UNIVERSITE BRETAGNE LOIRE

Centrale Nantes

Thèse de Doctorat

Louis-Marie GIVEL

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École centrale de Nantes
sous le sceau de l'Université Bretagne Loire

École doctorale : Sciences et technologies de l'information et mathématiques

Discipline : Informatique et applications, section CNU 27
Spécialité : Informatique
Unité de recherche : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)

Soutenue le 16 décembre 2016

Test de systèmes temps réel à l'aide du forçage en ligne

JURY

Président :	**M. Jean-Louis BOIMOND**, Professeur des universités, Université d'Angers
Rapporteurs :	**M. Philippe DHAUSSY**, Maitre de conférences, HDR, ENSTA Bretagne **M. Hacène FOUCHAL**, Professeur des universités, Université de Reims
Examinateur :	**M^me^ Karen GODARY DEJEAN**, Maitre de conférences, Université de Montpellier
Invité :	**M. Sébastien FAUCOU**, Maitre de conférences, Université de Nantes
Directeur de thèse :	**M. Olivier Henri ROUX**, Professeur des universités, École centrale de Nantes
Co-encadrant de thèse :	**M. Matthias BRUN**, Enseignant-chercheur, ESEO

Table des matières

Table des figures

Introduction

Contexte & Motivations

Les composants et les architectures des systèmes temps réel embarqués, du point de vue logiciel comme du point de vue matériel, font preuve d'une complexité grandissante.

Si l'on considère le standard AUTOSAR de l'industrie automobile pour les systèmes embarqués, les composants logiciels y sont générés à partir de modèles haut niveau puis inclus dans les tâches temps réel par un générateur de système. Ces tâches sont intégrées avec les composants AUTOSAR standards tels que le système d'exploitation temps réel, la couche de communication, les pilotes de périphériques, etc. L'architecture résultante pour un système embarqué automobile est une interaction complexe où le contrôle et les données circulent d'une tâche à l'autre.

De plus, d'un point de vue matériel, les architectures multicœurs, ou encore les micro-architectures orientées vers la performance telles que les processeurs superscalaires (capables d'exécuter simultanément plusieurs instructions) participent à l'augmentation de cette complexité.

L'intrication de ces différents composants et considérations tend à rendre le comportement des systèmes embarqués temps réel modernes difficiles à prédire et contrôler. Les comportements observés peuvent être considérés comme non déterministes, puisque la quantité de connaissance ainsi que la capacité d'analyse nécessaires pour prédire dans sa totalité le comportement des systèmes n'est pas disponible aujourd'hui.

Cependant, malgré et en raison de cette complexité grandissante, les systèmes temps réel doivent répondre à des contraintes de sûreté de fonctionnement qui doivent être testées. De part ces comportements qui peuvent apparaître non déterministes, il est compliqué pour une approche boîte noire de permettre le test des systèmes temps réel. Si l'on ne manipule que la trace d'entrée du système, il peut être difficile de lui faire atteindre l'état qui doit être testé.

Pour pouvoir permettre le test de ces systèmes, une solution peut être d'agir directement sur leur comportement interne, afin d'assurer qu'il corresponde aux attentes. Ce problème est connu sous le nom de forçage en ligne, ou runtime enforcement.

Un contrôleur est un composant logiciel dédié dont la fonction est de forcer un système à se conformer à un objectif défini formellement. Une approche formelle, basée sur des modèles mathématiques du système permet de générer ce contrôleur qui est ensuite inclut au système afin de garantir le comportement choisi.

Contributions

Dans ce travail de thèse, nous proposons une méthode qui utilise le forçage en ligne afin de faciliter le test de systèmes temps réel.

Notre démarche s'inscrit dans un contexte de conception basée sur les modèles, ou IDM (Ingénieurie Dirigée par les Modèles). Le modèle du système extrait des spécifications est utilisé de pair avec la formalisation d'un objectif de test pour générer un contrôleur temporel implémentant la stratégie de test permettant d'atteindre l'état qui doit être testé.

L'impact sur le système du contrôleur doit être minimal. Nous proposons un contrôleur qui ne peut qu'ajouter des délais au cours de l'exécution des tâches, et qui se situe à la frontière entre le système d'exploitation temps réel et l'application déployée sur celui-ci. Le contrôleur observe les appels systèmes effectués par les tâches de l'application, et peut agir sur leur exécution afin d'atteindre l'état choisi du système.

Cette approche peut être classifiée comme "boîte noire", puisque nous ne travaillons pas sur le contenu et la logique interne de l'application, mais uniquement sur son interface avec le système d'exploitation temps réel. Seul un modèle de l'interaction entre l'application et le système d'exploitation temps réel est nécessaire. Si ce modèle est disponible, il est possible de contrôler une application pour laquelle seul le binaire est disponible.

Nous proposons dans ce travail de thèse la théorie derrière cette solution de forçage en ligne, ainsi que l'implémentation d'une solution de contrôle pour le système d'exploitation temps réel Trampoline.

Organisation du manuscrit

Ce manuscrit est composé de trois parties.

La première se consacre au contexte et à l'état de l'art des différents domaines que nous utilisons dans ces travaux : le test des systèmes temps réel, l'injection de fautes, et le forçage en ligne.

La deuxième partie est consacrée au forçage en ligne des systèmes temps réel paramétrés. Elle présente tout d'abord la démarche dans son ensemble, puis se concentre sur chacune des étapes nécessaires à la réalisation du forçage, depuis la modélisation du système jusqu'à l'implémentation du forçage dans un système d'exploitation temps réel.

La dernière partie présente l'implémentation pour le système d'exploitation temps réel Trampoline.

Première partie

Contexte et état de l'art

Chapitre 1

Contexte

Nous proposons dans ce chapitre une vue d'ensemble des différents domaines et des différentes techniques employés au cours de ce travail de thèse : les systèmes temps réel, le test logiciel ainsi que le forçage en ligne.

Sommaire

1.1 Les systèmes temps réel

1.1.1 Définition

Les systèmes temps réel sont tous ces systèmes informatiques pour lesquels la contrainte d'exactitude de la réponse à un stimuli externe s'étend au domaine temporel en plus du domaine logique. En d'autres termes, une réponse sera considérée comme *correcte*, du point de vue d'un système temps réel, si le calcul effectué est correct et, surtout, si la

réponse est obtenue dans le laps de temps imparti, souvent avant l'expiration d'une limite temporelle, ou échéance (Martin, 1965 ; Shin & Ramanathan, 1994).

Ces systèmes, qui procurent ces garanties sur les aspects temporels de leur exécution, sont utilisés dans de nombreux domaines, tels que l'aéronautique, l'automobile, la production et la distribution d'énergie, les télécommunications, la robotique, etc.

La plupart des systèmes embarqués, présents dans nombre de ces domaines, sont d'ailleurs soumis à de telles contraintes temps réelles. Les systèmes embarqués sont les systèmes dont le fonctionnement est compris dans celui de systèmes plus larges. Il s'agit la plupart du temps de systèmes informatiques contrôlant des systèmes physiques et/ou électroniques. Ces systèmes sont soumis à des contraintes importantes, du point de vue du volume, des ressources disponibles (mémoire, capacité de calcul), et dans certains domaines de la criticité des fonctions auxquelles ils contribuent.

1.1.2 Temps réel dur vs souple

Le monde des systèmes temps réel est scindé la plupart du temps en deux catégories : le temps réel dur (ou strict), et le temps réel souple. Cette distinction vise à séparer les systèmes selon l'impact du non respect de l'échéance.

Pour le temps réel dur, l'impact est catastrophique si le résultat est obtenu après l'échéance (y compris dans une marge extrêmement faible après l'échéance). On peut imaginer l'impact sur la trajectoire d'une fusée d'un système de contrôle ne réussissant pas à répondre dans les temps.

Pour le temps réel souple, cet impact est faible et souvent lié à la qualité de service. On peut par exemple se permettre dans une application de télécommunication de perdre des paquets sans engendrer d'échec catastrophique. L'impact sera uniquement sur la qualité de la communication.

Une troisième catégorie que l'on peut rencontrer est celle du temps réel ferme : l'impact n'est pas catastrophique si le calcul n'est pas effectué dans les temps, mais le résultat est inutile. Considérons par exemple un système de prédiction (temps, résultats d'une compétition) : le résultat est inutile si l'évènement se produit avant la fin du calcul.

Dans ce travail de thèse, nous nous intéressons au domaine des systèmes embarqués temps réel. Les contraintes sur lesquelles nous nous concentrons sont des contraintes strictes : la réponse du système se doit d'être produite dans une fenêtre temporelle précise.

1.2 Le test logiciel

1.2.1 Définition

Afin de pouvoir mesurer l'adéquation des systèmes informatiques à leurs spécifications, il est nécessaire de les tester.

Le test logiciel consiste à exécuter le logiciel testé en lui fournissant des entrées et en observant ses sorties. Les sorties obtenues sont comparées avec des sorties attendues, et un verdict est rendu pour le test concerné. Contrairement à la vérification qui analyse le modèle pour garantir des propriétés du système, le test ne permet pas de prouver qu'un système est conforme à sa spécification. Il est cependant utile pour pouvoir détecter un maximum d'erreurs de l'implémentation, dans des situations où la vérification complète du logiciel n'est pas faisable.

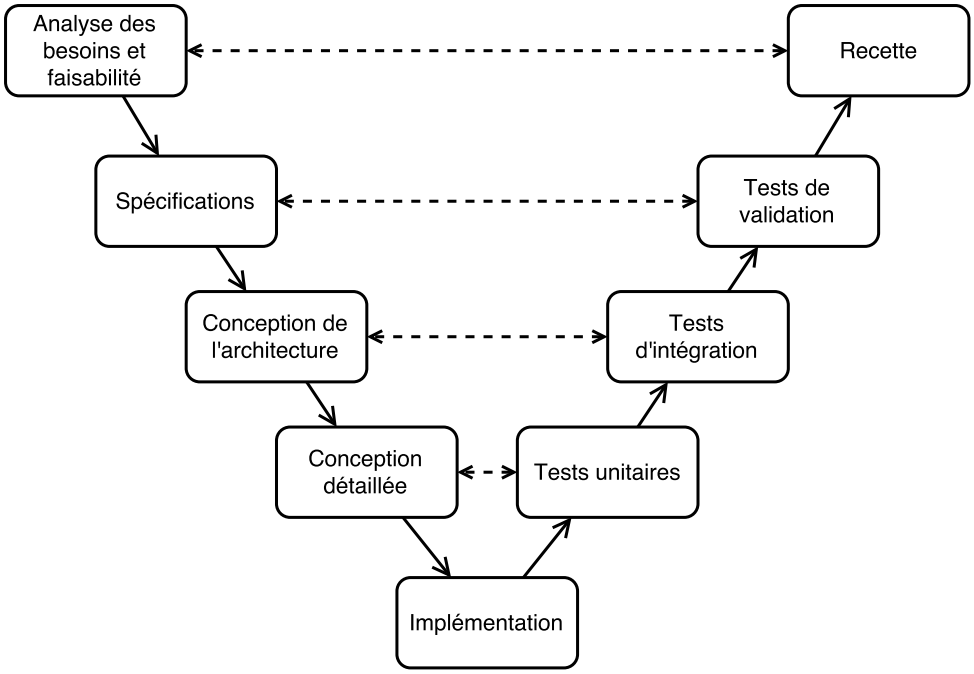


Figure 1.1 – Cycle en V

Le test logiciel permet de vérifier différentes propriétés d'un logiciel. On peut s'intéresser à la robustesse, vérifiant ainsi que le comportement de l'implémentation reste adéquat en présence de conditions extrêmes, d'entrées invalides, de pannes, etc. On peut d'autre part s'intéresser à la performance, mesurant le temps de réponse en fonction de la sollicitation du logiciel afin de connaître ses caractéristiques sous une charge lourde. On peut aussi s'intéresser au test des fonctionnalités d'un logiciel, permettant de valider l'implémentation des exigences spécifiées dans le cahier des charges.

Le test logiciel est intégré aux différentes méthodologies de développement logiciel à différents niveaux. Dans une démarche de développement guidé par les tests (Test-Driven Development, TDD), le test est présent avant même l'implémentation du programme à tester. L'ingénieur commence par programmer et automatiser des tests unitaires de manière à guider le développement (Janzen & Saiedian, 2005).

Dans le cycle en V, visible en figure 1.1, on retrouve le test à tous les niveaux de développement. Les tests unitaires des éléments du programme servent à valider l'implémentation de la conception détaillée, les tests d'intégration l'architecture, les tests de validation système l'adéquation aux spécifications.

1.2.2 Boîte noire et boîte blanche

Une classification fréquemment rencontrée en ce qui concerne le test logiciel est la distinction entre boîte noire et boîte blanche.

Dans le test boîte noire, le logiciel est vu, comme le nom de la catégorie l'indique, comme une boîte noire. Les seuls éléments manipulés et observés sont ses entrées et ses sorties, sans se soucier des éléments internes qui le composent. Le test boîte noire vise le plus souvent à garantir la conformité entre l'exécution du logiciel et ce que ses spécifications définissent. Sur le cycle en V, le test de validation est le plus souvent du test boîte noire.

Le test boîte blanche vise au contraire à ouvrir le logiciel et à venir valider son fonctionnement en ayant la connaissance de ses composants internes. Le test unitaire, par exemple,

s'intéresse au fonctionnement d'une "unité" du programme, qu'il s'agisse d'une fonction ou d'une classe (pour de la programmation orientée objet) et est le plus souvent réalisé en boîte blanche.

Contrairement au test boîte noire, le test boîte blanche se base sur une connaissance d'un modèle du fonctionnement du programme testé qui est le plus souvent une représentation du code source (graphe de flot de contrôle, formalismes tels que les automates ou les réseaux de Petri. . .). Cette connaissance de la structure du programme permet de le tester en garantissant des critères de couverture tels que tous les arcs, tous les sommets ou tous les chemins du graphe de flot de contrôle.

1.2.3 Processus de test

Un processus de test boîte noire vise à confirmer l'adéquation d'un système à ses spécifications en le soumettant à un ensemble de données d'entrée/sortie définies, et en comparant des sorties aux sorties attendues dans les spécifications. Cet ensemble de donnés d'entrées (qui peut inclure un aspect temporel) est ce que l'on appelle le cas de test. La figure 1.2 montre un processus classique de test logiciel. On peut y voir que le cas de test (test case) est obtenu à partir de l'objectif de test (test purpose), puis sert à nourrir le testeur afin de permettre d'effectuer le test.

L'objectif de test est la propriété du système à tester. Il décrit les conditions du test ainsi que le résultat attendu. Lorsqu'il est formalisé, c'est la spécification du cas de test.

Afin de pouvoir donner un verdict quant à l'adéquation du système à ses spécifications (ou à quantifier son (in)adéquation), le test doit être effectué sur une exécution du système. Il s'agit alors d'un test dynamique et non statique. L'approche statique pourrait inclure de la relecture de code, ou une exécution symbolique (King, 1976 ; Khurshid, Păsăreanu & Visser, 2003). Pour cela, il faut donc que le système s'exécute sur une plateforme, qui peut être réelle ou simulée, et qu'il communique avec un environnement qui là encore peut être réel ou simulé. Le testeur procure les entrées du système conformément au cas de test, puis compare les sorties obtenues avec celles qui sont attendues dans le cas de test. Il peut ensuite rendre un verdict sur l'inadéquation du système à ses spécifications.

1.3 Le forçage en ligne

1.3.1 Vérification en ligne

Le forçage en ligne est une technique qui permet de contraindre un système à se conformer à une propriété. Cette technique se base sur les avancées réalisées dans le domaine de la vérification en ligne. La vérification en ligne (Leucker & Schallhart, 2009 ; Bauer, Leucker & Schallhart, 2011) est la branche des méthodes formelles qui s'intéresse à la synthèse de moniteurs depuis des spécifications formelles. Ces moniteurs sont des machines logicielles ou matérielles (par exemple des machines à états) qui analysent la trace d'exécution d'un système, en ligne ou après l'exécution, pour vérifier que cette trace est conforme aux propriétés présentes dans les spécifications. Ils sont générés à partir d'une description mathématique (formelle) des spécifications du système, en s'appuyant généralement sur de la logique temporelle.

La vérification en ligne est une technique de vérification formelle, au même titre que la vérification de modèles. Comme on peut le voir en figure 1.3, la vérification en ligne

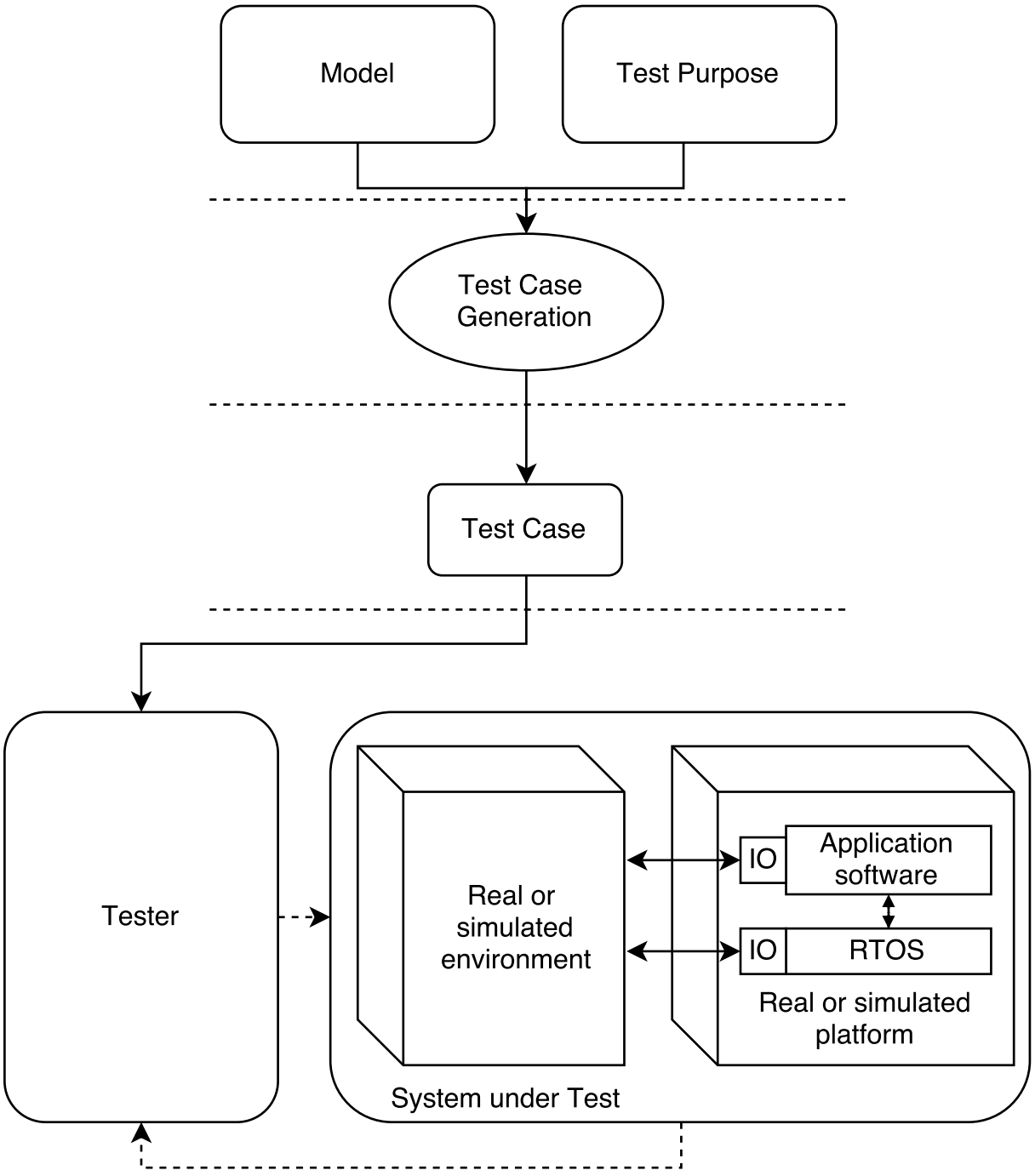


Figure 1.2 – Procédé de test logiciel

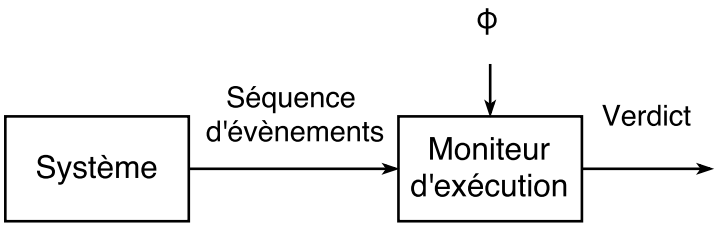


Figure 1.3 – Vérification en ligne

n'influence pas l'exécution : elle ne fait que rendre un verdict à partir de la trace observée sur le système. Le système est instrumenté afin de pouvoir observer son comportement. Les évènements sont fournis au moniteur d'exécution qui, à l'aide de la propriété ϕ à vérifier, renvoi un verdict quant au respect ou non de cette propriété. Formellement, le verdict consiste à déterminer si la trace d'exécution du système est inclue dans $[[\phi]]$, l'ensemble des traces valides données par la propriété ϕ.

La vérification en ligne peut être considérée dans une approche boîte noire, puisque le moniteur peut être synthétisé à partir de descriptions haut niveau des contraintes du système, venant des spécifications. Ceci est en contraste avec la vérification de modèles qui demande elle un modèle complet du système. Cependant, contrairement à la vérification de modèles, la vérification en ligne ne permet de valider que l'exécution observée du système, au lieu de valider une propriété pour toutes les exécutions d'un système.

1.3.2 Forçage en ligne

Le forçage en ligne (en anglais runtime enforcement) se base sur les concepts de la vérification en ligne. Au lieu de synthétiser un moniteur, cette technique vise à synthétiser un contrôleur garantissant au cours de l'exécution que le système est bien conforme aux attentes du concepteur. Pour ce faire, il est nécessaire au contrôleur de pouvoir influer sur la trace d'exécution. Il se doit donc de pouvoir capturer des évènements ciblés et de choisir pour chaque évènement l'action à effectuer, telles que annuler l'évènement, le faire parvenir à sa destination, le retarder, etc.

Sur la figure 1.4, on peut constater que le contrôleur transforme une séquence d'exécution potentiellement incorrecte en une séquence conforme à la propriété d'entrée.

Pour le forçage en ligne, tout comme pour la vérification en ligne, le comportement du système observé par le moniteur n'est pas affecté. Cependant, l'objectif du forçage en ligne est de permettre la modification de la trace d'exécution, là où la vérification en ligne ne permet que de rendre un verdict. Cette trace d'exécution modifiée, conforme à la propriété souhaitée, peut être utilisée par un autre système avec lequel communique le système observé.

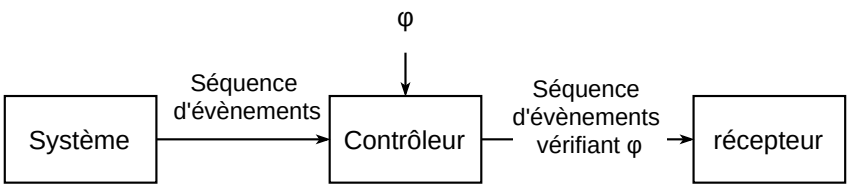


Figure 1.4 – Forçage en ligne

Le forçage en ligne, tout comme la vérification en ligne, ont été étendus pour inclure des propriétés temporelles sur des systèmes temporisés.

1.4 Positionnement

Systèmes temps réel Au long de cette thèse, nous travaillons sur des systèmes embarqués temps réel dur, notamment ceux que l'on peut rencontrer dans le secteur automobile.

Test Nous présentons nos travaux dans le contexte d'une approche de test qui se positionne à la frontière du test boîte noire et boîte blanche : Si nous définissons le système pour inclure le système d'exploitation sur lequel le logiciel tourne, nous proposons une technique qui, en travaillant sur l'interaction entre le système d'exploitation et le logiciel, se situe en boîte blanche. Si l'on considère le système comme étant le programme tournant sur le système d'exploitation, nous sommes donc en boîte noire.

Notre démarche se situe de plus au niveau du test de validation système.

Forçage en ligne Dans ce travail de thèse, nous proposons l'utilisation du forçage en ligne de propriétés temporelles afin de permettre l'implémentation d'une stratégie de test. Contrairement aux techniques traditionnelles de forçage en ligne, nous modifions directement le comportement du système observé. La trace d'exécution contrôlée est celle des appels systèmes effectués par l'application, mais la modification de celle-ci ne se fait pas de manière indépendante du comportement de l'application.

Chapitre 2

État de l'art

Nous proposons dans ce chapitre un état de l'art visant à détailler les avancées existantes dans les différents domaines employés au long de ce travail de thèse : le test de systèmes temps réel, l'injection de fautes, le forçage en ligne et le contrôle des systèmes temps réel.

Sommaire

2.1 Test des systèmes temps réel

Le domaine du test des systèmes temps réel est la cible de bon nombre de publications. Les travaux qui nous intéressent se situent dans la catégorie du test basé sur les modèles (en anglais model-based testing).

2.1.1 Test basé sur les modèles

Les spécifications d'un système sont souvent incomplètes ou ambigües, ce qui peut rendre le test et la définition des cas de tests difficiles. Le fait de centrer le test sur l'utilisation de modèles formels du système, sur une formalisation de sa spécification, permet de rendre ce procédé précis, complet, et sans ambigüité.

Le test logiciel centré sur l'utilisation de modèles fait généralement référence au test boîte noire : l'environnement est manipulé afin de valider la conformité de la trace de sortie

avec les contraintes fonctionnelles. Lorsqu'une entrée définie est fournie au système, il doit réagir de la manière décrite dans les spécifications.

Cela a été formalisé à l'aide de la relation de conformance entrée-sortie (Input-Output COnformance, IOCO. Tretmans, 1999). Cette relation a été étendue au domaine temporel par Krichen & Tripakis (2004) qui définissent TIOCO, Timed Input Output COnformance relation, la relation temporisée de conformance entrée-sortie.

Les auteurs définissent IOCO de la manière suivante :

Définition 2.1 (Relation de conformance entrée-sortie (IOCO))**.** A est conforme à B si pour chacun des comportements observables spécifiés dans B, les sorties possibles de A après ce comportement sont un sous-ensemble des sorties de B.

TIOCO augmente l'ensemble des comportements observables en y ajoutant les délais temporels.

Ces techniques agissent sur la trace d'entrée du système et comparent la trace de sortie avec une trace attendue.

2.1.2 Génération des cas de tests

De manière générale, les cas de test ainsi que leur formalisation sont souvent réalisés à la main. Dans le cadre d'une technique de test basée sur les modèles, il est possible de générer les cas de test à partir des spécifications formelles. On peut alors proposer un ensemble de tests garantissant des critères de couverture sur le programme testé (Rayadurgam & Heimdahl, 2001). Ces techniques ont été implémentées dans des outils tels que TGV (Jard & Jéron, 2005), STG (Clarke, Jéron, Rusu & Zinovieva, 2002) et TorX (Bohnenkamp & Belinfante, 2005).

Des travaux ont contribué à étendre ces techniques au domaine des systèmes temps-réel, générant des cas de tests en logique temporelle, par exemple à l'aide du logiciel UPPAAL (Hessel, Larsen, Mikucionis, Nielsen, Pettersson & Skou, 2008), ou bien TTG (Krichen & Tripakis, 2009) qui se base sur des automates temporisés non déterministes. Briones & Brinksma (2004) proposent une implémentation de la génération de tests basée sur TIOCO dans TorX.

Dans leurs travaux, Artho, Barringer, Goldberg, Havelund, Khurshid, Lowry, Pasareanu, Roşu, Sen, Visser et al. (2005) combinent la génération de tests avec la génération de moniteurs d'exécution afin de tester systématiquement le domaine d'entrée du programme pour une propriété donnée, et de confirmer que les traces y sont conformes.

2.2 Injection de fautes dans les systèmes temps réel

L'injection de fautes logicielle (Software Implemented Fault Injection, SWIFI), par contraste avec l'injection de fautes matérielle (PHYsical Fault Injection, Φ FI), propose une méthode pour introduire, à l'aide de moyens logiciels, des fautes dans un système.

