Méthode de génération d'exécutifs temps-réel

Toussaint Kabland Gautier Tigori

▶ To cite this version:

Toussaint Kabland Gautier Tigori. Méthode de génération d'exécutifs temps-réel. Système d'exploitation [cs.OS]. École Centrale de Nantes, 2016. Français. NNT : . tel-02990618v1

HAL Id: tel-02990618
https://theses.hal.science/tel-02990618v1

Submitted on 17 Sep 2020 (v1), last revised 5 Nov 2020 (v2)

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

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

UNIVERSITE BRETAGNE LOIRE

Centrale Nantes

Thèse de Doctorat

Kabland Toussaint Gautier TIGORI

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
Unité de recherche : IRCCyN, UMR 6597

Soutenue le 24 novembre 2016

Méthode de génération d'exécutifs temps-réel

JURY

Président :	**M. Gilles MULLER**, Directeur de recherche, INRIA-LIP6,
Rapporteurs :	**Mme Claire PAGETTI**, Ingénieur de recherche HDR, ONERA, Toulouse
	M. Emmanuel GROLLEAU, Professeur des universités, ISAE-ENSMA, Poitiers
Examinateurs :	**M. Hervé MARCHAND**, Chargé de recherche, INRIA, Rennes
	M. Jean-Luc BÉCHENNEC, Chargé de recherche, CNRS-IRCCyN, Nantes
Directeur de thèse :	**M. Olivier H. ROUX**, Professeur des universités, École Centrale de Nantes - IRCCyN, Nantes

À ma mère Hélène EBA, à ma fille Emmanuella TIGORI, à Ursule KORANDJI, à mes frères et sœurs ainsi qu'à la mémoire de mon défunt père Albert TIGORI AKA.

Remerciements

Tout d'abord, j'adresse mes sincères remerciements à Jean-Luc BÉCHENNEC et Olivier-Henri ROUX mes encadrants, qui ont largement contribué à la réalisation de ce travail de thèse. Sans oublier Sébastien FAUCOU qui grâce à ses idées pertinentes a permis d'apporter plus de précision au travail réalisé. Leurs conseils, leurs soutiens, et surtout leurs connaissances scientifiques avancées m'ont permis à la fois de prendre confiance en moi et d'acquérir toutes les compétences nécessaires au bon déroulement de cette thèse.

Mes remerciements s'adressent à Monsieur Gilles MULLER Directeur de recherche au LIP6 de Paris pour m'avoir fait l'honneur de présider mon jury de thèse. Profonds remerciements également à Madame Claire PAGETTI Ingénieur de recherche et HDR à l'ONERA de Toulouse et Monsieur Emmanuel GROLLEAU Professeur des universités à l'ISAE-ENSMA de Poitiers pour avoir accepté de rapporter mon manuscrit de thèse, pour l'intérêt qu'ils ont porté à mes travaux et pour leurs remarques constructives. Je remercie également Monsieur Hervé MARCHAND Chargé de recherche à l'INRIA de Rennes d'avoir accepté de participer à mon jury de thèse.

Je remercie mes compagnons de thèse, Louis-Marie GIVEL, Armel MANGEAN, Bertrand MIANNAY et Louis FIPPO FITIME ainsi que tous les autres doctorants de l'IRCCyN avec qui j'ai partagé de bon moments et qui resteront gravés dans ma mémoire.

Je remercie aussi tous les membres de l'équipe Système Temps-Réel de l'IRCCyN d'avoir facilité mon intégration au sein de l'équipe.

Je remercie tous mes amis pour leur soutien et pour tous les bon moments passés ensemble.

Je remercie tous les membres de ma famille surtout ma mère pour son soutien moral et mon frère pour ses conseils.

Enfin je remercie ma future épouse Ursule qui a bien voulu se lancer à mes côtés dans cette aventure.

Table des matières

Introduction générale **15**

Chapitre 1
Configuration des systèmes d'exploitation

1.1 Définitions . . . 20
1.1.1 Paradigme . . . 20
1.1.2 Granularité . . . 21
1.1.3 Intégrité . . . 21
1.2 Types de configuration . . . 21
1.2.1 Configuration statique . . . 22
1.2.2 Configuration dynamique . . . 24
1.3 Paradigme de développement . . . 25
1.3.1 Programmation orientée objet . . . 26
1.3.2 Programmation orientée sujet . . . 28
1.3.3 Programmation orientée aspect . . . 29
1.3.4 Langages dédiés (DSL) . . . 31
1.4 Conclusion . . . 32

Chapitre 2
Trampoline, un système d'exploitation temps réel

2.1 Le standard OSEK/VDX . . . 36
2.1.1 Architecture du système d'exploitation OSEK . . . 36
2.1.2 Gestion des tâches . . . 37
2.1.3 Politique d'ordonnancement . . . 38
2.1.4 Les modes d'application . . . 40
2.1.5 Traitement des interruptions . . . 40
2.1.6 Mécanisme des événements . . . 41
2.1.7 Gestion des ressources . . . 42
2.1.8 Alarmes . . . 45
2.2 AUTOSAR : un standard d'architecture logicielle . . . 46
2.2.1 Architecture logicielle . . . 46
2.2.2 Configuration dans AUTOSAR . . . 46
2.3 Trampoline . . . 46
2.3.1 Présentation . . . 46
2.3.2 Architecture de Trampoline monocœur . . . 48
2.3.3 Architecture de Trampoline multicœur . . . 51
2.4 Conclusion . . . 52

Chapitre 3
Modélisation

3.1 Outil de formalisme . . . 56
3.1.1 Automates finis . . . 56
3.1.2 Automates finis étendus . . . 56
3.1.3 Réseau d'automates finis étendus . . . 57
3.1.4 Utilisation des automates finis étendus pour l'abstraction d'un programme 58
3.1.5 Accessibilité et bornitude . . . 58
3.2 Principes de la modélisation . . . 59
3.2.1 Modélisation des branchements . . . 59
3.2.2 Modélisation des boucles . . . 59
3.2.3 Modélisation d'un appel de fonction . . . 60
3.3 Adaptation à UPPAAL . . . 60
3.3.1 UPPAAL : outil pour la vérification de systèmes temps réel . . . 60
3.3.2 Le traitement du temps . . . 62
3.3.3 Règles de modélisation en utilisant UPPAAL . . . 63
3.4 Modèle de Trampoline dédié aux architectures monocœur . . . 64
3.4.1 Modélisation des objets et des variables . . . 64
3.4.2 Modélisation du noyau . . . 65
3.4.3 Modélisation des services . . . 68
3.4.4 Modélisation de l'interruption matérielle liée au timer . . . 70
3.5 Modèle de Trampoline dédié aux architectures multicœur . . . 71
3.5.1 Modélisation du noyau . . . 71
3.5.2 Modélisation des services . . . 72
3.6 Modèle d'applications . . . 72
3.7 Propriétés du modèle . . . 73
3.8 Conclusion . . . 75

Chapitre 4
Synthèse formelle de systèmes d'exploitation

4.1 Méthodologie . . . 78
4.1.1 Modélisation . . . 79
4.1.2 Synthèse formelle . . . 79
4.1.3 Génération de code . . . 81
4.2 Outil développé . . . 83
4.2.1 Synthétiseur de modèle . . . 83
4.2.2 Générateur de code . . . 84
4.3 Étude de cas . . . 85
4.4 Conclusion . . . 88

Chapitre 5
Vérifications formelles

5.1 Les techniques de vérification formelle . . . 90
5.1.1 Model Checking . . . 90
5.1.2 Analyse statique . . . 91
5.1.3 Test . . . 91

5.2 Vérification formelle de la conformité OSEK/VDX 92
5.2.1 Cas de test 93
5.2.2 Séquence de test 93
5.2.3 Suite de test 94
5.2.4 Exemple 94
5.3 Vérification formelle de la conformité OSEK/VDX à partir d'observateurs génériques 97
5.3.1 Présentation de l'approche 97
5.3.2 Observateurs 98
5.3.3 Étude de cas 102
5.4 Conclusion 105

Conclusion générale et perspectives **107**

Publications **109**

Bibliographie **110**

Table des figures

1 Extrait du code du système d'exploitation Trampoline incluant des directives de pré-processeur pour la configuration à la compilation. 16

1.1 Illustration des différents types de configuration [Frö01] 22
1.2 Représentation de l'ordonnanceur de Trampoline sous forme de diagramme de classes 27
1.3 Composition de sujets 28
1.4 Modularisation selon la programmation orientée objet 30
1.5 Modularisation selon la programmation orientée aspect 30
1.6 Transformation d'un langage dédié en langage généraliste 31
1.7 Spécialisation d'un système d'exploitation par un langage dédié [PBC+97] 32

2.1 Diagramme d'état d'une tâche étendue 38
2.2 Diagramme d'état d'une tâche de base 38
2.3 Ordonnancement préemptif. 39
2.4 Ordonnancement non-préemptif. 40
2.5 Synchronisation par événement 42
2.6 Exemple de l'inversion de priorité 43
2.7 Exemple d'interblocage en utilisation des sémaphores 43
2.8 Utilisation des ressources dans l'ordonnancement des tâches. 45
2.9 Architecture Logicielle 47
2.10 Chaîne de compilation et configuration de Trampoline 47
2.11 Architecture de Trampoline monocœur [BBF15] 48
2.12 Structure du gestionnaire des appels système [BBF15] 49
2.13 Structure de données pour la gestion des alarmes 50
2.14 Structure de la liste des tâches prêtes 51
2.15 Architecture de Trampoline multicœur 52
2.16 Scénario d'exécution de tâches sur Trampoline multicœur 53

3.1 Représentation d'un branchement par un automate fini étendu. 59
3.2 Représentation d'une boucle par un automate fini étendu 59
3.3 Description d'un appel de fonction (fonction a à gauche et fonction b à droite) :*La variable partagée x permet le blocage de l'automate appelant jusqu'à ce que l'automate appelée termine son exécution. L'automate modélisant la fonction a reprend son exécution après la fin de l'exécution de l'automate modélisant la fonction b. Les actions h! et h? traduisent le point de synchronisation des deux automates.* 60
3.4 Descripteurs statique et dynamique d'une tâche dans le modèle 65
3.5 Représentation des pointeurs dans le modèle. 65
3.6 Modélisation de la fonction du noyau `tpl_terminate` 66

3.7 Modèle du gestionnaire de compteur 68
3.8 Code de la fonction `tpl_get_proc` dans le modèle (gauche) et dans Trampoline (droite) 69
3.9 Modèle de l'API TerminateTask 69
3.10 Modèle du service de terminaison d'une tâche (`tpl_terminate_task_service`) . . 70
3.11 Modèle du *Timer* 70
3.12 Modèle de la fonction `tpl_terminate` dans le cas multicoeur 71
3.13 Modèle de la fonction `tpl_terminate_task_service` dans le cas multicoeur 72
3.14 Modélisation d'une application 74

4.1 Positionnement de notre approche dans la chaîne de compilation de Trampoline . 78
4.2 Synthèse formelle du système d'exploitation 79
4.3 Synthèse de modèle 82
4.4 Code généré suite au modèle réduit de la fonction `tpl_terminate` 83
4.5 Architecture du synthétiseur du système d'exploitation 84
4.6 Modèle de la tâche `TASK_ACC_IDM` 87

5.1 Processus du *model checking* 90
5.2 Test de conformité 92
5.3 Modèle de séquence de test 95
5.4 Modèle d'un observateur de séquence de test 96
5.5 Processus de vérification formelle 97
5.6 Observateur générique *Obs*(`t`) pour un cas de test (propriété) `t` 98
5.7 Modification apportée au modèle initial de la fonction `tpl_terminate_task_service` 99
5.8 Exemple d'un modèle d'observateur 100
5.9 Exemple de cas de tests liés aux interruptions 101
5.10 Modèle du système de contrôle de caméra et du suivi 104

Liste des tableaux

4.1 Priorité, démarrage et période des tâches de l'application ADAS. 86
4.2 Résultat de la synthèse du système d'exploitation 88

5.1 Cas de test 93
5.2 Séquence de test 93

Introduction générale

Contexte général

Les systèmes embarqués ont considérablement amélioré la qualité de vie humaine ces trente dernières années et font maintenant partie intégrante de notre quotidien. Leur importance se ressent autant dans la vie active que dans le milieu industriel. Nous les retrouvons dans des petits objets connectés comme les montres intelligentes, les appareils électroménagers jusqu'aux systèmes industriels complexes comme les systèmes aéronautiques, automobiles et de télécommunications.

Les systèmes embarqués sont par définition un assemblage de systèmes électroniques et informatiques autonomes destinés à réaliser des tâches précises. Par exemple, dans le domaine automobile, par mesure de sécurité, les véhicules sont équipés de systèmes de freinage automatique ou de détection d'obstacles.

Compte tenu de leur évolution, ces systèmes nécessitent de plus en plus de fonctionnalités tant sur le plan logiciel que matériel en fonction des nouvelles applications mises au point et ceci se traduit par une augmentation de la complexité.

Les systèmes embarqués doivent alors implémenter toutes les fonctionnalités souhaitées, être plus performants en ce qui concerne le temps de calcul et exécuter les tâches qui leurs sont attribuées en un temps donné (par ex. systèmes temps réel).

Dans le domaine automobile, les nouveaux véhicules intègrent environ 270 fonctions déployées dans 70 unités de commande électronique (ECU) et ce nombre ne cessera de croître dans l'avenir avec les véhicules autonomes.

La gestion de toute cette complexité est réalisée par un composant des systèmes embarqués qui sert d'interface entre le matériel et le logiciel, qui implante toutes les fonctionnalités souhaitées et qui est responsable de la gestion des ressources matérielles ainsi que de l'ordonnancement des processus des applications : il s'agit du système d'exploitation.

Le système d'exploitation est une solution à la complexité des systèmes embarqués, mais entraîne un coût additionnel de mémoire car, la mémoire représente en effet une forte contrainte pour ces systèmes à cause de leur taille relativement petite.

Les systèmes embarqués étant en général de petite taille et présentant des ressources limitées, les systèmes d'exploitation ne peuvent intégrer toutes les fonctionnalités souhaitées pour une large gamme d'application. Par contre, d'une application à une autre, les besoins en terme de fonctionnalités varient. Pour cela, une solution à cette contrainte de taille serait de n'embarquer dans le système d'exploitation que l'ensemble des fonctionnalités réellement nécessaires à une application donnée. Cela aura pour avantage d'optimiser l'espace mémoire occupé par le système d'exploitation, d'améliorer la performance des systèmes embarqués et de réduire la probabilité d'occurrence de comportements fautifs.

Le système d'exploitation doit être implémenté de manière à être :

— Configurable : Afin de n'embarquer que les fonctionnalités nécessaires à une application donnée.
— Extensible : Dans la mesure où de nouvelles fonctionnalités sont nécessaires pour une application donnée (évite une ré-implémentation complète du système d'exploitation lors de l'implémentation de nouvelles fonctionnalités).

Problématique

Pour les raisons évoquées précédemment, les systèmes d'exploitation sont développés de manière à être configurables. La configuration des systèmes d'exploitation pour les systèmes embarqués à ressources limitées est la plupart du temps basée sur le pré-processeur C et la compilation conditionnelle.

Cette configuration relativement simple fait une utilisation intensive des directives de pré-processeur dans le code source du système d'exploitation.

L'extrait du code de la fonction `tpl_activate_task_service` du système d'exploitation Trampoline à la figure 1 en montre un exemple.

```
FUNC(StatusType, OS_CODE) tpl_activate_task_service(
  CONST(tpl_task_id, AUTOMATIC) task_id)
{
    ...
#if TASK_COUNT > 0
  IF_NO_EXTENDED_ERROR(result)
    result = tpl_activate_task(task_id);
    if (result == (tpl_status)E_OK_AND_SCHEDULE)
    {
      tpl_schedule_from_running();
# if WITH_SYSTEM_CALL == NO
      if (tpl_kern.need_switch != NO_NEED_SWITCH)
      {
        tpl_switch_context(
          &(tpl_kern.s_old->context),
          &(tpl_kern.s_running->context)
        );
      }
# endif
    }
  IF_NO_EXTENDED_ERROR_END()
#endif
    ...
}
```

FIGURE 1 – Extrait du code du système d'exploitation Trampoline incluant des directives de préprocesseur pour la configuration à la compilation.

À l'intérieur de cette fonction, les déclarations `#if` et `#endif` sont utilisées pour la sélection d'une section de code à la compilation lorsque la condition exprimée est vraie. Par exemple, le code complet de la fonction sera pris en compte dans le système d'exploitation lors de la compilation si la valeur de la variable `TASK_COUNT` est supérieure à 0. Ce type d'implémentation de systèmes d'exploitation semble risqué pour plusieurs raisons.

La première raison concerne le fait que le pré-processeur n'impose pas de règles sémantiques strictes. Il est donc possible qu'une erreur soit présente dans l'expression de la condition et que le mauvais code soit sélectionné lors de la compilation ce qui causerait éventuellement des *bugs* importants dans le système d'exploitation.

La deuxième raison concerne l'utilisation intensive de directives de préprocesseur au sein du code source du système d'exploitation qui entraîne une mauvaise modularisation du code, diminue sa qualité et le rend difficilement lisible [LHSP+09]. Cela constitue aussi un problème pour des outils d'analyse statique comme *Frama-C* [CKK+12] qui permettent de vérifier le comportement de programmes avant leurs exécutions car, ces outils ne sont pas capables de détecter des erreurs dans l'expression de condition traduite par les directives de préprocesseur.

De plus, une erreur dans l'expression de la directive de préprocesseur peut produire du code mort à l'intérieur du système d'exploitation et ainsi fausser possiblement son fonctionnement. Ces erreurs doivent donc être détectées lors d'une phase de vérification et de validation du système d'exploitation. Cependant seules les techniques d'analyse statique peuvent fournir une preuve de correction du code du système d'exploitation.

Le code mort est difficilement détectable et ne doit pas être considéré comme inoffensif car il peut impacter aussi bien l'empreinte mémoire du système d'exploitation que sa performance. D'un point de vue de sûreté de fonctionnement, le code mort devient un problème important à résoudre.

Objectifs

L'objectif principal de la thèse est de développer une méthode de configuration ou de synthèse du système d'exploitation afin que celui-ci ne contienne aucun code mort et que sa correction soit prouvée lors d'une phase de vérification et de validation.

Pour cela, nous proposons une configuration du système d'exploitation basée sur un modèle formel. Dans ce type d'approche, la configuration est effectuée sur le modèle du système d'exploitation qui est utilisé pour la génération de son code source. Une telle méthode est prometteuse pour plusieurs raisons.

Premièrement, le modèle est exécutable et donne la possibilité de simuler le système d'exploitation.

Deuxièmement, si le modèle est conçu correctement, l'analyse statique de ce modèle peut être utilisée pour déterminer toutes les branches qui ne sont pas nécessaires à parcourir lors de l'exécution du système d'exploitation. Il est ainsi plus simple de supprimer l'ensemble de ces branches qui traduisent en réalité du code mort. Le modèle résultant (élagué) peut donc servir à la génération du code source du système d'exploitation. Le système d'exploitation configuré ou synthétisé utilise ainsi une faible empreinte mémoire et présente des performances plus importantes.

Troisièmement, les techniques de vérification de modèles peuvent être utilisées pour prouver la correction du modèle élagué. Ce type de vérification assure la preuve que le comportement du système d'exploitation n'est pas impacté lors du processus de configuration. Cela permet alors d'accroître le niveau de confiance envers le système d'exploitation configuré.

Contributions

La contribution de la thèse comprend deux grands axes. Un premier axe traite de la configuration ou synthèse de systèmes d'exploitation à partir de leurs modèles et un second axe traite de la vérification formelle de systèmes d'exploitation.

Configuration ou synthèse formelle de systèmes d'exploitation : Nous proposons une méthode basée sur une approche formelle pour la synthèse automatique d'un système d'exploi-

tation temps réel spécialisé pour une application. À partir d'un système d'exploitation existant, un modèle formel et exécutable est conçu. Ce modèle supporte toutes les caractéristiques du système d'exploitation correspondant à savoir, les services et le comportement.

L'application est également modélisée et son modèle est synchronisé avec celui du système d'exploitation temps réel afin de formaliser le déploiement de l'application sur le système d'exploitation.

Sur le modèle obtenu (application + système d'exploitation), une recherche d'accessibilité est effectuée afin de détecter tous les chemins non parcourus et états inaccessibles.

Ces chemins représentent en réalité les sections de code du système d'exploitation qui ne seront jamais exécutées pour cette application. Tous ces chemins sont par la suite supprimés automatiquement du modèle. Suite à cela, un modèle spécialisé qui ne contient que les chemins faisables est obtenu et ce modèle décrit le code indispensable à l'exécution de l'application.

Enfin, à partir du modèle spécialisé, le code complet du système d'exploitation spécialisé correspondant est engendré au moyen d'un générateur de code. Cette méthode accélère le processus de développement de systèmes d'exploitation spécialisés et permet la validation de ceux-ci avant leur déploiement.

Vérification formelle de systèmes d'exploitation : Nous proposons deux différentes approches pour la vérification formelle du système d'exploitation.

Une première approche consiste à prouver la conformité du système d'exploitation selon la spécification OSEK/VDX. L'approche utilise le principe de test de conformité OSEK/VDX où des séquences de tests sont exécutées sur le système d'exploitation dans le but d'observer son comportement. Dans cette approche, toutes les séquences de tests sont modélisées puis appliquées au modèle du système d'exploitation. À l'issu de cette vérification, la conformité OSEK/VDX du modèle du système d'exploitation est déterminée et la déduction est faite sur celle du système d'exploitation réel.

Une seconde approche permet la vérification de la conformité du système d'exploitation OSEK/VDX mais cette fois basée sur des observateurs génériques. En effet, toutes les exigences ou propriétés de la spécification OSEK/VDX sont traduites en observateurs. Ces observateurs observent ainsi le comportement du système d'exploitation pour une application donnée et permettent de signaler si la propriété qu'ils traduisent est bien vérifiée par le système d'exploitation.

