind_plan, &run_plan[size_plan],
     &mut[mut_nb]);
go_to:= goto(&exe, &pi_plan, &ind_plan, &run_plan[size_plan],
     &mut[mut_nb]);
```

Listing 6.13: Instantiations (plan, UPPAAL)

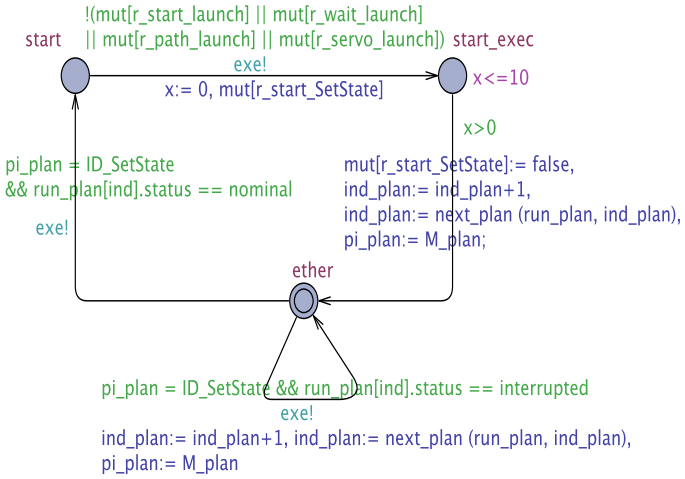


Figure 6.8: The process SetState

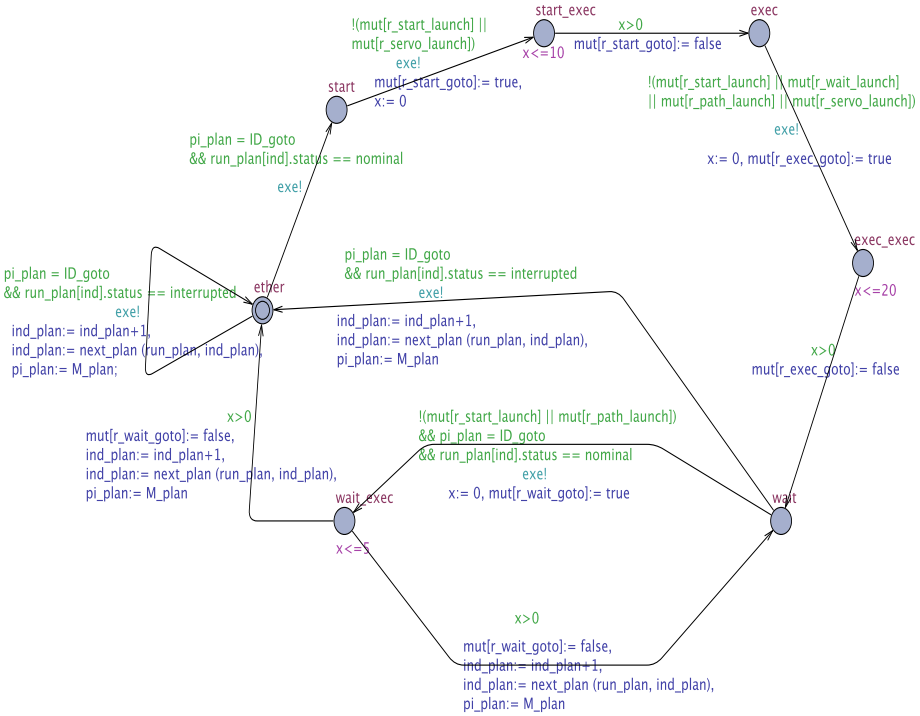


Figure 6.9: The process goto

And the urgency (listing 6.14).

```
urgency:= Urgency(&exe);
```

Listing 6.14: Instantiations (urgency, UPPAAL)

And finally, we compose in parallel all the declared instances, including the ones of task exec, using the keyword *system* (listing 6.15).

```
system urgency, tim_plan, man_plan, set_state, go_to, ....; /* plus
    the instances for task exec */
```

Listing 6.15: System definition (UPPAAL)

6.3.3 Mapping to BIP

The mapping to BIP varies according to the objective of the translation. Indeed, as seen in Sect. 3.6, RTD-Finder and the BIP Engine do not support the same options covered by the BIP language. In brief, RTD-Finder supports neither urgencies nor data variables while the BIP Engine implements the totality of the language. Therefore, the translation is quite straightforward for online verification and particularly delicate for offline verification. In this section, we briefly present online models then detail the translation for offline models. We show how we map the DUTA representations into BIP TA where neither urgencies nor variables are allowed. Since this mapping is quite complex, we only show how the mutual exclusion aspect, using both variables and urgencies, is translated into BIP as supported by RTD-Finder.

6.3.3.1 Online model

Since online models support data variables and urgencies on edges, the implementation of DUTA models is not particularly difficult. For urgencies, it is enough to use the BIP keyword *eager* on each $\wr$ edge. For data variables and functions, we may use external entities coded in $C++$. The main difference from the UPPAAL and Fiacre models is that here we can call the codels associated code, execute it, and read its return value. This makes these models runnable on the real robot but also no longer non-deterministic. Indeed, when executing the codel, it will return the next codel to execute after it does its internal computations. For a non-thread-safe codel c, the associated code is called when taking the edge from c to c_{exec} and the return value is checked at c_{exec}. On the other hand, we need an intermediate location c_{test} for thread-safe codels on which we can test the return value and decide, deterministically, which codel to execute afterwards. Let us see how this works using the activity *launch* in our example (Fig. 6.5). Without details on operations over shared variables, except the mutual exclusion variables (same names as in Fiacre and UPPAAL), we briefly see how the behavior of the BIP component atom for this activity is encoded from the DUTA implementation semantics. First, we define the external type of values returned by the execution of a codel:

```
extern data type genom_event
```

Then we define a simple port type without arguments:

```
port type Port()
```

The following listing shows the header of the BIP atom for activity *launch* as well as the behavior of the codel `wait`. We do not need here to specify the guard $x > 0$ on the outgoing edge from $wait\_exec$ because we are actually executing the codel and this will take a non-zero time (and if the execution lasts more than the specified WCET, the BIP engine will detect it).

```
atom type LAUNCH

data genom_event next_codel
```

```
clock x unit microsecond
/* ports */
...
port Port to_wait_exec()
port Port to_wait()
port Port to_path()
port Port to_servo()

state ether, ..., wait, wait_exec, wait_test, path, servo, ...
initial to ether

...
on to_wait_exec
from wait to wait_exec
provided (!(mut[r_start_SetState] || mut[r_exec_goto])/* has resources
     */)
eager
do (mut[r_wait_launch]=true /*take resources*/;
       next_codel = mv_exec_wait(...) /* execute the codel wait */;)
reset x

on to_path
from wait_exec to path
provided (next_codel == path)
do {mut[r_wait_launch]=false /*release resources*/;}

on to_servo
from wait_exec to servo
provided (next_codel == servo)
do {mut[r_wait_launch]=false /*release resources*/;}

on to_wait
from wait_exec to wait
provided (next_codel == wait)
do {mut[r_wait_launch]=false /*release resources*/;
       ... /* suspend activity (pause) */}

...
/* invariants */
...
invariant inv_wait at wait_exec when (x≤1000)
end
```

Now, if the codel `stop`, which is thread safe, had more than one successor, say *servo* in addition to *ether*, we would need an intermediate location *stop_test* between *stop* and its successors *ether* and *servo* to check what the execution of *stop* returns and solve non-determinism:

```
atom type LAUNCH2(.../*shared variables*/)
...
/* ports */
...
port Port to_stop_test()
port Port to_ether()
port Port to_servo()
...
on to_stop_test
from stop to stop_test
do {next_codel = mv_exec_stop() /* execute the codel stop */;}

on to_servo
from stop_test to servo
```

```
provided (next_codel == servo)

on to_ether
from stop_test to ether
provided (next_codel == ether)
do {... /* end activity */}

...
/* invariants */
...
invariant inv_stop at stop when (x≤500)
end
```

6.3.3.2 Offline model

Let us consider three non-thread-safe codels $c1$, $c2$ and $c3$ belonging to, respectively, activities $A1$, $A2$ and $A3$. These activities are run by three different tasks. $c1$ and $c3$ are in conflict with $c2$ but not in mutual conflict. Consequently, $c1$ (respect. $c3$) may not run if $c2$ is executing, and vice versa. However, $c1$ and $c3$ may run in parallel. For the sake of readability and simplicity, we suppose that each of these codels has only one *input* and only one *output* codel. That is, for instance, only one codel in $A1$ has a transition to $c1$ and $c1$ has only one outgoing transition. To reach the objective, *i.e* modeling the concurrency/mutual exclusion in pure BIP TA, we follow a two-step method. First, we get rid of the variables then we get rid of the urgencies. We will use UTA/TA BIP components (atoms) with ports associated with edges and connectors (thick colored lines in each figure) to synchronize edges only in a rendezvous fashion here (see Sect. 3.6 for ports and connectors). In each figure, The components $A1$, $A2$ and $A3$ are represented only partially (the discontinued edges denote missing parts before/after them).

With BIP UTA The idea is to get rid of the boolean array used in the UPPAAL model (Sect. 6.3.2) and replace it with BIP TA. Applying a connector interaction on edges with no conditions on clocks and with different urgencies produces a transition with the strongest urgency. That is, it is sufficient to have one $eager$ edge in the interaction to produce an $eager$ transition. Fig. 6.10 shows the modeling of concurrency/mutual exclusion between $c1$, $c2$ and $c3$ in BIP UTA. Each c has a *lock_c* BIP TA component that records the information on whether c is idle/waiting to execute (location *free*, equivalent to $mut[r\_c] = false$ in the UPPAAL model), or currently executing (location *taken*, equivalent to $mut[r\_c] = true$ in the UPPAAL model).

Let us see how it works through an example. The interaction involving the edge labeled *resfree* (component $A2$), the edge labeled *take* (component $lock\_c2$), and the edge labeled *check* (components $lock\_c1$ and $lock\_c3$) results in an $eager$ transition. Therefore, if the codel $c2$ is waiting to execute, i.e component $A2$ is in location *c2_wait* and $lock\_c2$ in location *free*, it will transit to *c2_exec* (start executing) as soon as the components $lock\_c1$ and $lock\_c3$ are in their respective locations *free*, which is equivalent to the truth of the guard $!(mut[r\_c1]||mut[r\_c3])$ in the UPPAAL model (in case $c1$ or $c3$ is also waiting to execute, non-determinism applies). Such a transition is concomitant to switching $lock\_c2$ to its location *taken* (equivalent to the assignment $mut[r\_c2] := true$ in the UPPAAL model), which will prevent any codel in conflict with $c2$ to execute until $c2$ finishes its execution. For the sake of readability, only one connector (that of $c1$) defining codels end of execution is represented in Fig. 6.10.

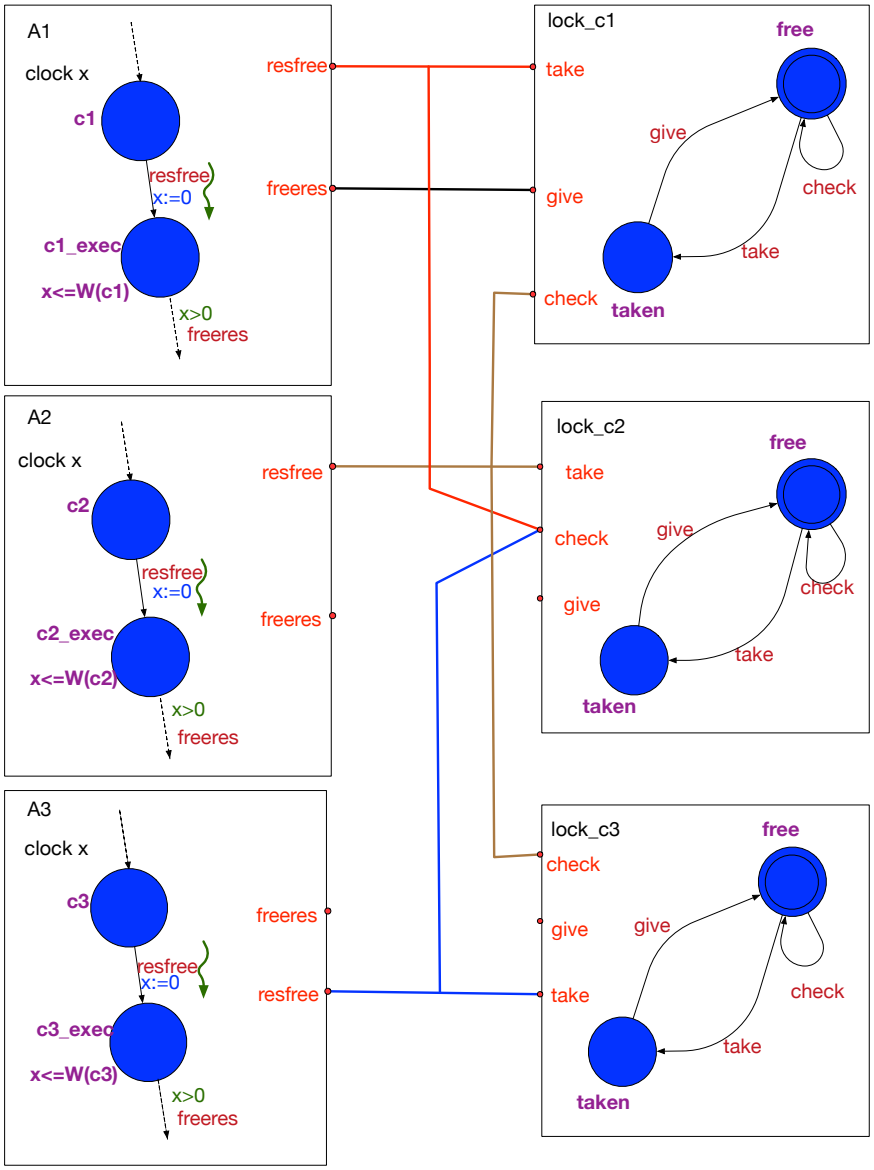


Figure 6.10: Concurrency and mutual exclusion in BIP UTA.

With BIP TA Now, we get rid of the urgencies to get the solution in Fig. 6.11. For the sake of readability, merely the connectors involving the execution and end of execution of $c2$ are shown. To eliminate urgencies, the *lock* components evolve considerably. Each $lock\_c$ BIP TA component records whether c may execute (location *ready*, equivalent to $mut[r\_c] = false$), is executing (location *execute*, equivalent to $mut[r\_c] = true$) or is idle/needs to wait further (otherwise, equivalent to $mut[r\_c] = false$).

Let us depict how it works with an example. The component $lock\_c2$ is initially at location *Init*. Unless one of the codels in conflict with $c2$, *i.e.* $c1$ or $c3$, starts executing, $lock\_c2$ transits to *ready* as $c2$ reaches the location $c2\_wait$. Here, an urgency

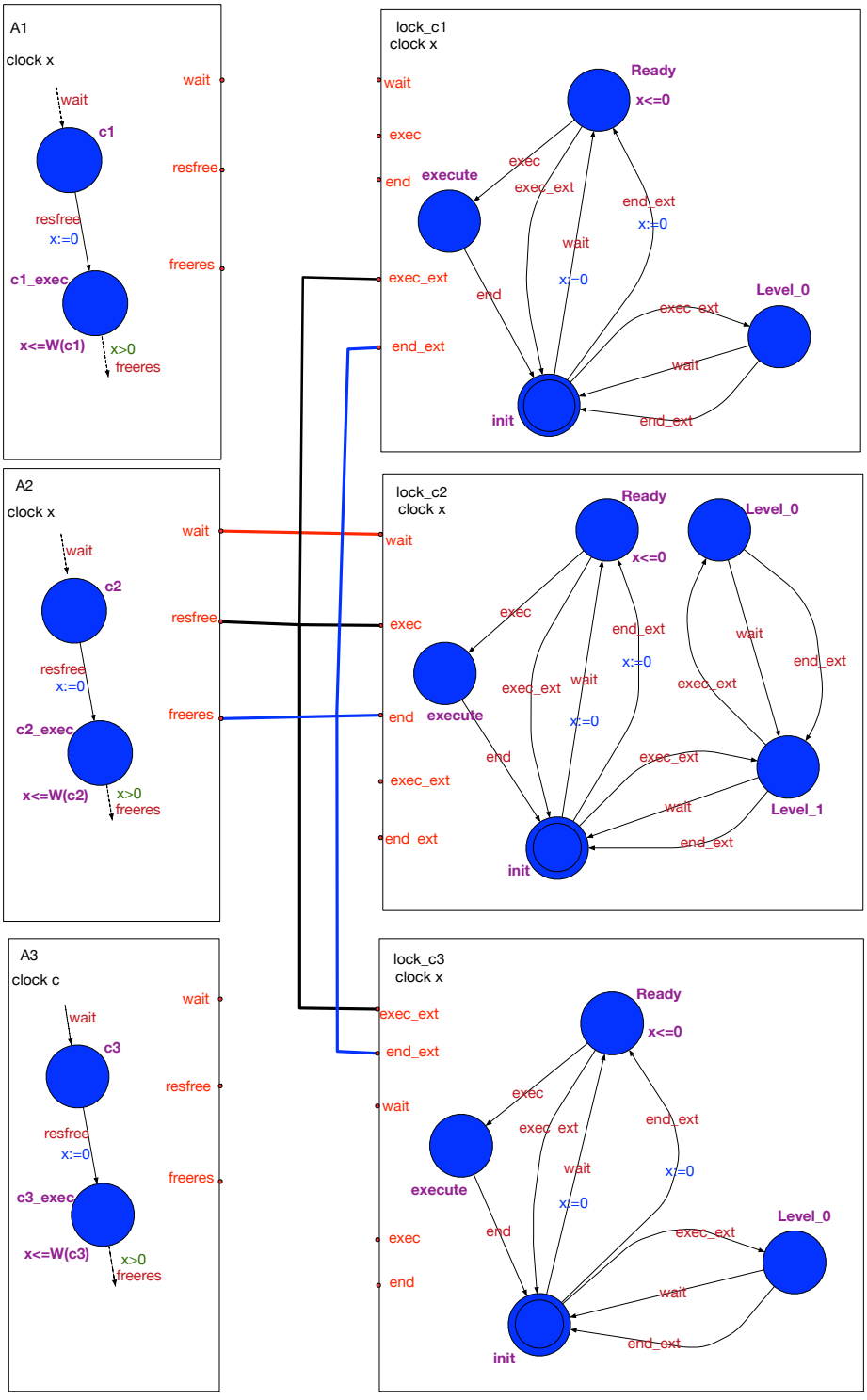


Figure 6.11: Concurrency and mutual exclusion in TA (without variables).

occurs as no time may elapse at the location *ready* (component $lock\_c2$). Therefore, $c2$ transits immediately to its location *c2_exec* while its lock transits to *execute* (in case $c1$ or $c3$ is also waiting to execute, non-determinism applies). By the end of execution, $lock\_c2$ transits back to *Init*. Now, if $c1$ starts executing, $lock\_c2$ transits to *level_1*. If $c2$ reaches its location *c2_wait* during $c1$'s execution, it is the end of the latter that allows the transition to *ready* (component $lock\_c2$) and consequently the possibility to execute $c2$.

TA do not express urgencies naturally, hence the heterogeneity of the *lock* components from a codel to another. Let c be a non-thread-safe codel and N the number of codels in conflict with it. The proposed solution induces a linear proportionality between N and the number of locations/edges of the *lock* component associated with c:

- The number of locations in the *lock* component associated with c is equal to N+3
- The number of edges within the *lock* component associated with c is equal to $3N$+5.

For instance, when N is equal to 8 (which is rather usual for a GenoM3 non-thread-safe codel), the number of locations and transitions is, respectively, 11 and 29. This is a main limitation of RTD-Finder that will influence the scalability of the models (Chapt. 7). The technical and theoretical difficulties accompanying the modeling and verification of GenoM3 with RTD-Finder are published in [Foughali, 2017] that we refer the interested reader to for in-depth details.

Coding in BIP Let us apply this modeling to the activities in our example (the GenoM3 component MANEUVER). We will see how we would implement the activities while dealing with mutual exclusion in an urgency-free, variable-free fashion as shown above. Anything besides nominal execution of activities and mutual exclusion is ignored in this example, including interruption edges, timers, managers, and other variables and functions such as run and $next$.

First, we need to define the ports and connectors types. For that, we need to know all the possible constructions of connectors from the mutual exclusion information:

- Each codel needs to synchronize its port *wait* with the port *wait* of its lock, we thus need a strong synchronization connector with two ports,
- Each of the codels `start` (*SetState*), `exec` (*goto*) and `start` (*launch*) is in conflict with four codels, we need thus for each two six-port strong synchronization connectors,
- Each of the codels `path` (*launch*) and `servo` (*launch*) is in conflict with three codels, we need thus for each two five-port strong synchronization connectors,
- Each of the codels `start` (*goto*), `wait` (*goto*) and `wait` (*launch*) is in conflict with two codels, we need thus for each two four-port strong synchronization connectors.

Now we are set to define the ports and connectors types (listing 6.16).