L'injection de fautes matérielle permet, à l'aide de composants matériels, de venir interagir avec le système concerné. Cela peut se faire avec contact, modifiant par exemple la tension d'un élément du circuit, ou bien à distance en utilisant des radiations électromagnétiques (Entrena, López-Ongil, García-Valderas, Portela-García & Nicolaidis, 2011). L'injection de fautes logicielle simule des fautes en ne modifiant pas les caractéristiques matérielles du système. Certaines catégories de fautes sont plus difficiles à implémenter

à l'aide de techniques logicielles, en raison de l'overhead qu'elle peuvent engendrer ou de la difficulté d'implémentation. On peut par exemple distinguer les fautes permanentes (par exemple fixer valeur d'un bit de mémoire) qui sont plus adaptées à l'implémentation matérielle des fautes temporaires qui peuvent se satisfaire d'une implémentation logicielle.

L'ensemble de ces techniques permet ainsi de venir tester la robustesse des systèmes ainsi que leurs mécanismes de tolérance aux fautes (mécanismes de sureté) (Arlat, Crouzet, Karlsson, Folkesson, Fuchs & Leber, 2003). On peut distinguer deux catégories relatives à l'injection de fautes logicielle : l'injection à la compilation et l'injection en ligne (Hsueh, Tsai & Iyer, 1997).

L'injection à la compilation permet l'injection d'erreurs dans le code afin de simuler des fautes, résultant ainsi dans l'exécution d'un logiciel contenant ces fautes, ce qui permet de tester la réponse du reste du système à la présence de ces fautes. Les fautes qui peuvent être simulées de cette manière incluent des fautes de conception, ou encore des fautes liées à l'implémentation (mauvaise utilisation des outils de développement, ou fautes venant de ces outils eux-mêmes). Certains outils proposent une implémentation de l'injection de fautes à la compilation, en se basant sur des données obtenues sur le terrain concernant les fautes les plus fréquentes(Christmansson & Chillarege, 1996), mais cela reste un problème complexe car il est difficile de générer des fautes réalistes vis-à-vis des fautes qui peuvent être rencontrées dans les systèmes réels (Madeira, Costa & Vieira, 2000).

L'injection de fautes en ligne utilise les évènements du système pour injecter des fautes dans la mémoire ou les registres du processeur, au cours de l'exécution. Ceci peut par exemple consister en l'ajout ou la modification des instructions de l'application. Pour permettre cela, certains outils (Kanawati, Kanawati & Abraham, 1992 ; Carreira, Madeira, Silva et al., 1998) utilisent des interruptions logicielles déclenchées sur des valeurs spécifiques du compteur ordinal, ou simplement des valeurs de l'horloge, afin d'émuler des catégories précises de corruption des données et de fautes matérielles.

Lorsque l'on utilise l'injection de fautes logicielles dans le contexte d'une application tournant sur un système d'exploitation temps réel, l'injecteur peut être positionné dans l'application elle même, ou bien dans une couche logicielle située entre l'application et le système d'exploitation. De manière similaire, Dawson, Jahanian, Mitton & Tung (1996) proposent une méthode et un outil pour injecter des fautes dans un système distribué au niveau du protocole de communication entre ses composants.

2.3 Forçage en ligne des systèmes temps réel

Comme précisé en section 1.3, le forçage en ligne se base sur les techniques de vérification formelle, et plus précisément des techniques de vérification en ligne.

Parmi d'autres, Bauer et al. (2011) proposent une vue d'ensemble de la vérification en ligne. Ils décrivent de plus une méthode de génération de moniteurs d'exécution pour LTL (Linear Temporal Logic, la logique temporelle linéaire) qui n'est pas temporisée, ainsi que pour TLTL (Timed Linear Temporal Logic, logique temporelle linéaire temporisée). Le forçage en ligne et la synthèse de moniteurs ont été implémentés avec LTL ou des variantes de LTL par de nombreux travaux (Havelund & Roşu, 2001 ; Havelund & Roşu, 2002 ; Stolz & Bodden, 2006). Cotard, Faucou, Béchennec, Queudet & Trinquet (2012) proposent de leur coté une implémentation de la vérification en ligne pour un système d'exploitation temps réel.

En ce qui concerne les propriétés temporisées, des variantes de LTL temporisées sont

utilisées telles que TLTL citée plus haut (Bauer et al., 2011), MTL pour Metric Temporal Logic (Maler, Nickovic & Pnueli, 2006 ; Thati & Roşu, 2005), ou encore des outils tels que AMT (Nickovic & Maler, 2007) ou LARVA (Colombo, Pace & Schneider, 2009).

Ces techniques ne permettent pas de changer l'état d'un système. Elles ne permettent que de confirmer (ou infirmer) formellement son adéquation avec une propriété donnée, à tout moment de son exécution. Dans ces travaux de thèse, il nous est nécessaire de manipuler la trace d'exécution du système afin de l'amener dans l'état choisi. C'est pourquoi il nous est nécessaire d'utiliser le forçage en ligne.

Le forçage en ligne des propriétés de sûreté (en anglais safety), c'est à dire : rien de mauvais ne peut se dérouler lors de l'exécution du programme, a été introduit par Schneider (2000). Des travaux tels que ceux de Falcone, Mounier, Fernandez & Richier (2011) étendent les propriétés qui peuvent être forcées avec la classe des propriétés dites *safety-progress*, qui incluent les notions de vivacité et permettent de garantir la terminaison.

Les actions disponibles au contrôleur incluent selon les travaux le blocage de la séquence d'entrée (Schneider, 2000), la suppression et/ou l'insertion d'évènements dans la séquence (Ligatti, Bauer & Walker, 2009), etc.

Le forçage en ligne de propriétés temporisés a été introduit par des travaux considérant le temps discret (Matteucci, 2007), voir basé sur des évènements incontrôlables représentant un coup d'horloge (Basin, Jugé, Klaedtke & Zălinescu, 2013). Ces travaux restent cependant théoriques et ne proposent pas d'implémentation des concepts introduits.

La considération du temps dense a été introduite dans des travaux tels que (Pinisetty, Falcone, Jéron, Marchand, Rollet & Timo, 2013). Les auteurs proposent une implémentation du forçage en ligne de propriétés temporisées à l'aide de l'outil UPPAAL.

Le forçage en ligne des propriétés temporisées ajoute dans les outils du contrôleur le délai des actions observées, certaines implémentations ne considérant que ce levier.

2.4 Positionnement

Dans ces travaux de thèse, nous proposons une technique permettant le test logiciel. Il ne s'agit pas, contrairement aux méthodes TIOCO, d'une méthode qui influe sur la trace d'entrée du système. Notre solution peut être intégrée à une méthode utilisant ces techniques, mais nous proposons d'influer sur l'interface entre l'application et le système d'exploitation. La seule action disponible est le délai qui peut être ajouté au cours de l'exécution du système. Nous proposons une génération de stratégies de test à partir des spécifications formelles du système et d'un cas de test spécifié manuellement.

Les méthodes actuelles de test de tolérance aux fautes requièrent des techniques d'injection de fautes invasives : le code de l'application doit être modifié, ou bien le système d'exploitation doit pouvoir venir modifier la mémoire de l'application. La solution que nous proposons est complètement découplée du code de l'application.

Nous utilisons le forçage en ligne des systèmes temporisés pour amener le système dans un état nous permettant de faciliter son test. Contrairement à la théorie décrite dans certains travaux, notre contrôleur peut être amené à modifier l'exécution du système observé. Ceci est similaire à ce qui est proposé par (Pinisetty et al., 2013).

Pour cette raison, et suivant la manière dont elle est utilisée, cette méthode s'apparente à de l'injection de fautes. Il est en effet possible de faire sortir le système de son exécution normale en manipulant le temps d'exécution des tâches.

La solution se découpe en deux sous parties : une première analyse hors-ligne, lors de la conception du système, et une seconde partie qui consiste au forçage en ligne du système.

Deuxième partie

Forçage en ligne des systèmes temps réel paramétrés

Chapitre 3

Présentation de la démarche

Nous proposons dans ce chapitre une vue d'ensemble de la contribution de ce travail de thèse. Nous présentons la démarche, ainsi que le plan de la partie 2 qui présente en détail les différentes étapes qui la compose.

Sommaire

3.1 Démarche pour le forçage en ligne

3.1.1 Objectif de la démarche

Nous proposons dans ce travail de thèse une technique permettant de forcer l'exécution d'un système de manière à atteindre un état choisi. Cette démarche permet de faciliter le test des systèmes temps réels en permettant de venir tester le système à partir de cet état Cet état peut en effet être difficile à atteindre si l'on ne manipule que les données d'entrée du système, en particulier en raison de contraintes temporelles qui peuvent conditionner l'accès à cet état.

Le cas d'utilisation du test de systèmes reste l'un des domaines applicables, mais pas l'unique domaine applicable. Considérons par exemple un modèle d'un système dans lequel l'estimation des WCET (Worst Case Execution Time, temps d'exécution dans le pire cas) serait peu fiable. On pourrait souhaiter utiliser ces travaux pour atteindre un état spécifique du système qui n'est pas atteignable avec l'estimation "optimiste" des WCET, et venir confirmer ou non les valeurs sélectionnées, ou bien la robustesse du système vis-à-vis des WCET calculés pour les contraintes les plus critiques.

Néanmoins, dans la suite de cette thèse et afin de présenter les différentes étapes, nous considérons un objectif de test des systèmes temps réels, par exemple le test d'un mécanisme de sûreté (tolérance aux fautes).

3.1.2 Test des systèmes temps réel

Comme évoqué en section 1.2, le test des systèmes temps réel vise à confirmer, par l'observation de son exécution sous des conditions spécifiques, que le comportement du système est conforme à ses spécifications. La figure 1.2 (page 17) propose un procédé permettant le test de systèmes temps réels. Un cas de test (Test Case) est généré à partir des spécifications (le modèle et l'objectif de test, Test Purpose), puis il est soumis au système testé (System Under Test, SUT). Un verdict est établi une fois le test réalisé, confirmant ou infirmant l'adéquation de l'implémentation avec les spécifications du système.

Dans ce travail de thèse, nous proposons d'ajouter à ce processus de test la génération d'une stratégie de test, utilisée par un contrôleur inclus dans le SUT, et dont l'objectif est de garantir la possibilité d'atteindre l'état dans lequel le système doit être testé et ce indépendamment des données d'entrée reçues.

La figure 3.1 représente ce procédé de test modifié, les parties en gras introduisant les étapes ajoutées ou modifiées par ce travail de thèse vis-à-vis d'un procédé de test classique.

Un modèle du système (extrait par exemple de ses spécifications) est transformé et utilisé avec l'objectif de test afin de générer une stratégie de test. Cette stratégie de test est finalement intégrée au SUT, dans l'interface de programmation (API, Application Programming Interface) du système d'exploitation. L'exécution du test permet au testeur d'observer l'exécution du système soumis au cas de test. Une fois l'état atteint, un verdict peut être rendu en fonction de l'exécution du système à partir de ce point.

3.1.3 Présentation du processus

La figure 3.2 présente le processus nous permettant d'effectuer la démarche du forçage au sein d'un procédé de développement d'une application pour un système d'exploitation temps réel.

3.1.3.1 Modèles

Issus d'une démarche d'ingénierie dirigée par les modèles, extrapolés à partir du code de l'application, ou encore réalisés à la main par les concepteurs du systèmes, les modèles structurels et comportementaux de l'application (Structural et Behavioral) sont utilisés par notre solution pour produire des modèles comportementaux des tâches du systèmes adaptés à nos besoins. Sur le schéma de la figure 3.2, il est considéré que ces modèles sont obtenus, dans une démarche d'ingénierie dirigée par les modèles, par extraction depuis le modèle de l'application.

Chaque tâche du système est modélisée du point de vue du système d'exploitation, dans un formalisme approprié. Les seuls évènements observables au cours de l'exécution de la tâche par le système d'exploitation sont les appels systèmes que la tâche effectue.

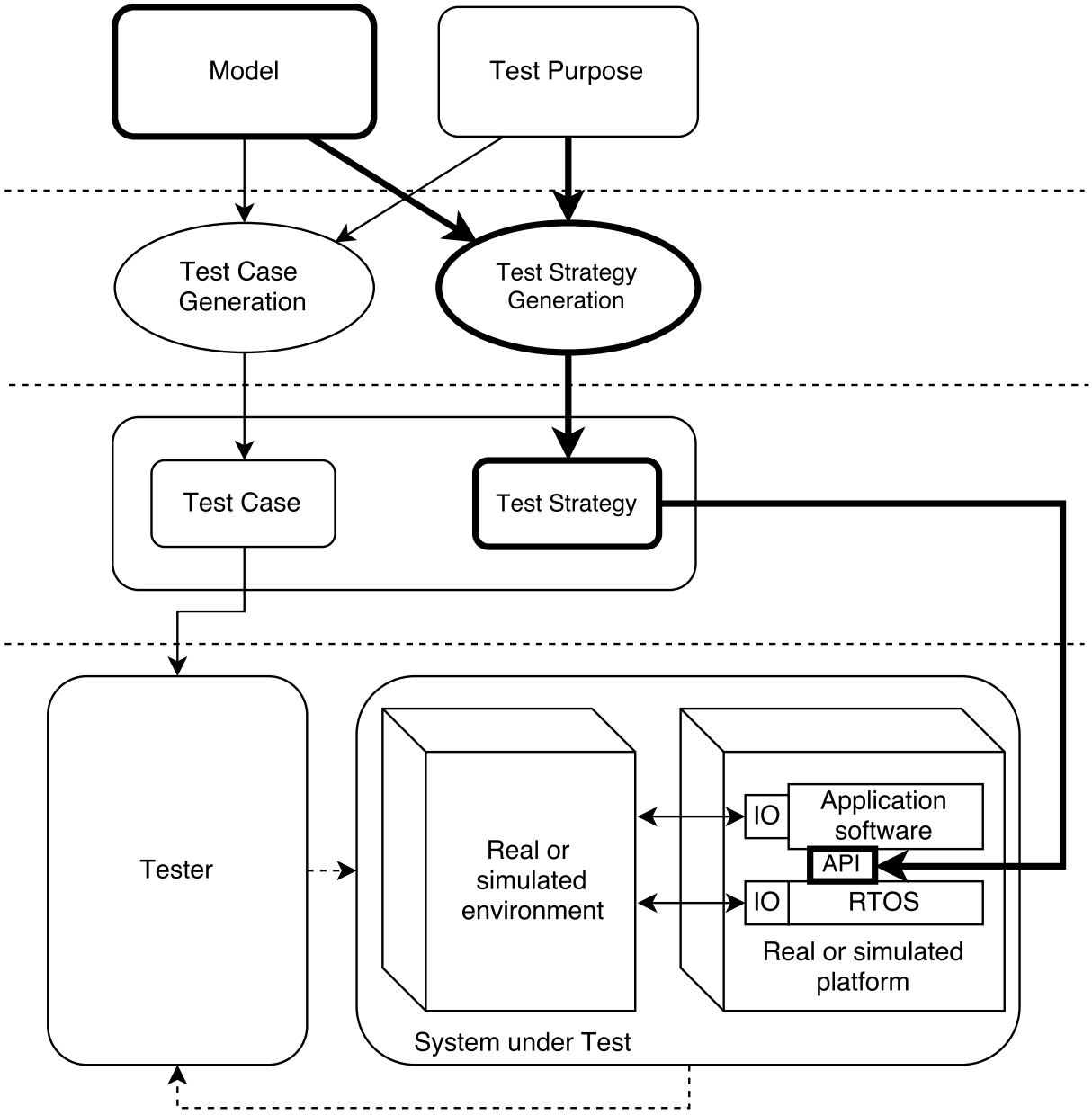


Figure 3.1 – Processus du point de vue du test de systèmes

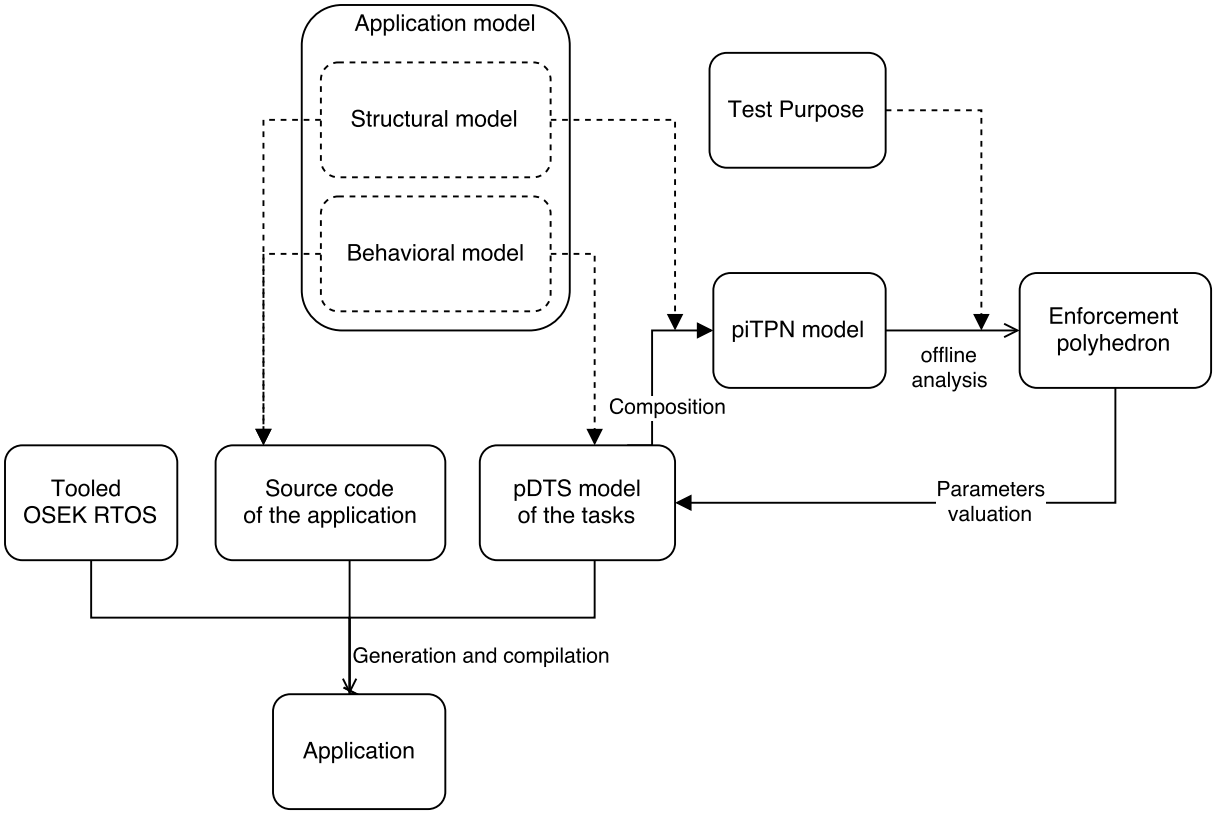


Figure 3.2 – Processus général

3.1.3.2 Analyse hors-ligne

Les modèles des différentes tâches du système sont transformés puis composés, à l'aide d'un modèle structurel du système, afin d'obtenir un modèle paramétré dont l'objectif est de permettre le calcul d'une stratégie de test à appliquer durant l'exécution. Cette stratégie prend la forme d'un ensemble de délais ajoutés par le contrôleur à l'exécution de la tâche contrôlée. Les délais à appliquer au cours de l'exécution sont déterminés à l'aide de l'analyse du modèle paramétré, guidée par l'objectif de test.

3.1.3.3 Forçage en ligne

Afin de réaliser le forçage, un ensemble de modèles décrivant la stratégie de test à appliquer sont embarqués dans le contrôleur. Le contrôleur est situé au niveau de l'implémentation de l'API du système d'exploitation temps réel. Il observe les appels systèmes effectués par les tâches et selon les informations données par les modèles embarqués, choisi ou non d'ajouter des délais au cours de l'exécution.

Cette étape de contrôle a lieu dans le cadre du test de systèmes temps réel. Le testeur fournit au SUT des données afin de simuler l'activité de l'environnement du système. Le contrôle à lieu en parallèle de ce test, garantissant l'accessibilité de l'état choisi. Une fois cet état atteint, le testeur peut continuer à observer le système afin de rendre un verdict sur l'adéquation de son comportement aux spécifications.

3.2 Présentation détaillée de la démarche

La partie suivante détaille successivement les différentes étapes permettant le forçage d'un système temps réel.

Nous présentons et justifions dans un premier temps le choix du formalisme retenu pour l'analyse hors-ligne du système. Pour cela, nous détaillons les capacités d'analyse paramétrique nécessaires à la démarche.

Nous présentons ensuite la manière dont le modèle permettant l'analyse du système est obtenu. Nous décrivons le formalisme utilisé pour modéliser les tâches du système, ainsi que la manière dont leur modèle initial est obtenu, transformé, paramétré et finalement composé avec celui des autres tâches afin d'obtenir le modèle sur lequel l'analyse sera effectué.

Enfin, nous proposons une vue d'ensemble des principes permettant la mise en œuvre du forçage en ligne. Tout d'abord, l'analyse hors-ligne du modèle précédemment obtenu dont les résultats permettent de sélectionner les valeurs de délais à appliquer pour atteindre l'état choisi. Ensuite, la génération des modèles embarqués suivie de la manière dont le contrôle est effectué dans un système d'exploitation temps réel.

Chapitre 4

Model checking paramétré

Nous proposons dans ce chapitre une vue d'ensemble de la vérification de modèles (model checking) et de notre capacité d'analyse paramétrique du formalisme choisi : les réseaux de Petri temporels.
Pour cela, nous revenons tout d'abord sur le choix de la technique de vérification de modèles paramétrés et du formalisme des réseaux de Petri temporels, puis nous présentons ce formalisme ainsi que son analyse.

Sommaire

4.1 Choix de la technique et du formalisme

4.1.1 Synthèse de paramètres

Il existe dans la littérature différentes techniques qui permettent d'obtenir formellement des garanties sur le comportement des systèmes temporisés en se basant sur l'analyse exhaustive de l'espace d'états du modèle du système concerné : le model checking (ou vérification de modèles), la synthèse de contrôleurs, et finalement la synthèse de paramètres.

L'analyse de modèles est une procédure automatique qui permet de vérifier qu'un modèle d'un système satisfait une spécification décrite par une propriété. Grâce aux progrès qui ont été fait dans ce domaine, l'analyse formelle des modèles temporisés est régulièrement utilisée pour des applications industrielles à l'aide d'outils tels que Uppaal (Larsen, Pettersson & Yi, 1997), ou Tina (Berthomieu & Vernadat, 2006). Cependant, pour ces techniques de model checking, les paramètres temporels du système considéré possèdent des valeurs dont le choix est déterminé « à la main » : l'analyse permet ensuite de déterminer si les contraintes de conception sont ou ne sont pas respectées avec ces valeurs.

La synthèse de contrôleur, quant à elle, consiste lorsque cela est possible à construire un programme (le contrôleur) qui pilote les mécanismes d'interaction avec le système contrôlé de façon à garantir un fonctionnement sûr et correct. L'ensemble des actions du système est partitionné en deux sous ensembles : l'ensemble des actions contrôlables et l'ensemble des actions incontrôlables. Le contrôleur peut agir sur (et seulement sur) les actions contrôlables.

La synthèse de contrôleurs propose ainsi une technique automatisée permettant de produire des modèles comportementaux de systèmes logiciels qui se conforment à un ensemble de contraintes. Les contraintes sont définies à l'aide de formules de logique temporelle, et un modèle comportemental les satisfaisant est généré dans un formalisme approprié. Ces techniques ont été étudiées d'un point de vue théorique, à la fois pour les systèmes discrets (Thomas, 2002 ; Kumar & Garg, 2012) et pour les systèmes temporisés (Bouyer, D'Souza, Madhusudan & Petit, 2003). D'un point de vue pratique, il existe des implémentations de ces concepts (Braberman, D'Ippolito, Piterman, Sykes & Uchitel, 2013), y compris dans des outils tels que UPPAAL-TIGA (Behrmann, Cougnard, David, Fleury, Larsen & Lime, 2007) mais cela reste une tâche ardue.

La troisième option est celle de la synthèse de paramètres. Les méthodes d'analyse paramétrique permettent au contraire des techniques de model checking ou de synthèse de contrôleurs de choisir des composants du modèle à paramétrer, l'analyse permettant ensuite de déterminer l'ensemble des valeurs pour ces paramètres qui respectent les contraintes choisies. Ces techniques sont plus simples à implémenter que la synthèse de contrôleurs, car il suffit d'ajuster des paramètres sur le système dont le modèle a été analysé (et sur son implémentation), et elles sont plus puissantes que celles du model checking, car le système peut être conçu en laissant une certaine incertitude représentée par les paramètres, l'analyse participant ensuite au choix des valeurs qui leur seront affectées.

4.1.2 Capacités d'analyse et décidabilité

Dans cette thèse, nous travaillons sur les aspects temporels des systèmes considérés. Dans cette optique, il est nécessaire de positionner les paramètres sur le temps.

Il existe dans la littérature deux principaux formalismes permettant de modéliser des applications avec des paramètres sur le temps : les automates temporisés paramétrés (PTA) (Alur,

Henzinger & Vardi, 1993) et les réseaux de Petri temporels paramétrés (Traonouez, Lime & Roux, 2009). Ces deux classes souffrent de résultats d'indécidabilité, même avec un nombre restreint de paramètres (Miller, 2000), des restrictions syntaxiques telles que des contraintes strictes (Doyen, 2007), ou encore des restrictions sur les domaines de paramètres : rationnels bornés (Miller, 2000) ou entiers (non-bornés) (Alur et al., 1993). La restriction syntaxique la plus connue qui permet la décidabilité est la classe des automates *lower/upper bound* (L/U) (Hune, Romijn, Stoelinga & Vaandrager, 2002)[1] mais même avec cette contrainte, trouver un algorithme de synthèse des paramètres reste un problème ouvert (Jovanović, Lime & Roux, 2015). Pour répondre à cette difficulté, des algorithmes symboliques ont été proposés pour synthétiser exactement les valeurs entières des paramètres quand ils sont bornés, sans les énumérer explicitement. Cette idée à été développée dans (André, Lime & Roux, 2015) pour produire des sous-approximations denses « suffisamment grandes » du résultat pour des paramètres rationnels bornés.

Pour beaucoup de systèmes temps réel, en particulier si ils sont soumis à un ordonnancement préemptif, ces formalismes ne sont pas assez expressifs. Bien que des extensions permettant de modéliser la préemption soient disponibles sous la forme d'automates paramétrés à chronomètres (Sun, Soulat, Lipari, André & Fribourg, 2013), ou de réseaux de Petri paramétrés à chronomètres (Traonouez et al., 2009), les approches de (Jovanović et al., 2015 ; André et al., 2015) ne sont pas disponibles pour ces formalismes.

Il faut donc utiliser la synthèse exacte pour des paramètres rationnels avec des semi-algorithmes. À notre connaissance, les seuls outils implémentant ces semi-algorithmes pour ces formalismes plus expressifs sont Imitator (André, Fribourg, Kühne & Soulat, 2012) pour les automates paramétrés à chronomètres et Roméo (Lime, Roux, Seidner & Traonouez, 2009) pour les réseaux de Petri paramétrés à chronomètres.

Les réseaux de Petri, contrairement aux automates temporisés, permettent de modéliser le parallélisme de manière concise : de multiples jetons peuvent être présents dans plusieurs états du réseau, et les transitions peuvent les faire évoluer en parallèle. Nous utilisons donc dans ce travail de thèse l'outil Roméo, puisque nous travaillons avec des réseaux de Petri temporels à chronomètres.

4.1.3 Principe

L'analyse paramétrique des réseaux de Petri vise donc à synthétiser les valeurs de paramètres telles qu'une propriété soit vérifiée sur un réseau. La vérification d'une propriété exprimée en logique TCTL (Time Computational Tree Logic) paramétrée nous donne un domaine pour les valeurs de paramètres. Ce domaine de valeurs admissibles est une union de polyèdres convexes.

Dans ce travail de thèse, nous utilisons l'analyse paramétrique des réseaux de Petri afin de déterminer les valeurs des paramètres nous permettant d'atteindre un état donné du système. Les transitions paramétrées correspondent aux temps d'exécution vus par les appels systèmes. Nous obtenons donc des valeurs indiquant le délai avant l'exécution des différents appels systèmes nous permettant d'atteindre l'état choisi.