Organisation du manuscrit

Le manuscrit est organisé comme suit :

— **Le chapitre 1** définit tout d'abord la notion de configuration ainsi que les différents types de configuration à savoir les configurations statique et dynamique. Enfin, les techniques de configuration existantes sont présentées.

— **Le chapitre 2** présente le cas du système d'exploitation Trampoline sur lequel nous appliquons la synthèse formelle.

— **Le chapitre 3** présente le formalisme utilisé pour la modélisation de systèmes d'exploitation et présente les modèles des architectures monocœur et multicœur de Trampoline.

— **Le chapitre 4** présente le processus de synthèse du système d'exploitation ainsi qu'une étude de cas d'une application d'aide à la conduite.

— **Le chapitre 5** présente les deux approches de vérification développées au cours de nos travaux pour la validation de la conformité d'un système d'exploitation selon la spécification OSEK/VDX à partir de modèles formels.

Chapitre 1

Configuration des systèmes d'exploitation

Sommaire

1.1 Définitions **20**
- 1.1.1 Paradigme 20
- 1.1.2 Granularité 21
- 1.1.3 Intégrité 21

1.2 Types de configuration **21**
- 1.2.1 Configuration statique 22
- 1.2.2 Configuration dynamique 24

1.3 Paradigme de développement **25**
- 1.3.1 Programmation orientée objet 26
- 1.3.2 Programmation orientée sujet 28
- 1.3.3 Programmation orientée aspect 29
- 1.3.4 Langages dédiés (DSL) 31

1.4 Conclusion **32**

Ce chapitre présente un état de l'art de la configuration des systèmes d'exploitation. Il présente en effet les différents critères à prendre en compte pendant la configuration d'un système d'exploitation, puis établit les différents types de configuration existants et les étapes auxquelles elles peuvent être réalisées. Enfin, plusieurs techniques de développement de systèmes d'exploitation configurables sont présentées.

1.1 Définitions

La configuration est par définition l'assemblage ou l'organisation d'un ensemble de composants dans le but de former un système complet. Par conséquent, la configuration d'un système d'exploitation permet l'inclusion, l'exclusion ou le remplacement de certaines parties du système d'exploitation.

De cette manière, le système d'exploitation peut être « taillé » de façon à répondre aux exigences de l'application, du matériel ou encore de l'utilisateur [Frö01].

Dans un contexte embarqué, plusieurs raisons motivent le choix du développement de systèmes d'exploitation configurables : il s'agit de l'amélioration de la performance des systèmes embarqués, de l'optimisation de l'empreinte mémoire occupée par les systèmes d'exploitation et de l'accroissement de leurs robustesses en ce qui concerne leurs sûretés de fonctionnement etc.

D'une part, concernant la charge CPU engendrée par le système d'exploitation, des travaux précédents [BRU+03, HSW+00, KLVA93, KTD+13] ont montré que pour des petits systèmes embarqués jusqu'aux systèmes ayant une puissance de calcul élevée, les systèmes d'exploitation non-configurables ou fixes ont un impact négatif sur la performance des applications. Comme exemple, dans [BRU+03] une comparaison basée sur une étude expérimentale entre le noyau Linux et un système d'exploitation hautement configurable pour des architectures multiprocesseurs a été effectuée et les résultats ont montré que le temps de transfert de message entre deux nœuds de calcul avec le système d'exploitation spécifique est 4 à 10 fois supérieur que celui observé avec le noyau Linux. Toutefois, la configuration de systèmes d'exploitation reste une opération délicate car dans bien des cas, lorsqu'elle est mal réalisée, elle peut causer la dégradation de la performance du système d'exploitation en insérant en son sein des *overheads* [Frö01].

D'autre part, l'optimisation de la mémoire occupée par les systèmes d'exploitation permet de satisfaire les contraintes de mémoire auxquelles les systèmes embarqués font face car seuls les services indispensables à l'exécution des applications sont pris en compte dans le système d'exploitation. Cela a aussi pour but de réduire la quantité de code inutilisé (source de comportements erronés) au sein des systèmes d'exploitation qui constitue un vecteur privilégié d'attaques.

Pour le développement de systèmes d'exploitation configurables, plusieurs critères sont à prendre en compte dont le choix du paradigme de développement, le niveau de granularité souhaité lors de la configuration et l'impact de la configuration sur l'intégrité du système d'exploitation. Dans cette section, nous présenterons en détails ces différents critères.

1.1.1 Paradigme

Le paradigme caractérise non seulement la manière dont est structuré le système d'exploitation mais aussi la manière dont l'application doit être développée. Dans le passé, plusieurs paradigmes ont été utilisés pour le développement de systèmes d'exploitation configurables. Cela va des librairies classiques jusqu'aux objets via les modules et les composants. De plus, plusieurs projets de recherches ont mis au point d'autres paradigmes tels que la programmation orientée aspect (AOP [1]), la programmation orientée objet (OOP [2]) ou encore le langage dédié (DSL [3]) que nous déraillerons à la section 1.3.

1. Aspect Oriented Programming
2. Object Oriented Programming
3. Domain Specific Language

1.1.2 Granularité

La granularité définit la finesse de configuration d'un système d'exploitation. Nous pouvons en distinguer deux niveaux à savoir :

- La configuration à *gros grain* : dans ce type de configuration, on ne peut configurer qu'un sous-système du système d'exploitation. Dans un système d'exploitation ayant une structure basée sur les composants, la configuration à gros grain revient à inclure ou non un composant au sein du système d'exploitation en fonction de son utilité. Par exemple, la configuration du système d'exploitation *eCos* [Tho00] est basée sur une configuration à gros grain et se fait à l'aide de *packages*. Le choix de sélection d'un *package* est fait en fonction des besoins de l'application. D'autres systèmes d'exploitation comme *Linux, Scout [Mos97], OSKit [FBB+ 97]* utilisent une configuration à gros grain.
- La configuration à *grain fin* : ce type de configuration utilise un niveau de configuration plus avancée que la précédente car dans ce cas, il est possible de modifier les fonctions du noyau du système d'exploitation, ses données et même son comportement. Comme exemples de systèmes d'exploitation utilisant une configuration à grain fin, nous pouvons citer *MMlite [HF98], Spin [BCE+ 95].*

1.1.3 Intégrité

L'intégrité est un critère crucial pour les systèmes d'exploitation configurables car elle adresse la question de validité d'une configuration donnée. En effet, pendant l'opération de configuration, des erreurs ou comportements indésirables peuvent être introduits dans le système d'exploitation configuré. Pour assurer l'intégrité d'un système d'exploitation lors de sa configuration, il est nécessaire de mettre en place des mécanismes de vérification (fonctionnelle et temporelle) du système d'exploitation.

1.2 Types de configuration

Nous pouvons distinguer deux grandes catégories de configuration, à savoir la configuration statique et la configuration dynamique. Une configuration est définie en fonction de l'étape à laquelle elle intervient dans le cycle de vie d'un système d'exploitation c'est-à-dire à la compilation, à l'édition des liens, à la génération ou à l'exécution du système d'exploitation. La figure 1.1 [Frö01] illustre les types de configuration et les étapes auxquelles elles sont réalisées.

La configuration statique est réalisée avant l'exécution du système d'exploitation soit à la compilation, à l'édition de liens ou à la génération du système d'exploitation. En revanche, la configuration dynamique est réalisée pendant l'exécution du système d'exploitation. Chaque type de configuration présente des avantages mais aussi des inconvénients.

La configuration statique permet une configuration stable et sûre car généralement réalisée avant l'exécution du système d'exploitation, son intégrité peut être vérifiée avant sa mise en exécution. Ce type de configuration n'engendre pas d'*overheads* ou encore permet une meilleure utilisation des ressources disponibles puisque tous les besoins en terme de services ou de ressources sont spécifiés avant l'exécution du système d'exploitation.

Par contre lorsque le système d'exploitation présente un caractère dynamique c'est-à-dire nécessite de nouvelles fonctionnalités au cours de son exécution, la configuration statique devient inapplicable. C'est pourquoi la configuration dynamique présente un intérêt car elle permet un changement ou une extension des services pendant l'exécution du système d'exploitation en fonction de la demande de l'application.

Le principal inconvénient d'une telle configuration est qu'elle peut favoriser l'insertion de *bugs* ou encore du code indésirable dans le noyau du système d'exploitation ce qui n'est pas tolérable dans les systèmes critiques.

La configuration statique est plus employée pour des systèmes d'exploitation utilisés dans le domaine des systèmes embarqués critiques où la performance et la stabilité sont des critères important tandis que la configuration dynamique est plutôt utilisée pour les systèmes d'exploitation généraux.

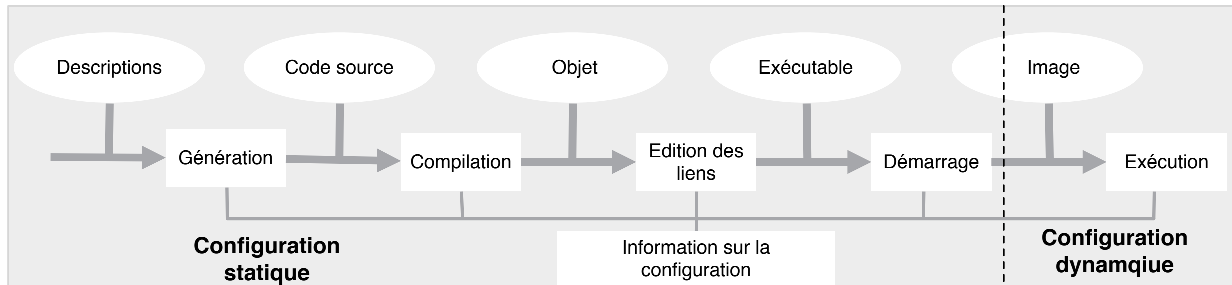


FIGURE 1.1 – Illustration des différents types de configuration [Frö01]

1.2.1 Configuration statique

Réalisée avant l'exécution du système d'exploitation, tous les critères de sélection permettant de connaître les différentes fonctionnalités à inclure sont établis à partir des exigences des applications. Lorsque ces critères ne sont pas bien spécifiés, cela conduit à une insertion dans le système d'exploitation de fonctionnalités inutiles ou encore à l'omission de fonctionnalités utiles. Traditionnellement basée sur une configuration de type gros grain, la configuration statique est réalisée manuellement. En d'autres termes, les services requis sont sélectionnés par le développeur d'application via un fichier de configuration. Une telle pratique peut favoriser l'inclusion de code inutile au sein du système d'exploitation. Ainsi plusieurs techniques de configuration statique ont été développées. Ces techniques s'appliquent à l'édition de liens, à la compilation ou à la génération du système d'exploitation.

1.2.1.1 Génération du code source

La configuration statique est possible à la génération du code source du système d'exploitation grâce à des outils de configuration. Dans ce cas, les outils sont essentiellement conçus pour générer une partie du code source du système d'exploitation à partir d'une spécification de haut niveau décrivant l'ensemble des fonctionnalités exigées par l'application.

Par exemple, le système d'exploitation PURE [SSPSS99], conçu pour des systèmes embarqués distribués (par ex. systèmes automobiles) opérant sous une extrême contrainte de ressource en terme de mémoire, CPU et de consommation d'énergie, supporte une configuration statique réalisée grâce à un outil de configuration.

Dans [Bau99], l'auteur propose une approche permettant de générer un système d'exploitation « taillé sur mesure » à partir de composants logiciels génériques. Les composants génériques sont des modules logiciels ou des blocs de construction préfabriqués fournissant des paramètres génériques par ex. des valeurs entières dénotant la taille d'une mémoire tampon.

Les valeurs des paramètres génériques sont choisies au niveau de l'application de manière à pourvoir définir l'ensemble des fonctionnalités requises par celle-ci. Un générateur de code prenant en entrée les valeurs des paramètres génériques et les différents composants logiciels génériques permet la génération du système d'exploitation spécifique à l'application mais nécessite une intervention humaine car elle n'est pas totalement automatisée.

Dans [FSP99, Frö01] une méthode de génération du système d'exploitation EPOS spécifique à l'application est présentée. L'application implémentée est analysée dans un outil afin de détecter tous les services nécessaires à l'exécution du système d'exploitation. Les informations produites par l'analyseur sont introduites au sein d'un configurateur qui rassemble l'ensemble des blocs de services utiles qui serviront enfin à la génération de code.

1.2.1.2 Compilation

À la compilation, la configuration statique se fait essentiellement à l'aide du préprocesseur C.

Cette technique permet de sélectionner à la compilation, par le biais de directives de préprocesseur (`#if, #ifdef, #endif` etc.) insérées dans le code source du système d'exploitation, les sections de code qui seront compilées ou la manière dont le code sélectionné devra être compilé.

Il existe en effet des outils comme *GNU autoconf* [MED06] ou *X11 imake* [DuB96] qui sont utilisés dans ce cadre.

Le concept de la compilation conditionnelle pour la configuration de systèmes d'exploitation suscite des opinions divergentes car, certains résultats de recherches montrent que cette technique conduit à une utilisation abusive des directives de préprocesseur.

Cela a pour effet, une illisibilité du code source du système d'exploitation [PPD+95, LHSP+09] qui devient ainsi difficile à maintenir. Par contre d'autres résultats de recherches montrent que lorsque la compilation conditionnelle est bien utilisée, elle n'engendre aucun *overhead* dans le système d'exploitation configuré [Cop99].

La configuration statique à la compilation est réalisée soit par une extension de langages existants ou par un nouveau langage. Tous les objets du système d'exploitation qui sont utiles à la conception de l'application sont spécifiés dans un fichier grâce à un langage donné. Ainsi à la compilation, le fichier est analysé dans le but de charger dans le système d'exploitation l'ensemble des services permettant la manipulation des objets précédemment décrit.

C'est le cas du système d'exploitation Trampoline [BBFT06] qui utilise le langage OIL (OSEK Implementation Language) pour décrire dans un fichier (*fichier OIL*), tous les objets de l'application (par ex. ressources, alarmes, tâches etc.) qui seront manipulés par le système d'exploitation pendant son exécution. Enfin, un compilateur (*GOIL*) sélectionne en fonction de la description faite, tous les services du système d'exploitation et les sections de code qui seront utiles à l'exécution de l'application.

La configuration menée dans Trampoline opère comme une configuration à *gros grain*. En effet, pour chaque objet défini, il existe un ensemble de services qui lui est associé et qui permet sa manipulation. Lorsqu'un objet est déclaré dans le *fichier OIL*, tous les services liés à cet objet sont automatiquement pris en compte dans le système d'exploitation. L'inconvénient se trouve dans le fait qu'il est impossible de sélectionner un sous ensemble des services qui sont liés à un objet dans le cas où ces services liés à cet objet ne sont pas tous utiles à l'application. Il en résulte ainsi une quantité non négligeable de code mort à l'intérieur du système d'exploitation qui impacte négativement sa robustesse et son empreinte mémoire.

Dans le système d'exploitation MARS [KFG+93], le langage MODULA/R est utilisé pour supporter sa configuration statique.

Dans [KTD+13, TKH+12], une approche permettant la configuration statique du noyau Linux est présentée. Dans cette approche, l'accent est mis sur la sécurité et la stabilité du noyau Linux après l'opération de configuration. Par cette approche, l'auteur montre à la fois que seul le code nécessaire à l'exécution de l'application est fourni mais aussi que le noyau configuré est beaucoup plus robuste.

Cette approche est ainsi réalisée en deux grandes phases. Une première phase consiste à analyser pendant l'exécution du noyau, l'ensemble des modules qui ont été chargés et le code qui a été véritablement exécuté. Suite à cette analyse, dans une seconde phase, une configuration optimale décrivant les services à désactiver ou à prendre en compte est ensuite générée. À partir des informations fournies par cette configuration, seuls les modules et services utilisés sont chargés dans le système d'exploitation à la compilation.

1.2.1.3 Édition de liens

À l'édition des liens, la configuration statique est possible par la sélection d'objets précompilés à partir d'une librairie ou d'un répertoire.

Les objets ayant des structures rigides, certains objets sélectionnés pour l'édition de liens peuvent renfermer des services qui ne seront pas utilisés pendant l'exécution du système d'exploitation et cela reste un inconvénient non négligeable.

Le système d'exploitation PEACE [SP94] supporte la configuration statique de son noyau à l'édition de liens en isolant automatiquement chaque classe de services dans des fichiers objets. Ainsi, un éditeur de liens prend le soin d'inclure les services sélectionnés dans l'exécutable. Le projet HARMONY [GoEE89] du conseil national de recherches du Canada utilise une table de configuration statique pour décrire l'ensemble des fichiers objets à inclure dans le système d'exploitation.

Le système d'exploitation OSKit [FBB+97] utilise un ensemble de composants organisés en bibliothèques. Lors de la configuration, l'utilisateur choisit parmi les bibliothèques et fichiers objets ceux qui ont été traités par l'éditeur de liens. Une nouvelle version du système d'exploitation OSKit [RFS+00] utilise un langage pour décrire des composants binaires qui permet de contrôler l'édition de liens afin de surmonter certaines restrictions imposées par les bibliothèques ordinaires et les éditeurs de liens.

1.2.2 Configuration dynamique

Un système d'exploitation est configurable dynamiquement lorsque ses fonctionnalités peuvent être changées ou étendues pendant son exécution. Dans le cas d'un système critique, ce type de configuration est moins approprié.

Les systèmes d'exploitation concernés par ce type de configuration sont les systèmes d'exploitation généraux tels que Windows, Linux, Solaris. Dans ce type de systèmes d'exploitation, il est impossible de savoir à l'avance c'est-à-dire avant le démarrage du système d'exploitation, tous les besoins des applications en terme de services. Dans ce cadre, il existe plusieurs approches traitant le cas de la configuration dynamique des systèmes d'exploitation.

1.2.2.1 Création dynamique de processus

La configuration dynamique peut être réalisée grâce à la création dynamique de processus. Dans une telle technique, certaines fonctionnalités du système d'exploitation sont implémentées en dehors du noyau puis des processus sont créés dynamiquement.

Ces processus sont ainsi choisis pour être exécutés lorsque la fonctionnalité qu'ils implémentent est requise par l'application. Ces processus peuvent être implémentés en partageant le même espace d'adressage que le système d'exploitation et s'exécuter en mode superviseur ou, comme des processus ordinaires s'exécuter en mode utilisateur.

Il existe un nombre important de systèmes d'exploitation procédant de cette manière pour réaliser la configuration dynamique notamment le noyau UNIX [RT78] qui permet de démarrer pendant son exécution certains services à la demande de l'application. Le système d'exploitation V-KERNEL [Che84] supporte une configuration dynamique en admettant la création et la destruction dynamique de processus. Le système d'exploitation CHORUS [RAA+91] utilise une configuration dynamique en permettant la création dynamique de processus s'exécutant en mode superviseur à l'intérieur de l'espace d'adressage réservé au noyau.

1.2.2.2 Extension du noyau

La configuration dynamique est réalisable par l'extension dynamique du noyau du système d'exploitation à travers plusieurs approches dont l'édition de liens dynamique, la compilation à la volée ou encore l'interprétation.

Lors de l'édition de liens dynamique, des modules précompilés sont liés au noyau. Ainsi, lors de la tentative d'accès à un module qui n'a pas été précédemment pris en compte dans l'édition de liens, un éditeur de liens dynamique intégré est invoqué afin de lier le module [DSS90, Dra93].

En ce qui concerne les techniques utilisant la compilation à la volée ou l'interprétation, le code source correspondant au module à insérer est extrait puis compilé ou interprété dans le noyau.

La technique utilisant l'édition de liens dynamique présente un avantage pour la performance du système d'exploitation car, en général, les compilateurs ont un temps de démarrage assez élevé et les interpréteurs doivent réinterpréter le code correspondant à chaque fois qu'ils sont invoqués. Par contre, assurer la sûreté de la technique de l'édition de liens dynamique est beaucoup plus compliqué [SESS96].

Dans [SS95] le système d'exploitation VINO utilise une configuration basée sur l'extension du noyau. Cela permet d'obtenir un noyau extensible puis réutilisable pour une large gamme d'applications. Mais, l'insertion de modules dans le noyau peut impacter négativement son intégrité car le code correspondant au module ajouté peut engendrer un *bug* dans le noyau. Pour cela, VINO utilise les techniques dites *d'isolation de fautes logicielles* tel que le *sandboxing* [WLAG93] qui permet d'écrire des extensions (des modules) dans un langage tel que le C.

Le *sandboxing* permet ainsi de tester si les différents modules ajoutés au noyau respectent bien l'espace d'adressage qui leur ont été attribué afin d'éviter des comportements inattendus au sein du noyau.

Le système d'exploitation SPIN [BSP+95] définit un noyau et un ensemble d'extensions chargeables dynamiquement écrit en MODULA/3. Un éditeur de liens intégré au noyau est invoqué à chaque fois qu'une application sollicite un nouveau service afin de charger l'extension qui l'implémente.

1.3 Paradigme de développement

Dans un souci de rendre les systèmes d'exploitation configurables, plusieurs approches adoptent le principe de la programmation modulaire. Cette technique permet le découpage du système en

sous parties où chaque sous partie représente un composant qui peut être inséré ou non dans le système d'exploitation en fonction de son utilité.

Dans cette sous section, nous présentons plus en détails les différentes techniques utilisées dans le développement des systèmes d'exploitation configurables.

1.3.1 Programmation orientée objet

Une approche permettant le développement de systèmes d'exploitation configurables est celle utilisant la programmation orientée objet. Un système d'exploitation conçu au moyen d'une telle technique est appelé *un système d'exploitation orienté objet.*

Ces systèmes d'exploitation sont attractifs car ils ont une structure modulaire qui permet facilement leur maintenance, leur portabilité ou encore leur extension.