```
port type Port()
connector type sync_2(Port p1, Port p2)
       define p1 p2
end
```

```
connector type sync_4(Port p1, Port p2, Port p3, Port p4)
       define p1 p2 p3 p4
end
connector type sync_5(Port p1, Port p2, Port p3, Port p4, Port p5)
       define p1 p2 p3 p4 p5
end
connector type sync_6(Port p1, Port p2, Port p3, Port p4, Port p5,
     Port p6)
       define p1 p2 p3 p4 p5 p6
end
```

Listing 6.16: Connectors types (BIP)

Now we define the atoms. Let us start with the locks. For each non-thread-safe codel c, we will have a lock atom $lock\_c$ whose behavior, and number of edges and locations depend on the number of codels c is in conflict with, as shown above. Since some codels share the same number of codels in conflict with them, we may define the same atom type for them. For instance, each of the codels `start` (*SetState*), `exec` (*goto*) and `start` (*launch*) is in conflict with four codels, that is, any couple $\{c, c'\}$ of these codels satisfies $|\mu(c)| = |\mu(c')| = 4$. We may thus define an atom type $lock\_4\,()$ and instantiate the locks of codels from it when defining the global component later:

```
atom type LOCK_4()
clock x
export port Port wait()
export port Port exec()
export port Port end_()
export port Port exec_ext()
export port Port end_ext()

state level_0, level_1, level_2, level_3, init, ready, execute
initial to init

on wait
from level_0 to level_1
... /* then blocks from level_1 to level_2, level_2 to level_3 */
on end_ext
from level_0 to level_1
... /* then blocks from level_1 to level_2, level_2 to level_3 */
on exec_ext
from level_3 to level_2
... /* then blocks from level_2 to level_1, level_1 to level_0 */

on wait
from level_3 to init
on wait
from init to ready
reset x

on end_ext
from level_3 to init
on end_ext
from init to ready
reset x

on exec_ext
from init to level_3
on exec_ext
from ready to init

on exec
from ready to execute
```

```
on end_
from ready to init

/* invariants */
invariant inv_ready at ready when (x≤0)
end
```

Similarly, we define the atom type $lock\_3$ (for the locks of codels `path` (*launch*) and `servo` (*launch*)) and $lock\_2$ (for the locks of codels `start` (*goto*), `wait` (*goto*) and `wait` (*launch*)). Note that all the ports need to be exported to build connectors later. Afterwards, we define an atom type A for each activity a from the DUTA implementation models (Fig. 6.5 and Fig. 6.7) while ignoring urgencies and variables-dependent guards and operations. Consequently, the atom type for the activity *SetState* will be as follows (the base unit is 10^{-1} ms, as in the UPPAAL model):

```
atom type SETSTATE()
clock x

export port Port wait_start()
export port Port resfree_start()
export port Port freeres_start()

state ether, start, start_exec
initial to ether

on wait_start
from ether to start

on resfree_start
from start to start_exec
reset x

on freeres_start
from start_exec to ether
provided (x>0)

invariant inv_start at start_exec when (x≤10)
end
```

Note how the ports are exported to be able to connect them with ports from $lock$ components when building connectors (see below). Similarly, we define the atoms for the remaining activities, GOTO for *goto* and LAUNCH for *launch*. When we are done with the definition of the atoms, we define the compound type that will englobe the connectors and the instantiated components (from atoms). The following listing shows such definition (only connectors involving *SetState* are shown as an example):

```
compound type maneuver_simple()
       /* locks */
       component lock_4 lock_start_SetState(), lock_exec_goto(),
            lock_start_launch()
       component lock_3 lock_path_launch(), lock_servo_launch()
       component lock_2 lock_start_goto(), lock_wait_goto(),
            lock_wait_launch()
       /* activities */
       component SETSTATE SetState()
       component GOTO goto()
       component LAUNCH launch()
       /* connectors */
       /* mutual exclusion : SetState codels */
```

```
connector sync2 wait_start_SetState(SetState.wait_start,
     lock_start_SetState.wait)
connector sync6 take_start_SetState(SetState.resfree_start,
     lock_start_SetState.exec, lock_start_launch.exec_ext,
     lock_wait_launch.exec_ext, lock_path_launch.exec_ext,
     lock_servo_launch.exec_ext)
connector sync6 give_start_SetState(SetState.freeres_start,
     lock_start_SetState.end_, lock_start_launch.end_ext,
     lock_wait_launch.end_ext, lock_path_launch.end_ext,
     lock_servo_launch.end_ext)
/* mutual exclusion : goto codels */
...
/* mutual exclusion : launch codels */
...
```

6.4 Automatic synthesis

After modeling a given component in different formal languages, we generalize the approach for automatic synthesis, one of the main contribution of this thesis. We develop six templates overall (table 6.1): four templates for offline models (Fiacre, UPPAAL, UPPAAL-SMC (Sect. 7.2.2) and BIP/RTD-Finder) and two for online models (BIP/Pocolibs and BIP/ROS). In this section, we show examples on how we use the template mechanism (Sect. 2.3) to automatically generate formal models from any GenoM3 component. We give excerpts from the Fiacre, UPPAAL and BIP/RTD-Finder templates for illustration purposes.

Template	Underl. formalism	Verification technique	Offline/online
Fiacre/TINA	TPN	Model checking	Offline
UPPAAL	TA	Model checking	Offline
RTD-Finder	TA	SAT solving	Offline
UPPAAL-SMC	STA	Statistical model checking	Offline
BIP (PocoLibs)	TA	Runtime enforcement	Online
BIP (ROS)	TA	Runtime enforcement	Online

Table 6.1: GenoM3 templates for formal models

The first example shows how to generate the BIP/RTD-Finder connector types for mutual exclusion, which would give the listing 6.16 for the component maneuver (Sect. 6.3.3). Listing 6.17 shows an excerpt from the BIP/RTD-Finder template that fulfills these requirements for any GenoM3 component *comp*. Line 1 creates the list *lengths* in Tcl with one element, 2. This is because we know that we will need in any case a connector of length 2 (that connects the port *wait* of the codel and the port *wait* of the codel's lock, Sect. 6.3.3). The lines 2-4 will fill the list *lengths* with further elements according to the number of codels in conflict with each non-thread-safe codel, returned by *mutex*, plus 2 (the addition of 2 ports is explained in Sect. 6.3.3). Line 5 will sort the list and remove doubles from it. Finally, lines 6-11 will generate a connector type based on each element of *lengths*. Note that line 4 would add the element 2 to *lengths* if the codel is thread safe. This is not a problem since 2 is already an element of *lengths* and line 5 will remove doubles.

```
1               <'set lengths {2}'>
2               <'foreach s [$comp services] {'>
```

```
<' foreach c [$s codels] {'>
<' lappend lengths [expr [length [$c mutex]] + 2]}'>
<'set lengths [lsort -unique $lengths]'>
<'foreach l $lengths {'>
connector type sync_<"$l">(Port p1, Port p2
<' for {set k 3} {$k≤$l} {', Port p<"$k"><'}'>)
      define p1 p2
<' for {set k 3} {$k≤$l} {'>p<"$k"><'}'>
end
<'}'>
```

Listing 6.17: Generation of connectors types in BIP

In the second example, we show how some parts of Fiacre activities processes are generated automatically. These parts are the header of the process and the behavior involving the additional edges in the TTS semantics (**Definition 6** in Chapt. 4). Listing 6.18 is a template excerpt that generates these parts for all activities in any GenoM3 component *comp*, which would give the corresponding same parts, for activity *launch*, in listing B.1 of Appendix. B. The automatic translation process from GenoM3 to Fiacre, applied to a terrestrial navigation case study, is published in [Foughali et al., 2016].

```
<'foreach t [$comp tasks]'>
<'  foreach s [$t services] {'>
<'  set stop 0'>
<'  set pauses [list]'>
<'#header'>
process <"[$s name]"> (&run_<"[$t name]">: RUN_<"[$t name]">,
&ind_<"[$t name]">: IND_<"[$t name]">, &pi_<"[$t name]">: PI_<"[$t name]">,
&mut: MUT) is
<'#states'>
states ether
<'    foreach c [$s codels] {'>
<'    if {[$c name] == "ether"} {continue}'>
<'      foreach y [$c yields] {'>
<'      if {[$y kind] == "pause"} {lappend pauses $y}}'>
<'      if {!$stop && [$c name]=="stop"} {set stop 1}'>, [$c name]_
<'      if {[llength [$c mutex]]} {'>, [$c name]_exec
<'    }} set pauses [lsort -unique $pauses]'>
<'#behavior'>
<'#from ether'>
from ether
wait [0,0];
on (pi_<"[$t name]"> = ID_<"[$s name]">);
if run_<"[$t name]">[ind_<"[$t name]">].status = nominal then
to start /* additional edge (starting) */
else
<'if {$stop} {'>
to stop /* additional edge (interruption) */
<'} else {'>
/* no stop codel. termination */
ind_exec:= ind_exec + 1;
ind_exec:= next_exec(run_exec, ind_exec);
pi_exec:= M_exec;
to ether
<'}'>
end
<'#interruption after "pause"'>
<'foreach c $pauses {'>
from <"[$c name]">
wait [0,0];
on (pi_<"[$t name]"> = ID_<"[$s name]">);
```

```
39              if run_<"[$t name]">[ind_<"[$t name]">].status = nominal then /*
        nominal behavior */
40              ...
41              else
42              <'if {$stop} {'>
43              to stop /* additional edge (interruption) */
44              <'} else {'>
45              /* no stop codel. termination */
46              ind_exec:= ind_exec + 1;
47              ind_exec:= next_exec(run_exec, ind_exec);
48              pi_exec:= M_exec;
49              to ether
50              <'  }'>
51              end
52              <'}'>
53
```

Listing 6.18: Generation of activities in Fiacre (excerpt)

In line 3, we define the variable *stop* which is initialized to zero, and becomes 1 only if the activity contains a `stop` codel (line 13). This will define what to do if the activity is interrupted before starting (at state *ether*) or when suspended (at a state that is a target of a pause transition).

In the case of interruption before start, lines 18-32 will generate the right behavior based on the existence or not of the codel `stop`. That is, if it exists, only lines 25 and 32 will be generated (inside the block 24-32) which corresponds to a transition to state *stop* if the activity is interrupted, and if it does not, the code corresponding to termination and a transition to *ether* will be generated (lines 27 to 31).

The idea is the same for the case of interruption after a pause, but we need first to locate the states that are targets of pause transitions. Lines 11-12 append to the list *pauses*, initially empty, the codels that are targeted by pause transitions. Line 15 removes the doubles from *pauses*. Finally, we simply need to go through the list *pauses*, generate the states from the names of its codels and the interruption behavior as seen with interruption at state *ether* above.

Now we show through the third and last example (listing 6.19) how the tail of the UPPAAL file, containing the instantiations of the processes and the definition of the system is generated for any GenoM3 component *comp* (examples of what would be generated for MANEUVER in listings 6.13, 6.14 and 6.15). After instantiating an urgency process (line 1), we instantiate a timer and a manager for each task, and one process instance for each activity (lines 4-11). The list *inst_names*, initialized empty at line 2, is filled in lines 5 and 9 with the instances names that are then used at line 14 to generate the elements of the system composition.

```
1               urgency:= Urgency(&exe);
2               <'set inst_names [list]'>
3               <'#instantiations'>
4               <'foreach t [$comp tasks] {'>
5               <'lappend inst_names [join [list "tim" [$t name]] _] [join [list "man"
                    [$t name]] _]'>
6               tim_<"[$t name]">:= timer_<"[$t name]">(&sig_<"[$t name]">);
7               man_<"[$t name]">:= manager_<"[$t name]">(&exe, &pi_<"[$t name]">,
                    &ind_<"[$t name]">, &run_<"[$t name]">, &sig_<"[$t name]">);
8               <' foreach s [$t services] {'>
9               <' lappend inst_names [join [list [$s name] ""] _]'>
10              <"[$s name]">_:= <"[$s name]">(&exe, &pi_<"[$t name]">, &ind_<"[$t
                    name]">, &run_<"[$t name]">[size_<"[$t name]">], &mut[mut_nb]);
11              <' }}'>
12              <'#system composition'>
```