Le formalisme exact que nous utilisons, présenté dans la section suivante, se rapproche des réseaux de Petri L/U (où le positionnement d'un paramètre est restreint à l'une

[1] Les automates L/U restreignent chacun des paramètres à l'ensemble des bornes supérieures ou l'ensemble des bornes inférieures des intervalles de l'automate. Un paramètre peut être présent sur plusieurs intervalles, mais pas sur les deux ensembles.

des deux bornes des intervalles). Cette classe de modèles nous donne la décidabilité de l'existence d'une solution, mais la synthèse des paramètres nécessite toujours un semi-algorithme (Jovanović, Lime & Roux, 2013).

4.2 Réseaux de Petri

4.2.1 Définition et sémantique des réseaux de Petri

Les réseaux de Petri sont un outil de modélisation introduit par Carl Adam Petri dans sa thèse de doctorat (Petri, 1962). Ils proposent une représentation graphique avec des *places* et des *transitions* pouvant contenir des *jetons*. Le réseau est formé par des arcs reliant les places aux transitions et vice-versa. Un exemple de réseau de Petri est visible en figure 4.1.

Les transitions peuvent être tirées, ce qui déplace les jetons des places en amont vers les places en aval. De plus, il est possible de rajouter un *poids* sur les arcs qui indique le nombre de jetons à consommer ou à produire lors du tir de la transition associée.

Les réseaux de Petri sont un formalisme mathématique qui dispose d'un ensemble d'outils permettant de les analyser afin d'étudier les propriétés des systèmes qu'ils modélisent.

Nous présentons ci-dessous la définition formelle des réseaux de Petri ainsi que la sémantique associée.

Définition 4.1 (Réseau de Petri)**.** Un réseau de Petri marqué est un n-uplet $\langle P, T, Pre, Post, m_0 \rangle$ où :

- P est un ensemble fini de places ;
- T est un ensemble fini de transitions, tel que $P \cap T = \emptyset$;
- Pre et $Post$ sont des applications de $(P \times T) \to \mathbb{N}$, qui décrivent les liens (appelés arcs) entre les places et transitions en indiquant le poids associé ;
- m_0 est le marquage initial du réseau de Petri (la notion de marquage est définie ci-dessous).

Le comportement d'un réseau de Petri est décrit à l'aide de sa sémantique opérationelle. Afin de pouvoir décrire la sémantique, nous définissons un certain nombre de notations :

- marquage : un marquage $M : P \to \mathbb{N}$ associe un entier avec chaque place du réseau ;
- m_0 est le marquage initial ;
- transitions sensibilisées : une transition t est dite sensibilisée si et seulement si $\forall p \in P, M(p) \geq Pre(p, t)$. Cela correspond à une fonction $enabled : M \times T \to \mathbb{B}$ l'ensemble des booléens ;

La sémantique d'un réseau de Petri peut être décrite par celle d'un système de transitions :

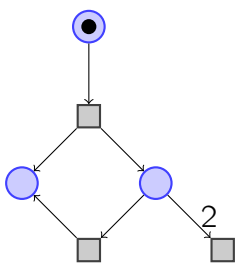


Figure 4.1 – Réseau de Petri

Définition 4.2 (Sémantique d'un réseau de Petri)**.** La sémantique d'un réseau de Petri $\mathcal{N} = (P, T, Pre, Post, m_0)$ est définie comme un système de transitions $\mathcal{T}_{\mathcal{N}} = (Q, q_0, \rightarrow)$, où :

- $Q = M$
- $q_0 = m_0$
- $\rightarrow \in Q \times T \times Q$ est la relation de transition telle que :

$$\forall t \in T, M \rightarrow M' \iff M(p) \geq Pre(p,t) \wedge M'(p) = (M(p) - Pre(p,t) + Post(p,t)), \forall p \in P$$

avec $M(p)$ le nombre de jetons dans p.

4.2.2 Définition et sémantique des réseaux de Petri temporels

Les réseaux de Petri temporels ajoutent au formalisme des réseaux de Petri la notion de temps. Il existe plusieurs manières de représenter le temps dans un réseau de Petri, selon le type de temporisation ou selon l'emplacement de la temporisation :

Concernant le type de temporisation :

- Réseaux de Petri temporisés : il s'agit d'une durée associée à chaque transition, indiquant le temps au bout duquel elle sera tirée.
- Réseaux de Petri temporels : ici un intervalle est associé à chaque transition, indiquant, en sémantique forte (respectivement faible), une durée minimale avant le tir de la transition et une durée maximum, forçant le tir de la transition (ou respectivement empêchant sa sensibilisation).

Concernant l'emplacement de la temporisation :

- Réseaux de Petri P-temporels : l'intervalle de temps est associé aux places
- Réseaux de Petri T-temporels : l'intervalle de temps est associé aux transitions.

Nous utilisons ici les réseaux de Petri T-temporels, c'est à dire avec les intervalles sur les transitions. Il s'agit du type de formalisme le plus répandu dans la littérature en ce qui concerne les réseaux de Petri avec du temps. Par ailleurs, il permet plus d'expressivité que les réseaux de Petri temporisés et sied mieux à notre modélisation. La sémantique que nous présentons est une sémantique forte : on ne peut pas laisser le temps s'écouler après la borne supérieure d'un intervalle d'une transition sensibilisée.

La figure 4.2 montre un exemple de réseau de Petri temporel.

Définition 4.3 (Réseau de Petri temporel)**.** Un réseau de Petri temporel est un n-uplet $\langle P, T, Pre, Post, m_0, \alpha, \beta \rangle$ où :

- $\langle P, T, Pre, Post, m_0 \rangle$ est un réseau de Petri ;
- $\alpha : T \to \mathbb{N}$, et $\beta : T \to \mathbb{N} \cup \{+\infty\}$ sont des applications décrivant la borne inférieure (resp. supérieure) de l'intervalle de tir statique de chaque transition, et $\forall t \in T, \alpha(t) \leq \beta(t)$.

Comme pour les réseaux de Petri, le comportement d'un reseau de Petri temporel est décrit par sa sémantique opérationelle. Nous définissons :

- transition tirable : le temps écoulé depuis qu'une transition t a été sensibilisée est noté $v(t)$. Une transition est tirable si $\alpha(t) \leq v(t) \leq \beta(t)$.
- tir d'une transition : on peut tirer une transition t si t est sensibilisée par le marquage courant et est tirable.
  On passe alors du marquage M au marquage M', tel que : $M'(p) = (M(p) - Pre(p,t) + Post(p,t))$, $\forall p \in P$, avec $M(p)$ le nombre de jetons dans p.
- transitions nouvellement sensibilisées : $\uparrow enabled(t_k, M, t_i) \in \mathbb{B}$ est $vrai$ si le tir de t_i avec le marquage M sensibilise t_k. La transition peut aussi être sensibilisée par son propre tir. Formellement :

$$\begin{aligned} \uparrow enabled(t_k, M, t_i) = \\ & (M - Pre(t_i) + Post(t_i) \geq Pre(t_k)) \wedge \\ & (M - Pre(t_i) < Pre(t_k) \vee (t_k = t_i)) \end{aligned}$$

La sémantique d'un réseau de Petri temporel est donnée par un système de transitions temporisé. Les systèmes de transition temporisés possèdent des transitions discrètes et continues (dans $\mathbb{R}^+$), mais sont par ailleurs similaires aux systèmes de transition classiques.

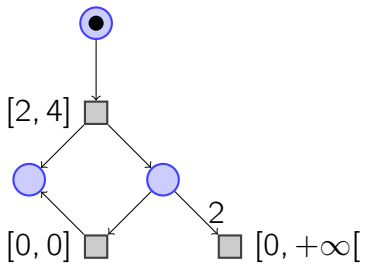


Figure 4.2 – Réseau de Petri temporel

Définition 4.4 (Sémantique d'un réseau de Petri temporel)**.** La sémantique d'un réseau de Petri temporel $\mathcal{N} = (P, T, Pre, Post, m_0, \alpha, \beta)$ est donnée par le système de transitions temporisé $\mathcal{T}_\mathcal{N} = (Q, q_0, \rightarrow)$, où :

- $Q = M \times v$, $v : T \rightarrow \mathbb{R}^+$ est l'ensemble des valuations d'horloges, tel que $v(t)$ représente le temps écoulé depuis que t a été sensibilisée ;
- $q_0 = (M_0, v_0)$ est l'état initial, avec M_0 le marquage initial et v_0 la valuation initiale, $\forall t \in T, v_0(t) = 0$;
- $\rightarrow \in Q \times (T \cup \mathbb{R}^+) \times Q$ est la relation de transition, composée de :
  - une relation de transition discrète, définie telle que :
    $\forall t \in T, (M, v) \rightarrow (M', v')$
    $$\iff \begin{cases} (M \geq Pre(t)) \wedge (M' = M - Pre(t) + Post(t)) \\ \alpha(t) \leq v(t) \leq \beta(t) \\ \forall t' \in enabled(M') : \\ \quad v'(t') = \begin{cases} 0 \textbf{ si } \uparrow enabled(t', M, t), \\ v(t') \textbf{ sinon} \end{cases} \end{cases}$$
  - une relation de transition continue, définie telle que :
    $\forall \delta \in \mathbb{R}^+, (M, v) \xrightarrow{\delta} (M', v')$
    $$\iff \begin{cases} M' = M \\ v'(t) = v(t) + \delta \\ \forall t \in T, M \geq Pre(t) \implies v'(t) \leq \beta(t) \end{cases}$$

4.2.3 Définition et sémantique des réseaux de Petri temporels à arcs inhibiteurs temporisés

Dans ce travail de thèse, nous utilisons les réseaux de Petri à chronomètres (Berthomieu, Lime, Roux & Vernadat, 2007). Plus précisément, nous utilisons les réseaux de Petri à arcs inhibiteurs temporisés, iTPN, ainsi que leur extension paramétrique, piTPN.

Les arcs inhibiteurs relient des places à des transitions. Ils permettent, lorsque la place source contient un ou plusieurs jetons (selon le poids associé), d'empêcher le tir tout en stoppant l'écoulement du temps pour la transition concernée (si elle est sensibilisée).

Les arcs inhibiteurs temporisés sont différents des arcs inhibiteurs classiques (Janicki & Koutny, 1995 ; Janicki & Koutny, 1991), qui eux ne décrivent que l'inhibition du tir de la transition ciblée. D'une certaine manière, on peut considérer les arcs inhibiteurs temporisés comme une extension des arcs inhibiteurs qui prend en compte les aspects temporels en retenant la valeur de l'horloge associée à la transition.

La figure 4.3 montre un exemple de réseau de Petri temporel à arcs inhibiteurs. Les arcs inhibiteurs y sont représentés par des losanges à leur extrémité.

Dans la définition ci-dessous, et puisque nous les utiliserons par la suite, nous ajoutons par ailleurs des étiquettes sur les transitions du réseau de Petri. La fonction d'étiquettage associe à chaque transition une étiquette permettant de l'identifier.

Définition 4.5 (Réseau de Petri temporel à arcs inhibiteurs)**.** Un réseau de Petri temporel à arc inhibiteur est un n-uplet $\langle P, T, \Sigma, \ell, Pre, Post, Inh, m_0, \alpha, \beta \rangle$ où :

- $\langle P, T, Pre, Post, m_0, \alpha, \beta \rangle$ est un réseau de Petri temporel ;
- Σ est un alphabet fini d'actions ;
- $\ell : T \to \Sigma$ est la fonction d'étiquettage ;
- $Inh : (P \times T) \to (\mathbb{N} \cup +\infty)$ est une fonction appelée «fonction d'incidence» et qui décrit les arcs inhibiteurs entre les places et les transitions.
  L'entier associé est appelé le poids de l'arc, et indique la valeur minimum de jetons dans la place source avant que la transition ciblée soit inhibée.
  Quand la valeur du poids de l'arc est $+\infty$, cela signifie qu'il n'y a pas d'arc inhibiteur entre une place et une transition spécifique.

Comme précédemment, le comportement d'un réseau de Petri temporel à arcs inhibiteurs est décrit par sa sémantique opérationelle. Nous définissons :

- Transition suspendue : t est dite *suspendue* si et seulement si $\forall p \in P, M(p) \geq Inh(p, t)$. Autrement, la transition est dite *active*. Ainsi, s'il n'y a pas d'arc inhibiteur entre une place p et une transition t alors il faut que $Inh(p, t) = +\infty$.
- Tir d'une transition : Le tir d'une transition t est similaire au comportement des réseaux de Petri temporels.
  Néanmoins, il faut prendre en compte l'inhibition des transitions. En conséquence, nous pouvons tirer t si t est sensibilisée par le marquage courant, est *active* et est tirable.
  On passe alors du marquage M au marquage M', tel que : $M'(p) = (M(p) - Pre(p, t) + Post(p, t))$, $\forall p \in P$.

Là encore, la sémantique est décrite à l'aide d'un système de transitions temporisé.

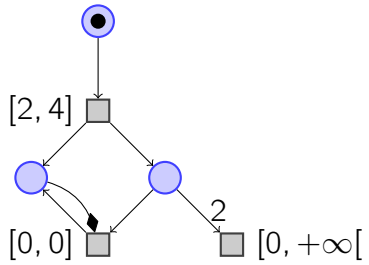


Figure 4.3 – Réseau de Petri temporel à arcs inhibiteurs

Définition 4.6 (Sémantique d'un réseau de Petri temporel à arcs inhibiteurs)**.** La sémantique d'un réseau de Petri temporel à arcs inhibiteurs $\mathcal{N} = (P, T, \Sigma, \ell, Pre, Post, Inh, m_0, \alpha, \beta)$ est définie par un système de transitions temporisé $\mathcal{T}_{\mathcal{N}} = (Q, q_0, \rightarrow)$, où :

- $Q = M \times v$, $v : T \rightarrow \mathbb{R}^+$ est l'ensemble des valuations d'horloge tel que $v(t)$ représente le temps écoulé durant lequel t a été active depuis qu'elle a été sensibilisée ;
- $q_0 = (M_0, v_0)$ l'état inital, avec M_0 le marquage initial, et v_0 la valuation initiale, $\forall t \in T, v_0(t) = 0$;
- $\rightarrow \in Q \times (T \cup \mathbb{R}^+) \times Q$ la relation de transition composée de :
  - une relation de transition discrète, définie telle que :
    $\forall t \in T, (M, v) \xrightarrow{\ell(t)} (M', v')$
    $$\iff \begin{cases} (M \geq Pre(t)) \wedge \neg(M \geq Inh(t)) \wedge \\ (M' = M - Pre(t) + Post(t)) \\ \alpha(t) \leq v(t) \leq \beta(t) \\ \forall t' \in enabled(M') : \\ \quad v'(t') = \begin{cases} 0 \textbf{ si } \uparrow enabled(t', M, t), \\ v(t') \textbf{ sinon} \end{cases} \end{cases}$$
  - une relation de transition continue, définie telle que :
    $\forall \delta \in \mathbb{R}^+, (M, v) \xrightarrow{\delta} (M', v')$
    $$\iff \begin{cases} M' = M \\ v'(t) = \begin{cases} v(t) \textbf{ si } M \geq Inh(t) \\ v(t) + \delta \textbf{ sinon} \end{cases} \\ \forall t \in T, M \geq Pre(t) \implies v'(t) \leq \beta(t) \end{cases}$$

4.2.4 Définition et sémantique des réseaux de Petri temporels paramétrés à arcs inhibiteurs

L'extension paramétrique des réseaux de Petri temporels à arcs inhibiteurs autorise l'ajout de paramètres sur les bornes des intervalles des transitions du réseau.

Les bornes paramétrées des intervalles correspondent à des expressions linéaires sur l'ensemble des paramètres. Lorsqu'une valeur est choisie pour ces paramètres, le remplacement des expressions par leurs valeurs donne un réseau de Petri temporel à arcs inhibiteurs qui n'est pas paramétré, et dont le comportement est équivalent au comportement du réseau paramétré pour ces valeurs de paramètres.

L'analyse formelle du réseau (par exemple d'accessibilité d'une place) permet de donner aux valeurs de paramètres un domaine de validité afin de respecter la propriété concernée.

Afin de définir les réseaux de Petri temporels paramétrés à arcs inhibiteurs (piTPN), on note :

$Par = \{\lambda_1, \ldots, \lambda_n\}$ l'ensemble des paramètres.

Une *valuation* des paramètres est une fonction $\nu : Par \rightarrow \mathbb{N}$.

Une *expression linéaire* γ sur l'ensemble des paramètres est une expression $\gamma = (\Sigma_{i=1..n}\ a_i * \lambda_i) + b$ où $a_i, b \in \mathbb{Z}$. On note $\mathcal{L}(Par)$ l'ensemble des expressions linéaires sur Par.

$\alpha : T \rightarrow \mathcal{L}(Par)$ associe une expression linéaire à la borne inférieure de l'intervalle de

tir de chaque transition.

$\beta : T \to \mathcal{L}(Par)$ associe une expression linéaire à la borne supérieure de l'intervalle de tir de chaque transition.

Un ensemble de contraintes initiales est donné sur les paramètres. Ces contraintes définissent un domaine D_0 pour les paramètres. Ce domaine est un polyèdre convexe. Ces contraintes doivent au minimum spécifier que :

$$\forall \nu \in D_0, t \in T \ , \ \alpha(t) \leq \beta(t)$$

Des contraintes linéaires supplémentaires peuvent être spécifiées.

La figure 4.4 montre un exemple de réseau de Petri temporel paramétré à arcs inhibiteurs. On y observe les paramètres sur les différents intervalles.

Définition 4.7 (Réseau de Petri temporel paramétré à arcs inhibiteurs)**.** Un réseau de Petri temporel paramétré à arcs inhibiteurs (piTPN) est un n-uplet $\langle P, T, \Sigma, \ell, Par, Pre, Post, Inh, m_0, \alpha, \beta, D_0 \rangle$, où :

- $\langle P, T, \Sigma, \ell, Pre, Post, Inh, m_0 \rangle$ est un réseau de Petri à arcs inhibiteurs sans les intervalles sur les transitions ;
- $Par = \{\lambda_1, \ldots, \lambda_n\}$ est un ensemble de paramètres ;
- $\alpha : T \to \mathcal{L}(Par)$ est la fonction qui associe une borne inférieure paramétrée à chaque transition ;
- $\beta : T \to \mathcal{L}(Par)$ est la fonction qui associe une borne supérieure paramétrée à chaque transition ;
- D_0 est un polyèdre convexe qui correspond au domaine des paramètres.

Définition 4.8 (Sémantique d'un réseau de Petri temporel paramétré à arcs inhibiteurs)**.** Étant donné un réseau de Petri temporel paramétré à arc inhibiteurs N et une valuation des paramètres ν, on note $\nu(N)$ le réseau de Petri temporel à arc inhibiteur (non paramétré) où toutes les occurences d'un paramètre λ_i on été remplacées par sa valuation $\nu(\lambda_i)$.
$\forall \nu \in D_0$, $\nu(N)$ est un réseau de Petri temporel à arcs inhibiteurs, et la sémantique du réseau de Petri temporel paramétré à arcs inhibiteurs N est définie par la sémantique de $\nu(N)$. La sémantique du réseau de Petri temporel paramétré à arc inhibiteurs est définie $\forall \nu \in D_0$ par la sémantique de $\nu(N)$.

4.2.5 Restriction des piTPNs

Pour la suite de ce travail de thèse, nous restreignons le modèle des réseaux de Petri temporels paramétrés à arcs inhibiteurs temporisés en contraignant les paramètres à la borne

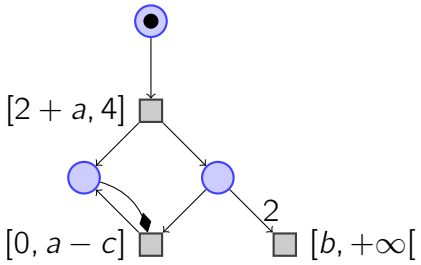


Figure 4.4 – Réseau de Petri temporel paramétré à arcs inhibiteurs

inférieure des transitions considérées. En effet, les transitions sont utilisées pour modéliser des appels systèmes, et les paramètres pour modéliser les délais que nous imposerons lors de l'exécution. Il n'est donc pas nécessaire d'utiliser les paramètres sur les bornes supérieures des intervalles.

La définition des piTPN est donc amendée, en remplaçant :

$$\beta \ : T \to (L)(Par)$$

(la fonction qui associe une borne supérieure paramétrée à chaque transition) par :

$$\beta \ : T \to \mathbb{N} \cup +\infty$$

la borne supérieure des intervalles.

Cette restriction nous permet de rentrer dans la classe des réseaux de Petri L/U, puiqu'il est impossible pour les paramètres d'être présents à la fois sur des bornes supérieures et inférieures d'intervalles. Comme évoqué ci-dessus, cela nous garantit des résultat de décidabilité sur l'existence d'une solution (Jovanović et al., 2013).

4.3 Analyse paramétrique

4.3.1 Algorithmes

Comme précisé plus haut, nous utilisons dans la suite de ce travail de thèse l'outil Roméo, qui implémente (entre autres) des algorithmes de synthèse de paramètre sur les réseaux de Petri temporels paramétrés à arcs inhibiteurs temporisés.

Les semi-algorithmes utilisés par Roméo sont présentés dans (Traonouez, Lime & Roux, 2008) dont voici un résumé.

Graphe de classes d'état paramétrique d'un piTPN : Pour un réseau de Petri temporel non paramétré, l'espace d'état est généralement infini en temps dense. En conséquence, il est nécessaire d'abstraire la notion de temps en regroupant les états dans des classes d'équivalences. On utilise donc des représentations symboliques de ces classes d'équivalence. L'une des approches pour le partitionnement de l'espace d'état en un ensemble fini de classes d'états infinies est le graphe de classes d'états.

Cependant, il existe aussi une infinité de valuations des paramètres. Il est donc nécessaire de représenter symboliquement le domaine des paramètres.

Pour les TPN et les automates temporisés, la représentation utilisée est celle des DBM (Difference Bound Matrix). Cependant, pour un réseau de Petri temporel paramétré à stopwatches (classe dont les réseaux de Petri temporels à arcs inhibiteurs font partie) le domaine de tir d'une classe est un polyèdre et ne peut pas être représenté par une DBM. Les classes d'états paramétriques des piTPN sont donc des polyèdres qui décrivent à la fois les domaines des variables des transitions, et les domaines des paramètres associés.

Calcul du graphe de classes d'états paramétrique : Le calcul du graphe de classes d'états paramétrique est effectué d'une manière similaire au graphe de classes d'états non paramétrique. Les paramètres sont inclus dans le domaine de tirs de la classe initiale, et les opérations qui calculent les classes successeurs ne concernent pas les paramètres. Cependant, au fur et à mesure de ce calcul, le domaine des paramètres dans une classe sera

automatiquement réduit pour ne considérer que les valuations qui permettent d'atteindre cette classe d'états.

Lorsque l'on remplace les paramètres par une de leur valuations dans le graphe des classes d'états, nous obtenons un graphe non paramétré. Cependant, certains domaines de tirs peuvent en conséquence devenir vides ce qui signifie que la classe n'est pas atteignable pour cette valuation des paramètres. Ces classes doivent être retirée du graphe, afin d'obtenir le graphe de classes d'états pour le réseau de Petri à arcs inhibiteurs correspondant à cette valuation.

Model-checking paramétré : La vérification de la satisfaction d'une propriété sur le réseau vise à obtenir un domaine pour les paramètres dans lequel la propriété est valide. On utilise pour exprimer les propriétés une extension paramétrique de la logique temporelle TCTL permettant d'ajouter des paramètres sur les bornes des intervalles que les expressions contiennent appelée PTCTL pour Parametric Time Computational Tree Logic (Wang, 1996).

Le problème du model-checking paramétré vise ainsi pour un piTPN $\mathcal{N}$ et une propriété PTCTL Φ à caractériser l'ensemble $F(\mathcal{N}, \Phi)$ de toutes les valuations de paramètres qui vérifie la propriété. Soit :

$$F(\mathcal{N}, \Phi) = \{\nu \in D_p \mid [[\mathcal{N}]]_\nu \models [[\Phi]]_\nu\}$$

avec ν une valuation, $[[\mathcal{N}]]_\nu$ le réseau $\mathcal{N}$ où les paramètres sont remplacés par leur valuation donnée par ν, et $[[\Phi]]_\nu$ la propriété où les paramètres sont remplacés par leur valuation donnée par ν.

Ce résultat est obtenu en calculant récursivement, pour chaque classe d'états, un prédicat logique correspondant à la vérification de la propriété par la classe concernée et ses successeurs.

(Traonouez et al., 2008) propose trois semi algorithmes permettant d'effectuer ce calcul, pour trois types de formules PTCTL différents.

4.3.2 Exemple

Afin d'illustrer les concepts présentés ci-dessus, nous vous présentons un exemple inspiré d'un problème d'ordonnancement présenté par (Bucci, Fedeli, Sassoli & Vicario, 2004), repris dans (Traonouez et al., 2009) pour contenir des arcs inhibiteurs temporisés ainsi que des paramètres sur les bornes des transitions.

Cet exemple, présenté en figure 4.5, contient trois tâches : la tâche 1 et la tâche 3 sont périodiques, et la tâche 2 est sporadique. Leurs périodes respectives sont représentés comme fonctions d'un paramètre a. Pour les trois tâches, il s'agit respectivement de a, $a \times 2$ et $a \times 3$.

Un autre paramètre est ajouté sur le temps d'exécution de la tâche 1. Il s'agit du paramètre b, sur la borne supérieure de la transition modélisant le temps d'exécution de la tâche : il décrit ainsi son pire temps d'exécution (Worst Case Execution Time, WCET).

Nous considérons la relation de priorité suivante : la tâche 1 a la priorité la plus importante, suivie de la tâche 2 et enfin de la tâche 3. Ces relations de priorité sont représentées par les arcs inhibiteurs temporisés présentés plus haut : une tâche de priorité supérieure vient inhiber les tâches de priorité inférieure à l'aide d'arcs partant des places la composant et visant les transitions des tâches concernées. On peut constater sur cet

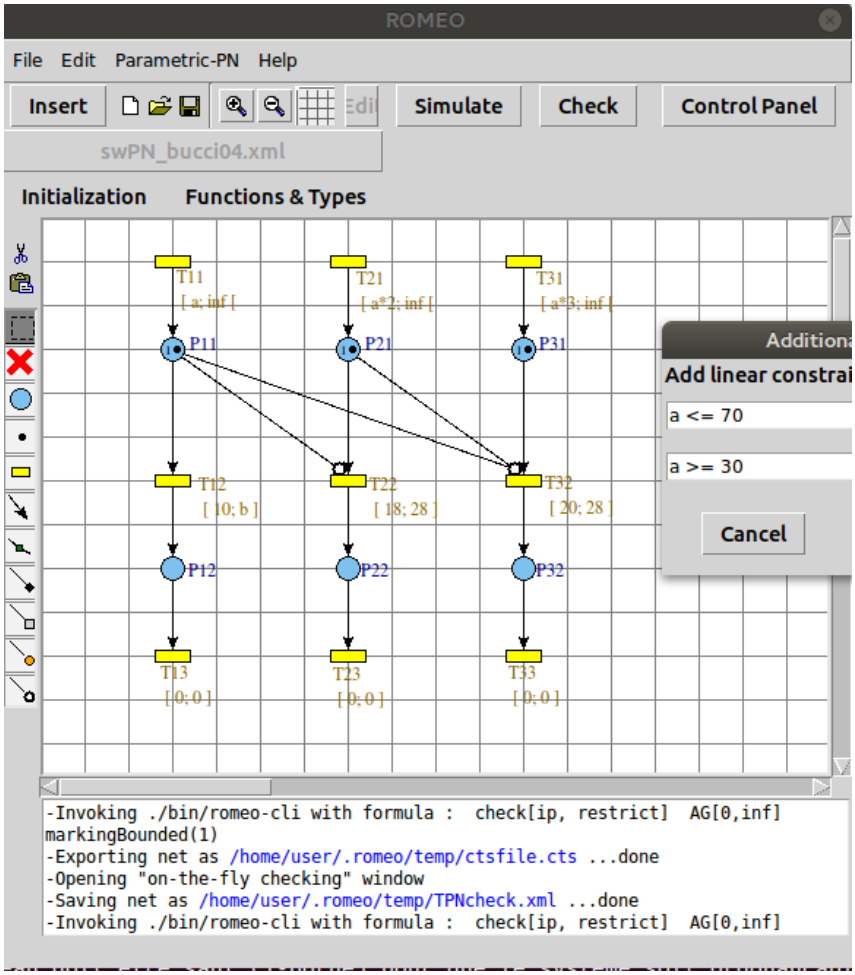


Figure 4.5 – Exemple de piTPN représentant trois tâches

exemple les arcs inhibiteurs partant de P11 et P21 pour inhiber les transitions des tâches 2 et 3, selon le cas.