En général, un système d'exploitation orienté objet est décomposé en plusieurs objets qui abstraient des fonctionnalités données. Ces objets forment ainsi des sous systèmes indépendants qui interagissent entre eux.

Dans le passé, plusieurs techniques de développement issues de la programmation orienté objet ont été proposées.

Le développement du système d'exploitation *Choices* [CJMR91] est basé sur la hiérarchie des classes et le principe de l'héritage. En effet, les composants du système d'exploitation tels que l'ordonnanceur, la gestion des fichiers etc. sont définis par des classes abstraites (super classes). De plus, des classes plus concrètes (sous classes) issues des classes abstraites suivant le principe de l'héritage sont définies dans le but de spécialiser des services déjà existants ou encore d'ajouter de nouveaux services.

Lors de la configuration du système d'exploitation, le développeur peut choisir parmi un ensemble de classes, celles qui seront parfaitement adaptées aux besoins de l'application ou du matériel. De plus, les classes peuvent être disponibles au niveau de l'application afin de pouvoir les spécialiser en fonction des besoins de l'utilisateur grâce au principe de l'héritage et du polymorphisme [ABB$^+$93].

À titre d'exemple, considérons la version mono-cœur du système d'exploitation Trampoline qui présente une structure monolithique. Dans cette version de Trampoline, il existe deux traitements possibles lors de l'ordonnancement des tâches applicatives. Le choix d'un traitement se fait par le biais de la compilation conditionnelle en fonction de la taille de la liste des tâches prêtes implémentée dans Trampoline comme une table de FIFO.

D'une part, lorsque la taille de la table de FIFO est une puissance de 2, un premier ordonnanceur spécifique est utilisé pour sa gestion pour des raisons de performance. D'autre part, lorsque la taille de la table de FIFO est quelconque, un second ordonnanceur plus générique est utilisé pour sa gestion.

En utilisant la programmation orientée objet et les principes de l'héritage et du polymorphisme, l'ordonnanceur pourrait être représenté comme une classe abstraite qui définit l'interface de ses sous classes (voir figure 1.2).

Grâce au polymorphisme, les fonctions héritées de la classe abstraite auront une exécution différente en fonction du type d'ordonnanceur sélectionné. L'extension de l'ordonnanceur pourrait se faire par la définition d'autres sous classes puis la configuration reviendrait à ne sélectionner que les ordonnanceurs nécessaires au fonctionnement de l'application.

Dans [Beu97], un langage appelé ALC et la programmation orientée objet sont utilisés pour le développement d'un système d'exploitation configurable. D'une part, le langage permet la définition et l'instance *d'attributs* où chaque attribut décrit une fonctionnalité ou un composant du système d'exploitation. D'autre part, un lien est établi entre les attributs et les classes définies

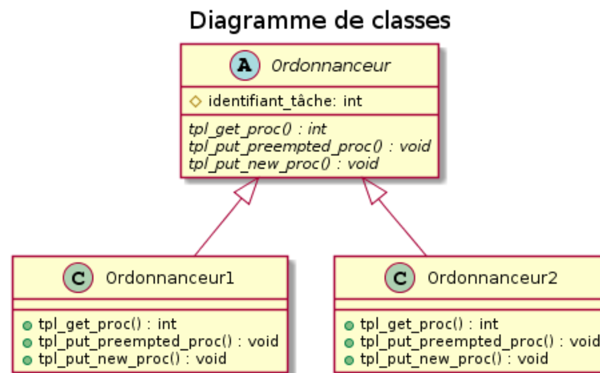


FIGURE 1.2 – Représentation de l'ordonnanceur de Trampoline sous forme de diagramme de classes

en C++ en annotant le code source du système d'exploitation via des commentaires (chaque classe est annotée par un attribut). Le code source du système d'exploitation est ainsi analysé par un outil appelé *analyser*. Cet outil génère un répertoire permettant de chercher l'ensemble des fonctionnalités nécessaires en fonction des exigences de l'application grâce à un configurateur prenant en entrée ce répertoire et les informations nécessaires à la configuration. Ce configurateur génère ainsi un ensemble de *makefiles* qui décrivent les fichiers source à compiler puis la manière dont ils doivent être compilés. Enfin, grâce à une commande `make` le système d'exploitation est généré. Malgré le fait qu'elle soit statique, cette configuration ne bénéficie d'aucun moyen de vérifier l'intégrité ou la correction du système d'exploitation configuré.

[Dru93] propose une méthode de configuration basée sur deux stratégies. La première stratégie concerne l'utilisation de la programmation orientée objet qui permet de structurer le système d'exploitation en objets dans le but de lui donner une structure modulaire. La seconde stratégie consiste à transférer un certain nombre de services vers le domaine de l'application. Cette approche permet l'obtention d'un noyau qui ne contient qu'un ensemble minimal de fonctions et d'un système d'exploitation où les services peuvent être remplacés ou modifiés à partir de l'application. La spécialisation du système d'exploitation est aussi basée sur les principes de l'héritage et du polymorphisme.

Le système d'exploitation PURE [SSPSS99] (Portable Universal Runtime Executive) fut principalement conçu pour les systèmes embarqués à ressources limitées. Les systèmes à ressources limitées sont ceux qui présentent de fortes contraintes au niveau de l'occupation de la mémoire, du processeur ou de l'énergie disponible. Le but des concepteurs de PURE était de fournir un système d'exploitation qui soit hautement configurable afin que le développeur d'application sélectionne parmi les services disponibles ceux qui sont réellement nécessaires. L'approche utilisée pour la conception de PURE est d'abord de considérer le système d'exploitation comme une *famille de programmes* [Par78] et d'utiliser la programmation orientée objet pour son implémentation. D'une part, le principe de *famille de programmes* fournit une conception dans laquelle un sous ensemble minimal d'un système de fonctions est utilisé comme une plateforme permettant d'implémenter les extensions minimales du système. Ces extensions sont ainsi établies quand il est nécessaire d'implémenter une nouvelle fonctionnalité supportée par l'application. D'autre part, la programmation orientée objet permet de concevoir un système d'exploitation hautement modulaire. PURE est ainsi représenté comme une bibliothèque de classes où chaque classe décrivant une fonctionnalité donnée peut être configurée grâce aux techniques de l'héritage et du

polymorphisme afin de satisfaire les besoins de l'application.

1.3.2 Programmation orientée sujet

Introduit par W. Harrison et H. Ossher [OKH+95, HO93], la programmation orientée sujet est une extension de la programmation orientée objet. Le principe fondamental de la programmation orientée sujet consiste à avoir une *vision subjective* d'un même objet c'est-à-dire qu'un objet peut avoir différentes propriétés ou peut être défini de différentes manières en fonction d'un contexte donné.

Comme exemple, considérons la figure 1.3. Pour cet exemple, nous supposons qu'une tâche peut être représentée d'une part comme une entité définie par une priorité et un type et d'autre part comme une entité définie par son état. Cette vision subjective permet la définition de plusieurs classes issues de la même tâche. Ainsi, un sujet est par définition une collection de classes séparées modélisant une vision subjective d'un même objet.

En programmation orientée sujet, les sujets peuvent être composés en vue de former un sujet regroupant toutes les propriétés des sujets composés.

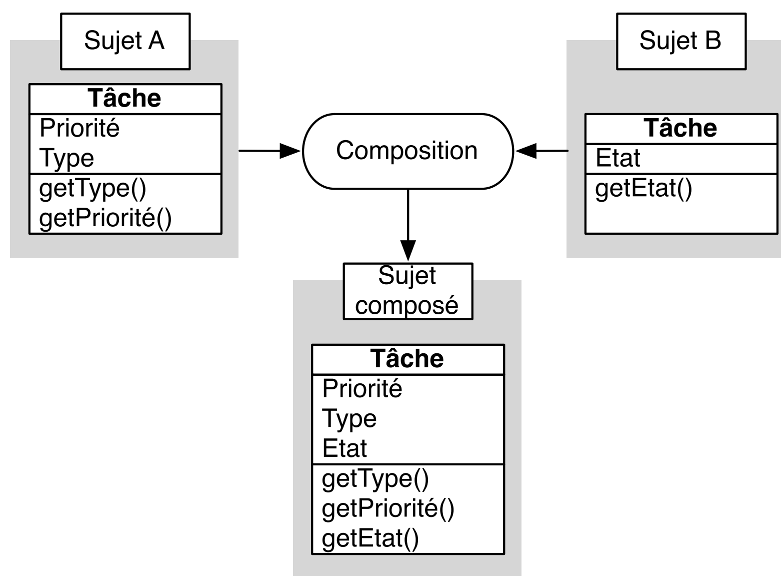


FIGURE 1.3 – Composition de sujets

De cette manière, la programmation orientée sujet permet le développement décentralisé et l'extension d'un système sans accéder à son code source. Dans [BC09], une méthode de développement de systèmes d'exploitation flexibles basée sur la programmation orientée sujet est présentée. Chaque module du système d'exploitation est représenté par un sujet qui décrit un certain nombre de fonctionnalités du système d'exploitation. Par la suite, le système d'exploitation est conçu par la composition des différents sujets. En fonction des exigences de l'application ou du matériel, les sujets indispensables sont sélectionnés puis composés dans le but de former le système d'exploitation complet.

L'insertion de nouvelles fonctionnalités est réalisée par l'écriture d'un nouveau sujet décrivant les fonctionnalités requises puis par composition de celui-ci avec les sujets existants. Cette approche permet le développement d'un système d'exploitation configurable, extensible et réuti-

lisable. Par contre, elle n'est applicable que sur des systèmes d'exploitation ayant une structure basée sur les sujets. Par conséquent, elle ne peut être utilisée sur des systèmes d'exploitation existants présentant une structure différente.

1.3.3 Programmation orientée aspect

Proposée par G. Kiczales et al [KLM+97], la programmation orientée aspect est un paradigme de programmation qui a été développé pour améliorer la programmation orientée objet sur certains points. La principale idée de ce paradigme est d'améliorer la séparation de préoccupations (*separation of concerns*) lors du processus de développement du logiciel.

Ces préoccupations forment les différents aspects techniques du logiciel, habituellement fortement dépendants entre eux. La programmation orientée aspect propose des mécanismes de séparation de ces aspects, afin de modulariser le code, le décomplexifier, et le rendre plus simple à modifier.

La programmation orientée aspect n'est pas particulièrement liée à un langage de programmation mais peut être mise en œuvre aussi bien avec un langage orienté objet comme le langage JAVA qu'avec un langage procédural comme le langage C.

Comme mentionné précédemment, la programmation orientée objet offre une structure modulaire aux systèmes logiciels dans le but de rendre ceux-ci configurable et réutilisable. Par contre, les modules formés suite au processus de modularisation contiennent des portions de code dont la représentation en objet est impossible. Communément appelées dans la littérature américaine *cross-cutting concerns*, ces portions de code implémentent les aspects non fonctionnels ou fonctionnalités transverses des systèmes logiciels comme le *logging*, le *monitoring*, la *gestion des exceptions*, le *débogage*. Il existe par conséquent plusieurs inconvénients liés à ces aspects dont la *dispersion de code* qui est due à la propagation des fonctionnalités transverses un peu partout dans les modules et à la confusion du code qui est due à la présence au sein d'un même module implémentant une fonctionnalité du système logiciel de diverses fonctionnalités transverses.

Ces aspects impactent négativement les systèmes logiciels parmi lesquels nous notons :

— Un code difficilement réutilisable : un module comportant plusieurs fonctionnalités transverses est difficilement réutilisable tel quel dans un autre système nécessitant ce module.
— Une diminution de la qualité du code : l'apparition dispersée des fonctionnalités transverses dégrade la qualité du code.
— Une baisse de la productivité du code : la redondance de fonctionnalités transverses au sein d'un module empêche le développeur de se concentrer sur la fonctionnalité principale de ce module.
— Une diminution de l'évolution du code : pour faire évoluer un système, il faut modifier de nombreux modules. Modifier chaque sous systèmes pour atteindre les modifications souhaitées peut entrainer des incohérences.

La figure 1.4 montre l'exemple d'un système formé de 4 modules dans lesquels apparaissent des fonctionnalités transverses ou *cross-cutting concerns* lors de la modularisation par la programmation orientée objet. Un autre inconvénient observé sur la figure concerne la possibilité de dupliquer certaines fonctionnalités transverses dans des modules où elles ne sont pas réellement nécessaires.

La programmation orientée aspect offre une solution à ce problème de modularisattion en permettant d'isoler les fonctionnalités transverses dans des *aspects*. Selon Georges Kiczales [KLM+97], lors de l'implémentation d'un système logiciel, les fonctionnalités qui peuvent être clairement encapsulées en utilisant la programmation orientée objet sont des *composants* tandis

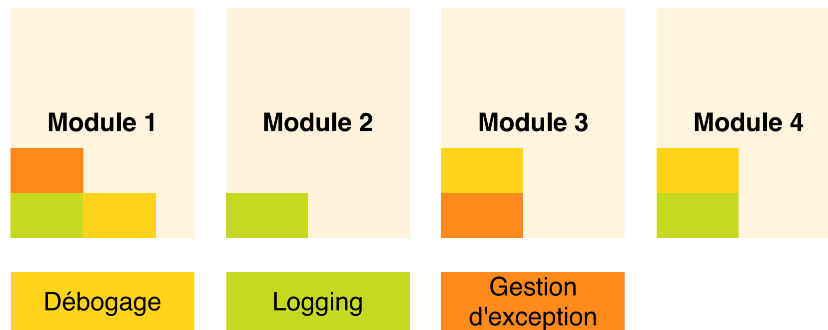


FIGURE 1.4 – Modularisation selon la programmation orientée objet

que celles qui ne peuvent être représentées par des objets (fonctionnalités transverses) sont des *aspects*.

La figure 1.5 montre le principe de développement de systèmes logiciels par le biais de la programmation orientée aspect. Trois étapes majeures sont ainsi identifiées à savoir :

— La décomposition des éléments du système : les fonctionnalités facilement identifiables par des objets constituent des composants et celles qui sont difficilement identifiables constituent des aspects.
— L'implémentation des sous systèmes : chaque sous système est implémenté soit dans des modules en utilisant un langage procédural ou un langage orienté objet, soit dans des aspects en utilisant des langages spécifiques à la programmation orientée aspect.
— l'intégration du système : les composants et les aspects sont composés à partir d'un métier à tisser ou *weaver* dans le but de former le système complet.

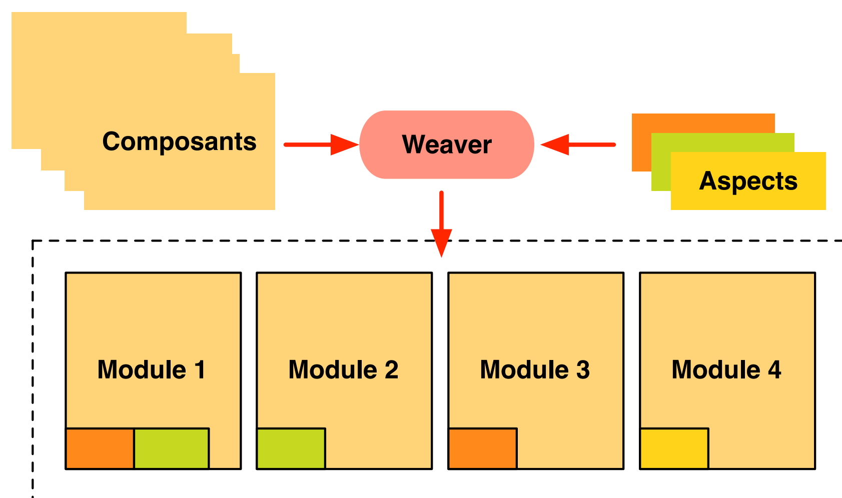


FIGURE 1.5 – Modularisation selon la programmation orientée aspect

La programmation orientée aspect offre alors une modularisation claire et concise tout en améliorant la qualité du code [Lad03] et la réutilisation des systèmes logiciels.

Dans le domaine des systèmes d'exploitation, de nombreux travaux ont proposé des approches de développement de systèmes d'exploitation configurables liées à la programmation orientée

aspect [LHSP+09, LHSPS11, LGS04, CKFS01, CKF00, SL04, MSGSP02].

[LGS04] présente une approche de développement de systèmes d'exploitation adaptables. Selon les auteurs, pour les petits systèmes embarqués (téléphones mobiles, montres connectées), les applications sont la plupart du temps dynamiques c'est-à-dire que leur besoin en termes de services peut varier à l'exécution. Par conséquent, le système d'exploitation doit être conçu de manière à s'adapter dynamiquement aux exigences des applications. Pour cela, la notion d'aspect dynamique est introduite. Ces aspects encapsulent les propriétés non fonctionnelles des systèmes d'exploitation et peuvent être chargés ou supprimés dynamiquement selon l'application grâce à un *tisseur dynamique* ou *dynamic weaver*. Par contre, l'adaptation n'est possible qu'après la suspension du système d'exploitation qui au redémarrage contiendra les fonctionnalités souhaitées. Aussi, aucun mécanisme permettant de vérifier la bonne intégration de nouvelles fonctionnalités durant l'exécution du système d'exploitation n'est présenté.

Dans [LHSP+09] le système d'exploitation CiAO implémenté en AspectC++ [SL07] qui est une extension du langage C++ par les concepts de la programmation orientée aspect est présenté. L'intérêt est porté aux systèmes embarqués à ressources limitées car en général, pour ce type de système, la configuration des systèmes d'exploitation est réalisée par la compilation conditionnelle via des directives de préprocesseur. Il en résulte ainsi un nombre important de déclarations `#ifdef, #if #endif` au sein du code source qui impactent la qualité du code du système d'exploitation. La solution proposée par les auteurs consiste à concevoir entièrement un système d'exploitation basée sur les principes de la programmation orientée aspect. Le système d'exploitation développé ne contient alors pas de déclaration `#ifdef, #if, #endif`, est hautement configurable, présente un code dont la qualité est améliorée et une structure modulaire.

1.3.4 Langages dédiés (DSL)

Le langage dédié ou DSL (*Domain Specific Language*) est un langage de programmation déclaratif qui traite d'un domaine bien précis contrairement aux langages généralistes comme le C, C++ ou Java qui traitent un ensemble de domaines. La conception d'un langage dédié nécessite une bonne expertise du domaine dans lequel il sera appliqué. Ce langage peut être vu comme un langage de spécification de haut niveau.

L'avantage d'un langage dédié est qu'il offre la possibilité d'établir des procédures de vérification au niveau de la spécification du système à développer dans le but d'éviter d'éventuels bugs.

Le langage dédié n'est pas directement exécutable sur les architectures matérielles. Il est généralement traduit dans un langage généraliste via un compilateur dédié (figure 1.6). Le passage par un langage dédié permet le développement d'un système très spécialisé en fonction d'un contexte donné.

Dans les systèmes embarqués, le langage dédié est utilisé pour le développement de pilotes de périphériques spécifiques [CE04, MRC+00] ou encore pour le développement d'ordonnanceurs temps réel spécifiques aux applications [BM02].

FIGURE 1.6 – Transformation d'un langage dédié en langage généraliste

En ce qui concerne la spécialisation des systèmes d'exploitation, le langage dédié est peu utilisé mais reste une piste intéressante car il permettrait de générer entièrement et automatiquement un système d'exploitation spécifique à l'application qui peut être vérifié fonctionnellement.

Dans [PBC+97] une approche pour la spécialisation de systèmes d'exploitation à partir d'un micro-langage est proposée. Le micro-langage intervient à plusieurs niveaux d'abstraction. Il est non seulement utilisé au niveau de l'application mais aussi au niveau du système d'exploitation. Le programme écrit dans un micro-langage est appelée *micro-programme*.

Au niveau de l'application, le micro-programme décrit les besoins de l'application, son comportement et la manière dont elle effectue des appels de services fournis par le noyau du système d'exploitation. Ce micro-programme est ensuite compilé via un *micro-compilateur* qui contient les informations sur les services du système d'exploitation. À la suite de la compilation, un second micro-programme est généré. Ce micro-programme décrit la stratégie de sélection des services indispensables à l'application. Il est enfin interprété par un *micro-moteur* qui génère l'ensemble des services requis par l'application. La figure 1.7 décrit de manière générale l'approche proposée.

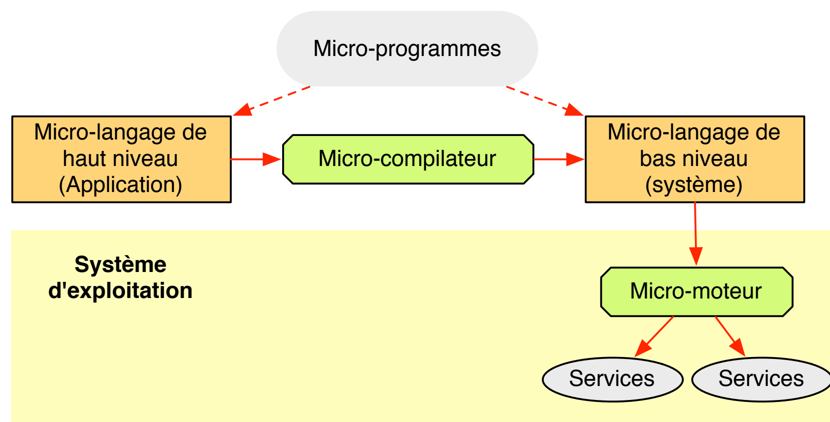


FIGURE 1.7 – Spécialisation d'un système d'exploitation par un langage dédié [PBC+97]

1.4 Conclusion

Dans ce chapitre, nous avons défini la notion de configuration des systèmes d'exploitation. Plusieurs approches sur le développement de systèmes d'exploitation ont également été proposées. Ces approches visent à offrir une structure modulaire aux systèmes d'exploitation dans le but de faciliter leur configuration, leur réutilisation et leur évolution en fonction des exigences des applications. Elles sont intéressantes pour tout concepteur souhaitant développer à partir de rien un système d'exploitation configurable.