```
13              system urgency
14              <'foreach i $inst_names {'>, <"$inst"><'}'>;
```

Listing 6.19: Instantiations and system definition (UPPAAL template)

6.5 Conclusion

In this chapter, we derive implementation semantics for execution tasks in both TTS and DUTA from the high-level operational semantics and its translation (Chapt. 5). We then show how we encode the implementation models in Fiacre, BIP, and UPPAAL. We explain how such a coding is automatized using the template mechanism presented in Chapt. 2. We thus have, at the end of this chapter, a correct automatic generation of formal models from any GenoM3 specification, which is a main contribution of this thesis. Indeed, as shown in Chapt. 1, one of the main issues of verification of functional robotic components is the lack of automation. That is, practitioners need to go through the tediousness and faillibility of formal modeling for each new robotic application. The approach shown in this chapter, Chapt. 5 and Chapt. 4 is a solution to this issue. The translations are proven correct, and since the process is automatized, this correctness is propagated no matter what the specification is. Moreover, a considerable amount of time and effort is saved with automatic translation. Furthermore, the variety of templates allows us to provide experimental feedback on how to use these templates as efficiently as possible (next chapter) which is also a part of the contributions of this thesis introduced in Chapt. 1.

Chapter 7

Verification

7.1 Introduction

In this chapter, we show how the automatically generated models (Chapt. 6) are used to specify and verify important behavioral and timed properties on the quadcopter and Osmosis case studies (Sect. 2.4). The verification can be offline or online. Under the offline class, we find exhaustive verification and statistical model checking. The former is based either on model checking (UPPAAL and Fiacre models) or SAT solving over super-approximations of reachable state spaces (BIP/RTD-Finder models) while the latter exploits the UPPAAL-SMC models. Under the online verification class, the BIP execution models are used to enforce desired properties at runtime. Within each section, we discuss the pros and cons of the used techniques throughout an experimental real-world feedback and derive guidance scheme to help practitioners benefit the maximum from our templates. We recall that the generated models include all the timing constraints, that is tasks periods and codels WCET, automatically extracted from the GenoM3 specifications (Chapt. 6). All the results given in this chapter follow verification performed on a typical mid-range computer; Intel Core i7 with 16 GB of RAM.

7.2 Offline verification

As shown in Sect. 2.2.3, clients are needed to run GenoM3 applications. Under the offline class of verification, clients are modeled separately and composed in parallel with the automatically generated models. We give examples within both categories of offline verification (exhaustive and statistical) of the clients models. Also, it is important to emphasize that, unless stated otherwise, all the results in this section follow the assumption that the hardware, in each application, has a sufficient number of cores to run all the tasks in parallel, so each task can be assigned to a certain core permanently without being interrupted.

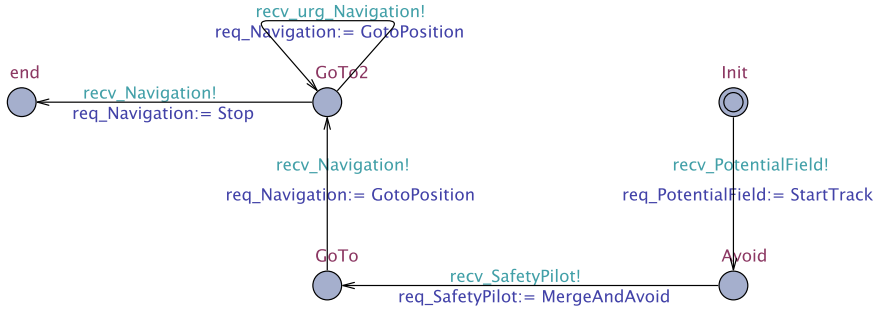


Figure 7.1: UPPAAL client (Osmosis).

7.2.1 Exhaustive verification

We automatically generate the Fiacre, BIP/RTD-Finder and UPPAAL models for Osmosis and quadcopter[1]. We add clients that model a navigation (for Osmosis in simulation mode) and a stationary flight (for quadcopter, excluding the component MANEUVER). For instance, Fig. 7.1 shows the UPPAAL model of the Osmosis client. Each component X in the application has a fixed-size queue $fifo\_X$ (not shown here) that models its PocoLibs mailbox (where it receives clients requests). Each channel $recv\_X$ is an urgent channel to insert requests of any activity of X in $fifo\_X$ through the shared variable req_X. The channel $recv\_urg\_X$ is a special channel to insert an activity Y in $fifo\_X$ only when the last requested activity has ended. Thus, the self-loop at location $GoTo2$ enables issuing a new *GotoPosition* request each time the last served *GotoPosition* activity has ended (goal invalid, reached, or unreachable). From the same location, a request *Stop* to interrupt the activity *GotoPosition* (to stop the navigation) can be sent at any moment.

7.2.1.1 Properties of interest

Let us now define the type of properties in which we are interested at this level. We mainly categorize these properties into four classes from a robotic programmer point of view:

Proper init (Osmosis & quadcopter): Is each port written at least once before being read for the first time? This property is very important as its violation may induce the use of garbage values that may lead to dangerous behaviors. This is a *safety property*.

Progress (Osmosis): Will an activity eventually finishes a codel it starts? Due to non-deterministic mutual exclusion, a non-thread-safe codel within a started activity may wait forever for resources it needs to acquire. We will see how this property is formulated as a *leads to property*.

Schedulability (quadcopter): Do execution tasks meet their deadlines? Summing the WCETs of the codels and comparing them to the task period would not help to reason on this property since the WCET refers to execution and does not consider waiting for IDS/ports resources, the latter being quite difficult to obtain. The

[1]The UPPAAL model of the Osmosis simulation application and the verification results are available at `git://redmine.laas.fr/laas/users/mfoughal/case-study-osmosis.git`

semantics proposed in Chapt. 4 will allow us to formulate this property as a *safety property*.

Bounded stop (Osmosis): Upon the reception of a *Stop* request (NAVIGATION), what is the maximum amount of time (upper bound) until the null speed is written to the **Cmd** port (SAFETYPILOT)? Indeed, the computation of such a bound, if it exists, is not obvious. This is a *bounded response property*.

Bounded processing (quadcopter): It is important to verify that the control task always finishes its execution in a "short" amount of time to make sure that further requests can be received and processed in the future. Here also, the mutual exclusion over the IDS might lead to the control task waiting too long for resources to execute a control service or a validate codel. This is a *bounded response property*.

Note that schedulability implies progress (the converse is false). The importance of each property depends on the application. This is why we choose to verify schedulability, the stricter property, for the stationary flight (quadcopter) where the frequencies are high and missing deadlines may lead to catastrophic behaviors. In contrast, it is sufficient for navigation (Osmosis) to check the weaker property, namely *progress*, as long as the *bounded stop* property holds with sufficiently small upper bound.

7.2.1.2 Verification with TINA (Version 3.4)

In Fiacre, one may express properties as *patterns*. The *frac* compiler will then translate such properties into LTL or modal μ-calculus for TINA (Sect. 3.3).

Proper init (Osmosis & quadcopter) A global array of initially false booleans *ports_read* (respect. *ports_write*) represents for each port in the application whether such a port has been already read (respect. written) at least once (*true*) or not (*false*). GenoM3 files specify in the arguments of codels which ports they read/write, if any (Sect. 2.2.2, Sect. 2.2.3), so it is possible to know when to switch the value of each variable $ports\_x[i]$ from *false* to *true*. Indeed, if a codel reads (respect. writes) a port p, then the operation $ports\_read[p] := true$ (respect. $ports\_write[p] := true$) is generated within the operations of the transition corresponding to the execution of that codel. Then, the property is an invariant that must evaluate to true for each port p:

```
always (ports_read[p] ⇒ ports_write[p])
```

This is a reachability property and depends only on the markings of the *State Class Graph (SCG)*. TINA offers a coarser and faster *Markings-Only* construction that computes the set of markings of TPN without preserving firing sequences. We take advantage of this construction to prove that the property holds for all ports in the stationary flight application (quadcopter). However, the property is violated for the port **Pose** (POM) in the navigation application of Osmosis. This helped us fix a bug where this port was read by NAVIGATION before being initialized which would lead to erroneous distance computations.

Progress (Osmosis) In the Fiacre modeling (Sect. 6.3.1), the *manager_T* process current state denotes whether the task T is executing activities (state *manage*) or idle (state *wait_*). Thus, to prove no codel is infinitely blocked (waiting for resources), it is enough to prove that whenever reached, the state *manage* is eventually left, e.g. for navigate (component NAVIGATION):

```
(Navigation/navigate_task/manager_navigate/state manage) leadsto
    (Navigation/navigate_task/manager_navigate/state wait_)
```

This pattern encodes the following LTL expression:

```
□((Navigation/navigate_task/manager_navigate/state manage) ⇒♢
                    (Navigation/navigate_task/manager_navigate/state wait_)
```

We build the SCG and prove that the property holds for all execution tasks in the Osmosis navigation.

Schedulability (quadcopter) If a task T is schedulable, then the variable *sig_T* is never *true* when the process *manager_T* is at state *manage* (Sect. 6.3.1). That is, when executing the activities, denoted by being at state *manage* of process *manager_T*, no new period signal is received, *i.e.* the process *timer_T* does not change the value of *sig_T* to *true*. It is thus sufficient to verify, e.g. for io (component POM):

```
always ((pom/io_task/manager_io/state manage) ⇒
                                    not (pom/io_task/value tick_io))
```

This is also, as it is the case for the proper init property, a reachability property that depends only on the markings of the SCG. We take advantage of the Markings-Only construction to prove all tasks schedulable in the quadcopter stationary flight application.

Bounded stop (Osmosis) As seen in Sect. 3.3, Fiacre provides also patterns to verify bounded response properties. When a bounded response property is given as a pattern in the Fiacre description, the *frac* compiler generates *observers* to verify it. The main advantage of this technique is reducing bounded response to *reachability*. However, observers induce a larger SCG which has a negative impact on scalability.

The bounded stop property aims at finding, if exists, the upper bound separating sending a *Stop* request (NAVIGATION) and writing a null speed to the **Cmd** port (SAFETYPILOT). This bound cannot be computed directly by estimating the bound between the event of sending the *Stop* request and the event of writing the **Cmd** port. Indeed, we need to make sure that the latter follows reading the value of **PFCmd** (POTENTIALFIELD) that was written after writing the value of **Target** (NAVIGATION) following taking the *Stop* request into account (the interruption of *GotoPosition*). This bound is thus the sum of the following maximum bounds:

- $b1$ between sending the *Stop* request and writing a new **Target** (NAVIGATION),
- $b2$ between writing **Target** (NAVIGATION) and reading **Target** (POTENTIALFIELD),
- $b3$ between reading **Target** (POTENTIALFIELD) and writing **PFcmd** (POTENTIALFIELD),
- $b4$ between writing **PFcmd** (POTENTIALFIELD) and reading **PFcmd** (SAFETYPILOT),
- $b5$ between reading **PFcmd** (SAFETYPILOT) and writing **Cmd** (SAFETYPILOT).

The events of read/write are localized thanks to the codels arguments (we know from the model which codels write/read which ports). So a port p writing is complete at the end of execution of each codel that writes it. We can therefore formulate the property for *e.g.* $b2$ as follows:

```
(leave Navigation/navigate_task/GotoPosition/state stop_exec) leadsto (leave
    PotentialField/plan_task/TrackTarget/read_ports) within [min,max]
```

with max being the sought upper bound (we fix min to 0) and *leave* denoting leaving the state, which is equivalent to the end of execution of the underlying codel. The inserted observer has a state *error* whose reaching implies the violation of the property. Thus, here also the property depends only on the markings, so we can use the Markings-Only construction. We use the *on-the-fly* mode to optimize the time needed to tune max. The overall maximum bound is equal to

$$b1 + b2 + b3 + b4 + b5 = 202.5 + 101.1 + 1.1 + 40.1 + 3.5 = 348.3 \text{ ms}.$$

This result is of a crucial importance as it allows the robotic programmer to deduce critical information from this current setup. For instance, we may conclude that the robot driving at 2 m/s will advance at most 0.7 m after sending the *Stop* request and before a null speed is sent to the controller.

Bounded processing (quadcopter) We use again the Fiacre pattern *leadsto within* to formulate this property, *e.g.* for MIKROKOPTER's control task:

```
(mikrokopter/CT/state busy) leadsto (mikrokopter/CT/state idle) within [min,max]
```

The control task is initially in the state *idle*. When a request is received, it transits to *busy*. Different behaviors corresponding to processing the request are possible from *busy* to *end* through intermediate states (a simplified model is given in Sect. 4.4.3, Fig. 4.9 without those intermediary states because we do not consider control services when presenting the semantics). When execution finishes, a transition is taken back to *idle* from *end*. The property above expresses that if the control task starts executing, it will eventually go back to *idle* (after execution finishes) in a bounded amount of time. We prove *max* to be equal to 0.1 ms for MIKROKOPTER. For each component, the computed value of max is acceptable (short enough) with no more than 0.2 ms, which is five times smaller than the smallest period in the specification.

7.2.1.3 Verification with UPPAAL (Version 4.1)

In UPPAAL, one may express the properties in the *verifier* tab of the graphical interface or use the *verifyta* command. Note that results are given using the *conservative* state-space reduction. The other available options *aggressive* and *extreme* proved to be slower in both of our case studies.

Proper init (Osmosis & quadcopter) Similarly to the Fiacre model, two global arrays of initially false booleans *ports_read* and *ports_write* are generated in each model. The *proper init* is then typically a safety property in UPPAAL that must evaluate to true for each port p:

```
A[] (ports_read[p] imply ports_write[p])
```

The verification result are identical to those obtained with TINA, that is the property being violated for the port **Pose** in Osmosis. The graphical diagnosis simulator in UPPAAL eases the analysis of the counterexample and, in consequence, acting accordingly (Fig. 7.2).

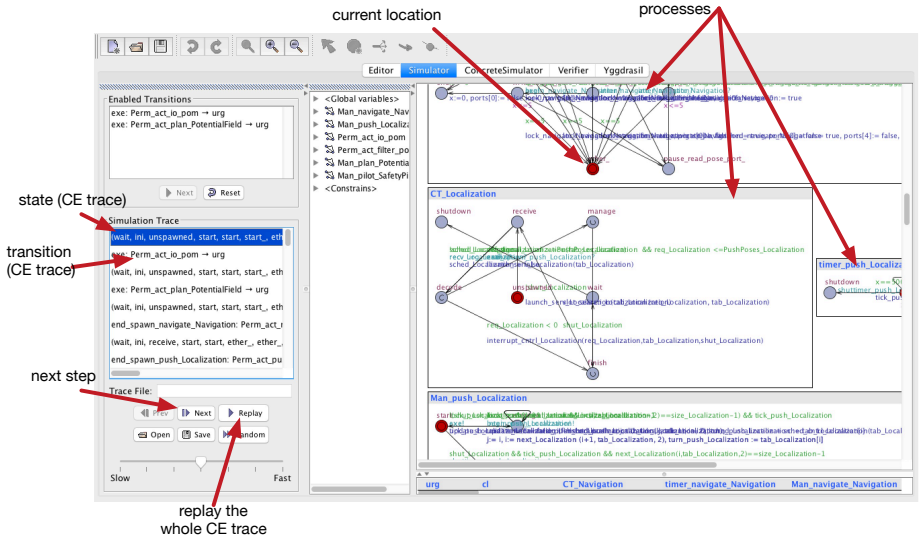


Figure 7.2: Diagnosis in UPPAAL simulator (screenshot from the Osmosis case study. CE abbreviates "counterexample").

Progress (Osmosis) As it is the case for Fiacre, this is a liveness property that we can formulate using the leadsto $--->$ operator, *e.g.* for task plan (POTENTIALFIELD):

```
manager_plan.manage --> manager_plan.wait
```

The results are in line with those obtained with TINA, *i.e.* the property holds for all execution tasks in the Osmosis navigation application.

Schedulability (quadcopter) Similarly to the Fiacre model, if a task T is schedulable, then the variable *sig_T* is never *true* when the process *manager_T* is at state *manage* (Sect. 6.3.2). It is thus sufficient to verify the corresponding safety property, e.g. for filter (component POM):

```
A[] (manager_filter.manage imply not tick_io)
```

The results agree with those obtained using TINA, that is all tasks are schedulable in the quadcopter stationary flight application.

Bounded stop (Osmosis) In contrast to Fiacre, UPPAAL does not provide an automatic support for modeling bounded response properties. Indeed, it is the user's task to model the observers and modify the original model accordingly in order to capture the events that appear in the property. Let us see how this is done through a generic example.

Fig. 7.3 shows a generic representation of the observer used to verify bounded response properties. It has two locations, *start*, where the triggering event *source* is captured, and *wait*, where the response event *target* is expected. The edge from *start* to *start* corresponds to ignoring some *source* events upon their reception (synchronized over the urgent channel *exe*). The edge from *start* to *wait*, on the other hand, models taking into account some *source* events upon their reception (also synchronized over

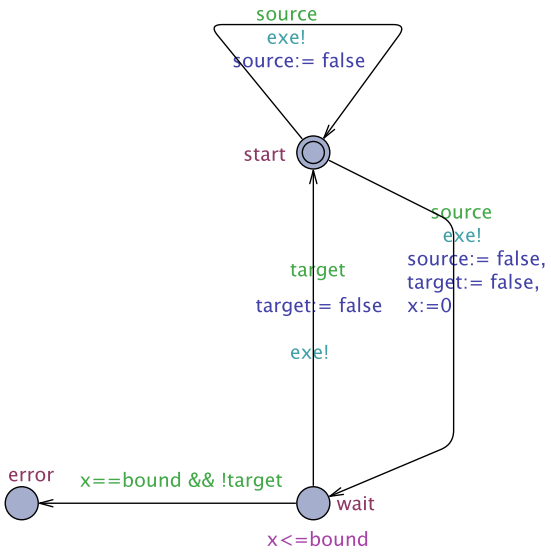


Figure 7.3: UPPAAL observer (bounded stop property).

exe). This allows computing the upper bound for all occurrences of *source*. That is, although the i^{th} occurrence of *source* is ignored in the scenario where it occurs at location *wait*, it is taken into account in the scenario where its predecessor, the $(i-1)^{th}$ occurrence, is ignored at location *start*. Now, at location *wait*, the event *target* is expected within $bound$ time units, tracked thanks to the clock x and checked through the invariant $x \leq bound$. Otherwise, the location *error* is reached.

Now, the events *source* and *target* need to be updated in the generated UPPAAL model for the GenoM3 specification. That is, if *e.g.* the objective is to compute *bound 3* (between reading **Target** and writing **PFcmd** (POTENTIALFIELD), see the same property verified with TINA previously), the end of execution of each codel in POTENTIALFIELD that reads **Target** (respect. writes **PFcmd**) must mark a new occurrence of *source* (respect. *target*). This means that each edge corresponding to the end of each codel in POTENTIALFIELD that reads **Target** (respect. writes **PFcmd**) must be augmented with the operation $source := true$ (respect. $target := true$).

Once the model is modified as explained above, the observer is added to the parallel composition and the value of *bound* is tuned for each bound (from *b1* to *b5*, see the same property verified with TINA above). This value corresponds to the smallest positive integer that satisfies the following safety property (for each bound):