Des contraintes supplémentaires sont définies initialement sur les paramètres : a est contraint à l'intervalle $[30, 70]$. On a donc $D_0 = \{30 \leq a \leq 70\}$.

Sur cet exemple, le problème qui nous intéresse est celui de l'ordonançabilité. Cela se traduit en logique temporelle par une contrainte, sur l'ensemble du réseau, d'un nombre de jetons maximum égal à 1. En d'autres termes, le réseau doit être sauf (1-borné) pour que le système soit ordonançable. La formule pTCTL correspondante est donc :

$$\forall P_i, \forall \Box_{[0,+\infty[}(M(P_i) \leq 1)$$

Avec $P_i \in P$, les places du réseau. Exprimée à l'aide de la syntaxe utilisée par Roméo, cette formule devient :

$$AG[0, inf[markingBounded(1)$$

L'analyse paramétrique nous donne le polyèdre suivant :

$$\begin{cases} 39 \leq a \leq 70 \\ 10 \leq b \leq 41 \\ a - b \geq 29 \end{cases}$$

Pour que le système soit ordonançable, la contrainte est que le paramêtre a qui contraint les périodes soit compris entre 39 et 70 unités de temps. Il ne faut pas, d'autre part que

le pire temps d'exécution de la première tâche dépasse 41 unités de temps. b est de plus contraint par une borne minimum de 10, qui correspond au meilleur temps d'exécution de la tâche déjà présent sur le modèle.

Une analyse plus poussée du WCET de la tâche 1 produit un résultat plus précis qui nous donne $\{b = 20\}$. En rajoutant cette contrainte sur le modèle, nous confirmons le résultat obtenu par Bucci et al. (2004) puisque l'on obtient alors comme polyèdre :

$$\left\{ 49 \leq a \leq 70 \right.$$

Ce qui correspond à $a = 50$, le résultat obtenu par Bucci et al. (2004).

Chapitre 5

Construction du modèle d'analyse

Ce chapitre est consacré à la présentation des étapes en amont de l'analyse paramétrique. Y est présenté le formalisme utilisé pour sa modélisation, ainsi que la manière dont il est utilisé.

Sommaire

5.1 Systèmes de transition à durée

Nous utilisons dans ces travaux les systèmes de transition à durée (Durational Transition Systems, DTS) afin de modéliser les tâches. Les modèles de tâches sont ensuite composés puis transformés afin d'obtenir le modèle d'entrée de l'analyse paramétrique (qui est lui un réseau de Petri temporel paramétré à arcs inhibiteurs temporisés). Le modèle embarqué et utilisé pour le forçage en ligne correspond lui aussi à un système de transitions à durée. Cette dernière étape est présentée dans le chapitre 7.

Nous commençons par présenter le formalisme des systèmes de transitions à durée ainsi que leur extension paramétrée, puis la manière dont sont obtenus les modèles des tâches. Sont ensuite présentés la paramétrisation, la composition puis la traduction vers les piTPNs de ces modèles de tâches afin d'obtenir le modèle d'entrée de l'analyse paramétrique.

5.1.1 Définition et sémantique des systèmes de transition à durée

Les systèmes de transition à durée (DTS) ont été proposés dans (Seidl, 1996). Un DTS est un système de transitions étiqueté où les transitions possèdent des durées admissibles spécifiées par un intervalle d'entiers. Ils peuvent aussi être vus comme une extension étiquetée des structures de Kripke à durée proposées dans (Laroussinie, Markey & Schnoebelen, 2002).

Soit I l'ensemble des intervalles fermés sur $\mathbb{N} \cup \{[a, +\infty[| a \in \mathbb{N}\}$.

Définition 5.1 (Système de transition à durée)**.** Un système de transitions à durée est un n-uplet $S = \langle Q, \Sigma, q_0, R \rangle$ où :

- Q est un ensemble fini d'états,
- $q_0 \in Q$ est l'état initial,
- Σ est un ensemble fini d'actions,
- $R \subseteq (Q \times \Sigma \times I \times Q)$ est un ensemble de transitions.

Une transition $(q, \sigma, \iota, q') \in R$ est notée $q \xrightarrow{\sigma,\iota} q'$ avec $q, q' \in Q$, $\sigma \in \Sigma$ et $\iota \in I$.

Définition 5.2 (Sémantique des sytèmes de transition à durée)**.** La signification voulue pour une transition $r = q \xrightarrow{\sigma,\iota} q' \in R$ est qu'il est possible de passer de q à q' par une action σ et avec une durée $d \in \iota$ ($d \in \mathbb{Q}$). Pour une durée $d \in \iota$ donnée, on écrit $q \xrightarrow{\sigma,d} q'$. Une séquence $\pi = q_0 \xrightarrow{\sigma_0,d_0} q_1 \xrightarrow{\sigma_1,d_1} q_2 \xrightarrow{\sigma_2,d_2} \dots q_n$ avec $\forall i$, $(q_i \xrightarrow{\sigma_i,d_i} q_{i+1}) \in R$ est un *chemin* si elle est finie, et un *run* si elle est infinie. Pour $q \in Q$, nous considérons $Exec(q)$ qui dénote l'ensemble des runs à partir de q.

La *durée* d'un chemin $\pi = q_0 \xrightarrow{\sigma_0,d_0} q_1 \xrightarrow{\sigma_1,d_1} q_2 \xrightarrow{\sigma_2,d_2} \dots q_n$ est $Time(\pi) = d_0 + d_1 + d_2 + \dots + d_{n-1}$ et le mot non temporisé de π est $Untime(\pi) = \sigma_0\sigma_1\sigma_2 \dots \sigma_n$.

Nous définissons les fonctions suivantes :

- $src(q \xrightarrow{\sigma,\iota} q') = q$, la source d'une transition ;
- $tgt(q \xrightarrow{\sigma,\iota} q') = q'$, la cible d'une transition ;
- $int(q \xrightarrow{\sigma,\iota} q') = \iota$, l'intervalle associé à une transition ;
- $lbl(q \xrightarrow{\sigma,\iota} q') = \sigma$, l'étiquette associée à une transition.

Un DTS est déterministe si $\forall (r, r') \in R^2$:
$(src(r) = src(r')) \wedge (tgt(r) \neq tgt(r')), \implies (lbl(r) \neq lbl(r'))$.

5.1.2 Définition et sémantique des systèmes de transition à durée paramétrés

Soit $Par = \{\lambda_1, \dots, \lambda_n\}$ l'ensemble des paramètres.

Une *valuation* des paramètres est une fonction $\nu : Par \to \mathbb{N}$.

Une *expression linéaire* γ sur les paramètres est une expression $\gamma = (\Sigma_{i=1..n}\ a_i * \lambda_i) + b$ où $a_i, b \in \mathbb{Z}$. On note $\mathcal{L}(Par)$ l'ensemble des expressions linéaires sur Par.

$\alpha : R \to \mathcal{L}(Par)$ associe une expression linéaire avec la borne inférieure de l'intervalle de chaque transition.

Définition 5.3 (Système de transition à durée paramétré)**.** Un système de transitions à durée paramétré est un n-uplet $S = \langle Q, \Sigma, q_0, R, Par, D_0 \rangle$ où :

- $\langle Q, \Sigma, q_0, R \rangle$ est un DTS,
- $Par = \{\lambda_1, \ldots, \lambda_n\}$ est un ensemble de paramètres,
- $\alpha : R \to \mathcal{L}(Par)$ est la fonction qui associe une borne inférieure paramétrée à l'intervalle de chaque transition,
- $\beta : R \to (\mathbb{N}^+ \cup +\infty)$est la borne supérieure de l'intervalle de chaque transition,
- D_0 est un polyèdre convexe qui est le domaine des paramètres.

Des contraintes initiales sont données pour les paramètres. Ces contraintes définissent le domaine initial D_0 pour les paramètres, et ce domaine est un polyèdre convexe. Ces contraintes doivent au minimum spécifier que pour tout $\nu \in D_0, r \in R : \alpha(r) \leq \beta(r)$

Des contraintes linéaires supplémentaires peuvent être fournies.

Définition 5.4 (Sémantique d'un système de transitions à durée paramétré)**.** Étant donné un système de transitions à durée paramétré S et une valuation des paramètres ν, nous notons $\nu(S)$ le système de transitions à durée non-paramétré où toutes les occurences d'un paramtère λ_i ont été remplacées par la valuation $\nu(\lambda_i)$.
$\forall \nu \in D_0$, $\nu(S)$ est un système de transitions à durée et la sémantique du système de transitions à durée paramétré S est définie par la sémantique de $\nu(S)$. La sémantique du système de transitions à durée paramétré est définie $\forall \nu \in D_0$ par la sémantique de $\nu(S)$.

5.2 Modèles des tâches du système

5.2.1 Démarche IDM

Ces travaux se placent dans le contexte plus large d'une démarche d'Ingénieurie Dirigée par les Modèles (IDM), dans la veine d'autres travaux (Lelionnais, Delatour, Brun, Roux & Seidner, 2014). Plus précisemment, l'un des objectifs à moyen terme est de permettre l'intégration des résultats de ce travail de thèse au sein de SexPIsTools, une chaine de développement basée sur les modèles pour les systèmes temps réel développée par l'équipe TRAME de l'ESEO.

5.2.1.1 Concepts et fonctionnement de SExPIsTools

Dans ces travaux, les auteurs présentent une démarche de conception de systèmes embarqués centrée sur l'IDM, à l'aide d'un ensemble d'outils qu'ils ont développé : la suite logicielle SExPIsTools qui permet à la fois de travailler sur la structure et le comportement des systèmes considérés. À l'aide d'un ensemble de modèles qui décrivent la structure et le comportement de l'application et de ses tâches d'une manière indépendante de la platforme considérée, ainsi que de modèles génériques décrivant la ou les platformes sur lesquelles elle doit être déployée, cette suite logicielle génère des modèles formels décrivant tout ou partie du système permettant son analyse à l'aide de méthodes formelles ainsi que le code permettant son implémentation sur la plateforme considérée.

La figure 5.1 montre la manière dont les différents modèles de la plateforme et de l'application interagissent dans cette suite logicielle. L'aspect novateur de cette approche réside dans le fait de pouvoir séparer la description de la plateforme de la description des applications et de leur comportement faites dans un langage dédié (RTEPML, voir :

Brun, 2010). De cette manière, les modélisations des différents systèmes peuvent être réutilisées pour d'autres applications.

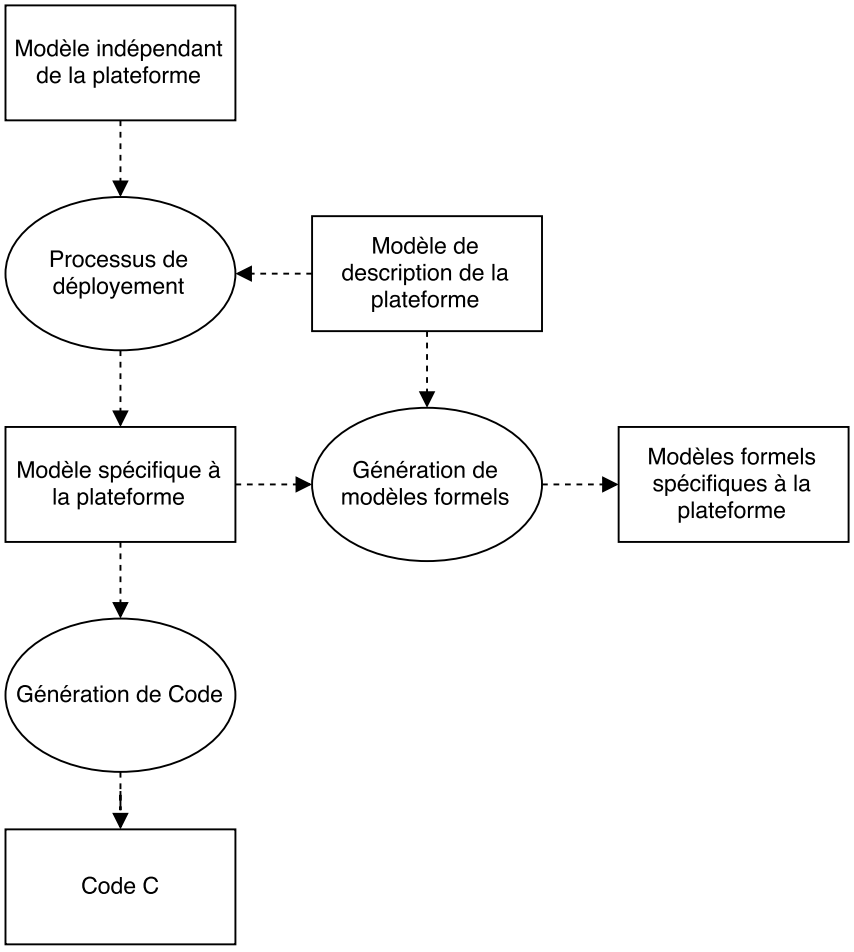


Figure 5.1 – Processus de génération de code et de modèles avec SExPIsTools

5.2.1.2 Lien avec notre travail

Le fait que nos travaux s'inscrivent dans cette continuité nous permet d'obtenir des modèles formels spécifiques pour châque tâche du système ainsi que des modèles nous donnant la structure générale de l'application et les liens entre les tâches. À l'aide de transformations de modèles, nous obtenons donc les modèles comportementaux des tâches exprimés avec le formalisme des DTS.

De plus, le niveau d'abstraction considéré est celui des traces du système d'exploitation temps réel : la plupart des transitions de modèles considérés correspondent à des appels système et le reste des comportements est masqué par les temps d'exécution inclus sur les transitions de la DTS. Ces temps d'exécutions, sous la forme $[a, b]$ correspondent a des estimations de temps d'exécution entre les différents appels système. a correspond au temps d'exécution dans le meilleur cas (Best Case Execution Time, BCET), b correpond au temps d'execution dans le pire cas (Worst Case Execution Time, WCET).

Ces méthodes IDM en amont de nos travaux nous permettent d'obtenir ces modèles afin de les manipuler.

5.2.2 Modèles de tâches (p)DTS

Cette section décrit avec plus de précision les modèles DTS des différentes tâches ainsi que la manière dont ils sont paramétrés avant leur composition.

5.2.2.1 Caractéristiques et description

La figure 5.2b présente un extrait du code d'une tâche réalisant le contrôle d'un système physique, implémentée selon la norme AUTOSAR OS. À partir de ce code, on produit un modèle DTS représentant le comportement de la tâche du point de vue du système d'exploitation temps réel.

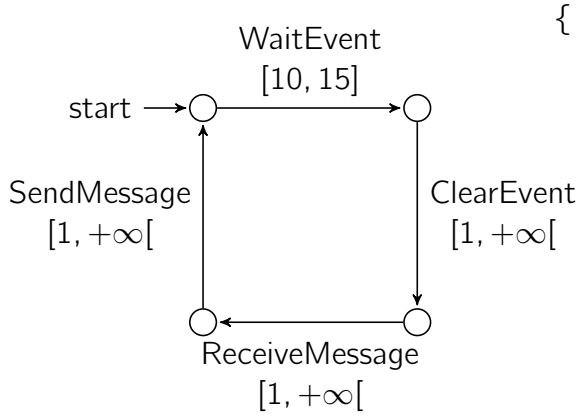
La figure 5.2a présente le modèle obtenu pour la tâche de contrôle. Le RTOS n'interagit avec les tâches que lors des appels système, c'est donc les évènements qui correspondent aux transitions à l'intérieur des modèles des tâches. Le temps spécifié sur les transitions des DTS de description des tâches reste local à chaque tâche : ces modèles ne modélisent pas les possibles préemptions, mais simplement le temps tel qu'il peut être observé par les tâches lors de leur exécution.

La tâche concernée est chargée du contrôle d'un système physique. On observe la boucle de contrôle, de part l'existence des appels système situés sur son tracé. Les intervalles de temps d'exécution sur les différentes transitions correspondent bien au BCET et WCET calculés pour ce système.

5.2.2.2 Restrictions

Nous considérons que l'ensemble Σ des étiquettes d'une pDTS $S = \langle Q, \Sigma, q_0, R, Par, D_0 \rangle$ représentent les appels système associés le cas échéant à chaque transition considérée. Si aucun appel système n'est associé à une transition, alors l'étiquette ϵ est associée à la transition.

Les modèles de pDTS ne considèrent que le temps local. En effet, une tâche n'est pas en mesure d'observer les préemptions ou les réordonnancements qui pourrait résulter d'une exécution de l'ordonnanceur lors d'un appel système. Sur l'exemple de la tâche de



(a) Modèle DTS

```
Task(Control)
{
  ...
  while(1) {
    WaitEvent(inputMsg);
    ClearEvent(inputMsg);
    ReceiveMessage(inputMsgData,
                   &data);
    output = compute(data);
    SendMessage(outputMsgData,
                &output);
  }
  TerminateTask();
}
```

(b) Code de la tâche Autosar

Figure 5.2 – Modèle et code tels qu'obtenus après extraction pour une tâche de contrôle

contrôle, on peut observer qu'un réordonnancement pourrait avoir lieu lors de l'attente de l'évènement, par exemple pour exécuter la tâche devant envoyer le message attendu. L'intervalle $[1\,;+\infty[$ sur cette transition correspond bien à un temps local à la tâche, sans tenir compte d'une telle préemption. Le temps d'attente possible sera modélisé dans la tâche émettant l'évènement, par l'appel à SetEvent.

5.2.2.3 Traduction des motifs d'exécution et paramétrisation

Les modèles de tâches sont obtenus par réduction des modèles comportementaux issus de l'analyse IDM. Pour obtenir ces modèles, il nous faut donc traduire un ensemble de motifs d'exécution, illustrés en figure 5.3 :

- Séquence (premier motif de la figure 5.3) : Il est possible de traduire directement ces motifs s'il n'y a pas d'appel système qui leur est lié. Sinon, il faut séparer le temps d'exécution de l'appel système lui même en deux transitions successives, séparées par un état : l'une avec le temps d'exécution et l'étiquette ϵ, l'autre avec l'appel système en temps zéro. Ceci permet de représenter le fait que les délais qui seront ajoutés lors de l'exécution afin de permettre le contrôle ne seront pas ajoutés durant l'appel système lui-même, mais bien avant que l'appel système ne soit effectivement exécuté. Cette séparation nous permet en aval d'effectuer l'analyse en tenant compte de ces considérations.
  Il s'agit cependant d'une approximation du comportement réel du système : nous considérons que le temps d'exécution de l'appel système est inclus dans l'intervalle de la transition précédente, et la séparation entre temps utilisateur et temps du noyau n'est pas considérée.
  Enfin, un paramètre est ajouté sur la borne inférieure de l'intervalle de temps d'exécution initialement associé avec cet appel système, soit la transition ϵ ajoutée.
- Alternative (second motif de la figure 5.3) : un motif si-alors-sinon. Si aucune des deux branches ne contient d'appel système, alors ce motif peut être réduit à une

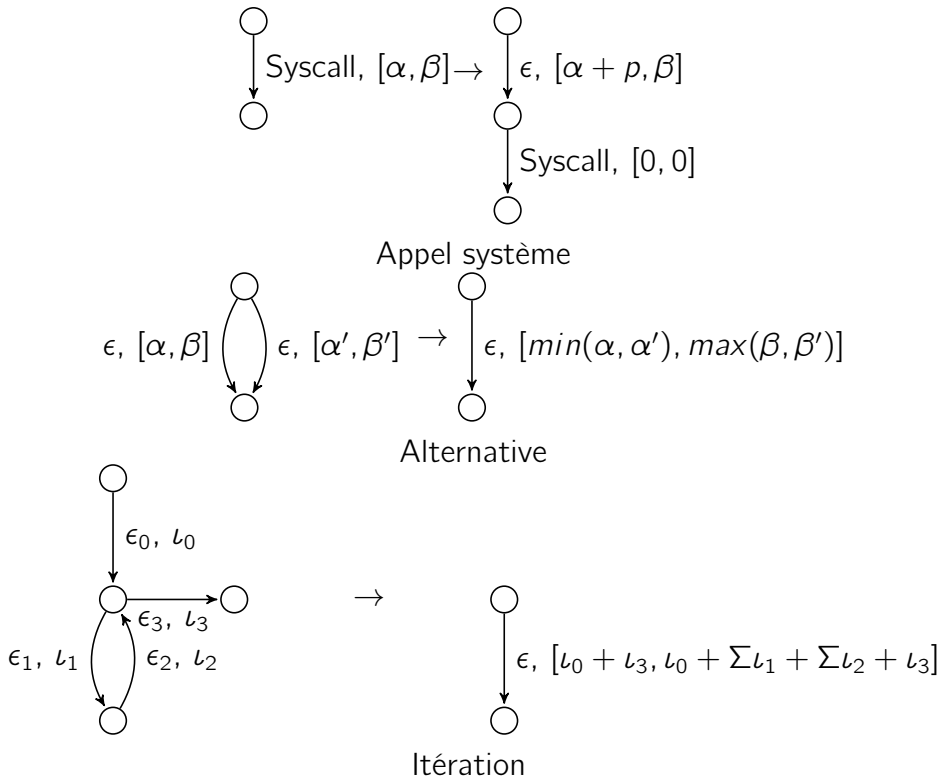


Figure 5.3 – Traduction des motifs d'exécution des pDTS

seule transition tel que défini ci-dessous dans la règle 1. Sinon, il peut être traduit directement, en appliquant la règle de traduction d'un motif de séquence pour les appels système concernés.

Rêgle 1. $\forall$ les états q d'une DTS S, si $\exists t, t'$ tel que
$q = src(t) \wedge q = src(t') \wedge tgt(t) = tgt(t')$,
avec $lbl(t) = lbl(t') = \epsilon$, $int(t) = [a, b]$, $int(t') = [c, d]$;
alors nous choisissons de le modéliser avec une seule transition :
$q \xrightarrow{[min(a,c),max(b,d)],\epsilon} q'$.

En effet, c'est le seul comportement qui sera observable du point de vue des appels système. On note que si $b < c$, des comportements sont ajoutés au modèle.

– Itération (dernier motif de la figure 5.3) : Si un motif itératif ne contient aucun appel système (et donc uniquement des étiquettes ϵ), alors il est abstrait par une seule transition considérant son temps d'exécution total, d'une manière similaire à celle présentée ci-dessus. Ici aussi, le seul comportement modélisé correspond à celui observable du point de vue des appels systèmes. Dans les autres cas, les règles présentées s'appliquent pour les appels système concernés.

L'application de ces règles de traduction des différents motifs nous donne pour le modèle DTS de la tâche de contrôle de la figure 5.2a une pDTS présentée en figure 5.4.

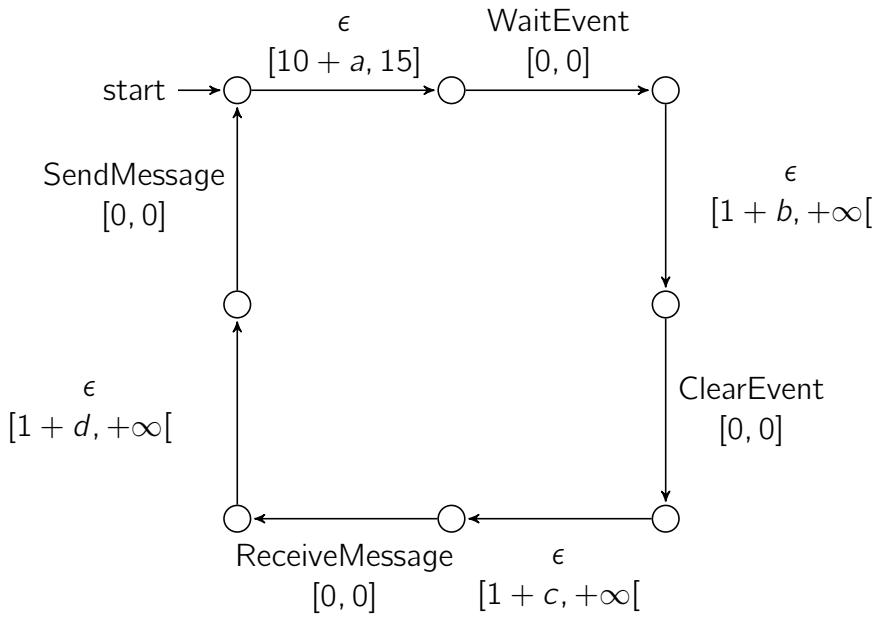


Figure 5.4 – Modèle pDTS de la tâche de contrôle

5.3 Transformation et composition des modèles d'entrée

Le modèle utilisé lors de l'analyse est un réseau de Petri T-temporel paramétré à arcs inhibiteurs, comme décrit en section 4.2.

Le choix des réseaux de Petri temporels à arcs inhibiteurs temporisés est motivé par le besoin d'avoir un modèle complet de l'application, comprenant à la fois le comportement des tâches et leurs interactions. En effet, ce modèle doit être capable de représenter la préemption qui peut avoir lieu lors des appels systèmes. Nous utilisons pour ce faire les arcs inhbiteurs temporisés, puisque leur sémantique signifie qu'une transition inhibée par un arc inhibiteur conserve la valeur de son horloge, ce qui nous permet de reprendre l'exécution d'une tâche là où elle en était rendue.

Ce formalisme peut être analysé à l'aide du logiciel Roméo(Lime et al., 2009).

5.3.1 Transformation de pDTS vers piTPN

Afin de composer le modèle global du système, il nous faut d'abord transformer chaque modèle de tâche de pDTS vers piTPN. Étant donné les restrictions syntaxiques évoquées plus haut dans les rêgles de traduction, cette transformation peut être simplement structurelle : les états de la DTS correspondant aux places du réseau de Petri, et les transitions entre les états aux transitions dans le réseau de Petri entre les places adéquates.

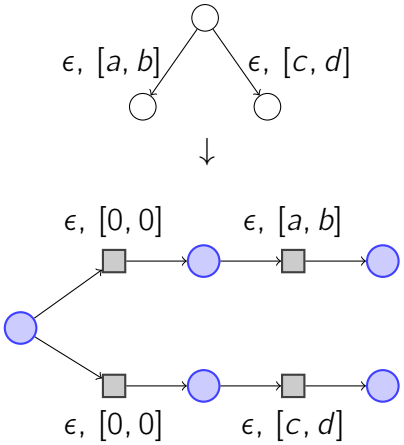


Figure 5.5 – Motif de choix lors du passage de pDTS vers piTPN

Définition 5.5. Transformation de pDTS vers piTPN :
Soit $S = \langle Q, \Sigma, q_0, R, Par, D_0 \rangle$ une pDTS.
S est équivalente à un piTPN $N = \langle P, T, \Sigma, \ell, Par, Pre, Post, Inh, m_0, \alpha, \beta, D'_0 \rangle$ avec :

- $\forall q \in Q$, $\exists p \in P$, tel que chaque q corresponde à un p ;
- m_0 le marquage, tel que $m_0(p) = 1$ pour p la place équivalente à q_0, et $m_0(p) = 0$ sinon,
- $D_0 = D'_0$,
- $\forall r = (q, \sigma, i, q') \in R$, $\exists t \in T$, tel que $\ell(t) = \sigma$, $(p, t) \in Pre$, $(t, p') \in Post$, et $i = [\alpha(t), \beta(t)]$, q correspondant à p et q' à p'.

Puisque les pTPN ont une sémantique forte, nous ajoutons un choix non-déterministe en temps zéro qui vient remplacer chaque branchement du modèle avant d'appliquer les rêgles ci-dessus. Soit :
$\forall q, q', q'' \in S$ t.q. $\exists q \xrightarrow{\sigma,\iota} q'$ et $q \xrightarrow{\sigma',\iota'} q''$ avec $\sigma = \sigma' = \epsilon$ et $\iota \neq \iota'$,
nous ajoutons q_a, q_b avec $q \xrightarrow{\epsilon,[0,0]} q_a$, $q \xrightarrow{\epsilon,[0,0]} q_b$, $q_a \xrightarrow{\epsilon,\iota} q'$ et $q_b \xrightarrow{\epsilon,\iota'} q''$.