Cependant, ces approches ne résolvent pas intégralement le problème de l'élimination de code mort. Cela présente un inconvénient majeur car la présence de code mort peut fragiliser la robustesse des systèmes d'exploitation. Un autre inconvénient réside dans le moyen de pourvoir vérifier la correction du code source du système d'exploitation après le processus de configuration. Ces approches ne disposent pas de moyens formels permettant de vérifier le système d'exploitation configuré.

Pour aborder ces différents problèmes, nous proposons dans les deux prochains chapitres de ce manuscrit, une approche de configuration de systèmes d'exploitation statiques à partir de

modèles formels. Cette approche à l'avantage d'effectuer une configuration agressive du système d'exploitation car elle prend en compte l'élimination de code mort afin de ne charger dans le système d'exploitation que du code réellement nécessaire à l'application. Aussi, étant basée sur une modélisation formelle du système d'exploitation, cette approche permet d'effectuer une vérification formelle du système d'exploitation configuré en utilisant la technique du *model-checking*.

Chapitre 2

Trampoline, un système d'exploitation temps réel

Sommaire

2.1 Le standard OSEK/VDX **36**
2.1.1 Architecture du système d'exploitation OSEK 36
2.1.2 Gestion des tâches 37
2.1.3 Politique d'ordonnancement 38
2.1.4 Les modes d'application 40
2.1.5 Traitement des interruptions 40
2.1.6 Mécanisme des événements 41
2.1.7 Gestion des ressources 42
2.1.8 Alarmes 45
2.2 AUTOSAR : un standard d'architecture logicielle **46**
2.2.1 Architecture logicielle 46
2.2.2 Configuration dans AUTOSAR 46
2.3 Trampoline **46**
2.3.1 Présentation 46
2.3.2 Architecture de Trampoline monocœur 48
2.3.3 Architecture de Trampoline multicœur 51
2.4 Conclusion **52**

Dans ce chapitre, nous présentons tout d'abord les standards OSEK/VDX et AUTOSAR qui proposent des règles d'implémentation de systèmes d'exploitation temps réel destinés au domaine de l'automobile. Ensuite, nous présentons le système d'exploitation Trampoline qui est implémenté suivant ces standards. Dans cette présentation de Trampoline, nous décrivons d'une part l'architecture de sa version monocœur et d'autre part l'architecture de sa version multicœur. Nous nous intéressons à Trampoline car notre travail de thèse se base sur sa modélisation.

2.1 Le standard OSEK/VDX

Initialement, OSEK a été fondé en 1993 par un consortium de constructeurs et équipementiers automobiles allemands dans le but de développer une architecture ouverte reliant les différents contrôleurs électroniques d'un véhicule. Ensuite, en 1994, le consortium a été rejoint par un groupe de constructeurs automobiles français qui présentaient des projets similaires (VDX pour *Vehicle Distributed eXecutive*). Cette fusion a abouti à la création du standard OSEK/VDX.

L'architecture ouverte présentée par OSEK/VDX est composée de trois parties dont la communication, la gestion du réseau et le système d'exploitation. Dans cette section, nous nous intéresserons particulièrement au système d'exploitation OSEK.

2.1.1 Architecture du système d'exploitation OSEK

La spécification du système d'exploitation OSEK [G+05] présente pour tout système d'exploitation qui l'implémente, un standard permettant une utilisation efficace des ressources matérielles pour une application logicielle de l'unité de contrôle électronique (*ECU*). Le système d'exploitation OSEK/VDX permet ainsi de répondre aux contraintes sévères présentes dans les systèmes embarqués automobiles (par ex. les contraintes de mémoire et les contraintes temps-réel). Il aide également à la portabilité de différents modules constituant la partie logicielle de l'application.

2.1.1.1 Niveaux de traitement

Pour les systèmes d'exploitation, OSEK définit trois niveaux de traitements distincts. Nous pouvons citer :

- Les interruptions.
- L'ordonnanceur.
- Les tâches.

Pour chaque niveau de traitement, une priorité est définie. Pour les tâches, la priorité est définie par l'utilisateur tandis que pour les interruptions et l'ordonnanceur, les priorités sont prédéfinies. La priorité des interruptions est supérieure à la priorité de l'ordonnanceur qui est elle-même supérieure à la priorité des tâches.

2.1.1.2 Classes de conformité

Le système d'exploitation OSEK est conçu pour être configurable dans le but d'être utilisé pour une large gamme d'applications qui peuvent être plus ou moins complexes.

Cette configurabilité permet ainsi au système d'exploitation de n'embarquer que les services qui sont réellement nécessaires aux applications. Pour cela, OSEK a introduit la notion de *classe de conformité* dont l'objectif est de proposer un niveau de fonctionnalités croissant pour les systèmes des plus simples au plus complexes. Il existe en effet 4 différents niveaux :

- **BCC1** : uniquement des tâches de base, une tâche par niveau de priorité et une seule demande d'activation par tâche.
- **BCC2** : BCC1, plusieurs tâches par niveau de priorité et plusieurs demandes d'activation par tâche.
- **ECC1** : BCC1 et des tâches étendues.
- **ECC2** : ECC1, plusieurs tâches par niveau de priorité et plusieurs demandes d'activation par tâche.

De cette manière, une classe de conformité de type BCC1 nécessitera moins de ressources qu'une classe de conformité de type BCC2.

2.1.2 Gestion des tâches

2.1.2.1 Le concept de tâche dans OSEK

Une tâche est un programme s'exécutant de façon séquentielle et encapsulé dans des méthodes ou fonctions dans le langage de programmation utilisé pour le développement de l'application. Le système d'exploitation permet une exécution concurrente et asynchrone des tâches. Deux types de tâches sont fournis par le système d'exploitation OSEK.

2.1.2.2 Tâche de base

Les tâches de base libèrent le processeur uniquement lorsqu'elles terminent leur exécution, préemptées par une autre tâche plus prioritaire ou lors d'une occurrence d'interruption. Ainsi, les points de synchronisation se situent au début et à la fin des tâches. Ces tâches ne peuvent pas effectuer un appel de services pouvant provoquer leur mise en attente, elles possèdent trois états : *suspendue, prête, en exécution.*

2.1.2.3 Tâche étendue

À la différence des tâches de base, les tâches étendues peuvent être mises en attente suite à un appel de services bloquant (par ex. `WaitEvent`), elles ont donc un état supplémentaire (*en attente*). La gestion des tâches étendues est plus complexe que celle des tâches de base et nécessite plus de ressources.

2.1.2.4 Diagramme d'état d'une tâche

Le nombre d'état que peut occuper une tâche varie selon son type. Les figures 2.1 et 2.2 décrivent respectivement les diagrammes d'états d'une tâche étendue et d'une tâche de base. Les différents états possibles sont :

- **En exécution** : dans cet état, le processeur est attribué à la tâche afin qu'elle soit exécutée. De plus, une seule tâche peut se trouver dans cet état [1].
- **Prête** : dans cet état, la tâche est en attente du processeur afin qu'elle soit mise en exécution une fois le processeur libre. La décision de mettre en exécution une tâche prête est faite par un ordonnanceur.
- **En attente** : dans cet état, la tâche est bloquée et reste en attente d'un événement pour la reprise de son exécution .
- **Suspendue** : dans cet état, la tâche est passive et peut être activée.

2.1.2.5 Activation d'une tâche

L'activation d'une tâche est faite grâce aux services `ActivateTask` et `ChainTask` du système d'exploitation. Après son activation, une tâche est prête à être exécutée.

En fonction de la classe de conformité, une tâche de base peut être activée une ou plusieurs fois. Une *activation multiple d'une tâche* signifie que le système d'exploitation OSEK reçoit et enregistre les requêtes d'activation d'une tâche de base qui a auparavant été activée. Pour chaque tâche, il existe un attribut spécifiant le nombre maximal d'activation possibles. Ainsi, si

1. Le standard OSEK n'envisage pas d'exécution sur une plateforme multicœur

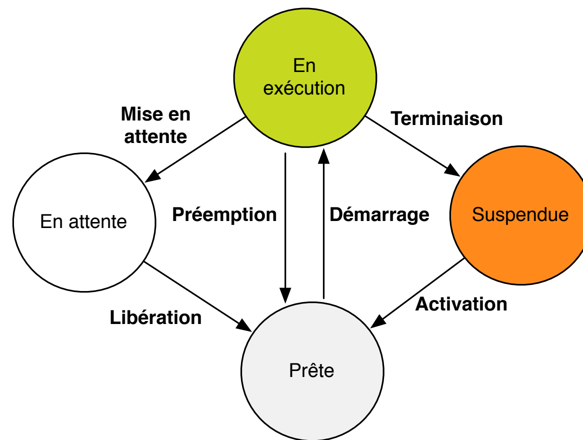


FIGURE 2.1 – Diagramme d'état d'une tâche étendue

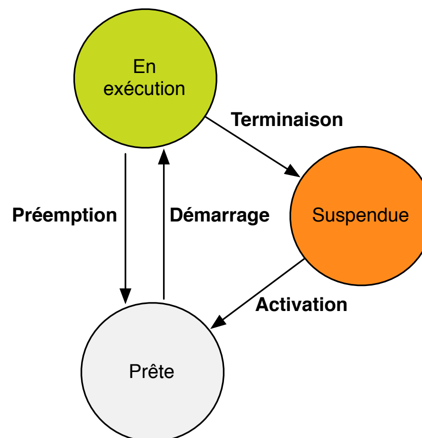


FIGURE 2.2 – Diagramme d'état d'une tâche de base

le nombre maximal n'est pas atteint, les requêtes d'activation sont placées en file d'attente par priorité définie selon l'ordre d'activation des tâches.

2.1.2.6 Services de l'API

Les différents services liés à la gestion des tâches sont présentés comme suit :

- `ActivateTask (<NomDeLaTache>)` : activation de la tâche désignée ;
- `TerminateTask` : terminaison de la tâche appelante ;
- `ChainTask(<NomDeLaTache>)` : terminaison de la tâche appelante et activation de la tâche désignée ;
- `Schedule` : attribution du processeur à la tâche prête la plus prioritaire. Ce service peut provoquer un réordonnancement.

2.1.3 Politique d'ordonnancement

Dans un système multi-tâche, les tâches sont exécutées de façon concurrente. L'entité permettant la gestion de l'ordre d'exécution des tâches est appelé un *ordonnanceur*.

En se basant sur la priorité, l'ordonnanceur choisit une tâche parmi les tâches prêtes à s'exécuter. Avec OSEK, la valeur 0 est toujours attribuée à la tâche ayant la priorité la plus basse. Dans un soucis d'efficacité, la gestion de priorité dynamique n'est pas supportée par le système d'exploitation OSEK sauf si les tâches détiennent des ressources. La priorité d'une tâche est donc définie de façon statique, en d'autres termes, elle ne peut être changée durant le temps d'exécution du système d'exploitation. Pour l'ordonnancement des tâches, plusieurs politiques d'ordonnancement sont définies.

2.1.3.1 Ordonnancement préemptif

Dans un ordonnancement préemptif, une tâche en cours d'exécution est préemptée aussitôt qu'une tâche ayant une priorité plus élevée devient *prête*. Le contexte de la tâche préemptée est sauvegardé afin qu'elle puisse reprendre son exécution à l'endroit où elle fut préemptée.

La figure 2.3 présente à titre illustratif un ordonnancement préemptif. Pendant la période A, la tâche *T1* est en exécution et la tâche *T2* est suspendue. Quand la tâche *T2* est activée, elle est aussitôt mise en exécution (Période B) en préemptant la tâche *T1* qui devient prête car la priorité de la tâche *T2* est supérieure à celle de la tâche *T1*. Lorsque la tâche *T2* termine son exécution, la tâche *T1* reprend son exécution (Période C) et se termine ensuite.

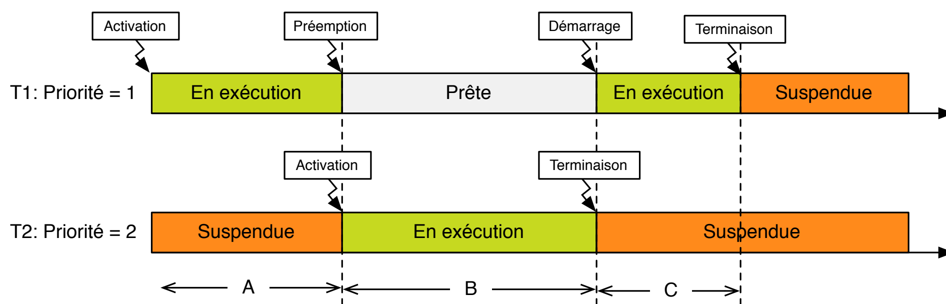


FIGURE 2.3 – Ordonnancement préemptif.

2.1.3.2 Ordonnancement non-préemptif

Un ordonnancement est dit non-preemptif lorsque le réordonnancement se produit aux différents points suivants :

— À la terminaison de la tâche (via le service `TerminateTask` du système d'exploitation).
— À la terminaison de la tâche et à l' activation explicite d'une autre tâche (via le service `ChainTask` du système d'exploitation).
— À l'appel explicite de l'ordonnanceur (via le service `Schedule` du système d'exploitation).
— À la mise en attente de la tâche (via le service `WaitEvent` du système d'exploitation).

Considérons la figure 2.4. Pendant la période A, la tâche *T1* est en exécution et la tâche *T2* est suspendue. Lorsque la tâche *T2* est activée, elle devient prête (Période B) et ne préempte pas la tâche *T1* qui reste toujours en exécution. À la fin de l'exécution de la tâche *T1*, elle est suspendue. La tâche *T1* démarre alors son exécution (Période C) et se termine ensuite.

2.1.3.3 Groupes de tâches

Il est possible de combiner les aspects préemptif et non-préemptif de l'ordonnancement par le biais de groupes de tâches. Ainsi, pour des tâches ayant une priorité inférieure ou égale à la

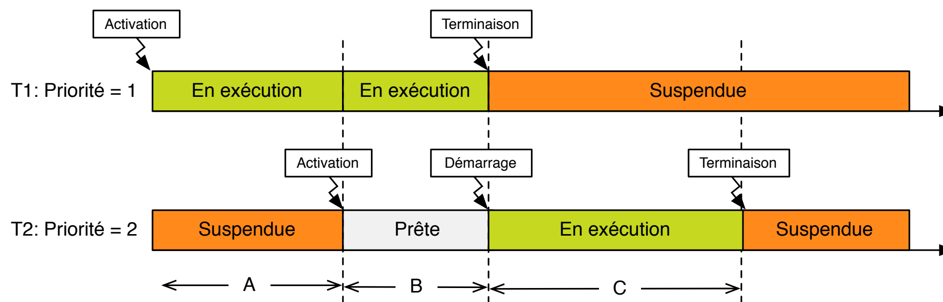


FIGURE 2.4 – Ordonnancement non-préemptif.

plus haute priorité dans un groupe, celles-ci se comportent comme des tâches non-préemptables. Pour des tâches ayant une priorité plus élevée que celle de la plus haute priorité dans un groupe, celles-ci se comportent comme des tâches préemptables.

2.1.3.4 Ordonnancement mixte

L'ordonnancement mixte a été introduit pour bénéficier à la fois des ordonnancements préemptif et non-préemptif. Dans ce cas, l'ordonnancement à réaliser dépend des propriétés de la tâche en cours d'exécution. Si elle est non-préemptable, l'ordonnancement non-préemptif est réalisé, dans le cas contraire, c'est l'ordonnancement préemptif qui est réalisé.

Une tâche non préemptable est utilisée dans un système d'exploitation préemptif si le temps d'exécution de la tâche est du même ordre de grandeur que le temps d'exécution du changement de contexte de la tâche et si la RAM doit être utilisée de façon économique afin de fournir plus d'espace pour la sauvegarde du contexte de la tâche.

2.1.4 Les modes d'application

Les modes d'applications permettent au système d'exploitation OSEK d'avoir différents modes d'opération. Les ECU (*Electronic Control Unit*) peuvent exécuter une même application dans des configurations différentes (test d'usine, programmation flash ou un fonctionnement normal). Le mode d'application est décidé au démarrage du système d'exploitation et sa modification n'est pas permise durant l'exécution.

Typiquement, chaque mode d'application renferme un ensemble de tâches, d'ISR et d'alarmes. Si plusieurs modes d'application nécessitent une fonctionnalité commune, il est alors recommandé de partager les différents objets (par ex. tâches, alarmes et ISR) qu'ils renferment entre eux.

Au démarrage, il appartient au code utilisateur (sans utilisation de services du système d'exploitation) de déterminer le mode d'application à démarrer afin de le passer en paramètre du service `StartOS` assurant le démarrage du système d'exploitation.

2.1.5 Traitement des interruptions

Une interruption se caractérise par un arrêt temporaire d'un programme par le processeur en raison de l'occurrence d'un événement extérieur. Le traitement d'une interruption est réalisé par une ISR (*Interrupt Service Routine*).

Pour la gestion des interruptions OSEK définit deux catégories d'ISR :

— ISR de catégorie 1 : ces ISR n'utilisent aucun service du système d'exploitation [2]. À la fin de leur traitement, le programme arrêté continue son exécution exactement à l'instruction où il fut interrompu.
— ISR de catégorie 2 : à la différence des interruptions de catégorie 1, les interruptions de catégorie 2 ont la possibilité de faire appel aux services du système d'exploitation.

À l'intérieur d'une ISR, il ne se produit aucun réordonnancement. Le réordonnancement se produit lors de la fin d'une ISR de catégorie 2 si une tâche préemptable a été interrompue et si aucune autre interruption n'est active.

Le nombre maximum de priorités d'une interruption dépend du matériel ainsi que de l'implémentation. L'ordonnancement des interruptions est fait en fonction du matériel et n'est pas spécifié dans OSEK. Les interruptions sont donc ordonnancées par le matériel tandis que les tâches sont ordonnancées par l'ordonnanceur. Si une tâche est activée à partir d'une ISR, elle est ordonnancée après la fin du traitement de toutes les interruptions actives.

2.1.5.1 Services de l'API

Les services qu'offre OSEK concernant les interruptions sont :
— `EnableAllInterrupts` : activation de toutes les interruptions ;
— `DisableAllInterrupts` : désactivation de toutes les interruptions ;
— `ResumeAllInterrupts` : reprise de toutes les interruptions ;
— `SuspendAllInterrupts` : suspension de toutes les interruptions ;
— `ResumeOSInterrupt` : reprise des interruptions liées au système d'exploitation ;
— `SuspendOSInterrupt` : suspension des interruptions liées au système d'exploitation ;

Ces services sont utiles pour assurer la protection des sections critiques courtes.

2.1.6 Mécanisme des événements

Les événements permettent la synchronisation des tâches. Un événement est toujours assigné à une tâche étendue. Chaque tâche étendue détient un nombre fini d'événements et est appelée *propriétaire* de ces événements. Un événement est identifié à partir de son propriétaire et de son nom (masque). Toutes les tâches peuvent signaler l'occurrence de tout événement mais seules les tâches étendues peuvent être en attente des événements dont elles sont propriétaires.

Quand une tâche est en attente d'un événement, elle passe à l'état *en attente* puis quand l'événement se produit, elle devient *prête* et un réordonnancement est réalisé. Si une tâche est en attente de plusieurs événements, elle devient *prête* lorsqu'un des événements se produit. Si une tâche étendue essaie d'attendre un événement qui s'est déjà produit, elle reste *en exécution.*

2.1.6.1 Services de l'API

Les différents services liés aux événements sont représentés comme suit :
— `SetEvent(<NomDeLatache>, <Evenement>)` : signale l'occurrence d'un événement nommé pour la tâche désignée.
— `WaitEvent(<Evenement>)` : met la tâche appelante en attente de l'événement nommé.
— `ClearEvent(<Evenement>)` : efface l'événement nommé de la tâche appelante.
— `GetEvent(<NomDelaTache>,<RefEvenement>)` : donne l'état de tous les événements de la tâche désignée.

La figure 2.5 montre l'exemple d'une synchronisation de tâches par événement.

2. excepté les services permettant de désactiver ou d'activer les interruptions

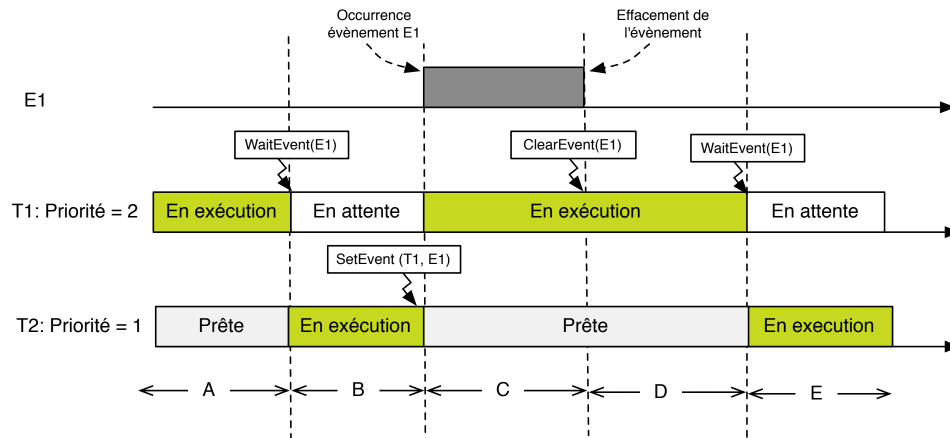


FIGURE 2.5 – Synchronisation par événement

.