```
A[] not observer.error
```

Finally, we sum the obtained bounds to get the same value as with TINA, that is $348.3ms$.

Bounded processing (quadcopter) We can use the same observer idea to verify this property, since it is a bounded response property as well. The results are in line with those obtained with TINA.

7.2.1.4 Verification with BIP

RTD-Finder uses SAT solving techniques. Typically, properties directly verifiable by the tool are safety properties, which is the case for *schedulability*, *proper init*, but also *bounded stop* when observers are modeled as in UPPAAL, after replacing properly the variables with timed automata due to the absence of support of data variables in RTD-Finder, as emphasized in Sect. 3.6 and Sect. 6.3.3. Unfortunately, this limited scope of the tool, combined with the absence of support of urgencies, makes the BIP TA models quite large due, for instance, to the proportionality between the number of locations/edges of a given *lock* and the codels in conflict with its associated codel (Sect. 6.3.3). The verification process leads quickly to a memory blow-up even when tested on our simplest components (*e.g.* DEMO, Sect. 2.2.2).

7.2.1.5 Discussion

When the theory of TA was introduced in [Alur and Dill, 1994], the authors argued that a main advantage of TA compared to other formalisms like Timed Petri Nets [Ramchandani, 1974] is the ability to express the time elapsed in a whole path (not only between taking two successive transitions). This advantage is contrasted with a less obvious, yet equal, compositional expressiveness when compared to TPN or UTA with regards to urgencies. Indeed, as seen in Sect. 6.3.3, expressing urgencies in TA is relatively painful in compositional contexts and leads to larger representations compared to those based on TPN or UTA. This was one of the motivations to introduce UTA and benefit from both the aforementioned advantage (expressing time easily through a whole execution path) and an easy and efficient expression of urgencies. In [Hsiung et al., 2006], the authors corroborate as they describe why UTA are needed in many real-world contexts (such as robotics).

The experience depicted here constitutes a useful feedback on RTD-Finder. RTD-Finder developers are currently investigating the memory blow-up. Consequently, they are developing a new version of the tool where the *linear method* [Bensalem et al., 2013] replaces Binary Decision Diagrams. They are also working on extending RTD-Finder to UTA to avoid scalability issues with the large TA models. TINA and UPPAAL, on the other hand, give promising results despite the scalability issues elaborated below. The advantages of both tools make their use complementary and might be optimized by choosing them according to the properties of interest.

Tables 7.1 and 7.2 present the cost of verification (in terms of time and memory consumption) of the Osmosis navigation and the quadcopter stationary flight, respectively. Hereafter, We shed the light on some advantages of both TINA and UPPAAL in verifying various properties of our real-world robotic applications. We thus enable practitioners to optimally use these tools depending on the property of interest. These conclusions are experience based and results may differ for other applications/properties. In any case, engineers may benefit from the less expensive and/or user-friendlier verification. Practitioners may also *cross-check* the properties of their specification using both tools if they judge it necessary.

TINA

- *Markings-Only construction:* A considerable amount of time and memory is saved with this construction. We advise practitioners to favor this construction, when using TINA, for an efficient verification, in terms of time and memory, of safety and bounded response properties.

Property	TINA		UPPAAL	
	time(s)	memory peak (MB)	time(s)	memory peak (MB)
Proper init	38	368	42	228
Progress	54	474	43	228
Bounded stop (bound *b2*)	61	505	84	560

Table 7.1: Verification results: Osmosis

Property	TINA		UPPAAL	
	time(s)	memory peak (MB)	time(s)	memory peak (MB)
Proper init	161	1051	199	978
Schedulability	159	998	200	980
Bounded processing	167	1012	201	902

Table 7.2: Verification results: quadcopter

- *Observers:* TINA observers are implemented automatically and do not need any effort from the user. That is, the practitioner needs only to express the bounded response properties as Fiacre patterns without having to specify the necessary observers to verify such properties. This is very useful as robotic practitioners usually have neither the time nor the required knowledge to edit formal models manually.

UPPAAL

- *Performance* UPPAAL is typically fast and shows a steady memory usage. Practitioners may benefit from UPPAAL's performance especially with *leads to* properties where execution sequences may not be ignored.

- *Diagnosis* UPPAAL offers several options for analyzing and replaying counterexamples, automatically loaded in its user-friendly simulator. Step-wise simulation in the graphical editor is very useful to practitioners. Indeed, they may replay the scenario leading to the violation of the property, without any additional effort from their end, which eases the diagnosis.

Scalability Despite promising results with both tools, scalability issues are quickly encountered when trying to verify larger applications. Indeed, if we generate the models corresponding to all the components in the Osmosis case study (real robot), the verification of properties fails due a memory blowup after dozens of hours of computation. The same outcome is reached when trying to verify properties on the navigation application (quadcopter) that involves all the components in the functional level (MANEUVER was not involved in the stationary flight application). At this stage, we hit the limits of the used model-checking tools due to combinatory explosion. This is where statistical model checking and runtime enforcement of properties give alternative solutions that are discussed, respectively, in Sect. 7.2.2 and Sect. 7.3 below.

```
1  process scheduler (&fifo: queue N of 1..N, &release: array 1..N of bool,
       &cores: 0..P) is
2  states start
3  from start
4         on (not empty fifo) and (cores > 0);
5         wait [0,0];
6         cores:= cores-1;
7         release [first fifo]:= true;
8         fifo := dequeue fifo;
9  to start
```

Listing 7.1: Fiacre model of the FCFS scheduler

7.2.1.6 Integrating hardware constraints

In the current practice, formal verification of robotic and autonomous systems usually ignores hardware constraints (numbers of processors/cores and scheduling policy). This restricts the validity of results to the cases where the number of processors/cores in the platform is at least equal to that of the robotic tasks, which is not always the case in reality. For instance, the stationary flight application can be run on the ODROID-C0 board, featuring four cores, while it has ten tasks. We give in [Foughali et al., 2018] some preliminary results of experiments on verifying this application while taking into account the hardware real characteristics, namely the number of cores and the scheduling policies. The considered schedulers are First Come First Served (FCFS) and Shortest Job First (SJF). The work uses the Fiacre template but can be generalized to other templates. We summarize the extension of the template and the results on schedulability below.

First-Come-First-Served (FCFS) Scheduler The preemptive FCFS scheduling policy is based on serving jobs in the order of their arrival while allowing higher priority tasks to preempt lower priority ones. Here, we choose a cooperative version of FCFS (preemption is not allowed)[2]. To model the scheduler, three shared variables are introduced into the Fiacre model. The first one is a queue, named `fifo`, which represents the list of tasks identifiers ordered by tasks activation dates. Tasks identifiers, *i.e.* the elements of `fifo`, are positive integers ranging from 1 to N, where N is the number of tasks in the robotic specification. The second shared variable is an array of boolean values, named `release`, that represents the signals to release tasks. Each task is statically associated to a specific index of this array, *i.e.* the task whose identifier is `i` can execute only when `release[i]` is set to `true`. The last shared variable, `cores`, is an integer that ranges from 0 to P. P is the number of cores provided by the platform and is thus the initial value of `cores`.

Listing 7.1 gives an overview of the scheduler Fiacre model. It is a process parameterized with `fifo`, `release` and `cores` (line 1). It has only one state called *start* (line 2). A self-loop transition is taken (line 9) (urgently (line 5)) only when the `fifo` is not empty **and** there is at least one available core (line 4). This allows the first task in the queue to execute by assigning an available core (line 6), sending the right release signal through `release` (line 7) and dequeuing the `fifo` (line 8).

Now, we show the relation between the scheduler and periodic tasks. The manager

[2]For more details about the policy, both versions (preemptive and cooperative) are studied in [Schwiegelshohn and Yahyapour, 1998].

```
1  process manager_n (..., &sig_n: bool, &fifo: queue N of 1..N, &release: array 1..N
       of bool, &cores: 0..P) is
2  states ask, start, manage
3  from ask
4         wait [0,0];
5         on sig_n;
6         sig_n:= false;
7         fifo:= enqueue(fifo, n); to start
8  from start
9         wait [0,0];
10        on release[n];
11        .../* update variables */;
12        to manage
13 from manage
14        /* if more activities to execute */
15        ... /* execute activities */
16        /* else */
17        ... /* update variables */
18        cores:= cores+1;
19        release[n]:= false;
20        to ask
```

Listing 7.2: Fiacre model of a the manager (with scheduling)

evolves (compared to the model given in Sect. 6.3.1) to adapt to schedulers. Listing 7.2 shows the new `manager` process for a generic execution task with identifier `n`, associated to the index `n` of the array `release`. The process `manager` has now three states (line 2): `ask`, `start` and `manage`. The transition from `ask` to `start` (lines 3-7) represents the activation of the task at the beginning of its period (when receiving the signal from the timer, equivalent to the transition from *start* to *manage* when schedulers were not a part of the model, Sect. 6.3.1). Upon activation, the task identifier is inserted at the back of `fifo` (using the primitive $enqueue$). State `start` (lines 8-12 in Listing 7.2) is used to wait on the release signal, that happens as soon as `release[n]` becomes *true*. Finally, when the manager finishes executing activities, it releases the core, updates the `release` array and transits back to `ask` (line 20) to wait for the next period signal.

Shortest-Job-First (SJF) Scheduler The SJF scheduler, *aka* SPN (Shortest Process Next), is a cooperative scheduler based on priorities related to the jobs estimated execution time (*aka* burst time) [Lupetti and Zagorodnov, 2006]. That is, the insertion of a job in the waiting queue is based on its Estimated Execution Time (EET) rather than its activation date. Jobs with equal EET will be sorted in a FIFO fashion, as in FCFS.

To encode this scheduler, we need first to define the EET for a GenoM3 task. We consider the period as the EET for a periodic execution task. For aperiodic tasks, including the control tasks, the programmers expect these to react and execute as promptly and quickly as possible. Their EET is thus considered shorter than any EET of a periodic task.

The Fiacre encoding is therefore similar to that of FCFS except that the insertion in the queue *fifo* is done through the Fiacre recursive function $insert\_sjf$ (Listing 7.3) rather than the classical *enqueue* primitive. This function ensures that the jobs are inserted according to their respective EET if different, and to their activation date otherwise (the primitive *append* is used to insert an element in the front of a queue).

The function eet, called within $insert\_sjf$, returns for each task its EET. Listing 7.4 shows how we can generate such a function from a GenoM3 component c using

```
1  function insert_sjf (q: queue N of 1..N, t: 1..N) : queue N of 1..N is
2  var temp: 1..N
3  begin
4         if (empty(q) or eet(t) < eet(first(q))) then
5               return append(q,t)
6         end if;
7         temp:= first(q);
8         return append(insert_sjf (dequeue(q), t),temp)
9  end
```

Listing 7.3: Queue insertion function for the SJF scheduler

```
1  function eet (t: 1..<"[expr [llength [$c tasks]] + 1]">) : nat is
2  begin
3         case t of
4         1 →return 0
5  <'set k 2
6  foreach task [$c tasks] {
7         if {![catch {$task period}]} {'>
8         | <"$k"> →return <"[$task period]">
9  <'} else {'>
10        | <"$k"> →return 0
11 <'}
12 incr k}'>
13        end
14 end
```

Listing 7.4: Generating the function eet for a component c

the Fiacre template. In line 1, the expression between markers will be replaced by the number of tasks in c (the $+1$ is for counting the control task as well, always present in a GenoM3 component). The statement **case ... of** (line 3) is similar to the **switch case** statement in the C language. The first clause of the **case ... of** statement (line 4) returns 0 for the control task, encoded by the integer 1. This ensures that the control task EET is always smaller than any EET of a periodic execution task. The same goes for aperiodic execution tasks (line 10). For periodic execution tasks, the function simply returns their periods (line 8).

Schedulability with FCFS (results) The cooperative FCFS scheduler can be easily implemented by using the real-time policy SCHED_FIFO on Ubuntu with all tasks given the same priority.

The schedulability property is expressed similarly as when schedulers where not involved. The main difference here is that now being at either state *start* or *manage* while *sig* becomes *true* means the violation of the property. For example, for the task io (component POM), schedulability is formulated as follows:

```
property sched_io is always (not (pom/manager_io/state ask)
                  ⇒not (pom/manager_io/value sig_pom))
```

We prove that all tasks are schedulable, considering the given hardware and a cooperative FCFS scheduling policy. We also prove that the minimum number of cores to ensure such schedulability is 3 as the task filter (component POM) misses its deadline when this number is reduced to 2. Table 7.3 summarizes the results according to the number of cores.

Cores	sched_main (MIKROKOPTER)	sched_main (NHFC)	sched_publish (OPTITRACK)	sched_io (POM)	sched_filter (POM)
4	True	True	True	True	True
3	True	True	True	True	True
2	True	True	True	True	False
1	False	False	False	False	False

Table 7.3: Schedulability results with FCFS

Schedulability with SJF (results) Since the EETs are assigned statically, this scheduler is easily implementable using the same real-time policy as that used for FCFS, namely SCHED_FIFO on Ubuntu, with the following differences:

- Priorities: Priorities are assigned according to the rules explained above: the smaller the period, the higher the priority, and aperiodic tasks have the same, highest priority.
- Preemption: To prevent higher priority tasks from preempting lower priority ones, a semaphore is initialized to the number of cores. A wait operation on this semaphore is added at the beginning of each task and a signal operation at the end.

The verification results are given in Table 7.4. With SJF, the minimum number of cores to ensure the schedulability of all tasks is 2.

Cores	sched_main (MIKROKOPTER)	sched_main (NHFC)	sched_publish (OPTITRACK)	sched_io (POM)	sched_filter (POM)
4	True	True	True	True	True
3	True	True	True	True	True
2	True	True	True	True	True
1	False	False	False	False	False

Table 7.4: Schedulability results with SJF

Discussion We prove that using SJF improves the schedulability of tasks as the application may be run on a dual-core platform while the minimum number of cores needed with FCFS is 3. By analyzing the traces of counterexamples, we realize that the scenarios where the task filter (component POM) misses its deadline correspond to execution paths where the task publish (component OPTITRACK) position in the queue precedes that of filter. These paths are eliminated in the SJF context thanks to the $insert\_sjf$ function (listing 7.3) since the EET of publish (4 ms) is larger than the EET of filter (1 ms).

Taking into account the hardware constraints gives a higher value to the results, because the model is the closest to the real hardware-software setting. The integration of the schedulers can be automatized thanks to the templates, which makes the approach accessible to robotic engineers. However, this needs to be generalized to more optimized cooperative schedulers such as cooperative EDF and Highest Response Rate Next (HRRN). Also, this approach is not suitable for preemptive schedulers, as preemption leads to unmanageable state spaces.

7.2.2 Statistical model checking

We use statistical model checking techniques as an alternative to model checking when the latter fails to scale. As an example, we rely on the navigation application (quadcopter) whose generated formal models scaled neither with UPPAAL nor with TINA[3]. Before elaborating on the properties of interest, presenting the results and discussing the advantages and limitations of this technique, we explain how GenoM3 models can be enriched with probabilities to fully benefit from the features of UPPAAL-SMC.

7.2.2.1 Enriching GenoM3 with probabilities

As seen in Chapt. 2, activities specifications may be non-deterministic. That is, a codel may have more than one successor, *e.g.* the codel `wait` of the activity *launch* in listing 6.1:

```
codel<wait> mv_exec_wait(in trajectory, in reference, out desired)
       yield pause::wait, path, servo wcet 1 ms;
```

This non-determinism, as explained in Chapt. 2, is solved only at runtime. Still, UPPAAL-SMC features augmenting non-deterministic edges with probabilities, as seen in Sect. 3.4. The idea is thus to collect *a posteriori* execution information to decorate the models with probabilities over non-deterministic transitions. For this, the regular middleware templates are enriched to generate, at the end of execution, the number of occurrences of each transition within each activity (in the same way they are enriched to generate execution times for WCET estimation in Sect. 2.3.2).

After running an application *app*, these occurrence numbers are generated automatically in an *app.proba* file where each line has the following syntax:

```
component_name/task_name/source_codel_name/target_codel_name <#occurrence>
```

This file is then passed as an argument to the UPPAAL-SMC template, together with the *dotgen* file, to generate the statistical model enriched with probabilities. Let us show, through an excerpt of the UPPAAL-SMC template (listing 7.5), how non-deterministic edges are automatically enriched with probabilities[4]. For simplicity, we show only the case where the source codel is thread safe and none of its ongoing transitions is a pause, interruption, or termination transition.