Cette dernière rêgle est illustrée en figure 5.5

Pour la tâche de contrôle présentée en exemple, le résultat final de la transformation de pDTS vers piTPN est présenté en figure 5.6

5.3.2 Composition

La prochaine étape pour obtenir le modèle complet de l'application est de fusionner l'ensemble des modèles de tâches obtenus en un modèle complet du sytème sur lequel réaliser la synthèse de paramètres.

Cette composition est obtenue par la fusion de places et de transitions des différentes tâches avec des places et transitions d'un ensemble de motifs. Les motifs correspondent aux interactions entre les tâches, et notamment aux appels systèmes impliqués dans ces interactions. La figure 5.7 montre deux exemple de motifs qui permettent de faire communiquer les tâches : le déclenchement et l'attente d'un évènement (SetEvent et WaitEvent), et le démarrage d'une tâche depuis une autre (ActivateTask).

On observe que dans chaque cas les motifs ajoutent des éléments (places, transitions, arcs) au modèle, tout en venant fusionner d'autres éléments avec les modèles de tâches

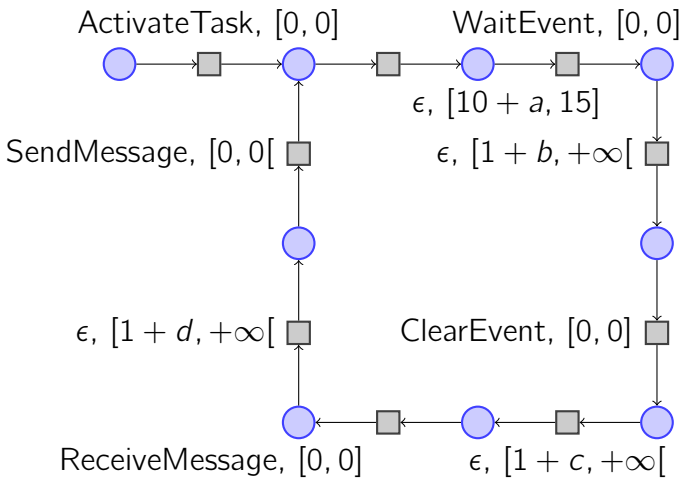


Figure 5.6 – Tâche de contrôle transformée en piTPN

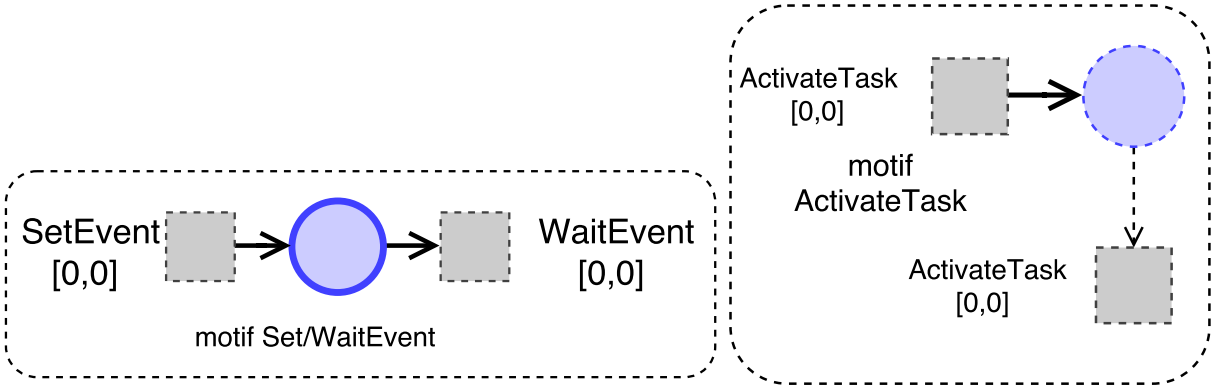


Figure 5.7 – Motifs pour la composition

(les transitions étiquetées avec les noms d'appels systèmes, leurs places associées, etc).

Afin de pouvoir correctement composer ces différents modèles, il nous faut recourir à un modèle structurel décrivant la manière dont les tâches interagissent entre elles. Dans la lignée des travaux réalisés précédemment (Brun, 2010 ; Lelionnais et al., 2014), nous utilisons un modèle conforme à AADL (Architecture Analysis & Design Language), issu de la démarche IDM, et décrivant l'architecture de l'application à un niveau structurel.

Ce modèle vient guider la composition, nous permettant de savoir quelles transitions et places fusionner. Il nous permet d'autre part de connaitre les priorités des différentes tâches du système. Dans le modèle composé, la relation d'ordre entre les priorités est modélisée à l'aide des arcs inhibiteurs temporisés décrit au chapitre 4 dans la sémantique des piTPN. L'ensemble des places des tâches de plus haute priorité viennent inihiber l'ensemble des transitions des tâches de plus faible priorité.

Définition 5.6 (Inhibition d'une tâche)**.** On dit qu'une tâche A inhibe une tâche B quand : $\forall p \in$ Tâche A, $t \in$ Tâche B, $\exists inh \in Inh$ tel que $inh = (p, t)$.

Afin d'alléger la représentation graphique de l'inhibition d'une tâche par une autre dans la suite de ce manuscrit, nous représentons cette inhibition par un seul arc inhibiteur dont la source et la destination sont des rectangles arrondis entourant les deux tâches concernées.

La figure 5.8 propose un exemple simple de modèle structurel d'une application. On y observe deux tâches de priorités différentes, communiquant l'une avec l'autre au moyen d'un

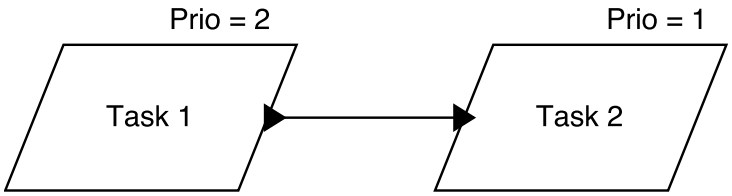


Figure 5.8 – Modèle structurel d'interaction entre les tâches

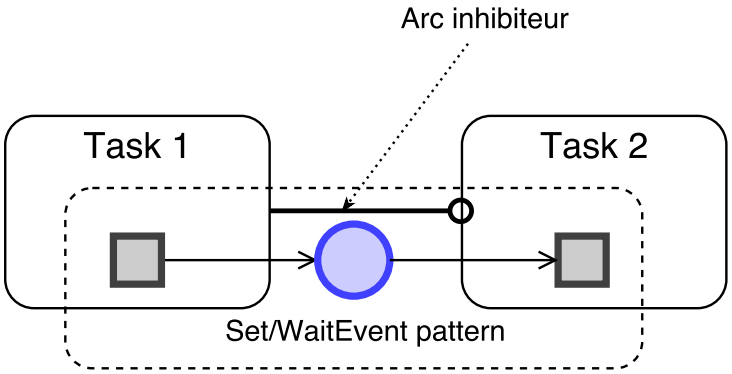


Figure 5.9 – Exemple d'utilisation d'un motif d'appel système

évènement. La composition des modèles comportementaux associés aux tâches concernées nous donne un modèle composé correspondant a celui de la figure 5.9.

Les comportements des tâches 1 et 2 n'y sont pas explicités, mais on peut observer la manière dont le motif de communication basé sur les évènements s'intègre par fusion de places dans les deux modèles comportementaux des tâches, ainsi que la notation décrite plus haut de l'inhibition d'une tâche par l'autre.

Chapitre 6

Principe de mise en œuvre

Nous proposons dans ce chapitre la description des étapes menant au forçage en ligne, ainsi qu'une vue d'ensemble de la manière dont celui-ci est implémenté dans un système d'exploitation temps réel.

Sommaire

Afin de pouvoir réaliser le forçage en ligne, le modèle obtenu à l'issu de la démarche présentée en chapitre 3 est utilisé pour obtenir la stratégie de test, et donc les délais à appliquer lors de l'exécution du test.

Un modèle de chaque tâche intégrant les valeurs de délais nécessaires au forçage est ensuite construit et intégré au contrôleur embarqué dans le système d'exploitation temps-réel.

6.1 Analyse hors-ligne du modèle du système

6.1.1 Analyse paramétrique

Le modèle complet et paramétré de l'application considérée, obtenu à la suite de la démarche décrite au chapitre 3, nous permet de réaliser l'analyse hors-ligne avec Roméo qui nous permet d'obtenir des valeurs pour les paramètres du système. Afin de pouvoir réaliser cette analyse, il nous faut considérer l'objectif de test, exprimé sous la forme d'une formule d'accessibilité en logique temporelle paramétrée.

6.1.1.1 Objectif de test

L'objectif de test est défini à l'origine de cette démarche. Il s'agit la plupart du temps d'un ensemble d'états à atteindre, objectif qui sera accompli lors de l'exécution à l'aide des différents délais ajoutés au programme par le contrôleur.

Afin de pouvoir accomplir l'analyse paramétrique et synthétiser des valeurs de paramètres, cet objectif de test doit tout d'abord être formalisé sous la forme d'une propriété de logique temporelle paramétrée (pTCTL, parametric Time Computational Tree Logic).

Par exemple, un objectif peut être l'atteinte d'un état dans lequel se déclenchera un mécanisme de sûreté, sur un système critique. Le déclenchement du mécanisme de sûreté n'est pas contrôlable uniquement par les entrées/sorties du système, mais nécessite soit l'occurence d'une faute temporelle, soit le suivi d'une certaine trajectoire interne du système. Il est donc nécessaire, afin de pouvoir tester ce mécanisme, de forcer son déclenchement à l'aide de notre solution.

Le déclenchement du mécanisme de sûreté a lieu sur une place nommée échec de la tâche adéquate. Il est souhaité d'atteindre cet état à chaque exécution du système, pour permettre le test du mécanisme de sûreté, et ce sans contrainte supplémentaire sur le temps d'exécution avant le déclenchement du mécanisme de sûreté. Formellement, cette contrainte s'exprime de la manière suivante à l'aide de la syntaxe pTCTL utilisée dans Roméo :

$$AF([0, +\infty[)\ M(\text{échec}) > 0$$

En d'autres termes, pour toutes les exécutions du système (AF), nous devons aboutir dans un état où la place *échec* contient au moins un jeton (M(échec)>0), et ceci sans contrainte temporelle ($[0, +\infty[$).

Une fois l'objectif de test formalisé à l'aide de pTCTL et le modèle composé de l'application obtenu, ces deux éléments peuvent être fournis à Roméo pour permettre l'analyse du système et l'obtention des valeurs admissibles pour les paramètres permettant d'atteindre l'état visé (dans notre exemple, l'état où au moins un jeton est présent dans la place échec).

6.1.1.2 Résultat de l'analyse

La synthèse de paramètres avec Roméo nous fournit à l'aide du semi-algorithme décrit en section 4.3 une union de polyèdres convexes qui sont une sous-approximation de l'ensemble des solutions possibles sur le domaine des paramètres. Cette sous-approximation contient néanmoins l'ensemble des solutions entières au problème.

Si Roméo nous propose plusieurs polyèdres, ceux-ci correspondent à des scénarios de contrôle différents. Par exemple, l'un des polyèdres pourrait revenir à ajouter des délais lors

du passage dans une première tâche, tandis que l'autre pourrait vouloir signifier l'ajout de délais dans une seconde tâche. Le choix du polyèdre à utiliser n'a pas d'importance dans le sens où l'état visé sera toujours atteignable. Néanmoins, ils vont permettre de venir tester de différentes manières, en prenant des chemins différents dans le graphe des classes du réseau de Petri, l'état qu'il est souhaité d'atteindre. Le choix du polyèdre à utiliser est laissé à la discretion de l'utilisateur de cette méthode, les critères pouvant inclure la minimisation du nombre de paramètres (en en forçant une partie à zéro), ou encore l'exclusion du forçage de certains appels systèmes (là aussi, en forçant les paramètres associés aux transitions correspondant à ces appels système à zéro).

Une fois ce choix effectué, n'importe lequel des ensembles de valeurs admissibles pour les paramètres (donc l'ensemble des valeurs contenues dans le polyèdre choisi) devrait théoriquement nous permettre d'effectuer le contrôle du système afin d'atteindre l'état choisi. Néanmoins, il nous reste à choisir un de ces points afin de passer à l'étape de contrôle en ligne.

6.1.2 Choix du point

6.1.2.1 Point optimal

Le choix du point n'a à priori pas de conséquence, puisque toutes les valeurs comprises à l'intérieur du ou des polyèdres pour les paramètres devraient satisfaire à nos exigences. Néanmoins, il faut tenir compte de deux problèmes :

- nous ne pouvons utiliser que les valeurs entières contenues dans ces polyèdres, ce qui élimine un certain nombre de solutions (voir de polyèdres qui ne contiennent pas de valeurs entières).
- Le mécanisme réalisant le forçage en ligne n'est pas idéal, sa précision n'est pas infinie. Afin de tenir compte de cette imprécision dans le forçage, il peut donc être souhaitable d'ajouter aux polyèdres des contraintes adéquates. Ces contraintes pourraient être de la forme $a > \Delta$, Δ représentant la précision temporelle du système de contrôle en ligne.
  Ce Δ étant dépendant de l'implémentation, nous ne considérons pas ces contraintes supplémentaires dans les exemples présentés dans la suite de ce travail de thèse. Cependant, la section 6.3.5 présente la manière dont il peut être considéré dans la description de l'implémentation.

Une autre considération est celle de l'impact de notre contrôleur sur le système : si une analyse doit être faite des résultats obtenus lors du test, il faut que le contrôleur ait un impact minimum sur l'exécution du système afin que les résultats soient les plus proches possibles de la réalité. Le choix du point doit donc être effectué en mettant le plus possible de paramètres à zéro, et en choisissant un point avec des valeurs de paramètres les plus faibles possibles : cela garantit que les délais appliqués seront minimes, puisque les paramètres ne sont présents que sur les bornes inférieures des intervalles.

L'algorithme permettant le choix du point est donc le suivant :

- Tout d'abord, il faut supprimer le maximum de paramètres des équations en forçant le maximum de paramètres possible à une valeur de zéro. Formellement :

Définition 6.1 (Réduction du nombre de paramètres)**.**
- Soit D un polyèdre sur Par.
- Soit $P_k(Par) = \{E | E \in \mathcal{P}(Par) \wedge |E| = k\}$ l'ensemble des parties de Par de cardinal k.
- Pour un élément $E \in P_k(Par)$, on définit $C_E = \wedge_{p \in E}(p = 0)$ l'ensemble des contraintes qui forcent à 0 les paramètres appartenant à E.
  On définit Sol_D^k l'ensemble des ensembles de solutions de D avec k paramètres nuls par $Sol_D^k = \{S | (\exists E \in P_k(Par)), (S = D \cap C_E) \wedge (S \neq \emptyset\}$
  Une solution maximisant le nombre de paramètres nuls est un élément $s \in S$ avec $S \in \max_{k \in [|1..2^{|Par|}]} Sol_D^k$

L'algorithme explore les sous ensembles de $P_k(Par)$ par cardinalité décroissante. Pour chaque combinaison, il teste leur appartenance à Sol, en vérifiant si le polyèdre a une solution lorsque les paramètres correspondants sont forcés à zéro. La première solution rencontrée est retenue. Il est possible que plusieurs solutions existent pour une taille de sous-ensemble donnée.

– Il faut ensuite résoudre un problème de programmation linéaire sur le polyèdre résultant, afin de choisir un des points entiers restant dans le polyèdre.

6.1.2.2 Programmation linéaire

La programmation linéaire propose des algorithmes visant à optimiser des fonctions sur des domaines délimités par des contraintes linéaires. Dans notre cas, le domaine est celui des paramètres et les contraintes sont celles (linéaires) fournies par le polyèdre résultant de l'heuristique que nous venons de présenter.

La fonction d'optimisation, quant à elle, découle des considérations exprimées ci-dessus vis-à-vis du choix d'un point optimal : l'impact sur le système doit être le plus faible possible. Cette contrainte peut être interprétée de différentes manières. Celle que nous avons choisie est d'utiliser la somme des valeurs de paramètres comme critère à minimiser.

D'autres solutions pourraient être de minimiser la valeur médianne des paramètres, ou encore de forcer le maximum possible de paramètres à leur contrainte de valeur minimale présentée par le polyèdre, tout en laissant les autres paramètres aux plus petites valeurs possibles autorisés par ces nouvelles contraintes (mais qui pourraient toutefois rendre la moyenne plus importante), réduisant ainsi les emplacements où les perturbations importantes du système seraient nécessaires. Ces solutions n'ont cependant pas été envisagées dans cette thèse en raison de leur caractère non-linéaire.

La définition 6.2 exprime formellement cette contrainte visant à minimser la somme des paramètres.

Définition 6.2 (Fonction à optimiser pour le problème de programmation linéaire)**.**

$$\begin{aligned} &\text{Minimiser } \sum_{i=1}^{l} \lambda_i \\ &\quad \text{tel que } (\lambda_1 ..., \lambda_l) \models D' \\ &\qquad \text{avec } D' \text{ un polyèdre sur } l \text{ paramètres, } \lambda_i \in Par. \end{aligned}$$

D' est le polyèdre de contraintes sur les paramètres, résultant de l'algorithme décrit ci-avant.

Étant donné le caractère linéaire de la fonction à optimiser et des contraintes définissant le polyèdre qui est de plus convexe, les algorithmes de programmation linéaire tirent le plus souvent partit du fait que les résultats optimaux se trouvent sur la frontière du polyèdre, et notamment qu'un sommet sera forcément inclus dans la solution. De fait, une méthode simple mais peu optimale pour résoudre un tel problème est de parcourir l'ensemble des sommets du polyèdre considéré. Néanmoins, il se peut que la solution ainsi trouvée ne soit pas entière.

Il existe un grand nombre d'algorithmes de résolution du problème de programmation linéaire :

- La méthode du simplex (Luenberger, 1973), la plus ancienne méthode, qui consiste à visiter les sommets du polyèdre en suivant l'augmentation (ou la diminution) de la fonction à optimiser, jusqu'à atteindre le sommet optimal.
- La famille des méthodes des points intérieurs, dont fait partie l'algorithme de Karmarkar (Karmarkar, 1984), qui consistent à traverser l'intérieur de l'ensemble des solutions réalisables. Cette méthode est plus récente et plus efficace que celle du simplex, permettant de résoudre des problèmes impossible à résoudre avec cette dernière.
- La génération de colonnes (Desaulniers, Desrosiers & Solomon, 2006), pour les problèmes de grande taille.
- Il existe aussi de nombreuses méthodes optimisées pour les problèmes entiers, dont la méthode des plans sécants (Wolsey, Rinaldi et al., 1993).

Nous utilisons pour résoudre le problème de programmation linéaire la suite logicielle GLPK (*GLPK (GNU Linear Programming Kit)*, 2012). GLPK utilise pour les problèmes entiers la méthode des plans sécants de Gomory (Gomory, 1960), ainsi qu'une version du simplex et de la méthode des points intérieurs pour les problèmes non entiers.

6.1.2.3 Implémentation de la recherche du point

La recherche du point à utiliser pour le forçage en ligne a été implémentée dans un script en Ruby. Le modèle en entrée est un réseau de Petri au format .cts, utilisé par Roméo. Il inclut de plus la formule pTCTL d'accessibilité utilisée pour l'analyse. Le script appelle Roméo et utilise GLPK pour obtenir la valeur de chacun des paramètres suivant l'algorithme décrit plus haut.

La minimisation du nombre de paramètres est implémentée en testant les différentes combinaisons de paramètres à zéro, en partant des combinaisons les plus larges et en s'arrêtant dès qu'une solution est trouvée. À chaque itération, la solvabilité du problème est testée à l'aide du solveur choisi par l'utilisateur.

Une fois la réduction du nombre de paramètres effectuée, le point optimal est calculé, et le script sort pour chaque paramètre du modèle initial une valeur correspondant au résultat de l'heuristique après l'analyse par Roméo.

6.1.2.4 Exemples

Exemple théorique : Soit un polyèdre issu de l'analyse d'un piTPN contenant les paramètres a, b, c, et d :

$$\begin{cases} 4 < a \leq 15 \\ b > 3 \\ 0 < c \leq 5 \\ d > 2 \\ a - b > 10 \\ b + d < 20 \end{cases}$$

Après utilisation de la technique de réduction, nous obtenons comme valeurs pour les paramètres :

$$\begin{cases} a = 15 \\ b = 4 \\ c = 0 \\ d = 3 \end{cases}$$

Tâche de contrôle : Nous reprenons l'exemple présenté en section 5.2.2, figure 5.2a[1]. L'objectif de l'analyse est de simuler l'envoi d'un message en réponse à l'évènement en un temps supérieur à 40 unités de temps. La manière dont cela peut être exprimé en TCTL, étant donné le modèle considéré, est la suivante :

$$AG[0, 40]M(9) == 0$$

En d'autres termes, il ne faut pas qu'il y ait de jetons dans la place P9 située en amont de l'envoi de la réponse avant 40 unités de temps.

Après analyse par Roméo, nous obtenons comme résultat le polyèdre suivant :

$$\begin{cases} a \geq 0 \\ b \geq 0 \\ c \geq 0 \\ d \geq 0 \\ a + b + c + d \geq 37 \end{cases}$$

Ce résultat correspond à l'intuition du problème : le temps total passé dans la boucle doit excéder 40 unités de temps.

Après résolution du problème d'optimisation, nous obtenons :

$$\begin{cases} a = 0 \\ b = 0 \\ c = 0 \\ d = 37 \end{cases}$$

[1]Nous ne considérons pas ici le modèle composé avec les autres tâches du système. Une application réelle nécessiterait bien entendu la présence d'une seconde tâche pour envoyer l'évènement (avec SetEvent). C'est cet envoi d'évènement qui pourrait inclure un délai supplémentaire lors de l'exécution du WaitEvent

Ce résultat indique que le paramètre d, associé sur le modèle à l'appel système SendMessage (voir figure 5.4, page 56) doit être déclenché après un minimum de 37 unités de temps par rapport à sa date de tir minimale.

6.2 Génération des modèles embarqués

6.2.1 Principe

Une fois les valeurs des paramètres obtenues, nous pouvons procéder à la génération des modèles qui seront embarqués pour le forçage en ligne. Les modèles embarqués sont des DTS issues des pDTS décrivant les tâches, et dans lesquelles les valeurs des paramètres ont été fixées. Ces modèles doivent être transformés afin de pouvoir faciliter le forçage.

Les DTS utilisées lors du forçage permettent au contrôleur de tenir son rôle d'observateur du comportement du système. La représentation interne au contrôleur des DTS évolue en fonction des appels systèmes observés, et donc des tirs de transitions correspondants. Chaque tâche est décrite par une DTS, ce qui permet au contrôleur de suivre son évolution, en temps local. On considère comme "local" le temps d'exécution propre à la tâche, et comme "global" le temps de réponse des tâches.

6.2.2 Injection des valeurs de paramètres

Les valeurs de paramètres obtenues par l'analyse du piTPN composé sont injectées dans les modèles pDTS des tâches correspondantes. On obtient ainsi pour chaque tâche une DTS issue de sa pDTS décrite au début de cette analyse.

Les figures 6.1 et 6.2 nous montrent une pDTS et la DTS correspondante après injection des paramètres, pour la tâche de contrôle présentée en exemple en section 5.2 et avec les valeurs obtenues en section 6.1.2.4.

6.2.3 Epsilon-réduction

Comme précisé plus haut, le modèle DTS des tâches doit servir à l'observation du système par le contrôleur. Le contrôleur n'est capable d'observer comme évènements que les appels systèmes effectués par les tâches. Il nous faut donc supprimer du modèle DTS toutes les transitions qui ne correspondent pas à des appels systèmes. C'est à dire, étant donné la définition de notre modèle, toutes les ϵ-transitions. Nous effectuons donc une ϵ-réduction de notre modèle afin que toutes les transitions considérées soient des appels systèmes.

L'ϵ-réduction n'est pas possible dans le cas général pour les modèles temporisés. Nous proposons ici une ϵ-réduction partielle où nous éliminons un certain nombre de transitions epsilon et de motifs ne pouvant pas être réduits, puis où nous réduisons les ϵ-transitions restantes en utilisant la structure spécifique à notre modèle. Nous présentons tout d'abord la manière dont les ϵ-transitions sont éliminées dans le cas général, grâce aux particularités de notre modèle, puis nous traitons deux cas particuliers pour lesquels il est nécessaire d'effectuer une modification avant de pouvoir retomber sur le cas général.

Réduction dans le cas général La structure des modèles de tâches obtenus à l'issue de la démarche IDM (5.2.1) ainsi que de la traduction des motifs d'exécution (5.2.2.3) nous permet d'affirmer que les ϵ-transitions contenant du temps sont, à l'exception des motifs

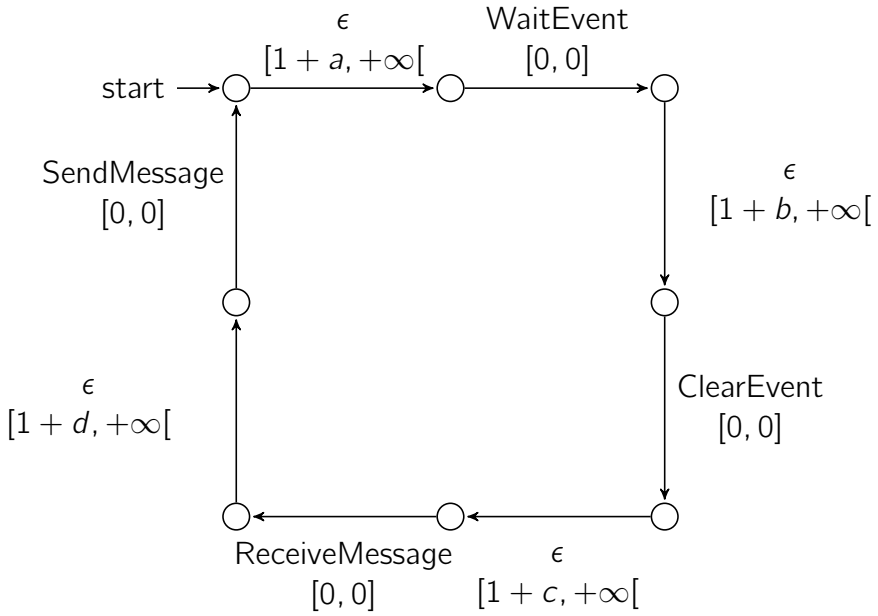


Figure 6.1 – pDTS initiale

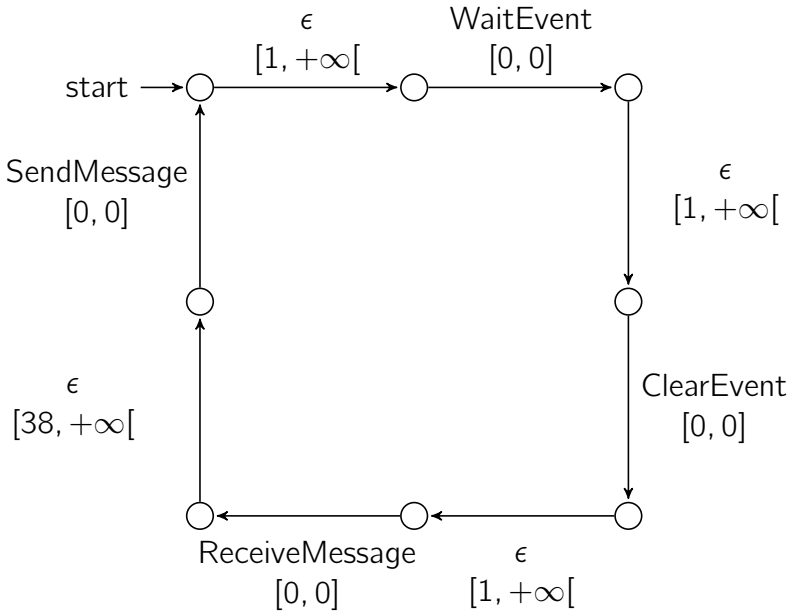


Figure 6.2 – DTS après injection

listés ci-dessous, toujours situées séquentiellement en amont d'une transition dont le tir est observable (un appel système). L'ϵ-réduction consiste donc simplement à aditionner le ou les temps d'exécution situés séquentiellement en amont des transitions d'appels système au temps d'exécution de la transition concernée, et à supprimer les ϵ-transitions restantes.

Ce modèle est déterministe en raison de la rêgle 1.