T1 est une tâche étendue qui a pour événement *E1*. Pendant la période *A*, *T1* est en exécution et *T2* est prête. *T1* fait ensuite appel au service `WaitEvent` et reste bloquée en attente de l'occurrence de l'événement *E1* puis *T2* démarre son exécution (période *B*). Lorsque *T2* signal l'occurrence de l'événement *E1* via le service `SetEvent`, *T1* est à nouveau mise en exécution puis *T2* est préemptée et devient prête (période *C* et *D*). T1 efface l'événement *E1* et continue son exécution jusqu'à être à nouveau en attente de l'événement *E1* (période E).

2.1.7 Gestion des ressources

La gestion des ressources est utilisée pour contrôler l'accès à une ressource partagée (par ex. ordonnanceur, zone mémoire, etc.) par plusieurs tâches ayant différentes priorités. Elle est utile dans les cas suivant :

- Lors de l'utilisation des tâches préemptables.
- Lorsque les tâches et les ISR de catégorie 2 partagent des ressources.
- Lorsque les ISR de catégorie 2 partagent des ressources.

Elle permet de satisfaire les points suivants :

- Deux tâches ne peuvent accéder à la même ressource au même moment.
- L'impossibilité d'inversion de priorité.
- L'absence d'*interblocage* en utilisant les ressources.
- Une tentative d'accès à une ressource ne conduit jamais à un état d'attente.
- Deux tâches ou routines d'interruption ne peuvent accéder à la même ressource au même moment.

2.1.7.1 L'ordonnanceur comme une ressource

Lorsqu'une tâche veut se protéger contre toutes préemptions par d'autres tâches, elle acquiert un accès exclusif à l'ordonnanceur. L'ordonnanceur peut ainsi être considéré comme une ressource accessible par toutes les tâches. Par conséquent, dans un système d'exploitation OSEK, une ressource prédéfinie appelée `RES_SCHEDULER` est automatiquement générée. Les interruptions sont reçues et traitées indépendamment de l'état de la ressource `RES_SCHEDULER`. Cependant, elle empêche le réordonnancement des tâches.

2.1.7.2 Problèmes liés aux mécanismes de synchronisation

Inversion de la priorité L'inversion de la priorité est un problème typique des mécanismes de synchronisation. Elle est causée par le fait qu'une tâche moins prioritaire retarde l'exécution d'une tâche plus prioritaire. Pour éviter le problème de l'inversion de la priorité, OSEK utilise le principe du protocole à priorité plafond (PCP : *Priority Ceiling Protocol*).

La figure 2.6 illustre le problème de l'inversion de priorité avec des sémaphores. La tâche *T1* a la plus haute priorité et la tâche *T3* a la priorité la plus basse. Initialement, la tâche *T3* qui est en exécution détient le sémaphore *S1* et est ensuite préemptée par la tâche *T1*. La tâche *T1* tente un accès au sémaphore *S1* qui lui est refusé car *S1* est occupé par la tâche *T3*. La tâche *T1* se met en attente du sémaphore *S1*. Lors d'un réordonnancement, la tâche *T2* se met en exécution et lorsqu'elle se termine, elle cède le processeur à la tâche *T3* qui se met en exécution. Lors de la libération du sémaphore *S1* par la tâche *T3*, elle est aussitôt préemptée par la tâche *T1* qui reprend son exécution. La tâche *T1* a été retardée par la tâche *T3* de priorité moins élevée.

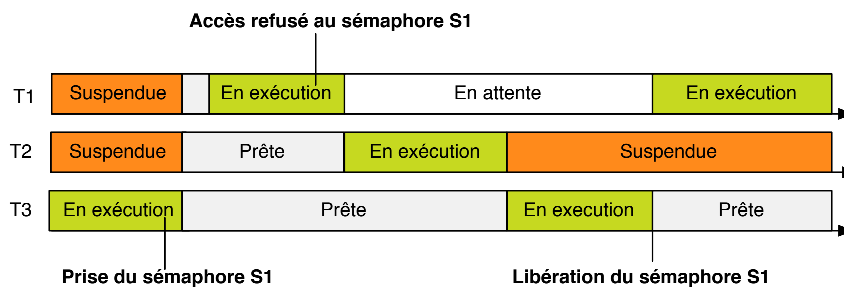


FIGURE 2.6 – Exemple de l'inversion de priorité

Interblocage Un autre problème des mécanismes de synchronisation est celui de l'interblocage. L'interblocage se traduit par une attente croisée de tâches.

Le scénario présenté à la figure 2.7 montre l'exemple d'un interblocage en utilisant des sémaphores. La tâche *T1* occupe le sémaphore *S1* puis se met en attente d'un événement. Ainsi, la tâche la moins prioritaire *T2* est mise en exécution puis occupe le sémaphore *S2*. Lors de l'occurrence de l'événement attendu par la tâche *T1*, elle se met en exécution puis tente un accès au sémaphore *S2* qui lui est refusé et se met en attente. La tâche *T2* reprend alors son exécution et tente un accès au sémaphore *S1* qui lui est refusé. Les deux tâches sont alors bloquées car elles sont en attente infini de sémaphores déjà occupés.

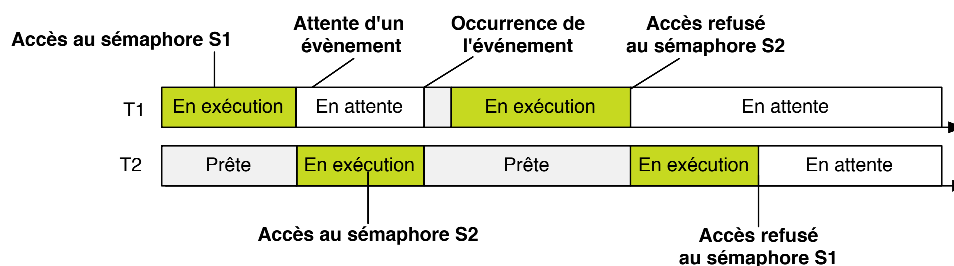


FIGURE 2.7 – Exemple d'interblocage en utilisation des sémaphores

2.1.7.3 Protocole à priorité plafond immédiat [SRL90]

Afin d'éviter le problème de l'inversion de priorité et celui du blocage, OSEK utilise le protocole à priorité plafond.

Son principe est défini comme suit :

- Une priorité plafond est attribuée à chaque ressource. La priorité plafond d'une ressource doit être au moins égale à la plus haute priorité de toutes les tâches accédant à cette ressource ou toute autre ressource liée à cette ressource.
- Si une tâche obtient l'accès à une ressource et si sa priorité courante est plus basse que la priorité plafond de la ressource, la priorité de la tâche est élevée à la priorité plafond de la ressource.
- Si la tâche libère la ressource, la priorité de cette tâche est remise à la priorité qu'elle avait avant d'accéder à la ressource.

Les tâches qui pourraient occuper la même ressource que celle de la tâche en cours d'exécution ne peuvent entrer en exécution du fait de leur priorité plus petite ou égale à celle de la tâche en exécution.

2.1.7.4 Les ressources internes

Les ressources internes sont des ressources qui ne peuvent être directement accessibles par les tâches de l'application. Elles sont plutôt manipulées par un ensemble de fonctions bas niveau. En outre, elles se comportent comme des ressources standards (basées sur le principe de la priorité plafond).

Les ressources internes sont limitées aux tâches et au plus une ressource interne doit être attribuée à une tâche. Si une ressource interne est détenue par une tâche alors, celle-ci est gérée de la façon suivante :

- La ressource est automatiquement prise lorsque la tâche commence son exécution à moins qu'elle la détienne déjà. Ainsi, la priorité de la tâche est remplacée par celle de la priorité plafond de la ressource interne.
- Au point de réordonnancement (voir section 2.1.3.2), la ressource interne est automatiquement libérée.

Le comportement des tâches ayant une même ressource interne est identique à celui des groupes de tâches (voir section 2.1.3.3). Les tâches préemptables sont un groupe spécial de tâches avec une ressource interne ayant une priorité égale à celle de la ressource `RES_SCHEDULER`. Les ressources internes sont utilisées lorsqu'aucun réordonnancement n'est souhaité dans un groupe de tâches. De plus, il existe autant de groupes de tâches que de ressources internes.

2.1.7.5 Services de l'API

La description des services assurant l'accès aux ressources partagées sont les suivants :

- `GetResource(<NomDeLaRessource>)` : prise de la ressource désignée.
- `ReleaseResource(<NomDeLaRessource>)` : libère l'accès à la ressource désignée.

La figure 2.8 décrit l'utilisation des services liés aux ressources en utilisant le protocole de priorité plafond.

`priorité(T0) > priorité(R) > priorité(T1) > priorité(T2)`. La ressource R est utilisée par les tâches *T1* et *T2*. Pendant la période A, la tâche *T2* est en cours d'exécution et les autres tâches sont suspendues. Ensuite la tâche *T2* prend la ressource R via le service `GetResource` et sa priorité est alors remplacée par celle de la ressource (période B à E). La tâche *T1* est ensuite activée mais ne préempte pas la tâche *T2* et reste prête (période C). *T0* est activée au début

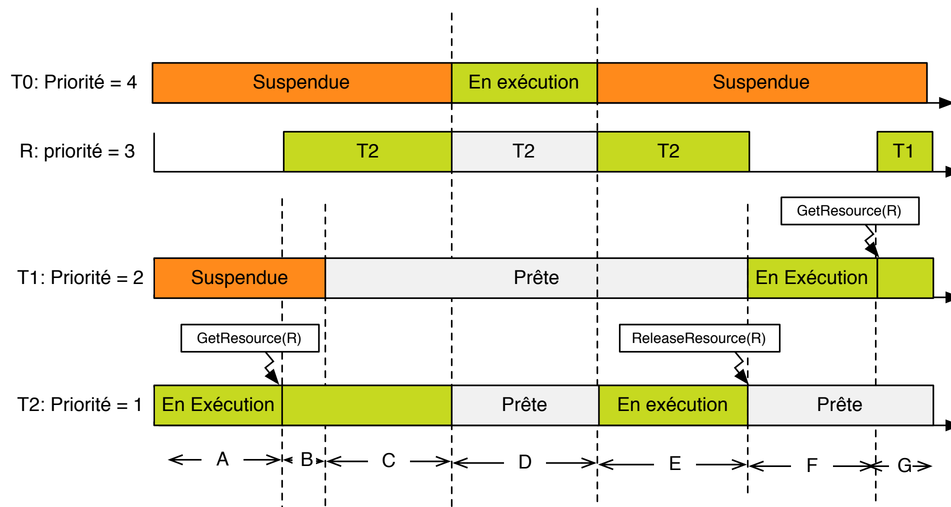


FIGURE 2.8 – Utilisation des ressources dans l'ordonnancement des tâches.

de la période *D* et puisqu'elle a une priorité supérieure à celle de la ressource occupée par *T2*, elle la préempte puis se met en exécution. À la fin de l'exécution de la tâche *T0*, la tâche *T2* reprend son exécution et la tâche *T1* reste toujours prête (période *E*). La tâche *T2* libère ensuite la ressource *R* via le service ReleaseResource et reprend sa priorité initiale. À cet instant, un réordonnancement est effectué et *T2* est aussitôt préemptée par la tâche *T1* qui a une priorité plus élevée (période *F*). T1 prend la ressource R puis récupère sa priorité (période *G*).

2.1.8 Alarmes

Le système d'exploitation OSEK offre des services permettant le traitement d'actions récurrentes dans le temps. Cela est possible grâce aux alarmes et aux compteurs. Ils permettent également la mise en œuvre des chiens de garde.

Un compteur est un objet qui permet l'enregistrement de *ticks* en provenance d'une horloge. Plusieurs alarmes peuvent être associées à un même compteur, ce qui permet de constituer facilement, par exemple, des bases de temps. À chaque alarme est associée une action, qui peut être soit l'activation d'une tâche soit la signalisation d'une occurrence d'événement. Une alarme peut être unique ou cyclique, absolue ou relative.

2.1.8.1 Services de l'API

Les services manipulant les alarmes sont les suivants :

— SeReltAlarm(<NomAlarme>, <Increment>, <Cycle>) : assure l'activation de l'alarme désignée avec une valeur relative (Increment) qui indique le nombre de ticks qu'il faudra attendre à partir de la date courante pour qu'une alarme expire.

— SetAbsAlarm(<NomAlarme>, <Start>, <Cycle>) : assure l'activation de l'alarme désignée avec une valeur absolue (start) qui indique la date absolue à laquelle l'alarme doit expirer.

— GetAlarm(<NomAlarme>, <Ticks>) : permet d'obtenir le nombre de *ticks* restant avant la prochaine expiration de l'alarme désignée.

— CancelAlarm(<NomAlarme>) : permet la désactivation de l'alarme désignée.

2.2 AUTOSAR : un standard d'architecture logicielle

Créé en 2003, AUTOSAR (*Automotive Open System Architecture*) est un standard présentant une architecture et une méthodologie permettant le développement de logiciels embarqués automobiles. Il a pour objectifs de définir un standard d'interopérabilité, d'échangeabilité et d'implantation permettant de favoriser la réutilisation des logiciels embarqués dans les véhicules. Il se base sur les fondements définis par le standard OSEK/VDX concernant le système d'exploitation.

2.2.1 Architecture logicielle

Le standard AUTOSAR présente une architecture composée des trois couches suivantes :

- La couche applicative : indépendante de la plateforme matérielle, cette couche contient les composants logiciels eux-mêmes composés de *runnables* qui contiennent le code mettant en œuvre les fonctions applicatives à exécuter.
- La couche basse BSW (*Basic Software*) : elle est divisée en plusieurs couches. La couche de services ou *Services Layer* contient les fonctionnalités de haut niveau accessibles à l'application et intègre le système d'exploitation AUTOSAR dérivé du système d'exploitation OSEK/VDX. La MCAL (*Microcontroller Abstraction Layer*) contient les pilotes de périphériques permettant l'accès au matériel. La ECUAL (*ECU Abstraction Layer*) s'interface avec la MCAL et fournit toutes les interfaces des services permettant d'accéder aux matériels et aux périphériques.
- Le RTE (*Runtime Environment*) : il sert de support de communication entre les différents composants logiciels (*Software Components*). Il permet également de faire le lien entre l'appel des services nécessaires à l'application logicielle et les services correspondant dans la couche basse BSW. Les appels de services peuvent passer par le système d'exploitation AUTOSAR.

2.2.2 Configuration dans AUTOSAR

AUTOSAR offre une configuration de type statique pour le développement de logiciel. Les différents modules configurables sont le BSW et la MCAL. Chaque module contient un ensemble de fichiers génériques contenant toutes les fonctionnalités disponibles et un fichier de configuration permettant de spécifier les différentes fonctionnalités requises par l'application. Pour décrire les spécificités de l'application, AUTOSAR utilise un fichier XML appelé AUTOSAR XML (*ARXML*). Une fois les besoins de l'application spécifiés, un générateur de code est utilisé pour synthétiser les modules BSW et MCAL spécifiques à l'application. Les fichiers obtenus sont enfin compilés puis liés pour former l'exécutable.

2.3 Trampoline

2.3.1 Présentation

Trampoline [BBFT06] est un système d'exploitation temps réel statique développé conformément au standard OSEK/VDX [G^{+}05] puis étendu au standard AUTOSAR [AUT13]. Il est développé dans l'équipe Systèmes Temps Réel de l'Institut de Recherche en Communications et Cybernétique de Nantes. Il regroupe l'ensemble des fonctionnalités offertes par le standard OSEK/VDX (voir section 2.1) et l'ensemble du standard AUTOSAR. Trampoline a été conçu

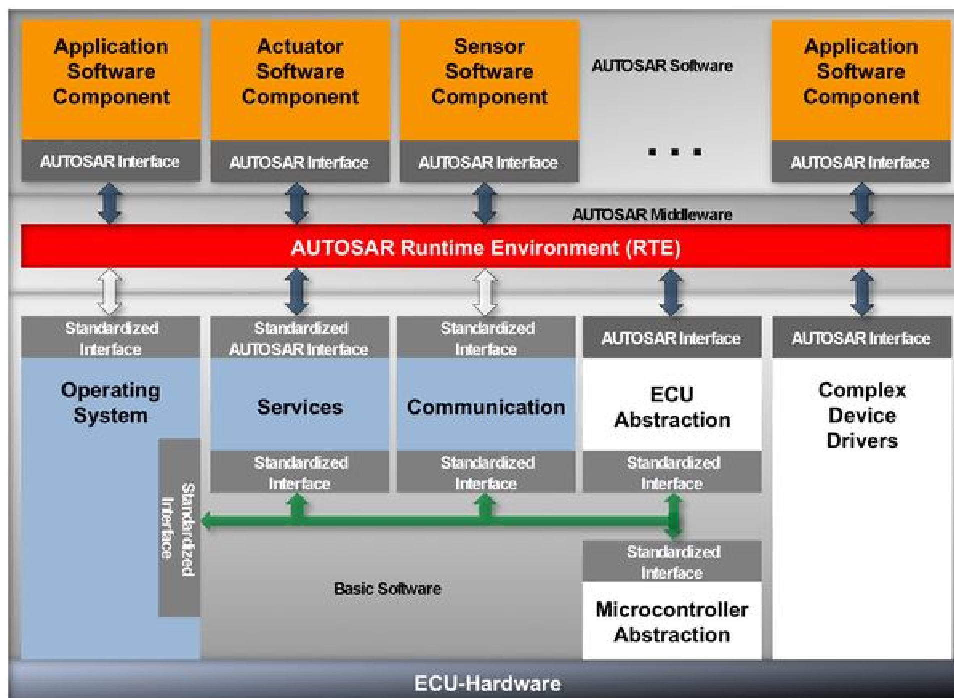


FIGURE 2.9 – Architecture Logicielle

dans un premier temps pour des besoins académiques mais est également utilisé dans le domaine automobile.

Par sa taille relativement petite et comportant un noyau monolithique, Trampoline est destiné aux systèmes à ressources limitées.

La configuration de Trampoline est faite de façon statique et est basée sur le préprocesseur C. La figure 2.10 présente la chaîne de compilation de Trampoline.

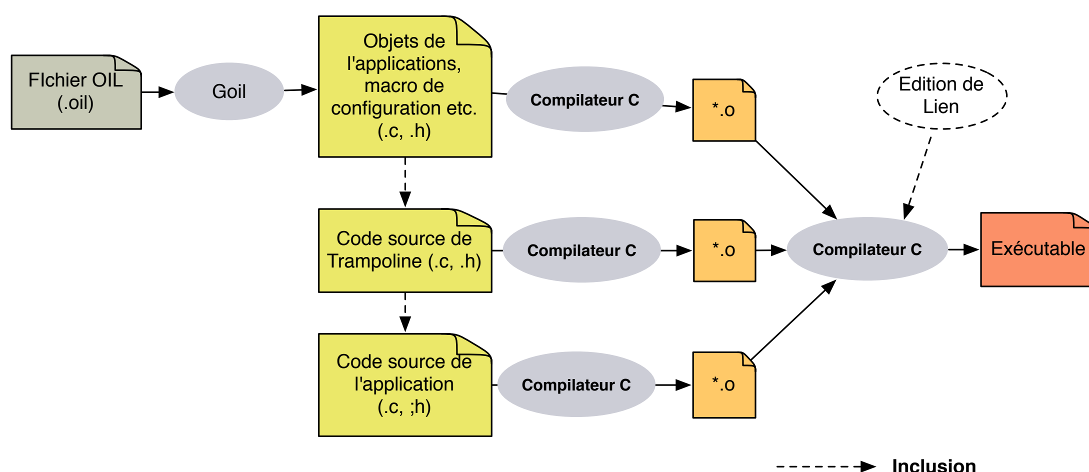


FIGURE 2.10 – Chaîne de compilation et configuration de Trampoline

L'application est avant tout décrite dans un fichier .oil. Ce fichier est ensuite compilé par le

compilateur *goil* qui génère un ensemble de fichiers de description C contenant la description de l'application (les structures de données des différents objets de l'application) ainsi que des macros de configuration. Les fichiers source de Trampoline incluent les fichiers (.h) contenant la configuration. Le code source de l'application inclut également les fichiers (.h) de Trampoline. Enfin, le code source de Trampoline, les fichiers de description ainsi que le code source de l'application sont compilés. Les fichiers objets obtenus suite à la compilation sont liés via un éditeur de liens pour former le fichier exécutable.

Trampoline comprend deux versions : une version monocœur et une version multicœur. Nous présentons ci-après ses architectures monocœur et multicœur.

2.3.2 Architecture de Trampoline monocœur

Trampoline est scindé en 3 composants (voir figure 2.11) :

— L'API contient tous les services accessibles à l'application (voir section 2.1) via une API standardisée (service de l'API des standards OSEK/VDX et AUTOSAR).

— Le Noyau contient toutes les fonctions de bas niveau sur lesquelles s'appuient les services de Trampoline. Ces fonctions permettent de réaliser l'ordonnancement des tâches, leur démarrage ainsi que leur synchronisation. Il contient également des fonctions assurant le démarrage et l'arrêt de Trampoline.

— Le BSP (*Board Support Package*) est la partie bas niveau de Trampoline dépendant de la machine cible. Les modules tels que *le gestionnaire des appels système* et *le gestionnaire du changement de contexte* doivent être implémentés en assembleur.