```
1  <'foreach comp [dotgen components] {'>
2  <'  foreach t [$comp tasks] {'>
3  <'    foreach s [$t services] {'>
4           ...
5  <'      foreach c [$s codels] {'>
6          ...
7  <'        if {[llength [$c yields]] > 1} {'>
8  <'        set p [join [list [$comp name] [$t name] [$tr cname] [$y cname]] /]'>
9  <"[$c name]"> → <"[$c name]">_b {guard x>0; },
10 <'          foreach y [$c yields] {'>
11 <"[$c name]">_b → <"[$y name]"> {; probability <"[dict get $argv $p]">; },
12 <'          }}'>
13         ...
14 <'      }'>
15       ...
16 <'    }}}'>
```

[3]The UPPAAL-SMC model of the quadcopter navigation application and the verification results are available at git://redmine.laas.fr/laas/users/mfoughal/case-study-quadcopter.git

[4]We recall that the number of occurrences is what UPPAAL-SMC refers to as probabilities, Sect. 3.5.

17

Listing 7.5: Automatic enriching of edges with probabilities (UPPAAL-SMC)

Line 7 conditions adding probabilities by the existence of non-determinism, *i.e.* the codel has more than one successor. Line 8 maps the edge from the source codel to a branchpoint (as shown in Sect. 6.3.2). Lines 9-12 generate the outgoing edges of the branchpoint and extract the probabilities from the *.proba* file.

For example, when applied on the codel `wait` of activity *launch* (MANEUVER of quadcopter, listing 6.1), listing 7.5 gives the following output[5]:

```
1   wait →wait_b {guard x>0;},
2   wait_b →wait {; probability 100},
3   wait_b →path {; probability 3},
4   wait_b →servo {; probability 30},
```

7.2.2.2 Adding the client

The client for the navigation application (quadcopter) is given in Fig. 7.4. The self-loop at location $navigate$ enables issuing a new *goto* request each time the last served *goto* activity has ended (goal invalid, reached, or unreachable). From the same location, a request *wait* can be sent, followed by a *take_off* request to land. The client covers thus all the possible scenarios of navigation. The channels $recv\_X$ and $recv\_urg\_X$ (X is a component) and the variables req_X are defined similarly to those in the UPPAAL client for Osmosis (Fig. 7.1). The main difference is that $recv\_X$ and $recv\_urg\_X$ are now broadcast channels, since handshake synchronization channels are not supported by UPPAAL-SMC (Sect. 3.5). To guarantee that broadcast channels behave as handshake ones, we use the boolean s_X for each component X that is true only when the component is ready to receive a request. It suffices then to guard each edge using the broadcast channel $recv\_X$ or $recv\_urg\_X$ with s_X (Fig. 7.4). The location $hold$ is for waiting between sending *servo_nhfc* and *servo* requests (NHFC and MIKROKOPTER) and the requests of MANEUVER. The waiting time is initially 3 seconds in the specification of the client[6]. This is because servoing must have already started before taking off, which is an important property to verify as we will see in the next section. The other difference between the clients is the exponential rates, necessary on invariant-free locations, that we fix here at 10000 (see the justification of using high exponential rates in Sect. 3.5).

7.2.2.3 Properties of interest

Schedulability: Estimate the probability of the schedulability of the periodic tasks in POM, MIKROKOPTER and NHFC, which must be the highest possible.

Progress: Estimate the probability of progress for each service, which must be the highest possible.

Readiness: When the client starts sending requests for MANEUVER, the previously requested services of MIKROKOPTER and NHFC must have already started executing. This is an important property since MANEUVER receives navigation goals while MIKROKOPTER and NHFC are in charge of low-level hardware and

[5] Once more, this listing is simplified to show probabilities, pause statements and mutual exclusion are ignored.

[6] 30000 in the UPPAAL-SMC model since the smallest time constraint in this application is 0.1 ms.

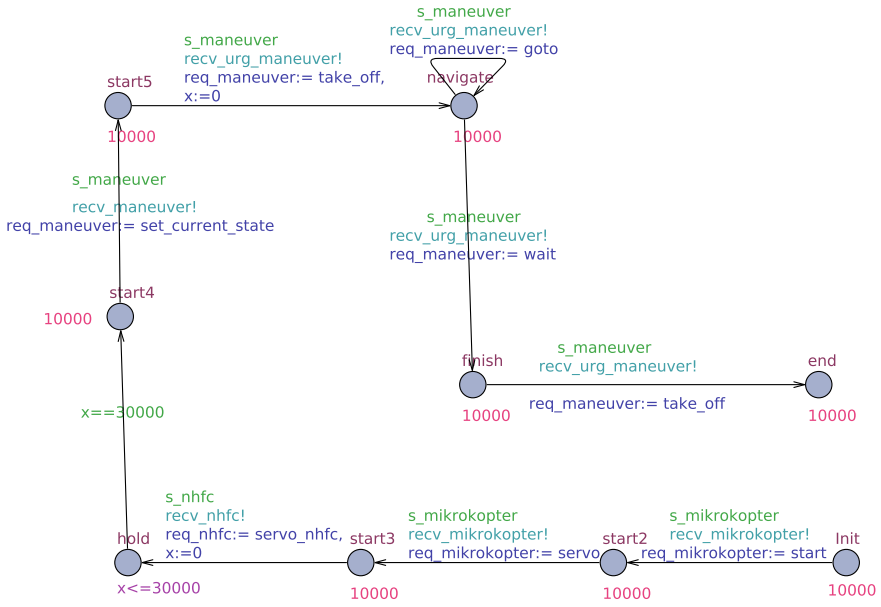


Figure 7.4: UPPAAL-SMC client (quadcopter).

servo control, respectively (Sect. 2.4). It follows that sending a navigation goal to MANEUVER while MIKROKOPTER or NHFC has not started yet is dangerous. Estimate the probability of satisfying this property that must be the highest possible.

7.2.2.4 Verification with UPPAAL-SMC

Statistical parameters are set to ensure a high confidence (0.98) and precision (0.005). The runs are bounded to 10 s ($Pr[<= 100000]$ in table 7.5, the base unit is 10^{-4} s) after verifying that the probability to reach the final state of the client within this bound is higher than 99%.

Schedulability It is reduced to a reachability property as seen in Sect. 7.2.1.3. The probability of violating this property is the lowest possible for all execution tasks of the critical components POM, MIKROKOPTER and NHFC ($< 1\%$, table 7.5) considering the value of the precision (0.005 ± 0.005). This means that schedulability is verified with the highest possible probability ($> 99\%$) given the chosen precision (0.005 ± 0.005). This result means also that we have one chance in 100 to violate the property. Trying to reduce the probability of violation (by reducing the precision parameter) leads to remarkably long computations (see discussion below).

Progress This is a "leads to" property, not supported by UPPAAL-SMC. As said previously, schedulability implies liveness (the first is stricter than the second). For MANEUVER and OPTITRACK, we propose then to verify schedulability as a stricter alternative to liveness (table 7.6). The highest possible probability is returned by the verifier. This means that, overall, we manage to verify that all tasks are schedulable with the high probability of 99%.

Task	Query	Results	Runs	Time (s)
Mikrokopter (main)	$Pr[<= 100000](<>$ $manager\_main.manage$ and $sig\_main)$	$Pr \in [0, 0.00998069]$	*390*	*948*
Pom (io)	$Pr[<= 100000](<>$ $manager\_io.manage$ and $sig\_io)$	$Pr \in [0, 0.00998069]$	*390*	*966*
Pom (filter)	$Pr[<= 100000](<>$ $manager\_filter.manage$ and $sig\_filter)$	$Pr \in [0, 0.00998069]$	*390*	*962*
Nhfc (main_nhfc)	$Pr[<= 100000](<>$ $manager\_main\_nhfc.manage$ and $sig\_main\_nhfc)$	$Pr \in [0, 0.00998069]$	*390*	*975*

Table 7.5: Analysis results for schedulability (Base unit 10^{-4} ms.)

Task	Query	Results	Runs	Time (s)
Maneuver (exec)	$Pr[<= 100000](<>$ $manager\_exec.manage$ and $sig\_exec)$	$Pr \in [0, 0.00998069]$	*390*	*964*
Maneuver (plan)	$Pr[<= 100000](<>$ $manager\_plan.manage$ and $sig\_plan)$	$Pr \in [0, 0.00998069]$	*390*	*958*
Optitrack (publish)	$Pr[<= 100000](<>$ $manager\_publish.manage$ and $sig\_publish)$	$Pr \in [0, 0.00998069]$	*390*	*927*

Table 7.6: Analysis results for progress (Base unit 10^{-4} ms.)

Readiness Readiness is typically a bounded response property, not supported by UPPAAL-SMC. We propose an alternative to express this property using the *Until* operator. An activity is ensured to have started once its codel `start` has began executing. Since none of the codels `start` in this context is thread safe, beginning to execute an activity is equivalent to reaching the location *start_exec*. Now, the location *start4* of the *client*, starting of which requests to MANEUVER are sent, must not be reached before the locations *start_exec* of each previously requested activity is reached. The bounded response property boils down then to the conjunction of three *Until* properties (table 7.7). Note that attempting to reduce these properties to only one using the conjunction of the right terms of *Until* would result in a stricter property (*e.g. start_exec* of *servo* may be left before *start_exec* of *servo_nhfc* is reached). Note also that one needs to estimate the probability of reaching location *start4* within the 10 s time bound. This probability is trivially maximal since it is maximal for reaching location *end* within the same bound (already estimated) and the latter cannot be reached before *start4* is reached (sequential behavior). The highest possible probability is returned by the verifier for each activity, which means that this property holds with a probability of 99%.

Service	Query	Results	Runs	Time (s)
Mikrokopter (start)	$Pr[<= 100000]$ $(not\ cl.start4$ $U\ start.start\_exec)$	$Pr \in$ $[0.990019, 1]$	390	2
Mikrokopter (servo)	$Pr[<= 100000]$ $(not\ cl.start4$ $U\ servo.start\_exec)$	$Pr \in$ $[0.990019, 1]$	390	2
nhfc (servo_nhfc)	$Pr[<= 100000]$ $(not\ cl.start4$ $U\ servo\_nhfc.start\_exec)$	$Pr \in$ $[0.990019, 1]$	390	2

Table 7.7: Analysis results for readiness (Base unit 10^{-4} ms.)

7.2.2.5 Discussion

While we cannot verify some properties in a precise way (due to scalability issues with model checking), the results we get with UPPAAL-SMC are encouraging. We may assert, up to a high probability, the truth of important properties, which is clearly better than classical scenario-based testing. Moreover, the verification is cost effective. Indeed, besides acceptable verification time (around 15 minutes in the worst case), UPPAAL-SMC shows a remarkably low memory consumption (less than 15 mb). This approach is therefore promising.

Nevertheless, three main issues are encountered, besides non exhaustivity. First, it is hard to set the probabilities at which we want the properties to hold. For this application, satisfying properties up to 99% may sound fair, but we actually do not know if it is[7] due to the absence of precise requirements expressed probabilistically. In the robotics domain, we need more regulatory work to define standards on fault tolerance, which we are severely lacking today. Second, despite a remarkable low consumption of memory, the verification time becomes unbearably long (in the order of days sometimes) when we tune the statistical parameters to reach higher probabilities. Third, the expressiveness of UPPAAL-SMC query language is quite limited. The absence of support for *leads to* and bounded response properties is particularly disabling. It is true that we often may manage, with some artefacts, to propose and verify closer alternatives. However, this limited scope of expressiveness remains a serious limitation since such artefacts need a proficiency with formal languages that robotic practitioners do not possess.

Finally, the process of obtaining some probabilities in the model can be enhanced. This is typical in "mode change" transitions where the number of occurrences is not quite representative of the probability. In the component NHFC, for instance, one of the codels of the permanent activity of task main is defined as follows:

```
codel<init> nhfc_main_init(ids in reference, in state, out rotor_input)
      yield pause::init, control;
```

Here, the number of occurrences of each transition does not really reflect the probability of taking it, because the idea is to repeat the self loop (from `init` to `init`) until servo control is needed (the first valid position is received). A more representative model that takes into account this mode switch would give a greater value to the verification results.

[7] Since, also, this result means also that we have one chance in 100 to violate the property.

7.3 Runtime enforcement of properties

We rely on runtime verification to enforce desired properties online. We carry out the experiments on the navigation application (Osmosis). We generate the BIP model for all the components in the application that involves the real robot (that does not scale either with UPPAAL or with TINA). This model is then augmented with a monitor to enforce a timed property. The BIP generated model, augmented with the monitor, is then run on the real Robotnik robot.

7.3.1 Properties of interest

In laser-based navigation, it is important to verify that laser data are regularly updated when the robot is moving. Indeed, failure to read new data from the sensor during motion may lead to collision with obstacles and therefore damaging the robot or injuring humans. The property consists thus in always writing new laser data in a bounded amount of time when the robot is moving. That is, each time the port **laser** (LASERDRIVER (Sect. 2.4)) is written, it will be rewritten before a *timeout* occurs, which means that there is a maximum amount of time separating two successive writes on the port. If the property is violated, it means that there is a serious problem such as a starving phenomenon (the codel is waiting forever to get access to the port it writes) or a sensor defect. The robot must thus urgently abandon its mission and stop moving.

We also visualize the violation of timing constraints extracted from the specification, that is the task periods and WCET of the codels.

7.3.2 Enforcement with BIP

After we automatically generate the online BIP model for the PocoLibs implementation[8], we augment it with a monitor to enforce the desired property.

First, we create a BIP atom *monitor* which verifies online the correctness of the property:

```
1   atom type property()
2
3   clock c unit millisecond
4   export port Port scan(), report(), go(), finish()
5
6   state idle, busy
7
8   initial to idle
9
10  on go
11  from idle to busy
12  eager
13  reset c
14
15  on scan
16  from busy to busy
17  provided (c≤ timeout)
18  eager
19  reset c
20
21  on finish
22  from busy to idle
23  provided (c≤ timeout)
```

[8]The model for ROS-Comm is also available but we choose PocoLibs for the reasons given in Sect. 2.3.2.

```
24              eager
25
26              on report
27              from busy to idle
28              eager
29              provided (c> timeout)
30
31              end
```

Starting to move is captured via the port *go*. The port *scan* corresponds to the event of writing the port **laser** within the constant *timeout*. The port *report* corresponds to detecting the violation of the property. Finally, the port *finish* must be triggered when the motion ends (goal reached, invalid, unreachable). It follows that state *idle* (respect. *busy*) corresponds to the robot at rest (respect. at motion), that is no (respect. one) instance of activity *GotoPosition* is being currently executed. Notice the non determinism at *busy* when $c \leq timeout$ with no real impact on the desired behavior (if *scan* is triggered first when both *scan* and *finish* are possible, *finish* will follow immediately anyway).

Second, we need to create connectors that link ports automatically generated in the model with the ports of the atom *monitor* so they correspond to the wanted events as explained above. We will see how this works for ports *scan* and *report*, for instance. The event that *scan* corresponds to is writing the port **laser**. We need thus a connector that involves both *scan* and the connector within the compound *laserdriver* (generated from the GenoM3 component LASERDRIVER) that corresponds to writing the GenoM3 port **laser**. This connector is defined as *Write_Laser*. Now, we create a broadcast connector involving both parties as follows:

```
connector trig2 Scan_OK (LaserDriver.Write_Laser, Monitor.scan)
```

Where *trig2* is a broadcast connector type with two ports (the first is the sender), *Monitor* an instance of atom *monitor* and *LaserDriver* an instance of *laserdriver*. The choice of a broadcast connector here is justified by the fact that writing the laser must be possible even when the robot is not moving. When the robot is moving, the maximal interaction is guaranteed by the BIP engine.

Now, the port *report* in the monitor must correspond to the property violation. As explained before, when the property is violated the robot must urgently abandon its mission and stop moving. In order to do so, we couple the triggering of *report* with the generation of a *Stop* request in the component SAFETYPILOT in order to force writing a null speed to its **Cmd** port[9]. The port triggering the behavior following a *Stop* request in the compound *safetypilot* (generated automatically from the GenoM3 component SAFETYPILOT) is defined as *Req_Stop*. We need thus to create a rendezvous connector involving *report* (from the monitor) and *Req_Stop*:

```
connector sync2 Scan_Failed (SafetyPilot.Req_Stop, Monitor.report)
```

Where *sync2* is a rendezvous connector type with two ports and *SafetyPilot* an instance of *safetypilot*.

We set the timeout to 100 ms, which is the period of POTENTIALFIELD, the component in charge of potential-field navigation. The robot fulfills its missions correctly. We inject then some delays in the codel responsible for writing **laser** (*e.g.* using the *sleep()* function) and visualize how the monitor interrupts the missions and forces the robot to stop quickly. We may see in this case within the execution trace that the engine

[9] Which makes the robot stop moving when applied by ROBOTDRIVER, Sect. 2.4.

forces taking the connector *Scan_Failed*, which results in executing the codel `stop` of the activity *MergeAndAvoid* (SAFETYPILOT)[10]:

```
[BIP ENGINE]: state #165351: 1 interaction:
[BIP ENGINE]: [0] ROOT.Scan_Failed: SafetyPilot.Req_Stop()
     Monitor.report() ] 26s591ms324us108ns, +INFTY ]
[BIP ENGINE]: →choose [0] ROOT.Scan_Failed: SafetyPilot.Req_Stop()
     Monitor.report() at global time 26s591ms324us109ns
...
[GenoM3] SafetyPilot Calling SafetyPilot_activity_MergeAndAvoid_stop
     codel.
[GenoM3] SafetyPilot Exiting SafetyPilot_activity_MergeAndAvoid_stop
     codel with ::SafetyPilot::ether.
```

We try different values for the timeout and realize that a constraint as small as 40 ms is too strict as the monitor stops the robot often. This might mean that we should reconsider the period that we give to *SafetyPilot* (which also relies on the LRF).

The BIP engine allows us also to visualize violating schedulability and when codels execution does not respect the WCET, which might be useful to further tune these constraints. For this, we use a flexible execution mode provided by the engine, that tolerates violating time invariants by emitting a warning instead of stopping the execution. Below is a warning issued by the engine pointing out that an invariant within the process *init* in the compound instance *PotentialField* (mapping the activity *init* in the component POTENTIALFIELD) is violated. The messages given by GenoM3 help localizing where the invariant was violated (in this example, the WCET of the codel `start`).