Motifs à éliminer Dans les modèles DTS peuvent se présenter des motifs de la forme présentée en figure 6.3a. Ces motifs ne peuvent pas être réduits de la manière décrite ci-dessus, puisque les ϵ-transitions ne sont pas séquentiellement en amont de transitions observables.

Afin de pouvoir réduire le modèle, lorsque une ϵ-transition avec du temps pointe sur un état avec deux ou plus transitions entrantes (tel que représenté en figure 6.3a), en amont d'une transition observable, nous ajoutons une place vers laquelle on fait pointer la transition ϵ, et dont la ou les transitions sortantes sont identiques à la place vers laquelle pointait initialement la transition ϵ. Ceci est visible en figure 6.3b.

ϵ-transitions pouvant être supprimées Il n'est possible d'observer que les appels système au cours de l'exécution. Toutes les séquences d'ϵ-transitions ne menant pas à une transition observable peuvent donc être supprimées. Nous supprimons donc du modèle toutes les ϵ-transitions pointant sur un état puit, jusqu'à temps qu'aucune ϵ-transition ne pointe sur un état puit.

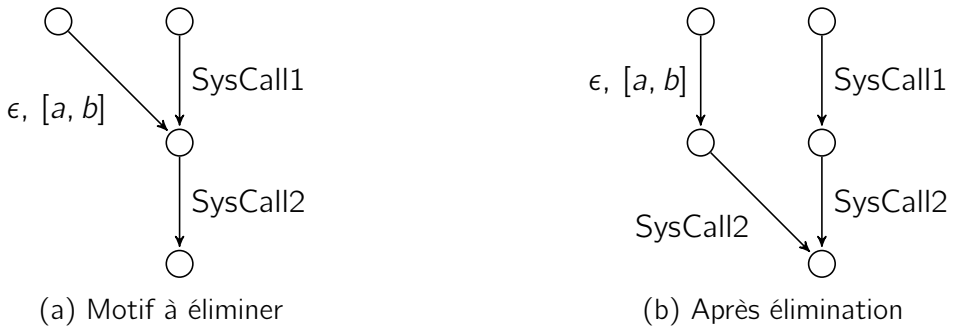
La figure 6.4 nous montre le résultat de l'ϵ-réduction de la DTS présentée en figure 6.2.

C'est ce modèle qui est ensuite intégré au contrôleur afin d'effectuer le forçage en ligne du système.

6.3 Principe de l'implémentation dans un RTOS

6.3.1 Principe général

L'objectif du forçage en ligne est d'assurer que l'exécution du système satisfasse les contraintes temporelles obtenues grâce aux étapes de l'analyse hors-ligne. Il est donc nécessaire pour ce faire d'ajouter des délais au cours de l'exécution qui, pour correspondre à ce qui a été considéré dans le modèle, doivent être ajoutés au cours de l'exécution de chaque tâche, comme si cette exécution prenait du temps supplémentaire.



(a) Motif à éliminer (b) Après élimination

Figure 6.3 – Motif particulier lors de l'ϵ-réduction partielle

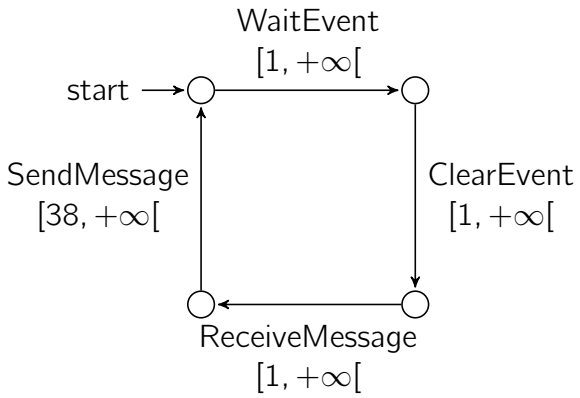


Figure 6.4 – ϵ-réduction d'une DTS

Le contrôle se déroule dans le cadre d'un système d'exploitation temps réel (Real-Time Operating System, RTOS). Chaque tâche est contrôlée séparément : le temps inclus dans les modèles les décrivant est bien un temps local, et non le temps global du système qui tiendrait compte des préemptions. C'est le temps d'exécution, et non le temps de réponse des tâches qui est considéré. Pour chaque tâche, nous incluons au contrôleur son modèle DTS dans lequel les valeurs de paramètres ont été injectées et qui ne contient que des transitions correspondant à des appels systèmes (grâce à l'étape d'ϵ-réduction).

Le contrôleur observe les appels systèmes émis par chaque tâche. Si le temps local à la tâche respecte la contrainte présente sur le modèle (l'intervalle dans lequel les valeurs de paramètres ont été injectées), alors l'appel système est exécuté immédiatement. Sinon, le contrôleur qui s'exécute dans le contexte de la tâche effectuant l'appel système garde la main sur le processeur jusqu'à ce que le temps local atteigne la valeur minimum définie dans le modèle. Le temps d'exécution ajouté aux tâches l'est en mode utilisateur, le mode d'exécution utilisé pour les tâches et non en mode superviseur ou noyau, le mode d'exécution sans restrictions d'accès utilisé par les routines du système d'exploitation. L'exécution reste ainsi normale du point de vue des autres tâches et du système d'exploitation temps réel. Il s'agit bien du temps local.

6.3.2 Architecture

6.3.2.1 Emplacement du contrôleur dans le système

Le contrôleur est placé à l'interface entre les tâches et le système d'exploitation temps réel. Son exécution se déroule lors des appels systèmes, mais alors que le passage en mode noyau n'a pas encore été fait. La figure 6.5 montre l'emplacement du contrôleur du point de vue du système d'exploitation. L'interception des appels systèmes se fait dans le wrapper (adapteur) d'appels systèmes du RTOS. Ce wrapper est une fonction en mode utilisateur appelée par les tâches, et qui contient l'appel système "réel" correspondant à l'interface binaire-programme (Application Binary Interface, ABI) du système d'exploitation considéré.

La figure 6.6 montre un diagramme d'interaction UML (Unified Modeling Language) qui déroule l'exécution du système lors de l'exécution de notre solution de contrôle. Cette figure montre deux facettes de la solution de contrôle :

- Le démarrage des tâches, en mode noyau, qui nécessite une intervention du contrôleur.
- Le contrôle d'un appel système quelconque , effectué en mode utilisateur.

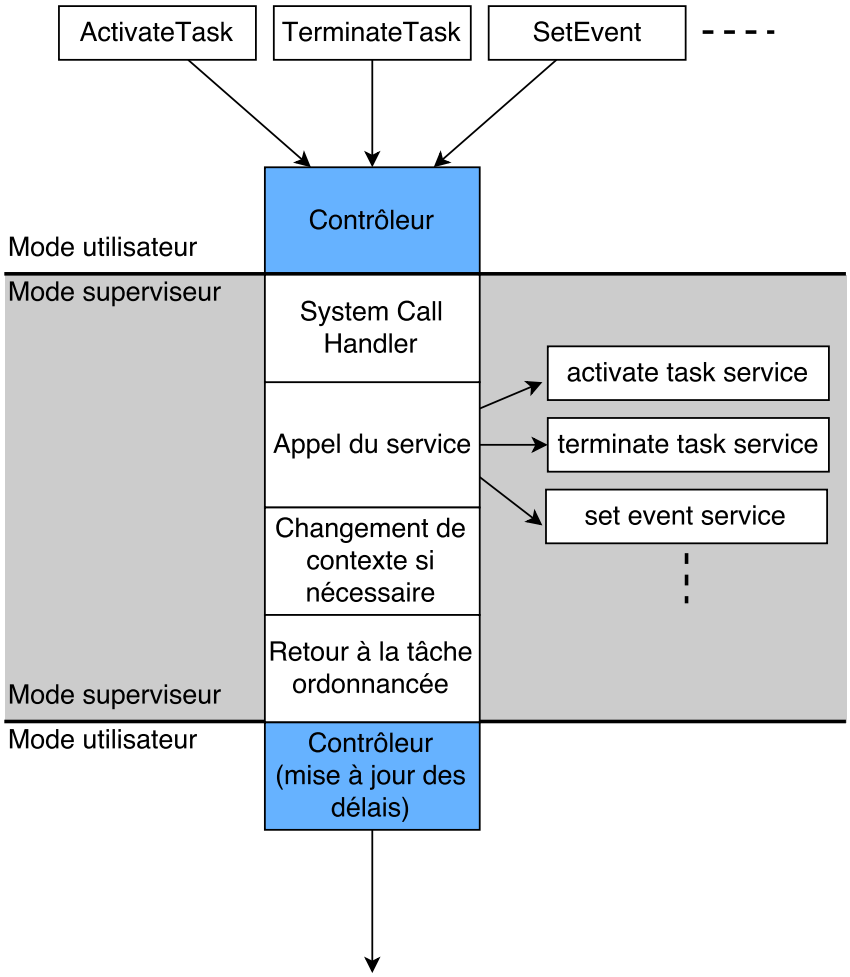


Figure 6.5 – Emplacement du contrôleur

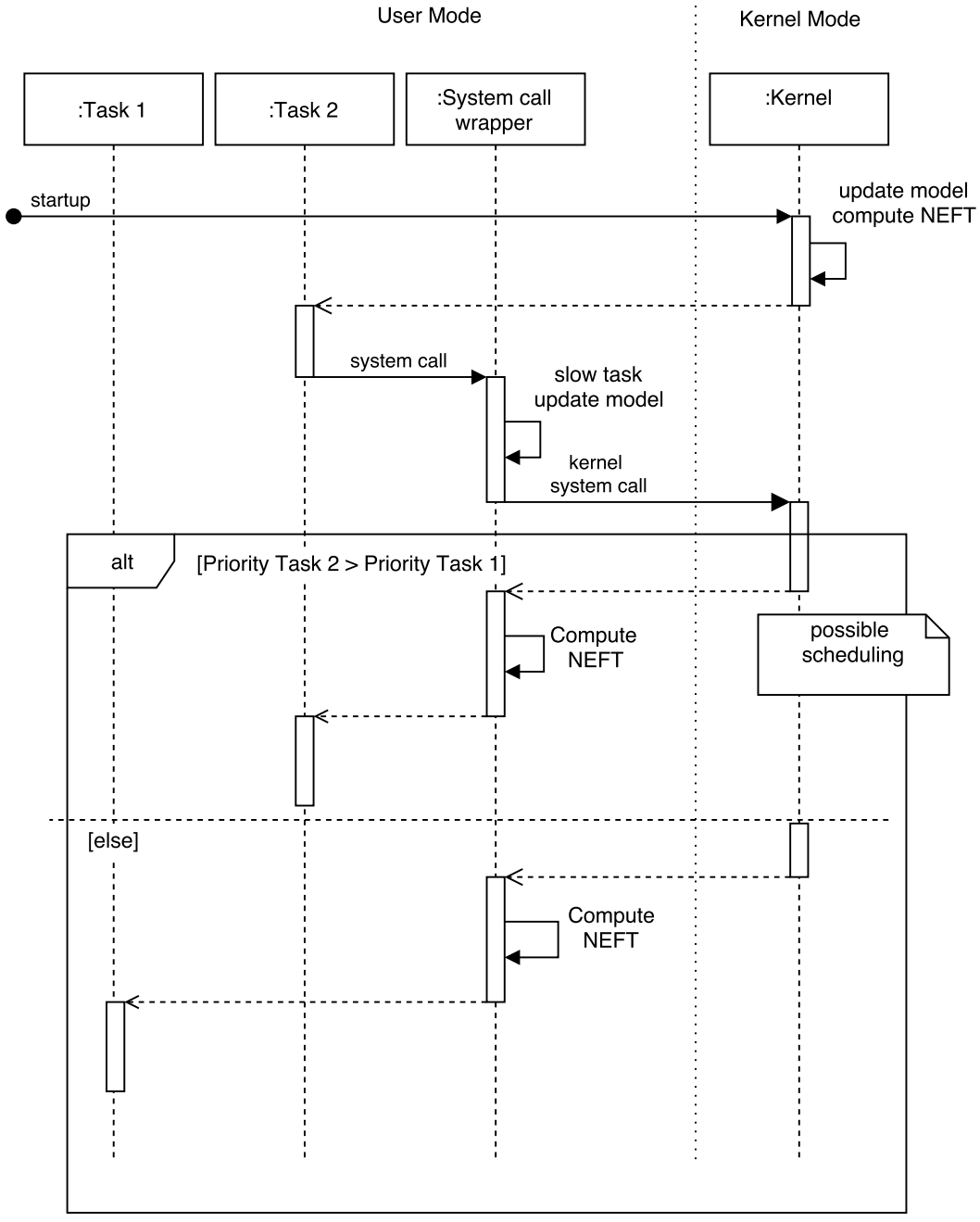


Figure 6.6 – Diagramme d'interaction de la solution de contrôle

```
typedef struct {
  int target; //Place cible, [0, MAX_NODES_NUM]
  int sc; //identifiant de l'appel systeme
  int lower_bound;  //Borne inferieure calculee hors-ligne
  int upper_bound;  //Borne superieure calculee hors-ligne
  time_t eft; //Date de tir minimale calculee en ligne
  time_t lft; //Date de tir maximale calculee en ligne
} transition;
```

Figure 6.7 – Structure représentant une transition

Lorsqu'une tâche appelle le wrapper d'appels systèmes, la procédure suivante, visible sur la figure 6.6 est exécutée :

1. Slow task : ralentissement de l'exécution. Cette étape est réalisée en utilisant la date de tir minimale calculée précédement (Next Earliest Firing Time, NEFT). Un délai est ajouté à l'exécution de la tâche jusqu'à ce que le NEFT soit atteint.

2. Update model : la mise à jour de l'état du modèle. L'appel système courant, ses paramètres et l'identifiant de la tâche en cours sont utilisés afin d'identifier la transition de la DTS à tirer dans le modèle.

3. Kernel system call : l'appel système noyau. Une fois les étapes ci-dessus effectuées, l'appel système est transféré au noyau où il peut s'exécuter normalement.

4. Compute NEFT : le prochain temps de tir pour les transitions qui peuvent être tirées depuis l'état actuel est calculé à l'aide du temps global du système au moment du retour de l'appel système. Cette étape est nécessaire, car l'appel système peut avoir donné lieu à une préemption et donc à un changement de contexte, comme montré sur la figure 6.6.

De plus, lors du démarrage des tâches, il est nécessaire de venir calculer le temps relatif à la date de ce démarrage pour les transitions qui peuvent être tirées depuis l'état initial.

Nous constatons donc que le temps utilisé par le contrôleur est bien le temps local à la tâche, le temps qui s'écoule lorsqu'elle est élue par l'ordonnanceur. Nous pouvons donc utiliser les modèles DTS locaux, qui ne contiennent pas d'information sur la préemption, afin d'effectuer ce forçage. Ceci est cohérent avec l'étape d'analyse où la préemption est modélisée à l'aide des arcs inhibiteurs temporisés ajoutés lors de la composition des modèles.

6.3.3 Structures de données

Pour permettre le forçage, il faut bien évidemment représenter en mémoire les modèles DTS des tâches du système.

Cette représentation est simplement basée sur des tableaux de structures listant les différents états de chaque DTS, et pour chaque état une structure représentant chaque transition sortante, ainsi que son intervalle, et l'état ciblé. En figure 6.7, on peut voir la déclaration de la structure représentant les transitions du système.

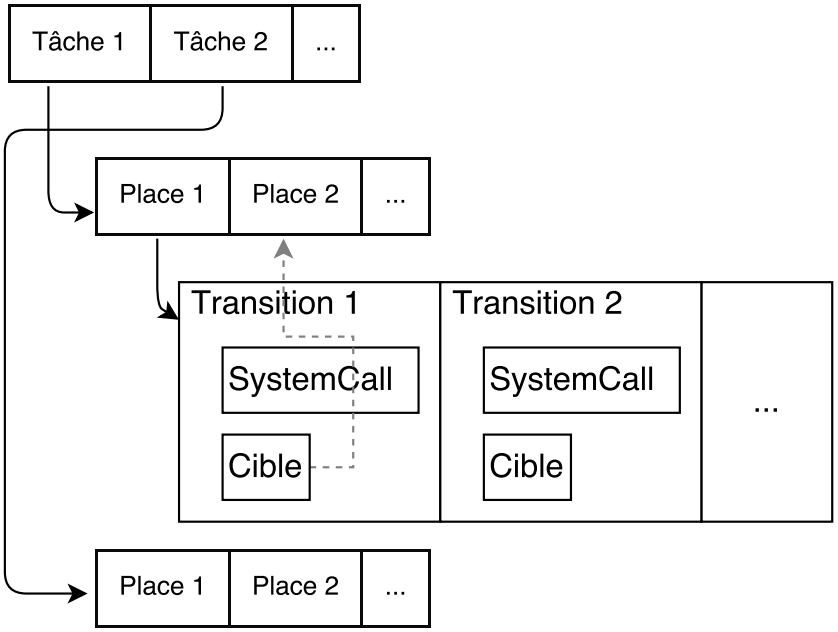


Figure 6.8 – Représentation en mémoire de la DTS d'une tâche de contrôle

La figure 6.8 montre la représentation en mémoire de la tâche présentée en section 5.2.2, figure 5.2a. Un tableau de tâches contient un tableau de places dont chaque élément contient un tableau de transitions correspondant à la structure de la figure 6.7.

L'évolution du modèle est suivie à l'aide d'un entier enregistrant pour chaque tâche l'identifiant de la place active. Les structures des différentes transitions sont mises à jour au fur et à mesure de l'exécution, suivant l'algorithme de contrôle.

6.3.4 Algorithme de contrôle

6.3.4.1 Définitions formelles

Contrôle : L'objectif de l'algorithme de contrôle est de transformer l'exécution du système pour qu'il se conforme à l'ensemble des DTS. Formellement :

Définition 6.3 (Contrôle du système)**.** Pour chaque transition r d'une DTS embarquée, nous définissons la fonction de date de tir minimale (Earliest Firing Time, eft). $eft : R \to \mathbb{Q}$ tel que $eft(r)$ est la borne inférieure de la transition r. À l'exécution, nous voulons implémenter la fonction $Control : Exec(q_0) \to Exec'(q_0)$ définie par :

$$Control(q_0 \xrightarrow{\sigma_0,d_0} q1 \xrightarrow{\sigma_1,d_1} q_2 \xrightarrow{\sigma_2,d_2} \ldots) = q_0 \xrightarrow{\sigma_0,d'_0} q_1 \xrightarrow{\sigma_1,d'_1} q_2 \xrightarrow{\sigma_2,d'_2} \ldots$$

tel que $\forall i, d'_i = \max(d_i, eft(r_i))$ avec $r_i = q_i \xrightarrow{\sigma_i,\iota_i} q_{i+1}$.

Ceci représente l'étape de contrôle effectuée à l'exécution : si un évènement est observé avant la date calculée hors-ligne, l'exécution est ralentie jusqu'à ce que le temps atteigne ce point. Cette définition est correcte vis-à-vis de la DTS, bien que l'implémentation soit effectivement incrémentale : lorsqu'un appel système est observé, des délais sont ajoutés si

nécessaire.

Calcul de la date de tir : Pour pouvoir ralentir les tâches de manière correcte, dans ce contexte mono-processeur, nous devons traduire le temps local exprimé dans la DTS en un temps global, le temps processeur, suivant l'exécution.

Cette opération doit avoir lieu après chaque appel système : si une préemption à eu lieu, une discontinuité temporelle est présente du point de vue du temps local de la tâche. Considérons par exemple l'appel système *WaitEvent*, qui correspond à l'attente d'un évènement. Lorsqu'une tâche l'utilise, l'ordonnanceur est appelé et peut rendre le processeur à une autre tâche durant l'attente de l'évènement par la tâche initiale. Cette possibilité est représentée en figure 6.6.

Cette opération doit aussi avoir lieu pour chaque transition sortante de l'état dans lequel le tir de la transition correspondant à l'appel système courant nous amène.

Cette traduction entre temps local et temps global se fait dans la fonction NEFT, pour Next Earliest Firing Time (Prochaine date de tir minimale). Formellement :

Définition 6.4 (Fonction NEFT)**.** Pour tout $r = q \xrightarrow{\sigma,\nu} q'$, nous définissons $Succ(r)$, l'ensemble des $r' \in R$ tels que $r' = q' \xrightarrow{\sigma',\nu'} q$".
Nous définissons la fonction $NEFT$ sur une transition r après un run π, qui calcule l'ensemble des dates de tir minimales associées : $NEFT(r) = \{Time(\pi) + eft(r') \mid r' \in Succ(r)\}$.

6.3.4.2 Fonctions implémentées

L'algorithme de contrôle est implémenté en plusieurs fonctions, qui s'exécutent avant, puis après l'appel système (en mode utilisateur), ainsi qu'au démarrage de chaque tâche. La figure 6.6 montre les emplacements des différents appels aux fonctions qui constituent la solution de contrôle, pour une exécution nominale.

Les fonctions qui constituent cette solution de contrôle sont les suivantes :

- compute_NEFT : formalisée ci-dessus, elle permet le calcul des nouvelles dates de tirs ;
- slow_task : cette fonction permet d'effectuer l'étape de ralentissement de l'exécution, et est apellée avant l'appel système. Elle effectue aussi la mise à jour du modèle.

Les actions du contrôleur lors de l'exécution d'un appel système par une tâche sont décrites dans l'algorithme 1.

La tâche est ralentie (ce qui inclut la mise à jour du modèle, comme visible en algorithme 2), puis l'appel système est exécuté par le noyau. Au retour de l'appel système, les nouvelles dates de tir sont calculées, grâce à la fonction visible en algorithme 3.

Une dernière nécessité pour le bon fonctionnement du contrôleur est la mise à jour du modèle et le calcul des nouvelles dates de tir lors du démarrage d'une tâche. Cette exécution est présentée en algorithme 4.

Algorithm 1 Behavior of the system call wrapper

```
1 : slow_task(identifier)
2 : syscall
3 : compute_NEFT(identifier)
```

Algorithm 2 Slowing a task

```
1 : function slow_task(identifier)
2 :     outgoing_transition ← get_outgoing(identifier)
3 :     while current_local_time < eft(outgoing_transition) do
4 :     end while
5 :     state ← target(outgoing_transition)
6 : end function
```

Algorithm 3 compute NEFT

```
1 : function Compute_NEFT(identifier)
2 :     for all outgoing_transitions of current node do
3 :         set_eft(current_local_time + eft(outgoing_transition))
4 :     end for
5 : end function
```

Algorithm 4 When starting a task

```
1 : outgoing_transition ← get_outgoing(identifier)
2 : state ← target(outgoing_transition)
3 : compute_NEFT(identifier)
```

6.3.5 Précision de l'implémentation

Étant donné le code qui doit être exécuté afin de réaliser le forçage d'un délai sur l'exécution au moment d'un appel système, l'implémentation n'est pas parfaite et il existe un décalage temporel entre l'instant calculé lors de l'analyse du modèle pour le tir d'un appel système et l'instant où l'appel système a effectivement lieu.

Le graphe de la figure 6.9 montre de manière symbolique le décalage entre la date de tir calculée hors ligne et la date de tir effective d'une transition, les dates étant relatives à la date d'observation de l'appel système. Le graphe n'est pas à l'échelle et ne représente pas la totalité des comportements observés sur le système en ce qui concerne les délais supplémentaires.

Lors de l'occurence d'un appel système, il est systématiquement nécessaire de mettre à jour le modèle. Le delta 1 correspond donc à ce coût qui est payé pour tous les appels systèmes, même ceux qui ne doivent pas être contrôlés. Si la date de tir souhaitée est supérieure à ce premier delta, relativement à la date d'observation de l'appel système, il existe néanmoins un délai supplémentaire lié à la précision temporelle permise par l'implémentation : la date actuelle n'est vérifiée qu'une seule fois par tour de boucle, ajoutant dans le pire cas la durée totale de la boucle. Nous notons ce délai supplémentaire delta 2.

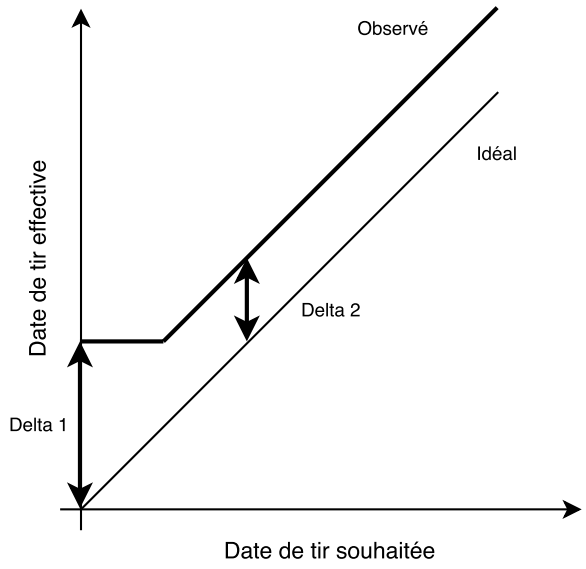


Figure 6.9 – Décalage temporel entre la date de tir calculée et celle obtenue

Comme proposé dans la section 6.1.2.1, il est possible d'ajouter un Δ au polyèdre sur les valeurs des paramètres, avant la synthèse des paramètres. Cependant, on considère les valeurs de ces variations comme négligeables au regard de l'échelle de temps du système. En effet, Δ_2 ne correspond qu'au temps d'exécution des quelques instructions correspondant à l'exécution d'une simple boucle. Nous considérons que la base de temps processeur d'un système temps-réel doit être d'un ou plusieurs ordres de grandeurs plus faible que celle des systèmes concernés.

Troisième partie

Mise en œuvre

Chapitre 7

Implémentation du forçage en ligne dans Trampoline RTOS

Nous proposons dans ce chapitre une vue d'ensemble de l'implémentation de notre méthode dans le système d'exploitation temps réel Trampoline RTOS.

Sommaire

7.1 Trampoline RTOS

7.1.1 AUTOSAR

7.1.1.1 Historique

AUTOSAR (Pour AUtomotive Open System Architecture, architecture ouverte pour les systèmes électroniques du secteur automobile) est une norme ayant pour but d'établir un

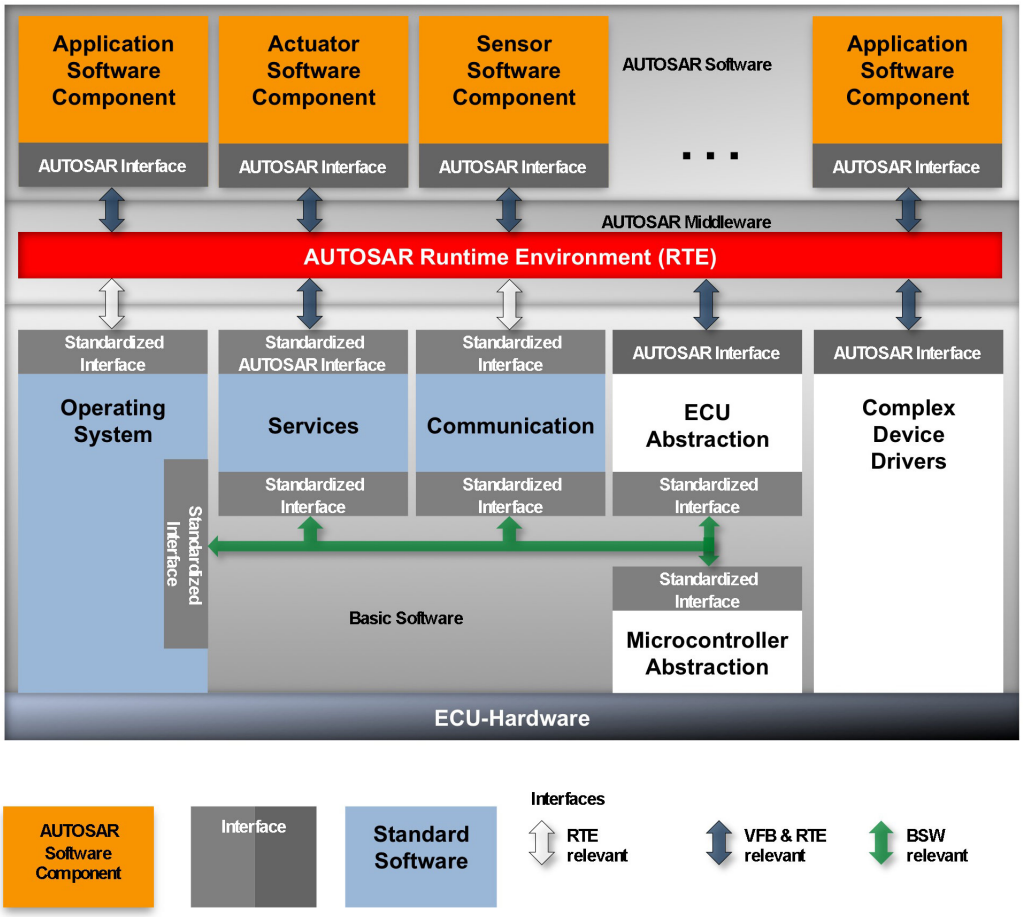


Figure 7.1 – Architecture logicielle d'AUTOSAR

standard ouvert pour les architectures logicielles de l'industrie automobile, permettant la communication entre les constructeurs et leurs sous-traitantsi (Fürst, Mössinger, Bunzel, Weber, Kirschke-Biller, Heitkämper, Kinkelin, Nishikawa & Lange, 2009). Il s'agit d'un ensemble de spécifications décrivant des composants logiciels et définissant leurs interfaces afin de permettre une interopérabilité entre - et une réutilisabilité des - implémentations des différentes fonctionnalités, ainsi que de permettre l'abstraction des considérations matérielles.