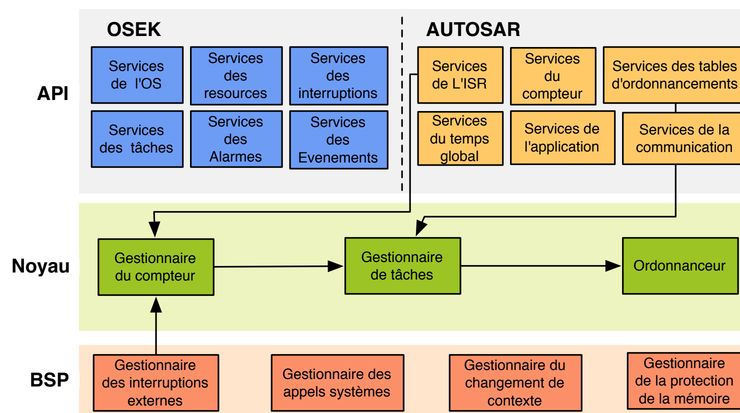


FIGURE 2.11 – Architecture de Trampoline monocœur [BBF15]

Le BSP Le *Board Support Package* est constitué de 4 modules :

Le gestionnaire des interruptions externes : Ce module assure la gestion des interruptions qui peuvent provenir du *timer* ou d'une autre source. Il interagit avec l'ordonnanceur et *le gestionnaire du changement de contexte* dans le cas où l'interruption déclenchée provoque un réordonnancement entraînant la perte du processeur par la tâche en cours d'exécution.

Le gestionnaire des appels système : Ce module constitue le point de passage pour l'appel d'un service de Trampoline. Au niveau de l'application, l'appel d'un service correspond

à l'appel d'une fonction de l'API. L'appel d'un service nécessite le changement du contexte d'exécution du mode utilisateur vers le mode superviseur dans le but de réaliser le service demandé.

Lorsque le service appelé nécessite un réordonnancement, *le gestionnaire des appels système* interagit avec *le gestionnaire du changement de contexte* afin de pouvoir sauvegarder le contexte de la tâche préemptée. La figure 2.12 [BBF15] montre la structure du gestionnaire des appels système.

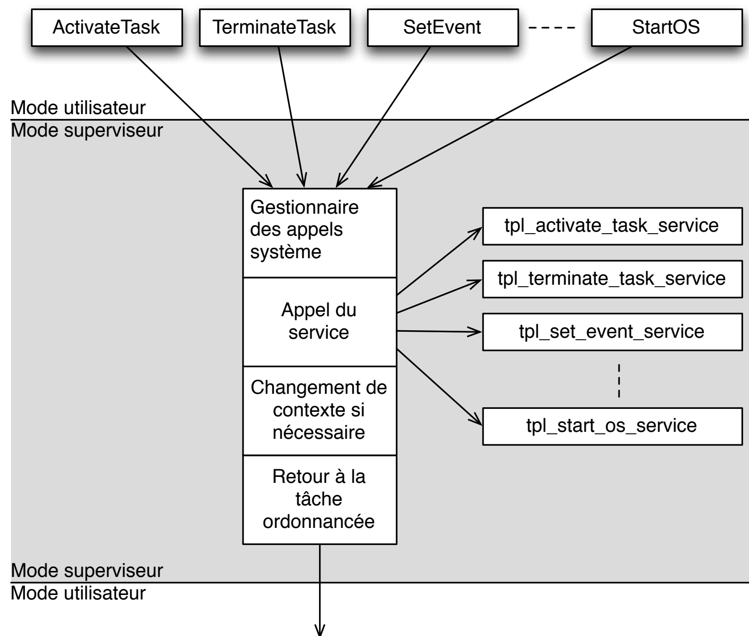


FIGURE 2.12 – Structure du gestionnaire des appels système [BBF15]

Le gestionnaire du changement de contexte : Ce module permet la sauvegarde du contexte de la tâche préemptée ainsi que la restauration du contexte de la tâche ordonnancée.

Le gestionnaire de la protection de la mémoire : Ce module assure la protection de la mémoire en fonction du contexte d'exécution. Il se charge de la programmation de la *Memory Protection Unit* en fonction de la tâche qui est ordonnancée dans le but d'isoler spatialement les tâches de l'application.

Le noyau Le noyau est formé de trois modules :

Le gestionnaire de compteurs : Le traitement des événements récurrents repose sur le concept proposé par le standard OSEK/VDX.

D'une part, un compteur se charge du comptage de l'occurrence des événements et d'autre part, les alarmes qui sont liées au compteur permettent de déclencher une action (e.g activation d'une tâche) lorsque celles-ci arrivent à expiration au bout d'un certain nombre d'occurrence d'un événement compté. De plus, Trampoline supporte les tables d'ordonnancement (*schedule table*) fournies par le standard AUTOSAR.

Tout comme les alarmes, les tables d'ordonnancement sont liées à un compteur. Mais, elles permettent le déclenchement d'une série d'actions ordonnancée dans le temps. Une table d'ordonnancement possède une période et peut être répétée.

Le gestionnaire de compteurs est appelé par *le gestionnaire des interruptions externes* lorsque la source d'interruption provient du *tick* compteur.

Dans Trampoline, la gestion des alarmes est faite en utilisant les listes chaînées. La figure 2.13 nous montre l'exemple d'une liste d'alarmes associées à un compteur.

Le compteur pointe à la fois sur le premier élément de la liste (`First`) et sur la prochaine alarme qui doit expirer (`Next`). Lorsque la date courante est égale à celle de l'expiration de l'alarme (`Next`), elle est enlevée de la liste et l'action correspondante est effectuée. Si l'alarme est cyclique, elle est réinsérée dans la liste avec une nouvelle date. Puisque le compteur admet une valeur maximale, cette date est calculée modulo cette valeur maximale et peut être plus petite que la date courante. Le compteur pointe toujours sur la prochaine alarme qui doit expirer et lorsqu'il arrive en fin de liste, il pointe à nouveau sur l'alarme qui se trouve en début de liste (si elle n'est pas vide). De plus, lorsque la valeur maximale du compteur est atteinte, celui-ci est remis à 0.

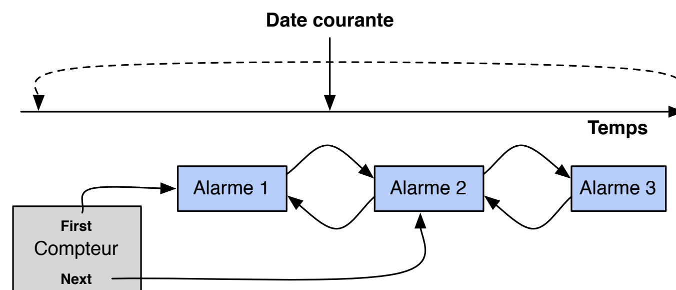


FIGURE 2.13 – Structure de données pour la gestion des alarmes

Ordonnanceur : L'ordonnancement de Trampoline est basé sur celui offert par le standard OSEK/VDX (voir section 2.1.3). L'ordonnanceur est le module central du noyau, il est à la fois invoqué par *le gestionnaire des interruptions externes* et *le gestionnaire de compteurs*. L'ordonnanceur gère une liste de tâches prêtes. Dans cette liste, les tâches sont rangées par ordre de priorité ou par ordre d'activation dans le cas où celles-ci ont des priorités égales. Cette liste est un tableau de liste FIFO représenté à la figure 2.14.

L'ordonnanceur utilise ainsi 3 fonctions pour la gestion de la liste des tâches prête à savoir :

— `tpl_put_preempted_proc:` cette fonction ajoute une tâche préemptée dans la liste des tâches prêtes en fonction de sa priorité.
— `tpl_put_new_proc:` cette fonction ajoute une tâche nouvellement activée dans la liste des tâches prêtes.
— `tpl_get_proc:` cette fonction retire de la liste des tâches prêtes la tâche ayant la priorité la plus élevée.

L'ordonnanceur gère également une structure de données, `tpl_kern`, permettant de mémoriser les informations relatives au processus en cours d'exécution. Les informations suivantes sont mémorisées :

— `running_id` : l'identifiant du processus en cours d'exécution ;
— `running` : un pointeur vers le descripteur dynamique du processus en cours d'exécution ;

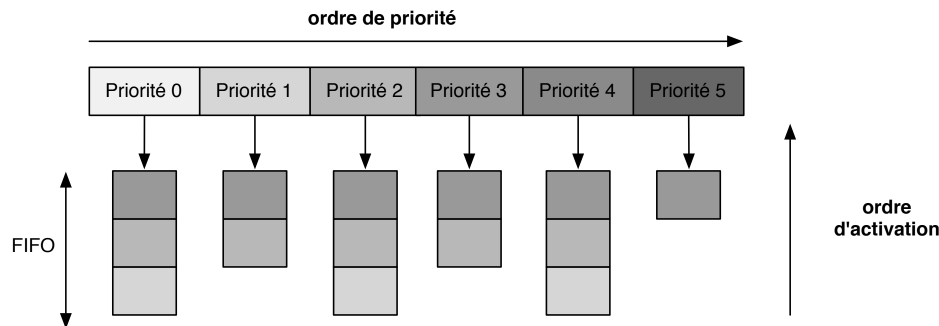


FIGURE 2.14 – Structure de la liste des tâches prêtes

- s_running : un pointeur vers le descripteur statique du processus en cours d'exécution ;
- old : un pointeur vers le descripteur dynamique du processus qui vient de perdre le CPU ;
- s_old : un pointeur vers le descripteur statique du processus qui vient de perdre le CPU ;
- need_switch : indique qu'un changement de contexte entre le processus qui vient de perdre le CPU, old/s_old, et le processus qui vient de le gagner, running/s_running, doit être effectué.

Le gestionnaire de tâches : Ce module est constitué d'un ensemble de fonctions bas niveau pour la gestion des tâches qui correspondent aux transitions sur le diagramme d'état des tâches (*cf.* figure 2.2) il s'agit de :

- tpl_activate_task: cette fonction est appelée soit à partir des services tpl_activate_task_service ou tpl_chain_task_service, soit à partir d'une action associée à une alarme, une table d'ordonnancement ou un message. Elle permet le passage d'une tâche de l'état *suspendue* à l'état *prête*.
- tpl_preempt: cette fonction est appelée par l'ordonnanceur. La tâche en cours d'exécution est préemptée et passe de l'état *en exécution* à l'état *prête*.
- tpl_block: cette fonction est appelée par le service tpl_wait_event_service. la tâche appelante passe de l'état *en exécution* à l'état *en attente*.
- tpl_terminate: cette fonction est appelée par les services tpl_terminate_task_service et tpl_chain_task_service. La tâche en cours d'exécution se termine et passe à l'état *suspendue*.
- tpl_set_event: cette fonction est appelée par le service tpl_set_event_service et par les actions d'envoi d'événements liées aux alarmes et aux tables d'ordonnancement. Lorsque la tâche cible reçoit au moins un événement attendu , elle passe de l'état *En attente* à l'état *prête*.
- tpl_start: cette fonction est aussi appelée par l'ordonnanceur, la tâche indiquée passe de l'état *prête* à l'état *en exécution*.

2.3.3 Architecture de Trampoline multicœur

La version de Trampoline multicœur présente une architecture à peu près similaire à sa version monocœur (voir figure 2.15). Des extensions ont été réalisées au niveau des services et du BSP. Des services de verrou ont été ajoutés. Ces services permettent le verrouillage du noyau lors d'un appel de service du système d'exploitation. De cette manière, il n'est pas possible d'exécuter

simultanément des services et des interruptions sur deux cœurs différents. Concernant le BSP, un module supplémentaire a été ajouté : il s'agit du *gestionnaire des interruptions intercœurs*. Il permet de gérer les interactions et la communication intercœur. Le BSP doit s'exécuter sur le cœur cible contrairement au noyau qui peut s'exécuter sur n'importe quel cœur.

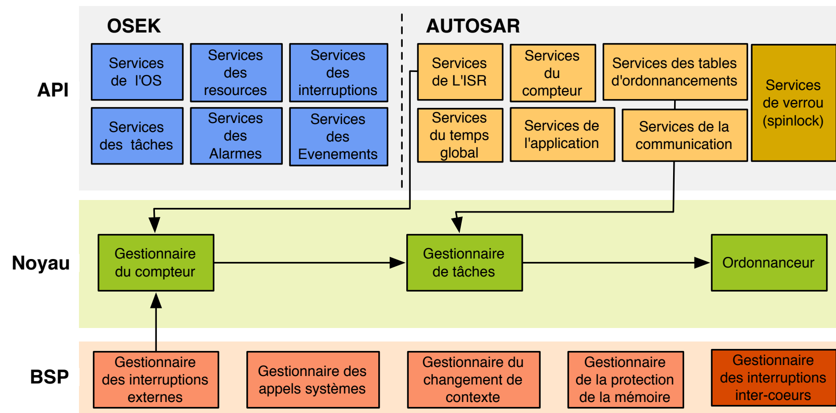


FIGURE 2.15 – Architecture de Trampoline multicœur

2.3.3.1 Stratégie d'ordonnancement de Trampoline multicœur

Trampoline multicœur supporte un ordonnancement de type partitionné comme le spécifie le standard AUTOSAR. Dans ce type d'ordonnancement, les tâches sont statiquement allouées aux cœurs et chaque cœur possède une liste de tâches prêtes. Une liste de tâches prêtes est implémentée dans Trampoline multicœur par un tas binaire. La figure 2.16 présente l'exécution d'un système formé de quatre tâches mettant en œuvre l'ordonnancement dans Trampoline ainsi que le principe de fonctionnement des interruptions intercœur et du verrou global.

Pour ce système, la tâche *T1* a la priorité la plus élevée et la tâche *T3* est la moins prioritaire. Initialement, les tâches *T0* et *T3* sont en exécution sur les *cœurs* 0 et 1. La tâche *T0* fait appel à l'API `ActivateTask` pour l'activation de la tâche *T1*. Le verrou noyau est alors activé et un réordonnancement est effectué pour le *cœur* 1 suivi d'une notification de changement de contexte qui lui est envoyée via une interruption intercœur différée à cause du verrou noyau qui est activé. Pendant ce temps, la tâche *T3* fait appel à l'API `ActivateTask` pour l'activation de la tâche *T2*. Le noyau étant verrouillé, le *cœur* 1 attend la désactivation de celui-ci pour exécuter le service demandé. Lorsque le verrou noyau est désactivé, un réordonnancement local est effectué sur le *cœur* 1 suivi d'un changement de contexte de *T3* à *T2*. Ensuite, l'interruption qui avait été envoyée auparavant au *cœur* 1 est prise en compte puis un deuxième changement de contexte est réalisé de la tâche *T2* à la tâche *T1* qui se met en exécution.

2.4 Conclusion

Dans ce chapitre, nous avons présenté les standards sur lesquels se base Trampoline ainsi que ses architectures monocœur et multicœur. Dans le chapitre suivant, nous présenterons le processus de modélisation de Trampoline. Ainsi, le modèle obtenu servira de base à notre approche de synthèse formelle de systèmes d'exploitation.

Cela permettra d'une part de réaliser une configuration à grain fin et d'autre part d'engendrer un système d'exploitation formellement vérifié.

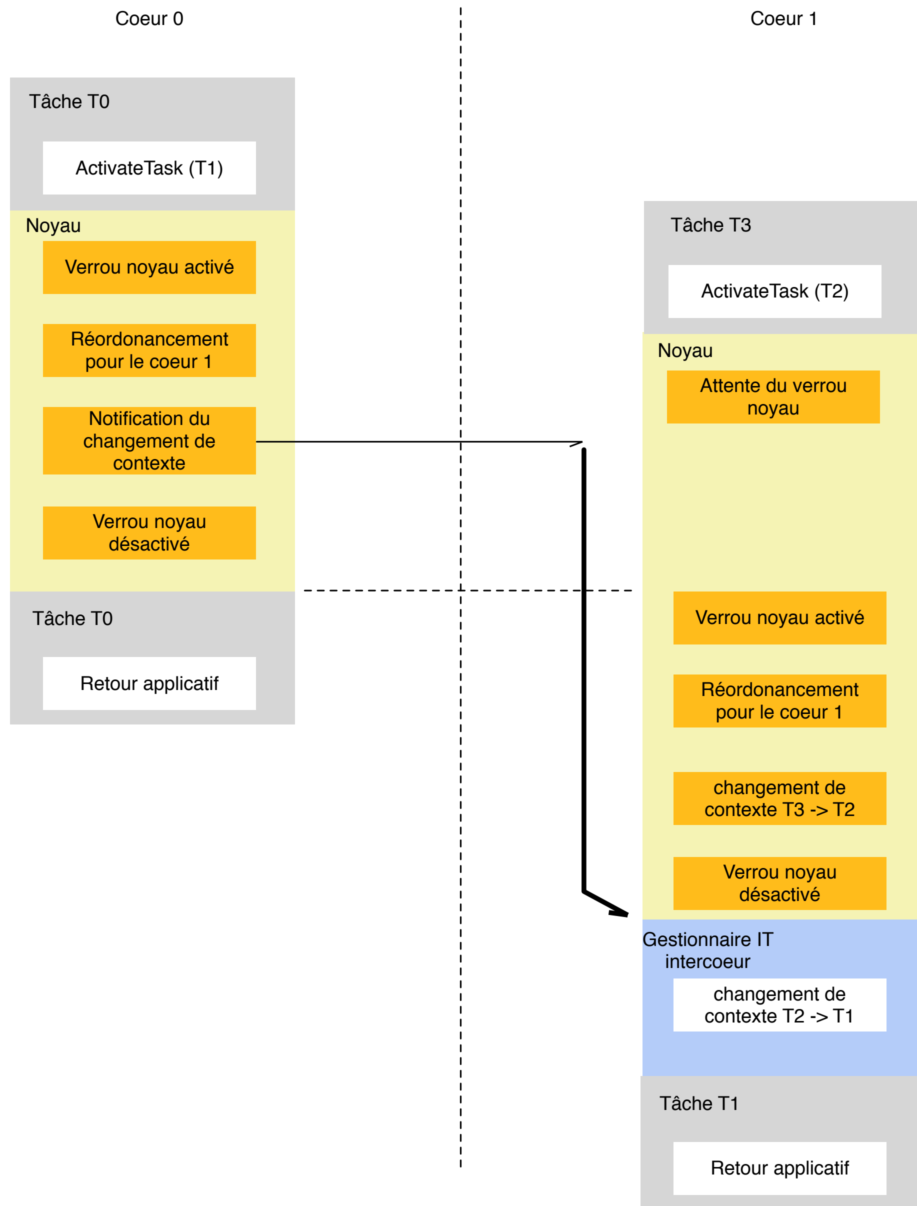


FIGURE 2.16 – Scénario d'exécution de tâches sur Trampoline multicœur

Chapitre 3

Modélisation

Sommaire

3.1 Outil de formalisme **56**
3.1.1 Automates finis 56
3.1.2 Automates finis étendus 56
3.1.3 Réseau d'automates finis étendus 57
3.1.4 Utilisation des automates finis étendus pour l'abstraction d'un programme 58
3.1.5 Accessibilité et bornitude 58
3.2 Principes de la modélisation **59**
3.2.1 Modélisation des branchements 59
3.2.2 Modélisation des boucles 59
3.2.3 Modélisation d'un appel de fonction 60
3.3 Adaptation à UPPAAL **60**
3.3.1 UPPAAL : outil pour la vérification de systèmes temps réel 60
3.3.2 Le traitement du temps 62
3.3.3 Règles de modélisation en utilisant UPPAAL 63
3.4 Modèle de Trampoline dédié aux architectures monocœur **64**
3.4.1 Modélisation des objets et des variables 64
3.4.2 Modélisation du noyau 65
3.4.3 Modélisation des services 68
3.4.4 Modélisation de l'interruption matérielle liée au timer 70
3.5 Modèle de Trampoline dédié aux architectures multicœur **71**
3.5.1 Modélisation du noyau 71
3.5.2 Modélisation des services 72
3.6 Modèle d'applications **72**
3.7 Propriétés du modèle **73**
3.8 Conclusion **75**

Comme nous l'avons précisé précédemment, l'objectif principal est la configuration automatique de systèmes d'exploitation basée sur une modélisation formelle de ceux-ci. Pour cela, il est nécessaire de construire le modèle du système d'exploitation. Selon [Mah11], un programme écrit en C peut être décrit sous la forme de systèmes de transitions.

Dans ce cas, les états du système de transitions décrivent l'état du programme et les transitions peuvent décrire un appel de fonction ou l'exécution d'un ensemble d'instructions qui mène le programme d'un état à un autre. Dans cette optique, les systèmes de transitions peuvent traduire le graphe de flot de contrôle d'un programme impératif.

En partant sur cette base, nous décrivons les différentes fonctionnalités de Trampoline à partir des systèmes de transitions.

Dans ce chapitre, nous verrons plus en détail le formalisme utilisé pour la description du système d'exploitation, les différentes règles de modélisation et les propriétés du modèle.

3.1 Outil de formalisme

3.1.1 Automates finis

Définition 1. *un automate fini G est un tuple $(S, \Sigma, \rightarrow, S_0)$ tel que :*
- *S est un ensemble fini d'états.*
- *Σ est un ensemble non vide d'actions réalisables.*
- *$\rightarrow \subseteq S \times \Sigma \times S$ est la relation de transition.*
- *S_0 est l'état initial.*

La relation de transition est écrite $p \overset{\sigma}{\rightarrow} q$ si et seulement s'il existe une transition étiquetée σ allant de l'état p à l'état q. De façon informelle, une telle transition s'explique par le fait que lorsque le système est dans un état p et réalise une action σ, il passe dans un état q.

Pour la réalisation d'un modèle de système d'exploitation devant contenir son code et inclure des variables qu'il manipule, un simple automate fini reste insuffisant. Il faudrait en effet étendre l'automate fini avec des variables de manière à décrire fidèlement le système d'exploitation. Nous définissons ainsi les *automates finis étendus*.

3.1.2 Automates finis étendus

Un automate fini étendu est une extension d'un simple automate fini par un ensemble de variables entières bornées. Dans ce cas, les transitions sont complétées avec des conditions et des actions sur ces variables. Les conditions sur les variables sont appelées *gardes* et les actions sont des fonctions de mise à jour des variables. La transition d'un automate fini étendu est franchissable si et seulement si sa garde est satisfaite.