```
[GenoM3] PotentialField Calling PotentialField_activity_Init_start codel.
[GenoM3] PotentialField Exiting PotentialField_activity_Init_start codel with
     ::PotentialField::ether.
[BIP ENGINE]: WARNING: state #903017 and global time 1min3s230ms653us976ns:
     violation of the following timing constraint ROOT.PotentialField.init:
[BIP ENGINE]: ROOT.PotentialField.Init invariant [ -INFTY, 1min3s229ms530us282ns ]
```

To detect the violation of periods constraints for execution tasks, the online model of the timer is more dependent on the task manager. It allows detecting the violation of schedulability by making the timer wait each time until all activities to execute in a cycle are paused or finished. Therefore, if the execution takes longer than the period, the period invariant in the timer atom is violated. The log below shows a warning from the engine relative to the violation of the schedulability property in the task filter (component POM):

```
[BIP ENGINE]: WARNING: state #1905764 and global time 2min57s370ms722us604ns:
     violation of the following timing constraint ROOT.pom.timer_filter:
[BIP ENGINE]: ROOT.Compound_pom.timer_filter invariant [ -INFTY,
     2min57s363ms458us605ns ]
```

7.4 Discussion

Using runtime verification can be efficient, especially when properties may be enforced online. With the use of the BIP engine and automatically generated models, we may build monitors to check the properties online and act accordingly when they are violated. The timeout property that we enforce is crucial to the safety in applications involving laser-based navigation.

[10] And therefore a zero speed is sent to the controller.

However, this efficiency of the engine comes at a visible cost in performance. We notice that using the engine in this application induces up to 15% of overhead in the processor load. Thus, acting on timing constraints in the specification according to the messages of the engine is not necessarily the right way to go. Indeed, the internal computations of the BIP engine induce delays and it is thus difficult to know whether period and WCET violations are due to the code itself or the engine performance. The same remark is valid with the fact that the value of the timeout is found too strict when set to 40 ms. The external constraints brought by the engine also makes it unsuitable for systems where timing constraints are critically small, such as the quadcopter case study. Another open issue is the way to act when properties are violated. Although our experiments show a proper stop of the robot, it is difficult to standardize the emergency routines that monitors need to implement when vital priorities do not hold at runtime.

7.5 Conclusion

We apply in this Chapter various formal verification techniques using automatically generated formal models (Chapt. 6) of real-world complex autonomous applications (Sect. 2.4). The obtained results are promising and the different techniques and tools can be used complementarily. The users are thus provided with advices on which techniques to resort to according to the properties to verify and what to do if some techniques fail to scale, which constitutes a contribution of this thesis. Moreover, getting the same results with both model checking tools used in this thesis (TINA and UPPAAL) in our case studies is reassuring with regard to the soundness of the generated models and the proper formulation of the properties. It also gives a greater value to the results, cross-checked using state-of-the-art model checking tools.

Our experiments also shed the light on real limitations and open challenges in formal verification of robotic and autonomous systems. Indeed, the feedback provided on the performance of the tools is of a high value since they are confronted to real-world, complex systems, as opposed to academic benchmarks. We believe that these results will not only help the practitioners to complementarily and optimally use the tools, but also the developers to enhance such tools and validate their capabilities on real systems. The conclusions derived after using multiple techniques and tools are potentially useful to better identify the major problems of applying formal methods to robotic systems and advance the state of the art in addressing them.

Chapter 8

Conclusion

We propose in this thesis a novel automatic generation approach that is mathematically sound, faithful to the robotic underlying components (without unrealistic abstractions) and with various target formal languages and tools. This automatization, combined with encouraging experimental results, advances the state of the art toward a more correct and less tedious formal verification of robotic and autonomous applications. We cover thus the contributions outlined in Sect. 1.5. This work may be, consequently, considered as a step forward in the direction of a safe deployment of autonomous systems in real-world critical applications in human environments.

Contribution 1: a clear semantics for the functional components First, we justify the choice of GenoM3, a component-based framework, for specifying and deploying the functional components of robotic applications (Chapt. 2). Then, we propose in (Chapt. 3) Timed Transitions Systems TTS as a suitable formalism to give unambiguous semantics to GenoM3. In Chapt. 4, we develop lightweight operational semantics for GenoM3 components in TTS, based on clear formal definitions of those components and their constituents. This helped us clarify several ambiguous aspects in GenoM3 and contributed to its evolution toward a true multithreaded model. Semanticizing a robotic framework for functional components that can be generated for popular middleware (*e.g.* ROS-Comm) is novel and crucial in order to develop sound translations into formal frameworks.

Contribution 2: automatic generation of formal models After giving complete unambiguous semantics to GenoM3 in TTS, we develop a translation into timed automata, that we prove sound using bisimulation (Chapt. 5). Then, we map the TTS and timed automata models into formal models in Fiacre, UPPAAL, UPPAAL-SMC and BIP, that we automatize using templates (Chapt. 6, Sect. 7.2.2). We ensure thus the automatic translation of any robotic specification written in GenoM3 into various formal languages and tools.

Contribution 3: rigorous mapping The formal models we develop are rigorous in the sense that no unrealistic abstractions are allowed. In particular, timing constraints are all taken into account, including task periods and WCETs of codels. These information are represented within the semantics model (Chapt. 4), carefully preserved when translating (Chapt. 5) and implemented in the target models through the target

languages constructs (Chapt. 6). The case studies that we consider are from real robotic and autonomous systems (Sect. 2.4). The results we obtain are thus valid and can be trusted (since the models are representative of the specifications) and exploited in practice (since the specifications are implemented on real systems).

Contribution 4: selecting the best verification method/tool We provide advices on how and when to use a given tool depending on the characteristics of the application and the properties to verify (Chapt. 7). Indeed, one of the results of our experiments—that justifies our multi-target approach—is that there is no "one-size-fits all" tool, which reflects a complementarity between the different approaches. For models that do not scale with classical model checking, we assist the robotic programmers on which alternative method to use depending on application- and property-related factors.

Future work Overall, the work presented in this thesis is of benefit to the robotics, the formal methods, and the real-time systems communities. It tackles the problem of the applicability of formal methods to the functional components of robotic and autonomous systems by confronting a range of formal verification techniques/frameworks/tools to complex real-time applications. Our experiments allow us, besides to evaluate the contributions of this thesis, to underline the problems encountered and consequently draw the directions for future work.

First, we lack a robotic-friendly language to express properties. This means that the practitioners need to formulate their properties in the targeted formal language (examples in Chapt. 7), which is often not convenient. Indeed, writing properties formally requires a good knowledge of the targeted language and the generated model which robotic programmers do not possess. It follows that an important axis of future work is to develop a property language for GenoM3 so robotic programmers can express properties conveniently (some directions may be investigated such that of [Py and Ingrand, 2004b] (section 5) or [Abdellatif et al., 2012] (section 4)). Dually, it would be easier and less costly if counterexamples generated by verification tools were also understandable for robotic programmers. We could envisage to translate such counterexamples into GenoM3 execution traces, understandable by robotic programmers, or even into scenarios that could be played directly in robotics simulators. Both of these objectives will require to extend the semantics of GenoM3 so it takes into account the properties/-traces languages. For instance, the semantics needs to define user-oriented "events" (as atomic propositions) that could be used to derive properties, which is already paved by the formalization work presented in this thesis. Examples on similar works (especially the interpretation of counterexamples) exist in the literature (mainly [Pecheur and Simmons, 2000]), but they are typically developed in untimed and higher level contexts. More research is needed in order to see whether it is possible to reproduce such developing environments and generic transformation chains in our lower level, timed setting.

Second, the generated models for offline verification (Fiacre, UPPAAL and UPPAAL-SMC) reduce the codels (that implement the actual code to execute) into their Worst Case Execution Time. That is, we consider that the code is written correctly. It would be more reliable if the verification of such models could be consolidated with the verification of the code itself by combining (possibly statistical) model checking and deductive techniques. For this, we need to further explore the existing methods and tools in order to come up with an approach that robotic programmers can benefit from with a minimal effort. As an intermediary step, we may enrich the models with more data-

related information such as codel arguments, which will allow the verification of other important properties such as data freshness in IDS/ports.

Third, in the current practice, the validity of verification results is restricted to the cases where the number of processors/cores in the platform is at least equal to that of the robotic tasks, which is not always the case in reality. We propose in this thesis a novel approach that allows the verification of robotic specifications while taking into account the specificities of the hardware. Some preliminary results are given considering a couple of cooperative schedulers. An obvious axis of future work is to explore further scheduling algorithms. We plan thus to add options to our templates to allow the user to specify, if they desire, the used scheduler in their application and the number of processors/cores in the platform. Comparing the results using different schedulers can further help the practitioner decide on how to schedule their tasks prior to the implementation.

Fourth, the use of either statistical model checking or runtime verification is subject to many interrogations. When it comes to statistical model checking, it is difficult to set the properties in terms of probabilities because we severely lack this kind of requirements in robotics. We need to investigate further the problem of interpreting probability estimation of properties. Also, the restricted query language of UPPAAL-SMC makes it hard to express a range of important properties in robotics. This forced us to reason on equivalent alternatives using the supported operators only. For a robotic programmer, this could be quite discouraging since it requires a good knowledge of the tool, the query language and the underlying logic. A possible future work consists therefore in developing query-to-query transformations that are transparent to the practitioner. Finally, UPPAAL-SMC supports stochastic behaviors where automata locations can have associated rates. This feature is interesting since it may be used in future work to verify some hardware-related properties such as energy consumption (as [Seceleanu et al., 2009]). For runtime verification, a major issue is to determine when one could deploy an unverified system (because of scalability issues) and rely only on enforcement of properties. Also, the behavior to adopt when a property does not hold is not obvious, and it is added manually. For future work, it will be useful to carry out a study on classes of properties that can be rather enforced online than checked offline, and those that we cannot deploy the system without checking them beforehand. For the latter, we may resort to statistical methods (assuming the probabilistic requirements are well defined) or try compositional approaches (*e.g.* ECDAR [David et al., 2010] and the newer versions of RTD-Finder that will take into account our feedback on the tool). Another proposition for future work is to automate runtime enforcement, which is quite challenging when we have realtime constraints.

Appendix A

Bisimilarity (Part II)

Discrete actions

Action st**:** From inference rules in table 5.9, to take st from γ, we must have:
$\gamma = (l, v) \models (v(xt) = Per)$ (1.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (1.a) and **Definition 21** (Rule (2)) we have:
$\phi(st) = I_{st} - Per$ and thus $\phi(st) = [0, 0]$ (1.b)
Now from inference rules in table 5.1, to take st we must have
$\psi = (s, \phi) \models (0 \in \phi(st))$ (1.c)
From (1.b) and (1.c) it follows that action st is possible at ψ.
We take now the action st from γ to reach the state γ'. From table 5.9 we have:
$\gamma' = (l', v') \models (l'(sig) = true \wedge v'(xt) = 0)$ and l' agrees with l otherwise (1.d)
We take the action st from ψ to reach the state ψ'. From table 5.1 we have:
$\psi' = (s', \phi') \models (s'(sig) = true \wedge \phi(st) = [Per, Per])$ and s' agrees with s otherwise (1.e)
From (1.d) and (1.e) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ', and from Sect. 5.3.2 (absence of effects on time constraints in activities) we conclude that the rule (5) in **Definition 21** is satisfied as well.
It follows that $\psi' \mathcal{R} \gamma'$.

Action sm**:** From inference rules in table 5.10, to take sm we must have
$\gamma = (l, v) \models (l(sig) = true \wedge l(\pi_M) = wait)$ (2.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (2.a) and **Definition 21** (Rules (1),(3)) we have at ψ:
$s(sig) = true \wedge s(\pi_M) = wait$ (2.b)
Now from inference rules in table 5.2, to take sm from ψ, we must have:
$\psi = (s, \phi) \models (s(sig) = true \wedge s(\pi_M) = wait)$ (2.c)
From (2.b) and (2.c) it follows that action sm is possible at ψ.
We take now the action sm from γ to reach the state γ'. From table 5.10 we have:
$\gamma' = (l', v') \models (l'(sig) = false \wedge l'(N) = N' \wedge l'(R) = R' \wedge l'(\pi_M) = manage)$ and l' agrees with l otherwise (2.d)
We take the action sm from ψ to reach the state ψ'. From table 5.2 we have:
$\psi' = (s', \phi') \models (s'(sig) = false \wedge s'(\pi_M) = manage \wedge s'(N) = N' \wedge s'(R) = R')$ and s' agrees with s otherwise (2.e)
From (2.d) and (2.e) and absence of external actions effect on the timer (Sect. 5.3.3), it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, from (2.a) and (2.d) (respect. (2.b) and (2.e)) we have $\Pi = M$ at both γ' and ψ' and thus both

satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action lm**:** From inference rules in table 5.11, to take lm we must have:
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing)$ (3.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (3.a) and **Definition 21** (Rule (1)) we have at ψ:
$s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing$ (3.b)
Now from inference rules in table 5.3, to take lm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing)$ (3.c)
From (3.b) and (3.c) it follows that action lm is possible at ψ.
We take now the action lm (rule $lm.1$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.d)
We take the action lm (rule $lm.1$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.e)
From (3.d) and (3.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.1$).

We take now the action lm (rule $lm.2$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.f)
We take the action lm (rule $lm.2$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.g)
From (3.f) and (3.g) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.2$).

We take now the action lm (rule $lm.3$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (3.h)
We take the action lm (rule $lm.3$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \notin s'(R))$ and s' agrees with s otherwise (3.i)
From (3.h) and (3.i) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.3$).

We take now the action lm (rule $lm.4$) from γ to reach the state γ'. From table 5.11 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise. (3.j)
We take now the action lm (rule $lm.4$) from ψ to reach the state ψ'. From table 5.3 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \notin s'(R) \wedge$
$(\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (3.k)
From (3.j) and (3.k) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3 with $\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}$)

are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.4$).
Note here that taking the action τ from γ' would give the state $\gamma'' = (l'', v'')$ such that $l''(\pi_A) = c$, $v''(x_A) = 0$ and l'' agrees with l' otherwise. We may easily deduce then that $\psi' \mathcal{R} \gamma''$ (Rule (5) through clause 5.2 with $\theta = 0$).

Action fm**:** From inference rules in table 5.12, to take fm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage)$ (4.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (4.a) and **Definition 21** (Rules (1),(3)) we have at ψ:
$s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage$ (4.b)
Now from inference rules in table 5.4, to take fm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage)$ (4.c)
From (4.b) and (4.c) it follows that action fm is possible at ψ.
We take now the action fm from γ to reach the state γ'. From table 5.12 we have:
$\gamma' = (l', v') \models (l'(\pi_M) = wait)$ and l' agrees with l otherwise (4.d)
We take the action fm from ψ to reach the state ψ'. From table 5.4 we have:
$\psi' = (s', \phi') \models (s'(\pi_M) = wait)$ and s' agrees with s otherwise (4.e)
From (4.d) and (4.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, from (4.a) and (4.d) (respect. (4.b) and (4.e)) we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ia**:** From inference rules in table 5.13 to take ia (rule $ia.1$ or $ia.3$) we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R)$ (5.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.a) and **Definition 21** (Rules (1), (5.1)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R)$ (5.b)
Now from inference rules in table 5.5, to take ia (rule $ia.1$ or $ia.3$) from ψ, we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R))$ (5.c)
From (5.a), (5.b) and (5.c) and edges equivalence (Sect. 5.3.4) it follows that action ia (rule $ia.1$ or $ia.3$) is possible at ψ.
We take now the action ia (rule $ia.1$) from γ to reach the state γ'. From table 5.13 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = stop \wedge v'(x_A) = 0 \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (5.d)
We take the action ia (rule $ia.1$) from ψ to reach the state ψ'. From table 5.5 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = stop \wedge ID_A \notin s'(R) \wedge (\phi' ea = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (5.e)
From (5.d) and (5.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking ia (rule $ia.1$).

We take now the action ia (rule $ia.3$) from γ to reach the state γ'. From table 5.13 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (5.f)
We take the action ia (rule $ia.3$) from ψ to reach the state ψ'. From table 5.5 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and

s' agrees with s otherwise (5.g)
From (5.f) and (5.g) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (5) are satisfied by γ' and ψ' ((5) is satisfied because $\Pi = M$ at both ψ' and γ' and thus $\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$). It follows that $\psi' \mathcal{R} \gamma'$ after taking $ia.1$ or $ia.3$.

From inference rules in table 5.13, to take ia (rule $ia.2$ or $ia.4$) we must have, besides the existence of an outgoing ia edge from c_{pause} (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.h)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.h) and **Definition 21** (Rules (1),(5.3)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \wedge ID_A \in s(R)$ (5.i)
Now from inference rules in table 5.5, to take ia (rule $ia.2$ or $ia.4$) from ψ, we must have, besides the existence of an outgoing ia edge from c (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \neq ether \wedge ID_A \in s(R))$ (5.j)
From (5.h), (5.i), (5.j) and edges equivalence (Sect. 5.3.4) it follows that action ia (rule $ia.2$ or $ia.4$) is possible at ψ.
Now, proving that $\psi' \mathcal{R} \gamma'$ after applying rule $ia.2$ (respect. $ia.4$) is identical to proving $\psi' \mathcal{R} \gamma'$ after applying rule $ia.1$ (respect. $ia.3$).

Action fa**:** From inference rules in table 5.14, to take fa we must have, besides the existence of an outgoing fa edge from c (to c'_{pause} (rule $fa.1$) or to $ether$ ($fa.2$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0)$ (6.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (6.a) and **Definition 21** (Rules (1),(5.2) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0)$ (6.b)
Now from inference rules in table 5.6, to take fa from ψ, we must have, besides the existence of an outgoing fa edge from c (to $c' \neq ether$ (rule $fa.1$) or to $ether$ ($fa.2$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0))$ (6.c)
From (6.a), (6.b), (6.c) and edges equivalence (Sect. 5.3.4) it follows that action fa is possible at ψ.
We take now the action fa (rule $fa.1$) from γ to reach the state γ'. From table 5.14 we have:
$\gamma = (l', v') \models (l'(\pi_A) = c'_{pause}, l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.d)
We take the action fa (rule $fa.1$) from ψ to reach the state ψ'. From table 5.6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = c' \neq ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.e)
From (6.d) and (6.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$). It follows that $\psi' \mathcal{R} \gamma'$.

We take now the action fa (rule $fa.2$) from γ to reach the state γ'. From table 5.14 we have:
$(l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.f)
We take now the action fa (rule $fa.2$) from ψ to reach the state ψ'. From table 5.6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and

s' agrees with s otherwise (6.g)
From (6.f) and (6.g) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ea**:** From inference rules in table 5.15, to take ea (rule $ea.1$) we must have:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether)$ (7.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.a) and **Definition 21** (Rules (1),(5.1)) we have at γ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether$ (7.b)
Now, from inference rules in table 5.7, to take ea (rule $ea.1$) from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether)$ (7.c)
From (7.b), (7.c) it follows that action ea is possible at ψ.
We take now the action ea (rule $ea.1$) from γ to reach the state γ'. From table 5.15 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = start \wedge v'(x_A) = 0)$ and l' agrees with l otherwise (7.d)
We take now the action ea (rule $ea.1$) from ψ to reach the state ψ'. From table 5.7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = start \wedge (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (7.e)
From (7.d) and (7.e) and absence of external actions effect on the timer (Sect. 5.3.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by γ' and ψ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 5.15, to take ea (rule $ea.2$) we must have, besides the existence of an outgoing ea edge from c:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0)$ (7.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.f) and **Definition 21** (Rules (1),(5.2)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge$
$\wedge (\phi(ea) = I_{ea} - \theta \vee \phi(fa) = I_{fa} - \theta)$ with $\theta = v(x_A)$ (7.g)
Now from inference rules in table 5.7, to take ea (rule $ea.2$) from ψ, we must have, besides the existence of an outgoing ea edge from c:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge \phi(ea) = I_{ea} - \theta \mid \theta > 0)$ (7.h)
From (7.f), (7.g) and (7.h) and edges equivalence (Sect. 5.3.4) it follows that action ea (rule $ea.2$) is possible at ψ.
We apply now the rule $ea.2$ from γ to reach the state γ' (table 5.15) then from ψ to reach the state ψ' (table 5.7). The proof that $\psi' \mathcal{R} \gamma'$ is similar to that when taking $ea.1$ (with replacing $start$ by c'.

Time actions From inference rules in table 5.17, to take d (rule $d.1$) we must have:
$(l, v) \models (v(xt) < Per \wedge l(sig) = false \wedge l(\pi_M) = wait)$ (8.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.a) and **Definition 21** (Rules (1),(2),(3)) we have at ψ:
$\phi(st) = I_{st} - a \wedge s(sig) = false \wedge s(\pi_M) = wait$ s.t. $a = v(xt)$ (8.b)
Now from inference rules in table 5.8, to take d (rule $d.1$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(sig) = false \wedge s(\pi_M) = wait)$ (8.c)
From (8.a), (8.b) and (8.c) it follows that action d ($d \in]0, Per - a]$) is possible at ψ.

We take now the action d ($d \in]0, Per - a]$) from γ to reach the state γ'. From table 5.17 (rule $d.1$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $v'(xt) = a + d$ (8.d)
We take now the action d from ψ to reach the state ψ'. From table 5.8 (rule $d.1$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d)$ (8.e)
From (8.d) and (8.e) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (the manager has the control and thus $\nexists A \in \mathcal{A} \mid s(A) = l(A) = ID_A$) are satisfied by γ' and ψ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 5.17 (rule $d.2$), to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \neq ether \wedge ID_A \notin R \wedge v(x_A) < W(c))$ (8.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.f) and **Definition 21** (Rules (1),(2),(5.2)) we have at ψ:
$\phi(st) = I_{st} - a \mid a < Per \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
s.t. $v(x_A) = b$ and $v(xt) = a$ (8.g)
Now from inference rules in table 5.8, to take d (rule $d.2$) from ψ, we must have:
$\psi = (s, \phi) \models (\phi(st) = I_{st} - a \mid a < Per \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
(8.h)
From (8.f), (8.g) and (8.h) it follows that action d ($d \in]0, min(W(c) - b, Per - a)]$) is possible at ψ.
We take now the action d ($d \in]0, min(W(c) - b, Per - a)]$) from γ to reach the state γ'. From table 5.17 (rule $d.2$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $(v'(xt) = a + d \wedge v'(x_A) = b + d)$ (8.i)
We take the action d from ψ to reach the state ψ'. From table 5.8 (rule $d.2$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d \wedge (\phi'(ea) = I_{ea} - b - d \vee \phi'(fa) = I_{fa} - b - d))$ (8.j).
From (8.i) and (8.j) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (through the clause 5.2 with $\theta = b + d$) are satisfied by γ' and ψ'.
It follows that $\psi' \mathcal{R} \gamma'$.

We have thus proven that for all discrete and time actions, if $\psi \mathcal{R} \gamma$ and action act is possible from γ s.t. $\gamma \xrightarrow{act} \gamma'$, then the same action act is possible from ψ s.t. $\psi \xrightarrow{act} \psi'$ and $\psi' \mathcal{R} \gamma'$. It follows that Ψ **(weakly) time simulates** Γ.

Appendix B

Mappings