La figure 7.1 présente l'architecture logicielle d'AUTOSAR. On peut remarquer la brique logicielle intitulée "Operating System", système d'exploitation. La norme OSEK/VDX[1], anciennement le standard pour les systèmes d'exploitation temps réel automobiles a été intégrée à AUTOSAR et étendue sous ce label OS.

7.1.1.2 Modèle de programmation

La partie OS de AUTOSAR est conçue spécifiquement pour le domaine d'application automobile. L'architecture est donc simple, et basée sur un nombre restreint de concepts. Le système d'exploitation répond à un certain nombre d'exigences, parmi lesquelles :

[1]OSEK/VDX : Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive

- Configuration statique : Tous les objets de l'application sont statiques, et la configuration est effectuée hors ligne. Les objets (tâches, ressources...) sont créés avant le démarrage de l'application et ne sont jamais détruits, il n'y a pas d'allocation dynamique.
- Configurabilité : il est possible de n'embarquer que les parties de l'OS qui sont utilisées par l'application.
- Comportement prévisible : une contrainte nécessaire pour les applications temps réel.

Tâches. Il existe deux types de tâches dans OSEK : les tâches basiques dont l'exécution ne peut pas être interrompue par un appel système, et les tâches étendues. L'état des tâches étendues peut être RUNNING (la tâche s'exécute), SUSPENDED (la tâche est inactive) et READY (la tâche a été activée et attend le processeur), mais aussi (et à la différence des tâches basiques) WAITING (la tâche est en attente d'un évènement).

Ordonnancement. L'algorithme d'ordonnancement des tâches est à priorités fixes. Il peut être préemptif ou coopératif, selon la configuration des tâches.

Synchronisation entre les tâches. Les tâches communiquent à l'aide d'un système d'évènements. Une tâche peut attendre un évènement à l'aide du service WaitEvent(), et une autre tâche ou une interruption peut le signaler à l'aide du service SetEvent().

Ressources. La gestion des ressources partagés est facilitée par l'utilisation d'un ensemble de services permettant aux tâches d'accéder aux ressources en exclusion mutuelle (GetRessource(), ReleaseRessource()).

Communication. La figure 7.1 montre l'existence d'une couche de communication, centrée sur l'envoi de messages, qui concerne les communications externes (bus CAN ou autre). Pour les communications internes, OSEK/VDX et AUTOSAR OS proposent la notion d'IOC (Inter Operation system application Communication). Cette couche de communication est elle aussi implémentée par les RTOS conformes à ces normes.

Développement. La figure 7.2 présente la manière dont une application OSEK/VDX est implémentée et la génération du binaire associé. Le langage OIL (OSEK Implementation Language) introduit par OSEK/VDX ou le langage ARXML (AutosaR XML) introduit par AUTOSAR OS, et leurs compilateurs associés sont utilisés pour décrire les objets de l'application, et générer les structures de données associées. L'application est implémentée en C puis compilée afin de pouvoir lier les différents fichiers objets issus de la compilation de l'OS, des structures de données et de l'application elle-même en un fichier binaire exécutable sur la cible.

7.1.2 Implémentation dans trampoline RTOS

7.1.2.1 Caractéristiques

Trampoline RTOS est un système d'exploitation temps réel statique développé à l'IRCCyN, par l'équipe systèmes temps réel. Il propose une implémentation des parties OS et COM

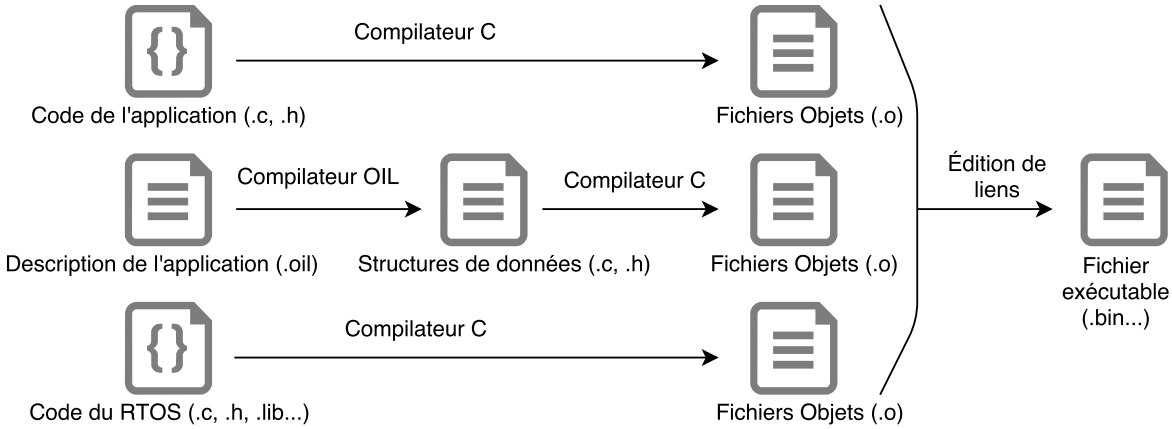


Figure 7.2 – Génération et compilation d'une application OSEK

(communication) de la norme AUTOSAR, et est proposé sous une license GPLv2 dans un processus de développement ouvert.

Ce système d'exploitation supporte un certain nombre de plateformes d'exécutions, parmis lesquelles :

- un ensemble de plateformes basées sur l'architecture ARM (plus spécifiquement la famille des microprocesseurs CORTEX, version 32 bits d'ARM) ;
- l'architecture PowerPC ;
- l'architecture AVR ;
- la plateforme POSIX, par le biais de VIPER (VIrtual Processor EmulatoR), un programme permettant l'émulation des interruptions à l'aide des signaux Unix.

Conformément à la norme AUTOSAR OS, Trampoline RTOS est un système d'exploitation statique, et est donc entièrement configuré hors-ligne. Il propose donc une implémentation de OIL et ARXML avec GOIL, le compilateur OIL/ARXML développé dans le projet.

7.1.2.2 Objectifs de développement

Le développement de Trampoline vise en premier lieu à proposer une implémentation libre de OSEK/VDX (puis AUTOSAR OS). Un objectif supplémentaire est la portabilité, comme on peut le constater par le nombre de plateformes supportées. Trampoline est d'autre part peu gourmand en termes de mémoire, ce qui est intéressant pour les plateformes à fortes contraintes et limitations de ce point de vue.

7.2 Plateforme d'exécution

Dans ce travail de thèse, nous proposons l'implémentation de notre solution pour deux plateformes d'exécution : POSIX (soit UBUNTU Linux) à des fins de prototypage et ARM CORTEX M4 (avec la carte STM32F4discovery). Malgré l'aspect multi-plateforme de Trampoline, il est nécessaire d'effectuer un portage de notre solution afin de supporter plusieurs plateformes. En effet, le wrapper d'appels systèmes est différent suivant la plateforme d'exécution. Le wrapper d'appels systèmes de Trampoline est implémenté en C sur la plateforme POSIX, mais en assembleur sur les plateformes embarqués, afin de pouvoir

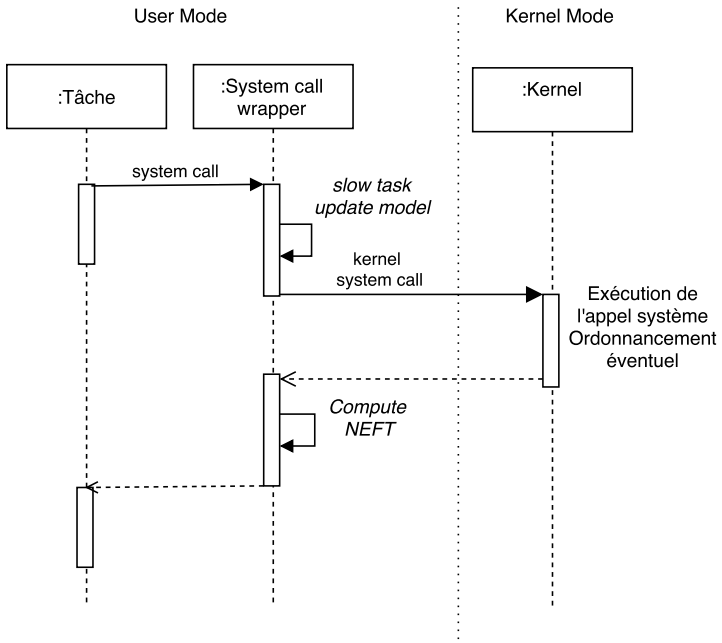


Figure 7.3 – Séquence d'appel système sur Trampoline RTOS

effectuer le passage du mode utilisateur au mode superviseur utilisé lors de l'exécution du noyau.

La figure 7.3 présente la manière dont un appel système Trampoline se déroule. Le code spécifique à chaque plateforme est situé dans le wrapper d'appels systèmes, afin d'appeler correctement les fonctions nécessaires à l'exécution du forçage en ligne, qui sont elles indépendantes de la plateforme d'exécution.

Comme évoqué ci-dessus, l'implémentation sous POSIX de notre solution n'a pas pour but de la tester en conditions réelles. En effet, l'utilisation de POSIX ne permet pas de garantir les temps d'exécution desquels notre modèle et implémentation sont fortement dépendants. Cette implémentation est utilisée pour prototyper le code et tester notre solution, de manière à faciliter le développement.

Dans le reste de ce travail de thèse, nous nous concentrons donc sur l'implémentation sur cible ARM, sur laquelle nous pouvons garantir des temps d'exécution fiables et ainsi tester notre solution.

7.3 Implémentation dans trampoline RTOS

L'implémentation de notre solution dans trampoline RTOS nécessite la modification de la séquence de démarrage des tâches, ainsi que la modification du wrapper d'appels systèmes. Il nous faut aussi embarquer le modèle du système afin qu'il soit accessible par le contrôleur.

7.3.1 Contrôleur

Le code du contrôleur consiste en un fichier .c contenant les différentes fonctions de mise à jour du modèle, calcul des délais et contrôle de l'application. Il peut être inclut à Trampoline si une directive dans le fichier de configuration .oil demande l'utilisation de ce module dans le noyau.

Le contrôle est réalisé sur la base du temps processeur, obtenu à l'aide d'une horloge non synchronisée (free-running timer, see (AUT, 2014)).

7.3.2 Modèle du système

Comme présenté en section 6.3.3, le modèle du système est représenté à l'aide d'un tableau imbriqué de structures représentant l'ensemble des transitions entre les différents états de toutes les tâches du système. Les DTS sont implémentés en C dans des fichiers .c situés dans le répertoire de développement de l'application.

Dans l'état actuel du travail, ils doivent être écrits à la main une fois le processus de génération aboutit. Un générateur pourrait aisément produire ce fichier à partir du modèle, dans une démarche d'extraction de modèle.

Le fichier est inclus par le contrôleur si la compilation de celui-ci est demandé par l'application, grâce à un attribut dans le fichier .oil.

7.3.3 Démarrage des tâches

Le démarrage des tâches nécessite une mise à jour du modèle, afin de tirer la transition de démarrage et de mettre à jour les différents temps de tir des prochaines transitions du modèle. Cette partie de l'implémentation n'est pas spécifique à une plateforme, puisqu'elle est implémentée dans la partie C de trampoline, au dessus de la couche d'abstraction matérielle.

La figure 7.4 montre les quelques lignes de code qui sont exécutées au démarrage d'une tâche trampoline si l'option de contrôle a été sélectionnée.

7.3.4 Wrapper d'appels systèmes

L'inclusion du code dans le wrapper d'appels systèmes, comme précisé en section 7.2, nécessite la modification de la partie spécifique à la cible de Trampoline. On inclut dans la fonction assembleur de chaque invoquation d'appel système quelques lignes d'assembleurs nous permettant d'appeler les fonctions nécessaires au forçage.

Ces wrapper d'appels systèmes étant générés grâce à un langage de template, cette modification est générique à tous les appels systèmes. En effet, suivant les appels systèmes utilisés par une application, GOIL utilise les templates pour produire le code assembleur nécessaire, incluant notre modification quel que soit l'appel système.

La figure 7.5 présente le code assembleur correspondant au wrapper d'appel système généré pour ActivateTask sur la cible Cortex M4.

7.4 Résultats

L'implémentation sur POSIX et Cortex M4 est fonctionelle. Nous proposons dans le chapitre 8 un exemple démontrant le fonctionnement de notre solution.

Il est cependant nécessaire d'évaluer les performances de notre solution en termes d'implémentation. Traditionellement, cela signifie mesurer l'overhead (surcoût temporel et en mémoire) de l'implémentation, afin de confirmer le peu d'impact sur l'exécution normale du système.

```
/**
 * Start the highest priority READY process
 */
FUNC(void, OS_CODE) tpl_start(CORE_ID_OR_VOID(core_id))
{
  [...]

  if (TPL_KERN_REF(kern).elected->state == READY_AND_NEW)
  {
    [...]

    #if WITH_CONTROL == YES
    if(TPL_KERN_REF(kern).elected_id != INVALID_TASK)
    {
      update_model(get_outgoing_transition(
        (void*) &proc.id,
        NULL,
        NULL,
        START_TASK));
      compute_NEFT(NULL, NULL, NULL, START_TASK);
    }
    #endif
  }
}
```

Figure 7.4 – Démarrage d'une tâche contrôlée

```
ActivateTask:
  [...]
  /* Runtime enforcement: save r0 thru r3
     (parameters + syscall id) */
  sub sp, #4 /* save space for r0 later on */
  push {r0-r3,lr}
  bl slow_task /* slow the task and update the model */
  ldmfd sp, {r0-r3,lr} /* restore registers for the next call,
                          but keep them in the stack */
  svc #OSServiceId_ActivateTask
  str r0, [sp,#20] /* save the return value in the stack*/
  pop {r0-r3} /* restore the parameters + syscall id */
  bl compute_NEFT /* compute the NEFT */
  pop {lr}
  pop {r0} /* restore r0, the return value of the syscall */
```

Figure 7.5 – Wrapper d'appel système outillé pour WaitEvent

Cependant, dans cette situation, l'objectif de notre solution est d'effectivement ajouter un temps d'exécution. Ce temps d'exécution néanmmoins ne doit pas dépasser les valeurs prescrites par le modèle. Certaines transitions, de plus, n'ont pas à subir cet ajoût de temps d'exécution, soit parce qu'elle ne sont pas contrôlées, soit parce que le temps système atteindra de toute manière la valeur minimum du temps d'exécution.

Afin de pouvoir qualifier notre solution, il est donc nécessaire de tester le cas où aucun temps d'exécution ne doit être ajouté, et de mesurer le délai (overhead) ajouté par l'exécution du contrôleur sur l'exécution de ces appels systèmes.

7.4.1 Overhead

Afin de tester l'overhead de notre solution, nous avons réalisé une application simple, effectuant un unique appel système.

Elle est composée de deux tâches, l'une de priorité 2, et l'autre de priorité 1. La tâche de priorité 2 invoque l'appel ActivateTask, puis se termine. La tâche de priorité 1 ne fait que se terminer.

Afin de tester l'overhead des différents composants, nous allumons puis éteignons une LED disponible sur la carte stm32f4discovery à différents emplacements au cours de l'exécution des tâches, et mesurons à l'aide d'un oscilloscope la durée de l'impulsion. Nous comparons ensuite le temps d'exécution pour la même application compilée avec et sans notre solution de contrôle.

Le modèle, pour la version contrôlée, ne contient pas de contraintes temporelles (c.à.d que tous les intervalles sont de la forme $[0, +\infty[$).

7.4.1.1 Appel système

Afin de tester l'overhead sur l'appel système, nous allumons la LED au démarrage de la tâche de priorité maximale, puis l'éteignons une fois l'appel d'ActivateTask effectué. À l'aide d'un oscilloscope, nous mesurons le temps durant lequel la LED est allumée.

La tâche activant étant de priorité supérieure, nous garantissons ainsi que le temps d'exécution mesuré est bien celui de l'appel système uniquement.

Les résultats avec et sans forçage sont disponibles en figure 7.6. On observe que le temps ajouté par l'exécution de notre solution est de $6\mu s$, soit un temps d'exécution de l'appel système $1.9\times$ plus important que pour un appel système non contrôlé.

C'est une augmentation conséquente, mais l'ordre de grandeur reste raisonnable par rapport à l'échelle de temps du système. Ceci est d'autant plus visible si l'on compare au temps d'exécution (même non contrôlé) du démarrage d'une tâche avec ActivateTask.

7.4.1.2 Démarrage de la tâche

Pour pouvoir tester le temps lié au démarrage d'une tâche, nous inversons les priorités. La tâche de priorité inférieure démarre la tâche de priorité supérieure. Nous allumons la LED avant de terminer l'exécution de la tâche de priorité inférieure, puis nous l'éteignons au démarrage de la tâche de priorité supérieure.

Étant donné les priorités des tâches, nous mesurons ainsi le temps pris par l'appel système ActivateTask additioné au temps pris pour le démarrage d'une tâche.

Les résultats avec et sans forçage sont disponibles en figure 7.7. On observe que le temps ajouté par l'exécution de notre solution est de $7.5\mu s$, soit un temps d'exécution de

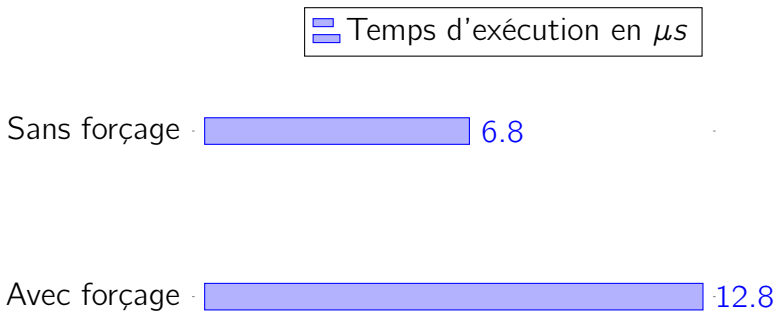


Figure 7.6 – Overhead de notre solution sur un appel système (ActivateTask)

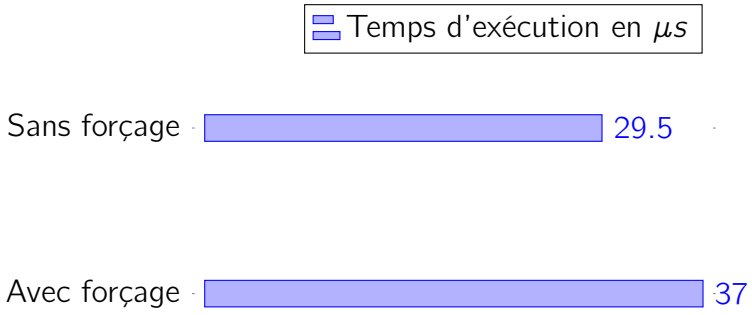


Figure 7.7 – Overhead de notre solution sur le démarrage d'une tâche.

l'appel système et du démarrage de la tâche suivante 1.3× plus important que pour un appel système et un démarrage de tâche non contrôlés. On remarque que le temps d'exécution aditionnel est proche de celui ajouté à l'exécution d'un appel système seul (6μs contre 7.5 μs). Ceci s'explique par le fait que les opérations effectuées sont sensiblement identiques : mise à jour du modèle et ralentissement (ici de zéro secondes) avant l'exécution de l'appel système, puis calcul de la date de tir des prochaines transitions après l'appel système ou le démarrage de la tâche. La différence provient du fait que le démarrage de la tâche doit de plus mettre à jour le modèle de la tâche concernée, afin de tirer la transition correspondant au démarrage de la tâche.

Néanmoins, étant donné les temps d'exécution plus conséquents des démarrages de tâches, l'impact sur le système reste faible.

Chapitre 8

Application à un exemple

Nous proposons dans ce chapitre d'appliquer les différents résultats à un exemple afin de présenter les résultats sur une application réelle.

Sommaire

8.1 Exemple

8.1.1 Protection temporelle AUTOSAR OS

Afin de pouvoir présenter la démarche dans sa totalité, nous proposons dans ce chapitre une étude de cas. Le système à tester est le mécanisme de protection temporelle du standard AUTOSAR OS, implémenté dans Trampoline RTOS (Bertrand, Faucou & Trinquet, 2009).

8.1.1.1 Fonctionnement

Ce mécanisme permet de détecter et contrôler les défaillances temporelles du système, c'est à dire les cas où le temps de réponse excède l'échéance fixée pour une tâche. Pour ce faire, la norme AUTOSAR OS propose un mécanisme de détection de fautes qui est composé des éléments suivants :

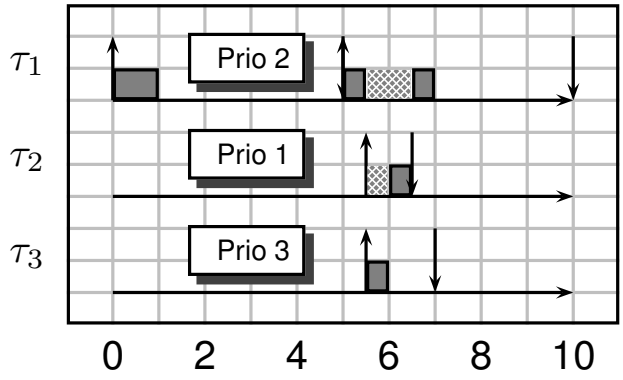


Figure 8.1 – Comportement du système à tester

- Contrôle de la date d'inter-arrivée pour une tâche, c'est à dire la durée entre deux activations d'une même tâche.
- Contrôle du temps d'exécution d'une instance d'une tâche.
- Contrôle de la durée d'une section critique entrainant le blocage d'autres tâches, qu'elle soit protégée par une ressource spécifique ou en bloquant les interruptions.

Ce mécanisme de protection temporelle est configuré pour chaque tâche dans sa decription OIL. Au cours de l'exécution, un ensemble de chiens de garde (watchdogs) sont utilisés par le noyau afin de restreindre le comportement des tâches. Si l'interruption correspondant à l'expiration d'un watchdog est levée, c'est que la tâche dépasse les contraintes imposées. Le système d'exploitation peut choisir de terminer ou réinitialiser la tâche, voir de cesser l'exécution du système.

8.1.1.2 Validation

L'implémentation de la protection temporelle AUTOSAR dans Trampoline RTOS nécessite d'être testée afin de garantir son fonctionnement correct. Pour ce faire, les concepteurs de Trampoline proposent un ensemble de cas de tests visant à couvrir les différentes caractéristiques du mécanisme de protection temporelle.

Le cas de test sur lequel ce cas d'étude se concentre, tiré de la suite de tests de Trampoline RTOS, est composé de trois tâches. La première tâche (tâche 1), de priorité 2 est démarée periodiquement, avec une période de 5 unités de temps. Elle est associée à une alarme, et se déclenche deux fois.

Cette tâche 1 déclenche les deux autres tâches. La première (tâche 2), de priorité inférieure (1) et la seconde (tâche 3), de priorité supérieure (3), sont déclenchées successivement durant la deuxième activation de la tâche 1. La figure 8.1 présente le système ainsi que son fonctionnement nominal. On observe l'attente de la tâche 2 lors de l'exécution de la tâche 3, puisque la priorité de la tâche 2 est plus importante.

Pour ce cas d'étude, l'échéance de chaque tâche est égale à sa période. Chaque tâche sera donc relancée à l'instant où elle doit se terminer.

8.1.2 Objectif de test

Le cas de test présenté dans la section précédente a pour but de venir tester la protection temporelle, dans le cas où le temps de réponse de l'une des tâches excède son échéance.

La manière dont ce test est implémenté dans la suite de tests de Trampoline RTOS est une simple boucle infinie (while(1)), forçant l'exécution d'une des tâches jusqu'à ce qu'elle soit supprimée par la protection temporelle.

Nous proposons ici d'utiliser la solution développée dans ce travail de thèse afin d'effectuer ce test. Les avantages sont multiples :

- L'implémentation ne nécessite pas de modifier l'application afin d'inclure la faute conduisant à l'erreur (mais, grâce à la protection temporelle, pas de défaillance, si elle fonctionne correctement). Notre solution permet l'injection de la faute sans modifier le code source de l'application.
- D'autre part, nous pouvons choisir n'importe lequel des polyèdre parmi ceux proposés dans l'union de polyèdres résultant de l'analyse (et par conséquent les valeurs de paramètres). Chacun des polyèdres de l'union correspondant à un scénario d'exécution différent, nous pouvons ainsi choisir l'emplacement de la faute. En d'autres termes, nous pouvons choisir la tâche sur laquelle la faute aura lieu, et ce en modifiant uniquement le modèle embarqué par le système.
- La finesse du test, enfin, est bien plus grande, l'amplitude du défaut provoqué bien plus faible : Le délai ajouté sera le délai minimal nécessaire à l'activation du mécanisme de protection temporelle. Contrairement à la solution par boucles infinies, cela permet de venir tester le degré de précision du déclenchement du mécanisme de protection temporelle.

L'objectif de test est donc d'atteindre dans le modèle l'état où la protection temporelle se déclenche.

8.2 Expérimentation

8.2.1 Modélisation du système

8.2.1.1 Modèle des tâches du système

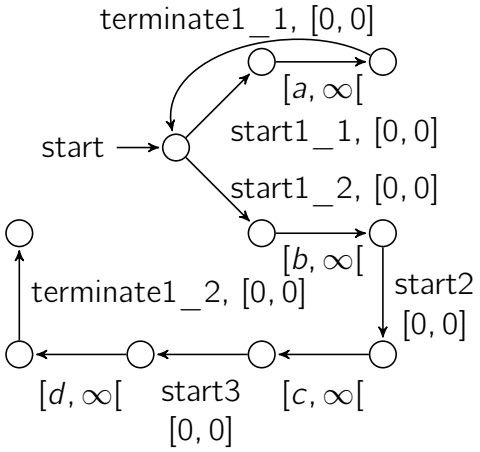
Le système présenté est modélisé à l'aide des DTS, à l'issu d'une démarche IDM. La figure 8.2 présente le modèle des différentes tâches du système.

8.2.1.2 Traduction et composition du modèle du système

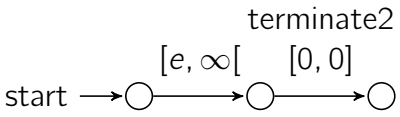
Afin de pouvoir effectuer l'analyse, nous traduisons le modèle du système en suivant les rêgles établies dans ces travaux de thèse, afin d'obtenir le modèle complet du système à tester, sous la forme d'un réseau de Petri temporel paramétré à arcs inhibiteurs temporisés. Ce modèle est représenté en figure 8.3.

Pour favoriser la concision du modèle, nous utilisons ici des arcs de vidange, en gris sur la figure. Ces arcs vident les places sources lorsque la transition de destination est tirée. L'utilisation de ces arcs ne change pas l'expressivité du modèle. (Dufourd, Finkel & Schnoebelen, 1998)

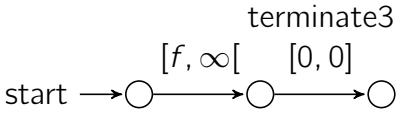
Ils nous permettent dans cette exemple de représenter le changement d'exécution provoqué par le déclenchement du ErrorHook lorsqu'un budget temporel est écoulé.