Une expression linéaire sur X est une expression générée par la grammaire suivante, pour $k \in \mathbb{Z}$ et $x \in X$:

$$\lambda ::= k \mid k \times x \mid \lambda + \lambda$$

Une contrainte linéaire sur X est une expression générée par la grammaire suivante avec λ une expression linéaire sur X, $\sim \in \{>, <, =, \geq \leq\}$ [1] et $a \in \mathbb{N}$:

$$\gamma ::= \lambda \sim a \mid \gamma \wedge \gamma$$

Nous notons $\mathcal{G}(X)$ l'ensemble des contraintes linéaires sur X. Une mise à jour de la variable $x \in X$ est définie par $x := k$ avec $k \in \mathbb{Z}$.
Nous notons $\mathcal{U}(X)$ l'ensemble des ensembles cohérents de mises à jour sur X définis par $\pi \in \mathcal{U}(X)$ ssi $\forall x \in X$, si $(x := k) \in \pi$ alors $\forall k' \neq k$, $(x := k') \notin \pi$.

Définition 2. *(Automate fini étendu).*
Un automate fini étendu F est un tuple $(S, s_0, X, \Sigma, \pi_0, \longrightarrow)$, tel que :
- *S est un ensemble fini d'états.*
- *$s_0 \in S$ est l'état initial.*
- *X est un ensemble fini de variables prenant un ensemble fini de valeurs* [2].
- *Σ est un ensemble fini d'événements.*
- *$\pi_0 \in \mathcal{U}(X)$ est l'ensemble des affectations initiales sur X.*
- *$\longrightarrow \subseteq S \times \mathcal{G}(X) \times \Sigma \times \mathcal{U}(X) \times S$ est un ensemble fini de transitions.*

1. Dans les figures, nous utiliserons == plutôt que = pour être proche de la syntaxe des gardes utilisées dans l'outil UPPAAL.
2. Nous considérerons les entiers relatifs de valeur absolue bornée

Soit $t = \langle s, g, \sigma, \pi, s' \rangle \in \longrightarrow$ représentant une transition allant de s à s' avec la garde g, l'événement σ et la mise à jour π. Si g est absent, la relation de transition est écrite $s \xrightarrow{\sigma,\pi} s'$ et il est supposé que g est toujours satisfait. Si π est absent (ou $\pi = \emptyset$), la relation de transition est écrite $s \xrightarrow{g,\sigma} s'$. Ainsi, aucune variable n'est mise à jour quand une telle transition est franchie. Une valuation v sur X est un élément de $\mathbb{Z}^{|X|}$, cela peut être vu comme un vecteur d'entiers de taille $|X|$. Étant donné une contrainte $\varphi \in \mathcal{G}(X)$ et une valuation $v \in \mathbb{Z}$, nous notons $\varphi(v) \in \{\mathbf{true}, \mathbf{false}\}$, la vraie valeur obtenue par la substitution de chaque occurrence de $x \in \varphi$ par $v(x)$. Nous considérons que $[\![\varphi]\!] = \{v \in \mathbb{Z} \mid \varphi(v) = \mathbf{true}\}$ (i.e. toutes les valuations v qui satisfont φ).
Pour $x \in X$, nous notons $v[x := k]$ la nouvelle valuation v' telle que pour tout $x' \in X$, $v'(x') = k$ si $x' = x$ et $v'(x') = v(x')$ autrement. Par extension, nous notons par $v[\pi]$ une valuation obtenue à partir de v en appliquant l'ensemble des mises à jour de π.

$\overline{0}$ est une valuation nulle avec $\forall x \in X, \overline{0}(x) = 0$. Ainsi, pour une affectation initiale π_0, la valuation initiale est obtenue à partir de la valuation nulle $\overline{0}$ par $\overline{0}[\pi_0]$.

Définition 3. *(Sémantique d'un automate fini étendu)*
La sémantique d'un automate fini étendu $(S, s_0, X, \Sigma, \pi_0, \longrightarrow)$ *est un système de transition* $S_H = (Q, q_0, \rightarrow)$ *tel que :*
- $Q = S \times (\mathbb{Z})^X$
- $q_0 = (s_0, \overline{0}[\pi_0])$ *est l'état initial.*
- $\rightarrow$ *est la transition définie pour tout* $\sigma \in \Sigma$ *par* $(s, v) \xrightarrow{\sigma} (s', v')$ *avec* $s \xrightarrow{g,\sigma,\pi} s'$, $g(v) =$ ***true***, $v' = v[\pi]$.

3.1.3 Réseau d'automates finis étendus

En général, le modèle d'un système complexe est conçu à partir de la modélisation des sous systèmes le constituant.

Dans notre cas, puisque nous modélisons entièrement un système d'exploitation à l'aide d'automates finis étendus, il est plus judicieux que chaque composant du système d'exploitation soit modélisé par un automate fini étendu et, par la suite, composer tous les modèles obtenus afin de former le modèle complet du système d'exploitation. Nous définissons pour cela *un réseau d'automates finis.*

Soit $F_1, ..., F_n$ n automates finis étendus tels que $F_i = (S_i, s_{i0}, X, \Sigma, \pi_{i_0}, \longrightarrow_i)$. Une fonction de synchronisation f est une fonction partielle $(\Sigma \cup \{\bullet\})^n \rightarrow \Sigma$ où $\bullet$ est un symbole spécial utilisé quand un automate n'est pas impliqué lors de l'évolution du système global. Notons que f est une fonction de synchronisation avec renommage. Nous notons par $(F_1 \mid ... \mid F_n)_f$ la composition parallèle des $F_i's$ faisant référence à f. $(F_1 \mid ... \mid F_n)_f$ est un automate fini étendu et un état de $(F_1 \mid ... \mid F_n)_f$ est noté $(s_1, ..., s_n) \in S_1 \times ... \times S_n$.

Définition 4. *(Produit synchronisé d'automates finis étendus).*
Soit $F_1, ..., F_n$, *n automates finis étendus avec* $F_i = (S_i, s_{i0}, X, \Sigma, \pi_{i_0}, \longrightarrow_i)$, *et f une fonction de synchronisation partielle* $(\Sigma \cup \{\bullet\})^n \rightarrow \Sigma$.
Le produit synchronisé $(F_1 \mid ... \mid F_n)_f$ *est un automate fini étendu* $\mathcal{A} = (S, s_0, X, \Sigma, \pi_0, \longrightarrow)$ *tel que :*
- $S = S_1 \times ... \times S_n$,
- $s_0 = (s_{10}, s_{20}, \cdots, s_{n0})$,
- $\pi_0 = \bigcup_{i=1 \rightarrow n} \pi_{i0}$
- $(s_1, s_2, \cdots, s_n) \xrightarrow{g,\sigma,\pi} (s'_1, s'_2, \cdots, s'_n)$ *ssi*
  - $\exists(\sigma_1, \ldots, \sigma_n) \in (\Sigma \cup \{\bullet\})^n$ *tel que* $f(\sigma_1, \sigma_2, \cdots \sigma_n) = \sigma$,
  - *pour tout i :*

  - — *Si* $\sigma_i = \bullet$ *alors* $s_i = s'_i$, $g_i = \textbf{\textit{true}}$, $\pi_i = \emptyset$,
  - — *Si* $\sigma_i \in \Sigma$ *alors* $s_i \xrightarrow{g_i,\sigma_i,\pi_i}_i s'_i$.
- — $g = g_1 \wedge g_2 \cdots \wedge g_n$,
- — $\pi = \bigcup_{i=1 \to n} \pi_i$

Cohérence des ensembles de mises à jour Les variables X sont partagées par les automates du produit mais nous supposons que π_0 ainsi que les ensembles de mises à jour du produit synchronisé sont cohérents i.e. $\forall \xrightarrow{g,\sigma,\pi} \in \longrightarrow$, $\pi \in \mathcal{U}(X)$ ($\forall x \in X$ si $(x := k) \in \pi$ alors $\forall k' \neq k$, $(x := k') \notin \pi)$. Les automates que nous manipulons respectent cette hypothèse quel que soit la fonction de synchronisation car pour une variable donnée $x \in X$, tous les automates peuvent lire x (garde g) mais un seul automate F_i peut écrire x (par ses mises à jour π_i impliquant x). De plus π_0 est cohérent, car les automates ont tous les mêmes conditions initiales sur X $(\pi_0 = \pi_{10} \cdots = \pi_{i0} \cdots = \pi_{n0})$.

Dans ce manuscrit, nous utilisons une classe particulière de fonctions de synchronisation pour laquelle au plus deux automates sont impliqués dans l'évolution du système global. Quand deux automates sont impliqués sur une action σ, nous notons classiquement $\sigma?$ et $\sigma!$. Ainsi f est soit $f(\bullet, \bullet \cdots \sigma, \cdots, \bullet) = \sigma$ soit $f(\bullet, \bullet \cdots \sigma?, \cdots, \bullet, \cdots, \sigma!, \cdots, \bullet) = \sigma$. Par conséquent, pour être concis dans la suite du manuscrit, nous omettrons la fonction de synchronisation et utiliserons la notation $\sigma?\sigma!$.

3.1.4 Utilisation des automates finis étendus pour l'abstraction d'un programme

Les automates finis étendus sont utilisés pour l'abstraction de programmes séquentiels. La structure et les gardes d'un automate fini étendu décrivent le flot de contrôle du programme. De ce flot de contrôle, le code séquentiel peut être abstrait par des actions tant que la consistance des mises à jour des variables est respectée. Alors une suite d'instructions peut être abstraite par une même action et leur exécution sera considérée comme atomique.

3.1.4.1 Atomicité et mise à jour

Une mise à jour π sur X associée à une transition est un ensemble de mises à jour sur X. Comme mentionné dans la sémantique d'un automate fini étendu, le franchissement de la transition t conduit systématiquement à $v' = v[\pi]$.

Supposons un automate fini étendu qui contient 2 états s_1 et s_2 et deux variables x et y qui prennent pour valeur initiale 0 et une transition $s_1 \xrightarrow{a,\{x=2;y=3\}} s_2$. Les états pour lesquels $x == 2$ et $y == 0$ ou $x == 0$ et $y == 3$ ne font pas partie de l'espace d'état du modèle.

Cela impose quelques restrictions sur le code qui peut être abstrait par un automate fini étendu.

3.1.5 Accessibilité et bornitude

Définition 5. *(Accessibilité)*
Soit $\longrightarrow^*$ *une clôture réflexive transitive de* $\longrightarrow$*. Pour un automate fini ayant pour état initial* s_0*, un état* s *est dit accessible si et seulement si* $s_0 \longrightarrow^* s$*, autrement,* s *est inaccessible.*

Définition 6. *(Bornitude)*
Un automate fini étendu $(S, s_0, X, \Sigma, \pi_0, \longrightarrow)$ *est dit borné si pour tous ses états accessibles, toutes les variables entières* $x \in X$ *sont bornées. Lorsque la borne est* k*, on peut affirmer que l'automate fini étendu est* $k-$*borné.*

La bornitude est en général non décidable pour les automates finis étendus. Cependant, un automate fini étendu peut être supposé borné si par définition, les variables prennent leurs valeurs dans un ensemble fini de valeurs (comme des entiers bornés).

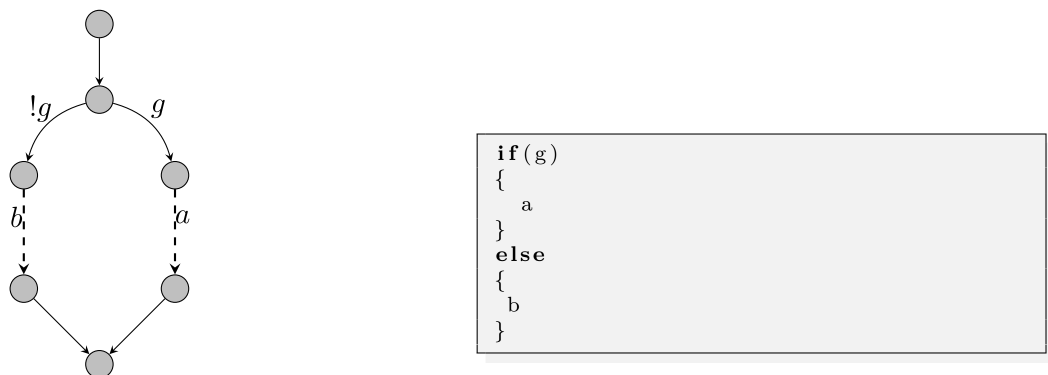
Proposition 1. *L'accessibilité est décidable pour un automate fini borné.*

Démonstration. Pour un automate fini étendu k-borné, le nombre d'états et l'espace d'état est borné par $|S|.k^{|X|}$, l'espace d'état est alors fini. □

3.2 Principes de la modélisation

3.2.1 Modélisation des branchements

Les instructions `if` et `if else` représentent un branchement car elles permettent d'orienter l'exécution du programme vers un traitement particulier ou vers un autre. Elles peuvent être parfaitement représentées par des automates finis étendus grâce aux gardes dont disposent ceux-ci. Soit g une garde traduisant la condition `if` à satisfaire. La figure 3.1 illustre la description d'un branchement par un automate fini étendu.

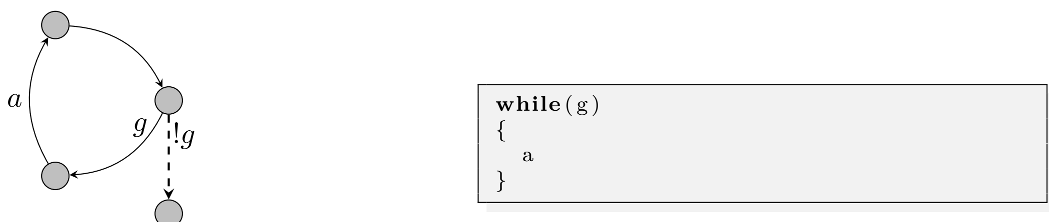



```
if(g)
{
  a
}
else
{
 b
}
```

FIGURE 3.1 – Représentation d'un branchement par un automate fini étendu.

3.2.2 Modélisation des boucles

Les instructions `for` et `while` permettent l'exécution d'une séquence d'instructions jusqu'à satisfaire une certaine condition. La figure 3.2 représente la description de la boucle `while` par un automate fini étendu. La boucle `for` est exactement représentée sous cette même forme.




```
while(g)
{
  a
}
```

FIGURE 3.2 – Représentation d'une boucle par un automate fini étendu

3.2.3 Modélisation d'un appel de fonction

Chaque fonction est modélisée par un automate. L'appel de fonction se traduit par une synchronisation des automates modélisant chacune des fonctions impliquées via des actions de synchronisation et des variables partagées. Soient a et b deux fonctions telles que a appelle b au cours de son exécution. Soit x une variable partagée par les deux fonctions et $f(h?, h!)$ la fonction de synchronisation. L'appel de la fonction b par a est illustré à la figure 3.3.

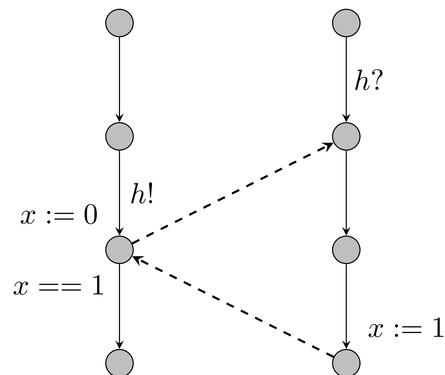


FIGURE 3.3 – Description d'un appel de fonction (fonction a à gauche et fonction b à droite) :*La variable partagée x permet le blocage de l'automate appelant jusqu'à ce que l'automate appelée termine son exécution. L'automate modélisant la fonction a reprend son exécution après la fin de l'exécution de l'automate modélisant la fonction b. Les actions* $h!$ *et* $h?$ *traduisent le point de synchronisation des deux automates.*

3.3 Adaptation à UPPAAL

3.3.1 UPPAAL : outil pour la vérification de systèmes temps réel

Développé conjointement par l'université d'Aalborg au Danemark et l'université d'Upsaala en suède, UPPAAL est un outil de modélisation et de vérification de systèmes temps réel à partir de réseaux d'automates temporisés incluant des variables bornées, des actions urgentes, des états urgents et différents mécanismes de synchronisation. Concernant la vérification, les propriétés pouvant être vérifiées sont essentiellement les propriétés d'accessibilité de vivacité ou d'absence d'état bloquant exprimées dans un sous ensemble de la logique *TCTL*(*Temporal Computation Tree Logic*). Dans la littérature, il existe plusieurs travaux qui décrivent l'outil [BLL+95, BLL+96, BDL04, BDL+01].

3.3.1.1 Description du modèle manipulé par UPPAAL

UPPAAL utilise un modèle qui est basé sur la théorie des automates temporisés présentée par *Alur* et *Dill* [AD94]. Le modèle est en effet une extension des automates finis avec des variables entières ou booléennes et des variables d'horloge qui servent à mesurer le temps. Tout comme les automates finis étendus, les variables manipulées par le modèle d'UPPAAL sont bornées afin de garantir la décidabilité des problèmes et la terminaison des procédures de vérification. Lorsqu'un système se trouve dans un état donné, les valeurs des variables d'horloge évoluent et représentent le temps écoulé dans cet état. Une transition entre deux états contient une garde sur les variables d'horloge qui compare leurs valeurs à des entiers. De plus, sur certaines transitions, les valeurs

des variables d'horloge peuvent être remises à 0. Les automates temporisés contiennent aussi des invariants qui sont des contraintes sur les variables d'horloge associées aux états. Les invariants permettent de restreindre le temps qu'un système peut prendre dans un état donné. D'une manière plus formelle, nous définissons les automates temporisés comme suit :

Définition 7. *(Automate temporisé) Soit H un ensemble de variables d'horloge et $B(H)$ l'ensemble des contraintes linéaires de la forme $x \sim a$ et $x - y \sim a$ où $x, y \in H$, $\sim \in \{>, <, =, \geq \leq\}$ et $a \in \mathbb{N}$. Un automate temporisé AT est un tuple $(L, l_0, \Sigma, T, g, r, Inv)$ où L est un ensemble d'états, $l_0 \in L$ est l'état initial, Σ est un ensemble fini d'actions, $g : T \mapsto B(H)$ associe une garde aux transitions, $r : T \mapsto 2^H$ est une fonction de mise à jour des variables d'horloge de la transition T, $Inv : L \mapsto B(H)$ est une fonction qui attribue un invariant à chaque état et $T \subseteq L \times \Sigma \times B(H) \times 2^H \times L$ est un ensemble fini de transitions.*

Si $(l, \sigma, \gamma, r, l') \in T$ alors, il existe une transition allant de l à l' ayant pour action σ, une garde γ et un ensemble de mises à jour des variables d'horloge r.

Définition 8. *(Sémantique d'un automate temporisé) Soit $v : H \mapsto \mathbb{R}_{\geq 0}$ une valuation d'horloge qui attribue une valeur réelle et non négative à chaque variable d'horloge et $v_0(x) = 0 \ \forall x \in H$. La sémantique d'un automate temporisé est un système de transition $(S, S_0, \rightarrow)$ où $S = L \times \mathbb{R}^H$ est un ensemble d'états, $s_0 = (l_0, v_0)$ est l'état initial et $\rightarrow \subseteq S \times S$ est la relation de transition définie par :*

- *$(l, v) \rightarrow (l, v + d)$ si $v \in Inv(l)$ et $v + d \in Inv(l)$.*
- *$(l, v) \rightarrow (l', v')$ si $t = (l, l') \in E$ tel que $v' \in g(t), [r(l) \rightarrow 0]v$, et $v' \in Inv(l')$*

où pour $d \in \mathbb{R}$, $v + d$ correspond à la valeur $u(x) + d$ de chaque variable d'horloge x et $[r \rightarrow 0]v$ dénote la valuation de l'horloge qui correspond à la valeur 0 de chaque variable d'horloge dans r.

Les automates manipulés par UPPAAL sont une extension par le temps, des automates finis étendus.

3.3.1.2 Quelques caractéristiques de UPPAAL

UPPAAL définit plusieurs notions pour la modélisation de systèmes temps réel. Parmi elles, nous notons :

La synchronisation binaire Dans UPPAAL, un canal de synchronisation ou *channel* permet de réaliser la synchronisation entre deux transitions contenant chacune des actions d'émission et de réception. Par exemple, un canal de synchronisation *c* est déclaré `chan c`. Ainsi, une transition contenant une action `c!` se synchronise à une autre transition contenant une action `c?`. Une paire de synchronisation est choisie de façon non déterministe si plusieurs combinaisons sont actives.

Les canaux de diffusion ou *broadcast channels* Dans une synchronisation par *broadcast channels*, une transition émettrice comportant une action `c!` peut se synchroniser à une ou plusieurs transitions réceptrices contenant une action `c?`. S'il n'existe aucune transition réceptrice au moment de la synchronisation, la transition émettrice peut effectuer l'action `c!`. On dit alors qu'un *broadcast channel* n'est jamais bloquant.

Les canaux urgents ou *urgent channels* Ce type de canal permet de réaliser une synchronisation *urgente*. Une synchronisation urgente de deux transitions ne doit pas être retardée. Les transitions comportant des *urgent channels* ne doivent contenir aucune contrainte sur le temps i.e. aucune garde. Dans la suite, nous appellerons les transitions comportant des *urgent channels* des transitions urgentes.

Les états urgents ou *urgent states* Sémantiquement, les états urgents sont équivalents à un état normal auquel on ajoute une variable d'horloge supplémentaire x qui est remise à 0 sur toutes ses transitions entrantes et ayant pour invariant $x \leq 0$. Ainsi, l'écoulement du temps n'est pas permis lorsqu'un système se retrouve dans un tel état. Un état urgent est toujours marqué par un "U".

Les états engagés ou *Committed states* À l'exécution, ces états sont plus restrictifs que les états urgents. Lorsque dans un réseau d'automates temporisés, plusieurs transitions sont franchissables au même moment et qu'il existe un état engagé qui a aussi sa transition sortante qui est franchissable, cette transition est franchie avant toutes les autres transitions. Ainsi, un automate se trouvant dans un état engagé doit le quitter immédiatement. Un état engagé est toujours marqué par un "C".