```
process launch (&run_exec: RUN_exec, &ind_exec: IND_exec, &pi_exec: PI_exec, &mut:
     MUT) is
states ether, start, start_exec, wait_, wait_exec, path, path_exec, servo,
     servo_exec, stop

from ether
       wait [0,0];
       on (pi_exec = ID_launch);
       if run_exec[ind_exec].status = nominal then
       to start /* additional edge (starting) */
       else
       to stop /* additional edge (interruption) */
       end

from start
       wait [0,0];
       on not (mut[r_start_SetState] or mut[r_start_goto] or mut[r_exec_goto] or
            mut[r_wait_goto]);
       mut[r_start_launch] := true;
       to start_exec

from start_exec
       wait ]0,1]; /* ]0, wcet] */
       mut[r_start_launch] := false;
       to wait_

from wait_
       wait [0,0];
       on pi_exec = ID_launch;
       if run_exec[ind_exec].status = nominal then /* nominal behavior */
       on not (mut[r_start_SetState] or mut[r_exec_goto]);
       mut[r_wait_launch] := true;
       to wait_exec
       else
       to stop /* additional edge (interruption) */
       end

from wait_exec
       wait ]0,1];
       mut[r_wait_launch] := false;
       select /* non determinism */
               to path
              [] to servo
```

```
                [] ind_exec:= ind_exec + 1;
                ind_exec:= next_exec(run_exec, ind_exec);
                pi_exec:= M_exec;
                to wait_ /* pause wait */
        end

from path
        wait [0,0];
        on pi_exec = ID_launch;
        if run_exec[ind_exec].status = nominal then /* nominal behavior */
        on not (mut[r_start_SetState] or mut[r_exec_goto] or mut[r_wait_goto]);
        mut[r_path_launch] := true;
        to path_exec
        else
        to stop /* additional edge (interruption) */
        end

from path_exec
        wait ]0,2];
        mut[r_path_launch] := false;
        /* pause operations */
        ind_exec:= ind_exec + 1;
        ind_exec:= next_exec(run_exec, ind_exec);
        pi_exec:= M_exec;
        select /* non determinism */
                to wait_
                [] to path
        end

from servo
        wait [0,0];
        on not (mut[r_start_SetState] or mut[r_start_goto] or mut[r_exec_goto]);
        r_servo_launch := true;
        to servo_exec

from servo_exec
        wait ]0,1];
        mut[r_servo_launch] := false;
        /* to pause wait */
        ind_exec:= ind_exec + 1;
        ind_exec:= next_exec(run_exec, ind_exec);
        pi_exec:= M_exec;
        to wait_

from stop
        wait ]0, 0.5];
        /* termination */
        ind_exec:= ind_exec + 1;
        ind_exec:= next_exec(run_exec, ind_exec);
        pi_exec:= M_exec;
        to ether
```

Listing B.1: activity *launch* (Fiacre)

Figure B.1: The process Urgency

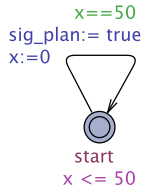


Figure B.2: The process timer_plan

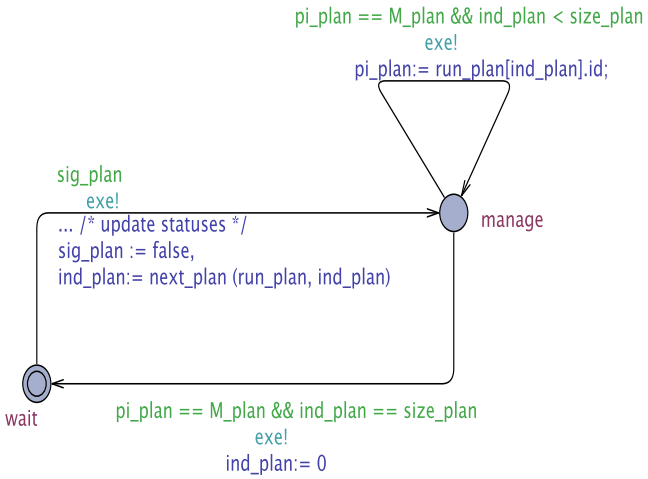


Figure B.3: The process manager_plan

Listing B.2: SetState process (UPPAAL)

```
process SetState(urgent chan &exe, int[M_plan, ID_goto] &pi_plan,
     int[0, size_plan] &ind_plan, CELL_plan &run_plan[size_plan], bool
     &mut[mut_nb]) {
clock x;
state ether, start, start_exec {x≤10};
init ether;
trans
 ether →start { guard pi_plan = ID_SetState && run_plan[ind].status
      == nominal; sync exe!; };
 ether →ether { guard pi_plan = ID_SetState && run_plan[ind].status
      == interrupted; sync exe!;
                     assign ind_plan:= ind_plan+1, ind_plan:=
                          next_plan (run_plan, ind_plan), pi_plan:=
                          M_plan; };
start →start_exec { guard !(mut[r_start_launch] || mut[r_wait_launch]
     || mut[r_path_launch] || mut[r_servo_launch]); sync exe!;
                     assign x:= 0, mut[r_start_SetState]:= true; };
start_exec →ether { guard x>0;
                           assign mut[r_start_SetState]:= false,
                                ind_plan:= ind_plan+1,
                                     ind_plan:= next_plan (run_plan,
                                          ind_plan), pi_plan:=
                                          M_plan; },
}
```

Listing B.3: Activity *goto* (UPPAAL)