(a) pDTS de la tâche $T1$



(b) pDTS de la tâche $T2$



(c) pDTS de la tâche $T3$

Figure 8.2 – Modèles pDTS des tâches de l'application

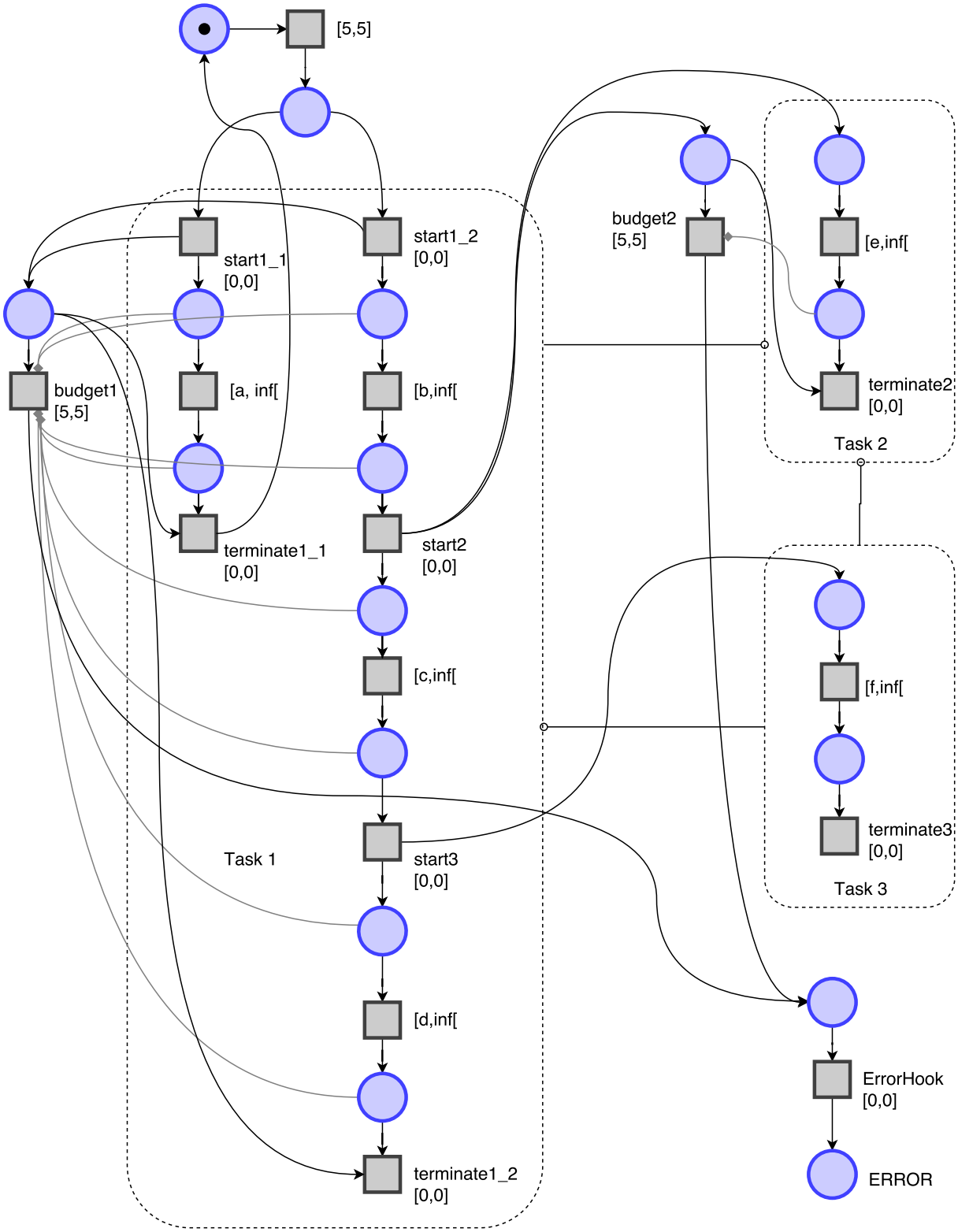


Figure 8.3 – Modèle complet de l'application à tester

$$D = \begin{cases} a > 5 \\ b > 0 \\ c > 0 \\ d > 0 \\ e > 0 \\ f > 0 \end{cases} \cup \begin{cases} a > 0 \\ b > 0 \\ c > 0 \\ d > 0 \\ e > 0 \\ f > 0 \\ b + c + d > 5 \end{cases} \cup \begin{cases} a > 0 \\ b > 0 \\ c > 0 \\ d > 0 \\ e > 5 \\ f > 0 \end{cases}$$

Figure 8.4 – Domaine des paramètres après analyse

8.2.2 Analyse hors ligne du système et obtention de la stratégie de test

8.2.2.1 Formalisation de l'objectif de test, analyse et choix du point

L'objectif de test est d'atteindre l'état correspondant à l'erreur. Il s'agit donc de la place étiquettée ERROR sur la figure. Formellement :

$$\Phi = AF[0, +\infty](M(ERROR) > 0)$$

Après l'exécution de la synthèse de paramètres à l'aide de Roméo, nous obtenons le domaine D présenté en figure 8.4.

Chacun des polyèdres de cette union correspond à un scénario d'exécution différent. Considérons le premier polyèdre : l'ajout d'un délai d'au moins 5 unités de temps durant la première exécution de la première tâche (paramètre a) nous permet d'atteindre l'objectif fixé. Il en va de même pour le deuxième polyèdre (avec les paramètres b, c et d) et la deuxième exécution, ainsi que le troisième polyèdre et l'exécution de la tâche 2 avec le paramètre e.

Notez que n'importe lequel de ces polyèdre atteindrait l'objectif fixé, il est donc possible d'effectuer le choix du point sur ces trois polyèdres. Nous choisissons ici d'utiliser la deuxième option.

Après utilisation du script évoqué plus haut, nous obtenons le résultat suivant : $d = 6$, et le reste des paramètres à 0.

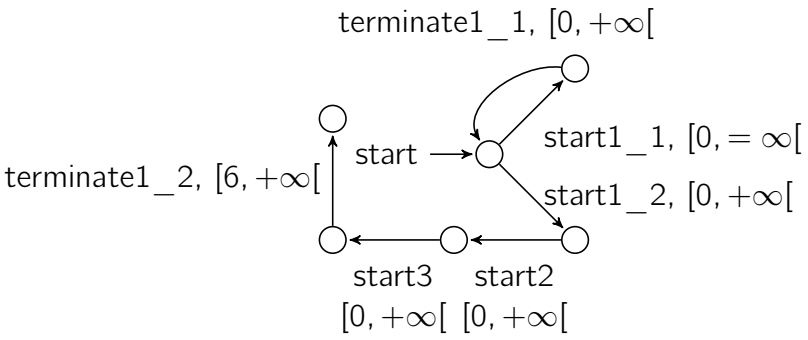
8.2.3 Configuration du forçage en ligne

Une fois le point obtenu, nous devons générer le modèle DTS des différentes tâches permettant d'effectuer le forçage en ligne. Après projection des valeurs de paramètres et ϵ-réduction, nous obtenons les modèles visibles en figure 8.5

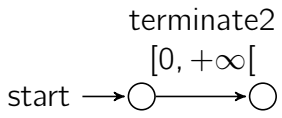
8.2.4 Exécution du test

L'exécution du test se fait sur la carte STM32f4discovery évoquée dans la section 7.2. Le banc de test est simplement composé de cette carte ayant été flashée avec l'exécutable généré incluant le modèle.

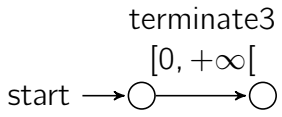
Dans l'exécutable embarqué, la fonction déclenchée lorsqu'une faute temporelle est détectée a été équipée avec du code lui permettant d'allumer une LED sur la carte.



(a) DTS de la tâche $T1$



(b) DTS de la tâche $T2$



(c) DTS de la tâche $T3$

Figure 8.5 – DTS embarquées

Nous constatons l'allumage de la LED lorsque le forçage est effectué, ce qui confirme le bon fonctionnement de notre solution et de la protection temporelle AUTOSAR.

Conclusion et perspectives

Nous revenons dans ce chapitre sur les apports de la présente thèse, et dégageons les quelques perspectives pour la suite de ces travaux.

Conclusion

Le test des systèmes temps réel est un champ de recherche crucial. Assurer la sûreté de fonctionnement et l'adéquation aux spécifications du logiciel embarqué temps réel moderne, dont la complexité ne cesse d'augmenter, demande l'utilisation de méthodes nouvelles. Il est nécessaire de permettre la considération de systèmes dont les comportements peuvent apparaitre non déterministes.

Dans ce travail de thèse, nous avons présenté une méthode facilitant le test des systèmes temps réel complexes. Cette méthode se place dans un contexte d'ingénierie dirigée par les modèles. Un modèle comportemental d'une application temps réel est utilisé pour déterminer une stratégie de test. Cette stratégie permet, à l'aide d'un contrôleur inclus dans le système d'exploitation temps réel, de forcer l'application à se rendre dans l'état dont l'exécution doit être testée.

L'état visé par notre stratégie n'est pas forcément atteignable si l'on ne manipule que les données d'entrées. Notre solution permet donc de l'injection de fautes, atteignant des états qui ne seraient pas atteignables autrement. Cela parmet aussi d'atteindre de façon déterministe un état dont le chemin dépend de la trajectoire interne du système.

Nous avons présenté les formalismes utilisés, la manière dont les modèles sont manipulés afin d'obtenir le modèle d'analyse puis les modèles embarqués, le choix de la stratégie de test optimale, ainsi que l'implémentation dans un système d'exploitation temps réel.

Nous avons de plus analysé l'impact de notre solution sur le système. Les résultats que nous proposons quant à la surcharge que notre solution impose sur l'exécution du système permettent ainsi d'effectuer des tests au plus proche des comportements attendus d'un système en conditions réelles, ne contenant pas notre solution de contrôle.

Perspectives

Stratégie coopérative Dans ces travaux, nous ne présentons que l'obtention de la stratégie de test et la réalisation du contrôle. En ce qui concerne le cas de test, il serait

intéressant de pouvoir les générer de paire avec la stratégie de test, permettant ainsi d'augmenter l'ensemble des actions contrôlables avec l'ensemble des entrées du système, et donc le contrôle effectif sur le système.

Élargir les propriétés considérées Les propriétés considérées dans ce travail de thèse sont uniquement des propriétés d'accessibilité (AF) ou d'invariance ou sûreté (AG). Il serait intéressant de pouvoir gérer des propriétés moins fortes, telles que l'existence (EF), en utilisant la coopération avec la stratégie de test proposée ci-dessus. Cette coopération pourrait prendre la forme d'une génération de cas de tests en parallèle de la génération de la stratégie de test, donnant ainsi un contrôle plus complet de la trajectoire du système. Cette approche permettrait d'augmenter le nombre de systèmes sur lesquels ces travaux peuvent être utilisés.

Applications Nos travaux sont appliqués ici au cas d'utilisation du test. Cependant, il est possible de les utiliser pour d'autres applications. Afin de valider et d'améliorer les contributions de ces travaux, il serait intéressant de les appliquer à des problèmes variés :

- *Identification des contraintes critiques :* Sur un système donné, il serait possible d'utiliser ces travaux pour permettre l'identification de contraintes critiques vis à vis de critères temporels (ou autres). Cela pourrait être effectué en modifiant les modèles comportementaux des tâches pour inclure des comportements non disponibles dans le modèle, et permettre ainsi à l'analyse de révéler des valeurs de délais permettant de tester ces chemins potentiellement critiques.
  Cette approche permettrait ainsi de venir préciser les WCETs dont l'estimation a été faite de manière peu précise dans un but d'économie de ressources, et ce uniquement pour les WCETs dont les conséquences sont critiques pour l'exécution du système.
- *Forçage en ligne et sûreté :* Une application possible de notre démarche serait de forcer sur un système critique une propriété de sûreté. On pourrait par exemple garantir une période minimale sur un ensemble de tâches, évitant la surcharge du système ou garantissant l'ordonnancabilité.

Bibliographie

Alur, R., Henzinger, T. A. & Vardi, M. Y. (1993), Parametric real-time reasoning, *in* 'STOC', ACM, pp. 592–601.

André, É., Fribourg, L., Kühne, U. & Soulat, R. (2012), IMITATOR 2.5 : A tool for analyzing robustness in scheduling problems, *in* 'FM', Vol. 7436 of *Lecture notes in computer science*, Springer, pp. 33–36.

André, É., Lime, D. & Roux, O. H. (2015), Integer-complete synthesis for bounded parametric timed automata, *in* 'RP', Vol. 9058 of *Lecture notes in computer science*, Springer.

Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E. & Leber, G. H. (2003), 'Comparison of physical and software-implemented fault injection techniques', *IEEE Trans. on Comp.Sci.* **52**(9).

Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Roşu, G., Sen, K., Visser, W. et al. (2005), 'Combining test case generation and runtime verification', *Theoretical Computer Science* **336**(2), 209–234.

AUT (2014), *Requirements on Free Running Timer*. Rev. 4.2.1.

Basin, D., Jugé, V., Klaedtke, F. & Zălinescu, E. (2013), 'Enforceable security policies revisited', *ACM Transactions on Information and System Security (TISSEC)* **16**(1), 3.

Bauer, A., Leucker, M. & Schallhart, C. (2011), 'Runtime verification for ltl and tltl', *ACM TOSEM* **20**(4).

Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K. G. & Lime, D. (2007), Uppaal-tiga : Time for playing games !, *in* 'International Conference on Computer Aided Verification', Springer, pp. 121–125.

Berthomieu, B., Lime, D., Roux, O. H. & Vernadat, F. (2007), 'Reachability problems and abstract state spaces for time Petri nets with stopwatches', *Journal of Discrete Event Dynamic Systems - Theory and Applications (DEDS)* **17**(2), 133–158.

Berthomieu, B. & Vernadat, F. (2006), Time petri nets analysis with tina, *in* 'Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)', IEEE, pp. 123–124.

Bertrand, D., Faucou, S. & Trinquet, Y. (2009), An analysis of the autosar os timing protection mechanism, *in* '2009 IEEE Conference on Emerging Technologies & Factory Automation', IEEE, pp. 1–8.

Bohnenkamp, H. & Belinfante, A. (2005), Timed testing with torx, *in* 'International Symposium on Formal Methods', Springer, pp. 173–188.

Bouyer, P., D'Souza, D., Madhusudan, P. & Petit, A. (2003), Timed control with partial observability, *in* 'International Conference on Computer Aided Verification', Springer, pp. 180–192.

Braberman, V., D'Ippolito, N., Piterman, N., Sykes, D. & Uchitel, S. (2013), Controller synthesis : From modelling to enactment, *in* 'Proceedings of the 2013 International Conference on Software Engineering', IEEE Press, pp. 1347–1350.

Briones, L. B. & Brinksma, E. (2004), A test generation framework for quiescent real-time systems, *in* 'International Workshop on Formal Approaches to Software Testing', Springer, pp. 64–78.

Brun, M. (2010), Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application, PhD thesis, Nantes.

Bucci, G., Fedeli, A., Sassoli, L. & Vicario, E. (2004), 'Timed state space analysis of real-time preemptive systems', *IEEE transactions on software engineering* **30**(2), 97–111.

Carreira, J., Madeira, H., Silva, J. G. et al. (1998), 'Xception : Software fault injection and monitoring in processor functional units', *Dependable Computing and Fault Tolerant Systems* **10**.

Christmansson, J. & Chillarege, R. (1996), Generation of an error set that emulates software faults based on field data, *in* 'Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on', IEEE, pp. 304–313.

Clarke, D., Jéron, T., Rusu, V. & Zinovieva, E. (2002), Stg : A symbolic test generation tool, *in* 'International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 470–475.

Colombo, C., Pace, G. J. & Schneider, G. (2009), Larva—safer monitoring of real-time java programs (tool paper), *in* '2009 Seventh IEEE International Conference on Software Engineering and Formal Methods', IEEE, pp. 33–37.

Cotard, S., Faucou, S., Béchennec, J., Queudet, A. & Trinquet, Y. (2012), A data flow monitoring service based on runtime verification for AUTOSAR, *in* 'HPCC-ICESS 2012, Liverpool, UK'.

Dawson, S., Jahanian, F., Mitton, T. & Tung, T.-L. (1996), Testing of fault-tolerant and real-time distributed systems via protocol fault injection, *in* 'Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on', IEEE, pp. 404–414.

Desaulniers, G., Desrosiers, J. & Solomon, M. M. (2006), *Column generation*, Vol. 5, Springer Science & Business Media.

Doyen, L. (2007), 'Robust parametric reachability for timed automata', *Information Processing Letters* **102**(5), 208–213.

Dufourd, C., Finkel, A. & Schnoebelen, P. (1998), Reset nets between decidability and undecidability, *in* 'Automata, Languages and Programming', Springer, pp. 103–115.

Entrena, L., López-Ongil, C., García-Valderas, M., Portela-García, M. & Nicolaidis, M. (2011), Hardware fault injection, *in* 'Soft Errors in Modern Electronic Systems', Springer, pp. 141–166.

Falcone, Y., Mounier, L., Fernandez, J.-C. & Richier, J.-L. (2011), 'Runtime enforcement monitors : composition, synthesis, and enforcement abilities', *Formal Methods in System Design* **38**(3).

Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K. & Lange, K. (2009), Autosar–a worldwide standard is on the road, *in* '14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden', Vol. 62.

GLPK (GNU Linear Programming Kit) (2012). `https://www.gnu.org/software/glpk/`

Gomory, R. (1960), An algorithm for the mixed integer problem, Technical report, DTIC Document.

Havelund, K. & Roşu, G. (2001), 'Monitoring java programs with java pathexplorer', *Electronic Notes in Theoretical Computer Science* **55**(2), 200–217.

Havelund, K. & Roşu, G. (2002), Synthesizing monitors for safety properties, *in* 'International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 342–356.

Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P. & Skou, A. (2008), Testing real-time systems using uppaal, *in* 'Formal methods and testing', Springer, pp. 77–117.

Hsueh, M.-C., Tsai, T. K. & Iyer, R. K. (1997), 'Fault injection techniques and tools', *Computer* **30**(4).

Hune, T., Romijn, J., Stoelinga, M. & Vaandrager, F. W. (2002), 'Linear parametric model checking of timed automata', *Journal of Logic and Algebraic Programming* **52-53**, 183–220.

Janicki, R. & Koutny, M. (1991), Invariant semantics of nets with inhibitor arcs, *in* 'International Conference on Concurrency Theory', Springer, pp. 317–331.

Janicki, R. & Koutny, M. (1995), 'Semantics of inhibitor nets', *Information and Computation* **123**(1), 1–16.

Janzen, D. S. & Saiedian, H. (2005), 'Test-driven development : Concepts, taxonomy, and future direction', *Computer* **38**(9), 43–50.

Jard, C. & Jéron, T. (2005), 'Tgv : theory, principles and algorithms', *International Journal on Software Tools for Technology Transfer* **7**(4), 297–315.

Jovanović, A., Lime, D. & Roux, O. H. (2013), Integer parameter synthesis for timed automata, *in* 'International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 401–415.

Jovanović, A., Lime, D. & Roux, O. H. (2015), 'Integer Parameter Synthesis for Real-Time Systems', *IEEE Transactions on Software Engineering (TSE)* **41**(5), 445–461.

Kanawati, G. A., Kanawati, N. A. & Abraham, J. A. (1992), Ferrari : A tool for the validation of system dependability properties, *in* 'FTCS-22. Digest of Papers.', IEEE.

Karmarkar, N. (1984), A new polynomial-time algorithm for linear programming, *in* 'Proceedings of the sixteenth annual ACM symposium on Theory of computing', ACM, pp. 302–311.

Khurshid, S., Păsăreanu, C. S. & Visser, W. (2003), Generalized symbolic execution for model checking and testing, *in* 'International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 553–568.

King, J. C. (1976), 'Symbolic execution and program testing', *Communications of the ACM* **19**(7), 385–394.

Krichen, M. & Tripakis, S. (2004), Black-box conformance testing for real-time systems, *in* 'Model Checking Software', Springer.

Krichen, M. & Tripakis, S. (2009), 'Conformance testing for real-time systems', *Formal Methods in System Design* **34**(3), 238–304.

Kumar, R. & Garg, V. K. (2012), *Modeling and control of logical discrete event systems*, Vol. 300, Springer Science & Business Media.

Laroussinie, F., Markey, N. & Schnoebelen, P. (2002), On model checking durational kripke structures, *in* 'Foundations of Software Science and Computation Structures', Springer, pp. 264–279.

Larsen, K. G., Pettersson, P. & Yi, W. (1997), 'Uppaal in a nutshell', *International Journal on Software Tools for Technology Transfer (STTT)* **1**(1), 134–152.

Lelionnais, C., Delatour, J., Brun, M., Roux, O. H. & Seidner, C. (2014), 'Formal synthesis of real-time system models in a mde approach', *IARIA Journals* **7**(1&2), pp–115.

Leucker, M. & Schallhart, C. (2009), 'A brief account of runtime verification', *The Journal of Logic and Algebraic Programming* **78**(5), 293–303.

Ligatti, J., Bauer, L. & Walker, D. (2009), 'Run-time enforcement of nonsafety policies', *ACM Transactions on Information and System Security (TISSEC)* **12**(3), 19.

Lime, D., Roux, O. H., Seidner, C. & Traonouez, L.-M. (2009), Romeo : A parametric model-checker for Petri nets with stopwatches, *in* S. Kowalewski & A. Philippou, eds, '15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)', Vol. 5505 of *LNCS*, Springer, York, United Kingdom, pp. 54–57.

Luenberger, D. G. (1973), *Introduction to linear and nonlinear programming*, Vol. 28, Addison-Wesley Reading, MA.

Madeira, H., Costa, D. & Vieira, M. (2000), On the emulation of software faults by software fault injection, *in* 'Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on', IEEE, pp. 417–426.

Maler, O., Nickovic, D. & Pnueli, A. (2006), From mitl to timed automata, *in* 'International Conference on Formal Modeling and Analysis of Timed Systems', Springer, pp. 274–289.

Martin, J. (1965), Programming real-time computer systems, Technical report.

Matteucci, I. (2007), 'Automated synthesis of enforcing mechanisms for security properties in a timed setting', *Electronic Notes in Theoretical Computer Science* **186**, 101–120.

Miller, J. S. (2000), Decidability and complexity results for timed automata and semi-linear hybrid automata, *in* 'HSCC', Vol. 1790 of *Lecture notes in computer science*, Springer, pp. 296–309.

Nickovic, D. & Maler, O. (2007), Amt : A property-based monitoring tool for analog systems, *in* 'International Conference on Formal Modeling and Analysis of Timed Systems', Springer, pp. 304–319.

Petri, C. A. (1962), 'Kommunikation mit automaten'.

Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A. & Timo, O. L. N. (2013), Runtime enforcement of timed properties, *in* 'Runtime Verification', Springer.

Rayadurgam, S. & Heimdahl, M. P. E. (2001), Coverage based test-case generation using model checkers, *in* 'Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the', IEEE, pp. 83–91.

Schneider, F. B. (2000), 'Enforceable security policies', *ACM Transactions on Information and System Security (TISSEC)* **3**(1), 30–50.

Seidl, H. (1996), A modal mu-calculus for durational transition systems, *in* 'Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science', LICS '96, IEEE Computer Society, Washington, DC, USA.

Shin, K. G. & Ramanathan, P. (1994), 'Real-time computing : A new discipline of computer science and engineering', *Proceedings of the IEEE* **82**(1), 6–24.

Stolz, V. & Bodden, E. (2006), 'Temporal assertions using aspectj', *Electronic Notes in Theoretical Computer Science* **144**(4), 109–124.

Sun, Y., Soulat, R., Lipari, G., André, É. & Fribourg, L. (2013), Parametric schedulability analysis of fixed priority real-time distributed systems, *in* C. Artho & P. Ölveczky, eds, 'Second International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'13)', Vol. 419 of *Communications in Computer and Information Science*, Springer, pp. 212–228.

Thati, P. & Roşu, G. (2005), 'Monitoring algorithms for metric temporal logic specifications', *Electronic Notes in Theoretical Computer Science* **113**, 145–162.

Thomas, W. (2002), Infinite games and verification, *in* 'International Conference on Computer Aided Verification', Springer, pp. 58–65.

Traonouez, L.-M., Lime, D. & Roux, O. H. (2008), Parametric model-checking of time petri nets with stopwatches using the state-class graph, *in* 'Formal Modeling and Analysis of Timed Systems'.

Traonouez, L.-M., Lime, D. & Roux, O. H. (2009), 'Parametric model-checking of stopwatch Petri nets', *Journal of Universal Computer Science* **15**(17), 3273–3304.

Tretmans, J. (1999), Testing concurrent systems : A formal approach, *in* 'CONCUR'99 Concurrency Theory', Springer.

Wang, F. (1996), 'Parametric timing analysis for real-time systems', *Information and Computation* **130**(2), 131–150.

Wolsey, L., Rinaldi, G. et al. (1993), 'Integer programming and combinatorial optimization'.

UNIVERSITE BRETAGNE LOIRE

Centrale Nantes

THÈSE DE DOCTORAT

LOUIS-MARIE GIVEL

TEST DE SYSTÈMES TEMPS RÉEL À L'AIDE DU FORÇAGE EN LIGNE

TESTING REAL-TIME SYSTEMS USING RUNTIME ENFORCEMENT

Résumé

Les systèmes temps réel doivent répondre à des contraintes d'exactitude et de sureté de fonctionnement, ce qui nécessite de pouvoir les tester. Dans ces travaux de thèse, nous proposons une méthode permettant de forcer une exécution d'un système temps réel de manière à atteindre un état choisi.

Cette technique peut permettre le test de systèmes temps réel dans des situations où l'état choisi pourrait être compliqué à atteindre en ne manipulant que les données d'entrée du système. Un exemple d'application pourrait être celui d'un état déclenchant un redondant, et dont l'exécution serait peu fréquente, mais que l'on souhaite néanmoins tester. Dans ce cas d'utilisation, il s'agit effectivement de l'injection de fautes dans le système.

Cette solution est basée sur une analyse hors-ligne d'un ensemble de modèles comportementaux du système, utilisant le formalisme des structures de Kripke à durée. Les modèles du système sont issu d'une démarche IDM, et présentent le comportement des tâches du point de vue du système d'exploitation temps réel.

À l'aide d'une paramétrisation, transformation et composition des modèles, nous effectuons une analyse paramétrée du réseau de Petri temporel résultant. Grâce au résultat de cette analyse, nous déterminons un ensemble de délais à ajouter à l'exécution du système afin de forcer le système à atteindre l'état choisi.

En utilisant un modèle embarqué des différentes tâches, et par le biais d'un contrôleur intégré au système d'exploitation temps réel, nous effectuons le forçage en ligne du système afin de permettre l'atteinte de l'état choisi, et éventuellement le test de l'exécution découlant de cet état.

Ce travail de thèse détaille les formalismes utilisés pour modéliser l'application, la manière dont ils sont modifiés et analysés pour obtenir les délais d'exécution, ainsi que l'implémentation dans un système d'exploitation temps réel : TrampolineRTOS.

Mots clés

systèmes temps réel, forçage en ligne, réseaux de Petri, test

Abstract

Real-time systems must conform to requirements of correctness and safety, which requires testing them. In this thesis, we propose a method that allows to force an execution of a real-time system so as to reach a chosen state.

This technique can be used to test real-time systems in situations where the chosen state could be hard to reach when dealing with the input sequence of the system alone. One example of an application could be that of a state that triggers a redundancy mechanism when reached, the execution of which would be infrequent, but that still requires testing. In this use case, it is effectively fault injection in the system.

This solution is based on an offline analysis of a set of behavioral models of the system, using the formalism of durational Kripke structures. The models of the system are obtained through a Model Driven Engineering approach, and present the behaviors of the tasks from the point of view of the real-time operating system.

Using a parametrization, transformation and composition of the models, we perform a parametric analysis of the resulting Time Petri net. Thanks to the result of this analysis, we establish a set of delays to be added to the execution of the system so as to reach the chosen state.

With the help of an embedded model of the different tasks of the system, and through the means of a controller embedded in the real-time operating system, we perform the runtime enforcement of the system so as to make it reach the chosen state, and possibly allow for testing the execution starting from this state.

This thesis details the formalisms used to model the application, the way in which the models are modified and analyzed to obtain the execution delays, as well as the implementation in a real-time operating system: TrampolineRTOS.

Key Words

real-time systems, runtime enforcement, Petri nets, test