```
process goto(urgent chan &exe, int[M_plan, ID_goto] &pi_plan, int[0,
     size_plan] &ind_plan, CELL_plan &run_plan[size_plan], bool
     &mut[mut_nb]) {
clock x;
state ether, start, start_exec {x≤10}, exec, exec_exec {x≤20}, wait,
     wait_exec {x≤5};
init ether;
trans
 ether →start { guard pi_plan = ID_goto && run_plan[ind].status ==
      nominal; sync exe!; };
 ether →ether { guard pi_plan = ID_goto && run_plan[ind].status ==
      interrupted; sync exe!;
                    assign ind_plan:= ind_plan+1, ind_plan:=
                         next_plan (run_plan, ind_plan), pi_plan:=
                         M_plan; };
start →start_exec { guard !(mut[r_start_launch] ||
     mut[r_servo_launch]); sync exe!;
                    assign x:= 0, mut[r_start_goto]:= true; };
start_exec →exec { guard x>0; assign mut[r_start_goto]:= false; };
exec →exec_exec { guard !(mut[r_start_launch] || mut[r_wait_launch]
     || mut[r_path_launch] || mut[r_servo_launch]); sync exe!;
                    assign x:= 0, mut[r_exec_goto]:= true; };
exec_exec →wait { guard x>0; assign mut[r_exec_goto]:= false; };
wait →wait_exec { guard !(mut[r_start_launch] || mut[r_path_launch])
                           && pi_plan = ID_goto &&
                                run_plan[ind].status == nominal;
                                sync exe!;
                           assign x:= 0, mut[r_wait_goto]:= true; };
wait →ether { guard pi_plan = ID_goto && run_plan[ind].status ==
     interrupted; sync exe!;
                    assign ind_plan:= ind_plan+1, ind_plan:=
                         next_plan (run_plan, ind_plan), pi_plan:=
                         M_plan; };
wait_exec →wait { guard x>0;
                           assign mut[r_wait_SetState]:= false,
                                ind_plan:= ind_plan+1,
                                     ind_plan:= next_plan (run_plan,
                                          ind_plan), pi_plan:=
                                          M_plan; },
wait_exec →ether { guard x>0;
                           assign mut[r_wait_SetState]:= false,
                                ind_plan:= ind_plan+1,
                                     ind_plan:= next_plan (run_plan,
                                          ind_plan), pi_plan:=
                                          M_plan; },
}
```

Bibliography

Abdeddaim, Y., Asarin, E., Gallien, M., Ingrand, F., Lesire, C., and Sighireanu, M. (2007). Planning robust temporal plans: A comparison between cbtp and tga approaches. In *International Conference on Automated Planning and Scheduling*, pages 2–10.

Abdellatif, T., Bensalem, S., Combaz, J., De Silva, L., and Ingrand, F. (2012). Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578.

Abid, N. and Dal Zilio, S. (2010). Real-time extensions for the fiacre modeling language. *International Summer School on Modeling and Verifying Parallel Processes*.

Abid, N., Dal Zilio, S., and Le Botlan, D. (2014). A formal framework to specify and verify real-time properties on critical systems. *International Journal of Critical Computer-Based Systems*, 5(1-2):4–30.

Adam, K., Hölldobler, K., Rumpe, B., and Wortmann, A. (2017). Modeling robotics software architectures with modular model transformations. *Journal of Software Engineering for Robotics*, 8(1):3–16.

Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337.

Albus, J. (1995). Rcs: A reference model architecture for intelligent systems. In *Working Notes: AAAI Spring Symposium on Lessons Learned for Implemented Software Architectures for Physical Agents*, pages 1–6.

Alur, R. and Dill, D. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

Andersen, S. and Romanski, G. (2011). Verification of safety-critical software. *Communications of the ACM*, 54(10):52–57.

Aniculaesei, A., Arnsberger, D., Howar, F., and Rausch, A. (2016). Towards the verification of safety-critical autonomous systems in dynamic environments. *arXiv preprint arXiv:1612.04977*.

Aştefănoaei, L., Rayana, S. B., Bensalem, S., Bozga, M., and Combaz, J. (2014). Compositional invariant generation for timed systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 263–278. Springer.

Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J. (2011). Rigorous component-based system design using the bip framework. *IEEE software*, 28(3):41–48.

Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K., and Lime, D. (2007). Uppaal-tiga: Time for playing games! In *International Conference on Computer Aided Verification*, pages 121–125. Springer.

Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. Springer.

Bensalem, S., Bozga, M., Boyer, B., and Legay, A. (2013). Incremental generation of linear invariants for component-based systems. In *International Conference on Application of Concurrency to System Design*, pages 80–89. IEEE.

Bensalem, S., Bozga, M., Nguyen, T.-H., and Sifakis, J. (2009). D-finder: A tool for compositional deadlock detection and verification. In *International Conference on Computer Aided Verification*, pages 614–619. Springer.

Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79:1270–1282.

Bérard, B., Cassez, F., Haddad, S., Lime, D., and Roux, O. H. (2005). Comparison of the expressiveness of timed automata and time Petri nets. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 211–225. Springer.

Berard, B., Cassez, F., Haddad, S., Lime, D., and Roux, O. H. (2013). The expressive power of time Petri nets. *Theoretical Computer Science*, 474:1–20.

Bernat, G., Burns, A., and Llamosi, A. (2001). Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321.

Berthomieu, B., Bodeveix, J.-P., Farail, P., Filali, M., Garavel, H., Gaufillet, P., Lang, F., and Vernadat, F. (2008). Fiacre: an intermediate language for model verification in the topcased environment. In *European Congress on Embedded Real-Time Software and Systems*.

Berthomieu, B., Dal Zilio, S., and Fronc, Ł. (2014). Model-checking real-time properties of an aircraft landing gear system using fiacre. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 110–125. Springer.

Berthomieu, B. and Menasche, M. (1983). An enumerative approach for analyzing time Petri nets. In *International Federation for Information Processing Congress*.

Berthomieu, B., Peres, F., and Vernadat, F. (2006). Bridging the gap between timed automata and bounded time Petri nets. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 82–97. Springer.

Berthomieu, B., Ribet, P., and Vernadat, F. (2004). The tool Tina – construction of abstract state spaces for Petri nets and time Petri nets. *Journal of Production Research*, 42(14).

Bjørner, D. and Havelund, K. (2014). 40 years of formal methods. In *International Symposium on Formal Methods*, pages 42–61. Springer.

Bornot, S., Sifakis, J., and Tripakis, S. (1998). Modeling urgency in timed systems. In *International Symposium on Compositionality: the significant difference*, pages 103–129. Springer.

Bourdil, P.-A., Berthomieu, B., and Jenn, E. (2014). Model-checking real-time properties of an auto flight control system function. In *International Symposium on Software Reliability Engineering Workshops*, pages 120–123. IEEE.

Boussinot, F. and de Simone, R. (1991). The ESTEREL Language. In *Proceeding of the IEEE*, volume 79, pages 1293–1304.

Bowen, J. and Stavridou, V. (1993). Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209.

Brafman, R., Bar-Sinai, M., and Ashkenazi, M. (2016). Performance level profiles: A formal language for describing the expected performance of functional modules. In *International Conference on Intelligent Robots and Systems*, pages 1751–1756. IEEE.

Brewer, E. (2012). Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29.

Brugali, D. (2015). Model-driven software engineering in robotics. *IEEE Robotics & Automation Magazine*, 22(3):155–166.

Bruyninckx, H. (2001). Open robot control software: the OROCOS project. In *International Conference on Robotics and Automation*, pages 2523–2528. IEEE.

Bryant, R. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

Bulychev, P., David, A., Larsen, K., Legay, A., Li, G., and Poulsen, D. (2012). Rewrite-based Statistical Model Checking of WMTL. In *International Conference on Runtime Verification (RV)*, pages 260–275. Springer.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L.-J. (1992). Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170.

Cimatti, A., Roveri, M., and Bertoli, P. (2004). Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1-2):127–206.

Clarke, E., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.

Clarke, E., McMillan, K., Campos, S., and Hartonas-Garmhausen, V. (1996). Symbolic model checking. In *International Conference on Computer Aided Verification*, pages 419–422. Springer.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., and Quesada, J. F. (2002). Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243.

Clements, P., Garlan, D., Little, R., Nord, R., and Stafford, J. (2003). Documenting software architectures: views and beyond. In *25th International Conference on Software Engineering*, pages 740–741. IEEE Computer Society.

Coad, P. and Nicola, J. (1993). *Object-oriented programming*. Yourdon Press Englewood Cliffs.

David, A., Larsen, K., Legay, A., Nyman, U., and Wkasowski, A. (2010). Ecdar: An environment for compositional design and analysis of real time systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 365–370. Springer.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

De Nicola, R., Ferrari, G. L., and Pugliese, R. (1998). Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on software engineering*, 5:315–330.

Demathieu, S., Thomas, F., André, C., Gérard, S., and Terrier, F. (2008). First experiments using the uml profile for marte. In *International Symposium on Object Oriented Real-Time Distributed Computing*, pages 50–57. IEEE.

Dennis, S., Alex, L., Matthias, L., and Christian, S. (2016). The smartmdsd toolchain: An integrated mdsd workflow and integrated development environment (ide) for robotics software. *Journal Of Software Engineering In Robotics*, 7(1):3–19.

Desai, A., Dreossi, T., and Seshia, S. A. (2017). Combining model checking and runtime verification for safe robotics. In *International Conference on Runtime Verification*, pages 172–189. Springer.

Dhouib, S., Kchir, S., Stinckwich, S., Ziadi, T., and Ziane, M. (2012). Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer.

Dwyer, M., Avrunin, G., and Corbett, J. (1999). Patterns in property specifications for finite-state verification. In *International Conference on Software engineering*, pages 411–420. ACM.

Echeverria, G., Lassabe, N., Degroote, A., and Lemaignan, S. (2011). Modular open robots simulation engine: Morse. In *International Conference on Robotics and Automation*, pages 46–51. Citeseer.

Elkady, A. and Sobh, T. (2012). Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*.

Emerson, A. and Srinivasan, J. (1988). Branching time temporal logic. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 123–172. Springer.

Faugere, M., Bourbeau, T., De Simone, R., and Gerard, S. (2007). Marte: Also an uml profile for modeling aadl applications. In *International Conference on Engineering Complex Computer Systems*, pages 359–364. IEEE.

Foughali, M. (2017). Toward a correct-and-scalable verification of concurrent robotic systems: Insights on formalisms and tools. In *International Conference on Application of Concurrency to System Design*, pages 29–38. IEEE.

Foughali, M., Berthomieu, B., Dal Zilio, S., Hladik, P.-E., Ingrand, F., and Mallet, A. (2018). Formal verification of complex robotic systems on resource-constrained platforms. In *International Conference on Formal Methods in Software Engineering*, pages 2–9.

Foughali, M., Berthomieu, B., Dal Zilio, S., Ingrand, F., and Mallet, A. (2016). Model checking real-time properties on the functional layer of autonomous robots. In *International Conference on Formal Engineering Methods*, pages 383–399. Springer.

Fu, M., Li, Y., Feng, X., Shao, Z., and Zhang, Y. (2010). Reasoning about optimistic concurrency using a program logic for history. In *International Conference on Concurrency Theory*, pages 388–402. Springer.

Gabel, M. and Su, Z. (2010). Online inference and enforcement of temporal properties. In *International Conference on Software Engineering*, pages 15–24. ACM/IEEE.

Gat, E. and Bonnasso, P. (1998). On three-layer architectures. *Artificial intelligence and mobile robots*, 195:210.

Genc, S. and Lafortune, S. (2003). Distributed diagnosis of discrete-event systems using Petri nets. In *International Conference on Application and Theory of Petri Nets*, pages 316–336. Springer.

Gibson-Robinson, T., Armstrong, P., Boulgakov, A., and Roscoe, A. (2014). Fdr3a modern refinement checker for csp. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer.

Giridhar, A. and Kumar, P. R. (2006). Distributed clock synchronization over wireless networks: Algorithms and analysis. In *International Conference on Decision and Control*, pages 4915–4920. IEEE.

Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F., Pinciroli, C., Brambilla, M., Birattari, M., and Dorigo, M. (2012). Towards a formal verification methodology for collective robotic systems. In *International Conference on Formal Engineering Methods*, pages 54–70. Springer.

Gobillot, N., Lesire, C., and Doose, D. (2014). A modeling framework for software architecture specification and validation. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 303–314. Springer.

Green, C. (1981). Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier.

Guerra, M., Efimov, D., Zheng, G., and Perruquetti, W. (2016). Avoiding local minima in the potential field method using input-to-state stability. *Control Engineering Practice*, 55:174–184.

Hähnel, D., Burgard, W., and Lakemeyer, G. (1998). Golex - bridging the gap between logic (golog) and a real robot. In *Annual Conference on Artificial Intelligence*, pages 165–176. Springer.

Halder, R., Proença, J., Macedo, N., and Santos, A. (2017). Formal verification of ros-based robotic applications using timed-automata. In *International Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50. IEEE/ACM.

Havelund, K. and Rosu, G. (2001). Java pathexplorer: A runtime verification tool. Technical report, NASA Ames Research Center.

Hazim, M., Qu, H., and Veres, S. (2016). Testing, verification and improvements of timeliness in ros processes. In *Conference Towards Autonomous Robotic Systems*, pages 146–157. Springer.

Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., and Vaandrager, F. (2004). Adding symmetry reduction to uppaal. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 46–59. Springer.

Henzinger, T., Manna, Z., and Pnueli, A. (1991). Timed transition systems. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 226–251. Springer.

Henzinger, T., Nicollin, X., Sifakis, J., and Yovine, S. (1994). Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244.

Hoare, C. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Holzmann, G. (1997). The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295.

Hsiung, P.-A., Lin, S.-W., Chen, Y.-R., Huang, C.-H., Yeh, J.-J., Sun, H.-Y., Lin, C.-S., and Liao, H.-W. (2006). Model checking timed systems with urgencies. In *International Symposium on Automated Technology for Verification and Analysis*, pages 67–81. Springer.

Huang, J., Erdogan, C., Zhang, Y., Moore, B., Luo, Q., Sundaresan, A., and Rosu, G. (2014). Rosrv: Runtime verification for robots. In *International Conference on Runtime Verification*, pages 247–254. Springer.

Huang, X., Kwiatkowska, M., Wang, S., and Wu, M. (2017). Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer.

Huet, G., Kahn, G., and Paulin-Mohring, C. (1997). The coq proof assistant a tutorial. Technical report, Institut National de Recherche en Informatique et en Automatique.

Ingrand, F. and Ghallab, M. (2017). Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44.

Ingrand, F., Lacroix, S., Lemai-Chenevier, S., and Py, F. (2007). Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580.

Jang, C., Lee, S.-I., Jung, S.-W., Song, B., Kim, R., Kim, S., and Lee, C.-H. (2010). Opros: A new component-based robot software platform. *Electronics and Telecommunications Research Institute journal*, 32(5):646–656.

Kapoor, A., Deguet, A., and Kazanzides, P. (2006). Software components and frameworks for medical robot control. In *International Conference on Robotics and Automation*, pages 3813–3818. IEEE.

Katz, G., Barrett, C., Dill, D., Julian, K., and Kochenderfer, M. (2017). Reluplex: An

efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer.

Kazanzides, P., Kouskoulas, Y., Deguet, A., and Shao, Z. (2012). Proving the correctness of concurrent robot software. In *International Conference on Robotics and Automation*, pages 4718–4723. IEEE.

Kim, M. and Kang, K. (2005). Formal Construction and Verification of Home Service Robots: A Case Study. In *International Symposium on Automated Technology for Verification and Analysis*, pages 429–443. Springer.

Knight, R., Chien, S., Gat, E., Starbird, T., Gostelow, K., Keller, B., and Smith, W. (2000). Integrating model-based artificial intelligence planning with procedural elaboration for onboard spacecraft autonomy. Technical report, California Institure of Technology, Pasadena Jet Propulsion Lab.

Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *International Conference on Intelligent Robots and Systems*, pages 2149–2154. IEEE.

Kortenkamp, D. and Simmons, R. (2008). Robotic systems architectures and programming. In *Springer Handbook of Robotics*, pages 187–206. Springer.

Kouskoulas, Y., Renshaw, D., Platzer, A., and Kazanzides, P. (2013). Certifying the safe design of a virtual fixture control algorithm for a surgical robot. In *International Conference on Hybrid Systems: Computation and Control*, pages 263–272. ACM.

Kramer, J. and Scheutz, M. (2007). Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132.

Kress-Gazit, H., Fainekos, G., and Pappas, G. (2008). Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359.

Kress-Gazit, H., Wongpiromsarn, T., and Topcu, U. (2011). Correct, reactive, high-level robot control. *IEEE Robotics & Automation Magazine*, 18(3):65–74.

Kwiatkowska, M., Norman, G., and Parker, D. (2011). Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer.

Larsen, K., Behrmann, G., Brinksma, E., Fehnker, A., Hune, T., Pettersson, P., and Romijn, J. (2001). As cheap as possible: effcient cost-optimal reachability for priced timed automata. In *International Conference on Computer Aided Verification*, pages 493–505. Springer.

Legay, A., Delahaye, B., and Bensalem, S. (2010). Statistical model checking: An overview. In *International Conference on Runtime Verification*, pages 122–135. Springer.

Leucker, M. and Schallhart, C. (2009). A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303.

Leveson, N. G. and Stolzy, J. L. (1987). Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, 3:386–397.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. (1997). Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83.

Ligatti, J. and Reddy, S. (2010). A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security*, pages 87–100. Springer.

Lomuscio, A., Qu, H., and Raimondi, F. (2009). Mcmas: A model checker for the verification of multi-agent systems. In *International Conference on Computer Aided Verification*, pages 682–688. Springer.

Lupetti, S. and Zagorodnov, D. (2006). Data popularity and shortest-job-first scheduling of network transfers. In *International Conference on Digital Telecommunications*, pages 26–26. IEEE.

Macek, K., Govea, D. A. V., Fraichard, T., and Siegwart, R. (2008). Safe vehicle navigation in dynamic urban scenarios. In *IEEE Conference on Intelligent Transportation Systems*.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (2010). GenoM3: Building middleware-independent robotic components. In *International Conference on Robotics and Automation*, pages 4627–4632. IEEE.

McCarthy, J. (1968). Situations, actions, and causal laws. *Semantic Information Processing*, pages 410–417.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl-the planning domain definition language. Technical report, Yale Center for Computational Vision and Control.

Medina, J. and Cuesta, A. G. (2011). Model-based analysis and design of real-time distributed systems with ada and the uml profile for marte. In *International Conference on Reliable Software Technologies*, pages 89–102. Springer.

Meng, W., Park, J., Sokolsky, O., Weirich, S., and Lee, I. (2015). Verified ros-based deployment of platform-independent control systems. In *NASA Formal Methods Symposium*, pages 248–262. Springer.

Merlin, P. and Farber, D. (1976). Recoverability of Communication Protocols: Implications of a Theoretical Study. *IEEE Transactions on Communications*, 24(9):1036–1043.

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., and Timmis, J. (2017). Automatic property checking of robotic applications. In *International Conference on Intelligent Robots and Systems*, pages 3869–3876. IEEE.

Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., and Woodcock, J. (2016). Robochart: a state-machine notation for modelling and verification of mobile and autonomous robots. Technical report, University of York.

Mohamed, N., Al-Jaroodi, J., and Jawhar, I. (2008). Middleware for robotics: A survey. In *International Conference on Robotics, Automation and Mechatronics*, pages 736–742. IEEE.

Molnar, L. and Veres, S. (2009). System verification of autonomous underwater vehicles by model checking. In *OCEANS-EUROPE Conference*, pages 1–10. IEEE.

Mowbray, T. and Zahavi, R. (1995). *The essential CORBA: systems integration using distributed objects*. Wiley New York.

Murray, R., Burdick, J., Perona, P., Cremean, L., Kriechbaum, K., Pfister, S., Foote, T., Gillula, J., Lamb, J., and Stewart, A. (2005). Darpa technical paper: Team caltech. Technical report, California Institure of Technology, Pasadena Jet Propulsion Lab.

Nayak, P., Kurien, J., Dorais, G., Millar, W., Rajan, K., Kanefsky, B., Bernard, D., Gamble, E., Rouquette, N., Smith, B., Muscettola, N., Taylor, W., and Tung, Y.-W. (1999). Validating the ds-1 remote agent experiment. In *Artificial Intelligence, Robotics and Automation in Space*, page 349.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media.

Pecheur, C. (2000). Verification and validation of autonomy software at nasa. Technical report, NASA Ames Research Center.

Pecheur, C. and Simmons, R. (2000). From livingstone to smv. In *International Workshop on Formal Approaches to Agent-Based Systems*, pages 103–113. Springer.

Petri, C. A. (1962). *Communication with automata*. PhD thesis, Technische Hochschule Darmstadt.

Platzer, A. (2008). Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189.

Py, F. and Ingrand, F. (2004a). Dependable execution control for autonomous robots. In *International Conference on Intelligent Robots and Systems*, pages 1136–1141. IEEE.

Py, F. and Ingrand, F. (2004b). Real-time execution control for autonomous systems. In *European Congress ERTS, Embedded Real Time Software*, pages 21–23.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, page 5.

Ramamoorthy, C. and Ho, G. (1980). Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Transactions on software Engineering*, 5:440–449.

Raman, V., Piterman, N., and Kress-Gazit, H. (2013). Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *International Conference on Robotics and Automation*, pages 4075–4081. IEEE.

Ramchandani, C. (1974). Analysis of asynchronous concurrent systems by Petri nets. Technical report, Defense Technical Information Center.

Rangra, S. and Gaudin, E. (2014). SDL to Fiacre translation. In *European Congress on Embedded Real-Time Software and Systems*.

Roscoe, A. (2010). *Understanding concurrent systems*. Springer Science & Business Media.

Schlegel, C., Haßler, T., Lotz, A., and Steck, A. (2009). Robotic software systems: From code-driven to model-driven designs. In *International Conference on Advanced Robotics*, pages 1–8. IEEE.

Schwiegelshohn, U. and Yahyapour, R. (1998). Analysis of first-come-first-serve parallel job scheduling. In *Symposium on Discrete Algorithms*, pages 629–638.

Seceleanu, C., Vulgarakis, A., and Pettersson, P. (2009). Remes: A resource model for embedded systems. In *International Conference on Engineering of Complex Computer Systems*, pages 84–94. IEEE.

Seshia, S., Sadigh, D., and Sastry, S. (2016). Towards verified artificial intelligence. *arXiv preprint arXiv:1606.08514*.

Shalev-Shwartz, S., Shammah, S., and Shashua, A. (2017). On a formal model of safe and scalable self-driving cars. *arXiv preprint arXiv:1708.06374*.

Sierhuis, M. and Clancey, W. (2002). Modeling and simulating practices, a work method for work systems design. *IEEE Intelligent Systems*, 17(5):32–41.

Simon, D., Espiau, B., Kapellos, K., and Pissard-Gibollet, R. (1997). Orccad: software engineering for real-time robotics. a technical insight. *Robotica*, 15(1):111–115.

Simon, D., Pissard-Gibollet, R., and Arias, S. (2006). Orccad, a framework for safe robot control design and implementation. In *National workshop on control architectures of robots: software approaches and issues*.

Smart, W. D. (2007). Is a common middleware for robotics possible? In *IROS workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*.

Smith, D., Frank, J., and Cushing, W. (2008). The anml language. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Sotiropoulos, T., Waeselynck, H., Guiochet, J., and Ingrand, F. (2017). Can robot navigation bugs be found in simulation? an exploratory study. In *International Conference on Software Quality, Reliability and Security*, pages 150–159. IEEE.

Sowmya, A., Tsz-Wang So, D., and Hung Tang, W. (2002). Design of a Mobile Robot Controller using Esterel Tools. *Electronic Notes in Theoretical Computer Science*, 65(5):3–10.

Stocker, R., Dennis, L., Dixon, C., and Fisher, M. (2012). Verifying brahms human-robot teamwork models. In *Proccedings of the International Conference on Logics in Artificial Intelligence*, pages 385–397. Springer.

Täubig, H., Frese, U., Hertzberg, C., Lüth, C., Mohr, S., Vorobev, E., and Walter, D. (2012). Guaranteeing functional safety: design for provability and computer-aided verification. *Autonomous Robots*, 32(3):303–331.

Todorov, V., Boulanger, F., and Taha, S. (2018). Formal verification of automotive embedded software. In *6th Conference on Formal Methods in Software Engineering*, pages 84–87. ACM.

Tomatis, N., Terrien, G., Piguet, R., Burnier, D., Bouabdallah, S., Arras, K., and Siegwart, R. (2003). Designing a secure and robust mobile interacting robot for the long term. In *International Conference on Robotics and Automation*, pages 4246–4251. IEEE.

Vardi, M. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, pages 322–331. IEEE Computer Society.

Veksler, O. (2003). Fast variable window for stereo correspondence using integral images. In *Computer Society Conference on Computer Vision and Pattern Recognition*, pages 556–561. IEEE.

Volpe, R., Nesnas, I., Estlin, T., Mutz, D., Petras, R., and Das, H. (2001). The claraty architecture for robotic autonomy. In *Proccedings of IEEE Aerospace Conference*, pages 1–121.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. (2008). The worst-case execution-time problemoverview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36.

Williams, B. and Nayak, P. (1996). A model-based approach to reactive self-configuring systems. In *National Conference On Artificial Intelligence*, pages 971–978.

Woodcock, J., Larsen, P., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM computing surveys*, 41(4):19.

Yu, W., Chen, T., Franchetti, F., and Hoe, J. C. (2010). High performance stereo vision designed for massively data parallel platforms. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(11):1509–1519.