3.3.1.3 Le *Model-checker* d'UPPAAL

Le but principal du model-checker est de vérifier des propriétés sur le modèle afin de prouver que le modèle repond bien aux exigences souhaitées. Comme le modèle, les propriétés à vérifier doivent être formellement décrites dans un langage bien défini. Pour spécifier formellement les propriétés à vérifier, UPPAAL utilise une version simplifiée du langage TCTL (*Timed Computation Tree Logic*) qui est une extension du langage CTL (*Computation Tree Logic*). UPPAAL permet la vérification de plusieurs types de propriétés à savoir, les propriétés d'accessibilité, de sûreté et de vivacité.

Propriétés d'accessibilité Les propriétés d'accessibilité sont, pour de nombreux systèmes les propriétés les plus simples à vérifier. Elles permettent d'indiquer si à partir de l'état initial d'un système, il existe un état du système satisfaisant une propriété φ. Plus simplement, elle indique si un état du système est accessible ou non à partir de l'état initial (par ex. l'état critique du système peut être atteint). La formule permettant de vérifier une propriété d'accessibilité est notée $E \diamond \varphi$. En utilisant la syntaxe d'UPPAAL, la propriété s'écrit $E <> \varphi$.

Propriétés de sûreté Les propriétés de sûreté expriment que quelque chose de mauvais n'arrivera jamais (par ex. une imprimante ne peut être accessible simultanément par deux utilisateurs). Dans UPPAAL, ces propriétés sont formulées d'une autre manière (par ex. quelque chose de bon se produira toujours). Pour exprimer qu'une propriété φ est satisfaite pour tous les états accessibles, la formule est écrite $A\Box\varphi$. Selon la syntaxe d'UPPAAL, elle est écrite $A[]\varphi$.

Propriétés de vivacité Les propriété de vivacité expriment que quelque chose arrivera sûrement. Dans sa forme simple, une propriété de vivacité est écrite $A \diamond \varphi$ signifiant que la propriété φ est sûrement satisfaite. La forme la plus pratique est la propriété de réponse écrite $\varphi \leadsto \psi$ qui exprime que lorsque la propriété φ est satisfaite alors ψ sera sûrement satisfaite (par ex. un message envoyé est sûrement reçu). Dans UPPAAL les propriétés de vivacité sont écrites sous la forme $A <> \varphi$ et $\varphi - - > \psi$.

3.3.2 Le traitement du temps

Le modèle du noyau du système d'exploitation n'est pas temporisé. En effet, les variables d'horloge ne sont pas utilisées dans le modèle du système d'exploitation Trampoline. Par ailleurs, le modèle de Trampoline est constitué de transitions urgentes et d'états urgents.

Selon UPPAAL, la notion d'urgence est utilisée pour éviter l'écoulement du temps. Cela signifie que dans le modèle, l'exécution d'une séquence de code du point de vue des autres composants du modèle se fait en temps nul.

Cette abstraction est correcte car :

- Les variables du noyau ne sont pas partagées entre les différentes composantes du modèle. Ainsi, la date réelle de leurs mises à jour reste sans importance en dehors du noyau et l'ordre des opérations de lecture et de mise à jour des variables à l'intérieur du noyau est correcte.
- Quand le mode noyau est accédé, les tâches de l'application ne s'exécutent pas et ne peuvent donc effectuer d'appels système.
- Quand le mode noyau est accédé, les sources d'interruptions sont bloquées et les différentes requêtes issues des interruptions ne sont pas prises en compte jusqu'à ce que le mode noyau soit quitté.

En outre, dans un système réel, le pire temps d'exécution et toute séquence de code à l'intérieur du système d'exploitation - le temps de latence du noyau - devrait être plus petit que le temps minimum entre chaque arrivée d'interruptions. Dans un système d'exploitation temps réel typique, le système d'horloge est de deux ordres de grandeurs plus grand que le temps de latence du noyau. Autrement, le système passerait son temps à n'exécuter que le gestionnaire du système d'horloge.

Le temps est utilisé pour les sources d'interruptions et les tâches de l'application. Les variables d'horloge sont utilisées par d'autres composants du système tels que les sources d'interruptions et les tâches de l'application. L'utilisation des variables d'horloge permet d'éliminer certains comportements qui ne sont pas permis dans un système réel.

Concernant les sources d'interruptions, les variables d'horloge sont utilisées pour modéliser le temps minimum d'inter-arrivée entre deux requêtes d'interruptions. Si les sources d'interruptions n'étaient pas temporisées, l'espace d'état du modèle inclurait des interblocages actifs ou *livelock* dus au traitement d'un nombre infini des requêtes d'interruptions en un temps nul. Cette abstraction préserverait la recherche d'accessibilité des états du système d'exploitation mais augmenterait la taille de l'espace d'état.

Concernant les tâches de l'application, les variables d'horloge sont utilisées pour modéliser le pire temps de réponse des tâches. Cela est bien évidemment une sur-approximation du comportement du système réel qui inclut certains comportements impossibles (par ex. l'exécution en un temps nul de certaines tâches). Ainsi, cette abstraction préserve la recherche d'accessibilité des états du système d'exploitation temps réel. Si les tâches de l'application ne sont pas temporisées, plusieurs comportements impossibles tels que les *livelocks* dus à une infinité d'exécution peuvent faire partir de l'espace d'état du modèle.

3.3.3 Règles de modélisation en utilisant UPPAAL

Dans UPPAAL, le code impératif exprimé dans un sous ensemble du langage C est associé à une transition. Ce code est exécuté de façon séquentielle mais son exécution est considérée comme atomique dans l'espace d'état et respecte le principe de l'atomicité (voir 3.1.4.1).

Le code source entier du système d'exploitation est abstrait par un réseau d'automates finis étendus à partir des bases suivantes :

- La structure des automates décrit le flot de contrôle du système d'exploitation.
- Les variables utilisées dans le modèle sont les variables de contrôle du système d'exploitation.

— Les actions et les conditions sur les variables associées à chaque transition sont des actions et les conditions issues du code source du système d'exploitation.
— Le code associé à chaque transition des automates est similaire au code du système d'exploitation et doit être consistant vis-à-vis de l'atomicité du modèle.

Atomicité Dans les modèles UPPAAL, l'atomicité de la mise à jour associée à une transition est une propriété très importante car elle garantit l'absence d'entrelacement des mises à jour associées aux différentes transitions.

Dans la sémantique des automates finis étendus et du modèle UPPAAL, notre règle de modélisation doit garantir la consistance de notre modèle conformément à l'atomicité.

Afin de garantir que le code du système d'exploitation respecte l'atomicité du modèle, les règles suivantes sont définies :
— Le code associé à une transition est un code non interruptible du système d'exploitation conformément à l'atomicité.
— Si deux transitions sont synchrones (dans le sens du produit synchronisé), seulement l'une d'elles peut effectuer une mise à jour de variables. Le code du système d'exploitation correspondant à ce code associé est non interruptible et respecte l'atomicité.

3.4 Modèle de Trampoline dédié aux architectures monocœur

Le modèle de Trampoline monocœur est essentiellement composé de l'ensemble des services liés aux standard OSEK/VDX et du noyau. Il contient au total 84 automates finis étendus, 1 automate temporisé et 34 fonctions décrites sous une syntaxe proche du langage C. Dans cette section, nous présentons le modèle de Trampoline, à savoir la description des objets qu'il manipule, la représentation de pointeurs, la représentation de son noyau et enfin la représentation de ses services.

3.4.1 Modélisation des objets et des variables

3.4.1.1 Descripteurs d'objets

Les objets manipulés par Trampoline sont représentés par des descripteurs. Ces descripteurs sont des variables de type `structure` écrites en langage C et contiennent toutes les données des objets qu'ils décrivent.

Par exemple, dans Trampoline, une tâche est représentée par deux descripteurs. D'une part un descripteur statique qui contient les données de la tâche qui ne varient pas durant l'exécution du système d'exploitation à savoir, son type ou son identifiant. D'autre part, un descripteur dynamique qui contient les données qui peuvent varier durant l'exécution du système d'exploitation comme son état ou sa priorité (dans le cas où cette tâche accède à une ressource).

La représentation des objets dans le modèle est faite de façon similaire à celle faite dans Trampoline. La figure 3.4 montre les descripteurs de tâche dans le modèle. Ainsi, lors de la modélisation d'une tâche, elle est avant tout instanciée (dans le modèle) à partir de ses descripteurs dynamique et statique.

De plus, Trampoline utilise des tableaux pour ranger les différents descripteurs d'objets afin de faciliter leur manipulation. Dans le modèle, des tableaux sont aussi utilisés pour enregistrer les objets. En considérant toujours l'objet de type `tâche`, deux tableaux à savoir, `tpl_dyn_proc_table` et `tpl_stat_proc_table` enregistrent respectivement les descripteurs dynamique et statique des tâches. De cette manière, l'accès aux descripteurs d'une tâche se fait à partir de son identifiant.

```
typedef struct {
tpl_internal_resource internal_resource;
tpl_task_id id;
int base_priority;
int max_activate_count;
int type;
} tpl_proc_static;
```

Listing 3.1 – Descripteur statique

```
typedef struct {
  tpl_resource resources;
  tpl_task_id id;
  int activate_count;
  int prior;
  int state_d;
} tpl_proc;
```

Listing 3.2 – Descripteur dynamique

FIGURE 3.4 – Descripteurs statique et dynamique d'une tâche dans le modèle

3.4.1.2 Modélisation des pointeurs

La représentation des pointeurs dans le modèle est faite par des tableaux car il n'est pas possible dans UPPAAL de déclarer un pointeur. Par exemple un pointeur vers un objet de type `tâche` sera représenté par un tableau de type `tâche` dont chaque index sera l'identifiant de la *tâche* sur laquelle elle pointe (*cf.* figure 3.5).

```
CONSTP2VAR(tpl_task_id,
  AUTOMATIC, OS_APPL_DATA) task_id;
```

Listing 3.3 – Déclaration d'un pointeur vers un identifiant de tâches dans Trampoline

```
tpl_task_id task_id[ ];
```

Listing 3.4 – Représentation dans le modèle

FIGURE 3.5 – Représentation des pointeurs dans le modèle.

3.4.2 Modélisation du noyau

Le noyau de Trampoline est composé principalement de 3 modules (voir section 2.3.2). Dans cette section, nous détaillons la représentation de chacun des modules dans le modèle.

3.4.2.1 Modélisation du gestionnaire de tâches

Le gestionnaire de tâches contient un ensemble de fonctions manipulant les différentes tâches d'une application. Il gère l'activation, la synchronisation et la terminaison des tâches. Toutes les fonctions contenues dans le gestionnaire de tâches sont représentées dans le modèle à l'aide d'automates finis étendus.

La modélisation de la fonction du gestionnaire de tâches `tpl_terminate` est représentée à la figure 3.6.

Description de la fonction tpl_terminate Lorsque la fonction `tpl_terminate` est appelée, elle effectue ensuite l'appel de la fonction `tpl_release_internal_resource` et lui attribue comme paramètre la variable `tpl_kern.running_id` contenant l'identifiant de la tâche en cours d'exécution (voir section 2.3.2 concernant la structure de données `tpl_kern`).

La fonction `tpl_release_internal_resource` permet la libération de la ressource interne détenue par la tâche en cours d'exécution. Lorsqu'elle termine son exécution, la fonction `tpl_terminate` reprend son exécution. Ensuite, le compteur d'activation de la tâche en cours d'exécution est testé via une condition `if`. Si le compteur d'activation est supérieur à 0 (`tpl_kern.running.activate_count`

```
FUNC(void, OS_CODE) tpl_terminate(void)
{
#if WITH_AUTOSAR_STACK_MONITORING == YES
  tpl_check_stack((tpl_proc_id)tpl_kern.running_id);
#endif /* WITH_AUTOSAR_STACK_MONITORING */

  /*
   * a task switch will occur. It is time to call the
   * PostTaskHook while the soon descheduled task is RUNNING
   */
  CALL_POST_TASK_HOOK()

  tpl_release_internal_resource((tpl_proc_id)tpl_kern.running_id);

  /* and checked to compute its state. */
  if (tpl_kern.running->activate_count > 0)
  {
    tpl_kern.running->state = READY_AND_NEW;

#if EXTENDED_TASK_COUNT > 0
    if (tpl_kern.running_id < EXTENDED_TASK_COUNT)
    {
      CONSTP2VAR(tpl_task_events, AUTOMATIC, OS_APPL_DATA) events =
      tpl_task_events_table[tpl_kern.running_id];
      events->evt_set = events->evt_wait = 0;
    }
#endif
  }
  else
  {
    tpl_kern.running->state = SUSPENDED;
  }
   /* copy it in old slot of tpl_kern */
  tpl_kern.old = tpl_kern.running;
  tpl_kern.s_old = tpl_kern.s_running;

}
```

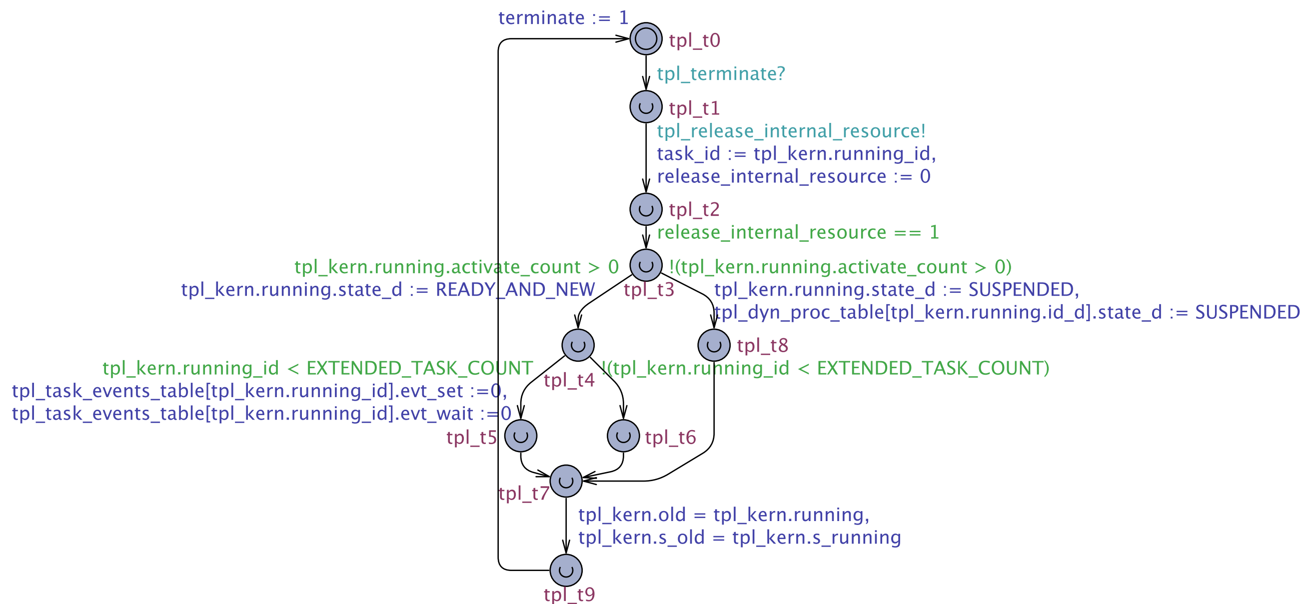


FIGURE 3.6 – Modélisation de la fonction du noyau `tpl_terminate`

`> 0`), la tâche en cours d'exécution passe à l'état `READY_AND_NEW` (`tpl_kern.running->state = READY_AND_NEW`). Dans le cas contraire, elle passe à l'état *suspendue* (`tpl_kern.running->state = SUSPENDED`).

Si la tâche en cours d'exécution est étendue (`tpl_kern.running_id < EXTENDED_TASK_COUNT`), les événements qui lui sont associés sont initialisés.

Enfin, les informations à propos de la tâche en cours d'exécution sont enregistrées afin de réaliser un changement de contexte si besoin est.

Description du modèle de la fonction tpl_terminate Nous rappelons que les états marqués par un U signifient qu'ils sont urgents. Le double cercle représente l'état initial de l'automate et les actions se terminant par `?` ou `!` sont des actions de synchronisation utilisées pour décrire les appels de fonctions (voir section 3.2.3). Les étiquettes en vert représentent les gardes et les actions en bleu représentent les actions de mise à jour.

Dans le modèle, l'action de synchronisation `tpl_terminate?` permet de réaliser l'appel de l'automate modélisant la fonction `tpl_terminate`.

L'action de synchronisation `tpl_release_internal_resource!` décrit l'appel de l'automate modélisant la fonction `tpl_release_internal_resource` par l'automate modélisant la fonction `tpl_terminate` et la variable `task_id` contient la valeur qui lui est passée en argument.

Ensuite, les conditions exprimées par l'instruction `if` sont décrites par des gardes (par ex. `tpl_kern.running.activate_count > 0`).

Enfin, les instructions de la fonction sont décrites par les fonctions de mise à jour (par ex. `tpl_kern.running.state_d := READY_AND_NEW`.

Cette figure montre le niveau d'abstraction d'une fonction de Trampoline par un automate. Nous pouvons remarquer que l'automate fini étendu embarque le code source de la fonction qu'il modélise. Toutes les autres fonctions contenues dans le gestionnaire de tâches sont décrites de la même manière.

3.4.2.2 Modélisation du gestionnaire de compteurs

Le gestionnaire de compteurs assure la gestion de toute interruption provenant du *Timer*. Dans le modèle, trois automates décrivent son fonctionnement. Il s'agit des automates `tpl_call_counter_tick`, `tpl_counter_tick` et `tpl_raise_alarm`.

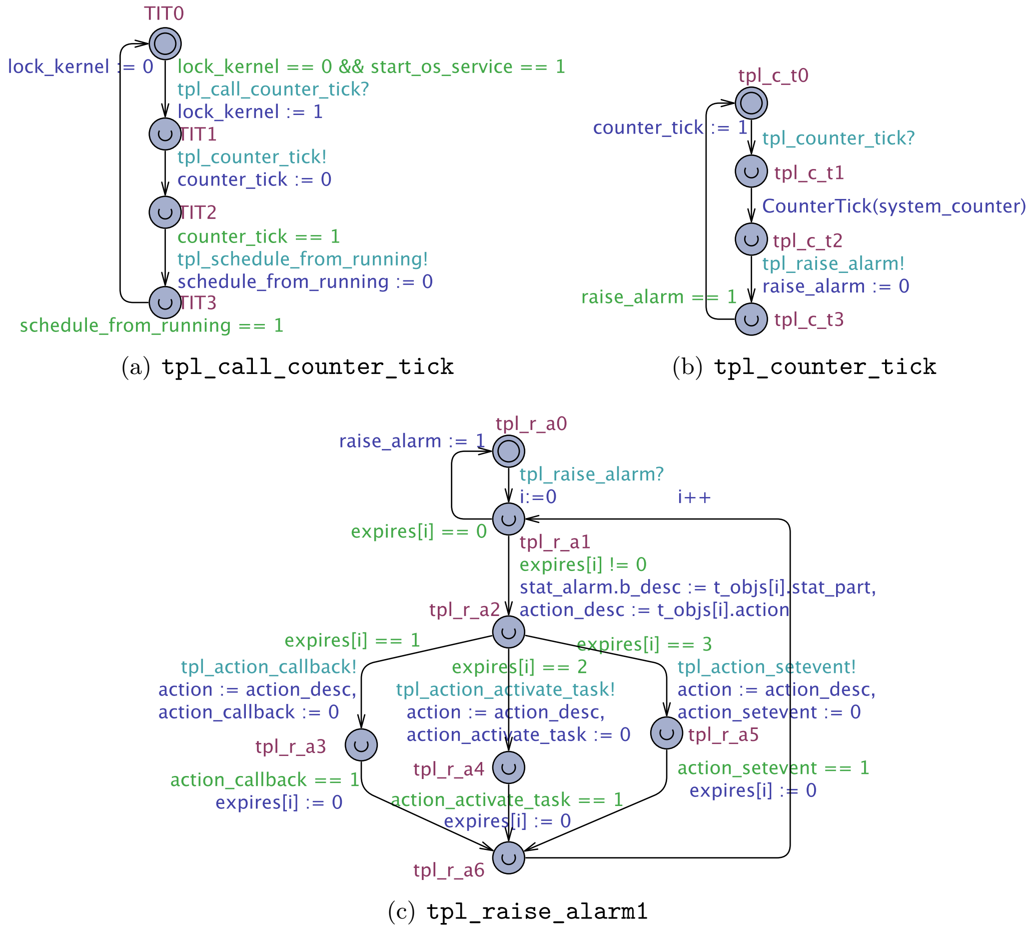
L'automate `tpl_call_counter_tick` est appelé suite à l'interruption liée au *Timer*. Il appelle ensuite l'automate `tpl_counter_tick` qui lui, permet d'incrémenter la valeur de l'ensemble des compteurs du système à partir de la fonction `CounterTick()` implémentée par une fonction UPPAAL. Ensuite, l'automate `tpl_raise_alarm` est appelé. Il permet l'exécution de l'action liée à l'ensemble des alarmes qui expirent à cet instant.

Enfin, l'automate `tpl_schedule_from_running` est appelé, car l'interruption causée par le *Timer* peut provoquer un réordonnancement.

3.4.2.3 Modélisation de l'ordonnanceur

L'ordonnanceur est constitué principalement de trois fonctions permettant la gestion de la liste des tâches prêtes (voir section 2.3.2). Toutes ces fonctions sont décrites dans le modèle par des fonctions UPPAAL. Le code de la fonction `tpl_get_proc` est représenté à la figure 3.8.

Nous remarquons une similarité entre le code de la fonction `tpl_get_proc` dans le modèle et dans Trampoline. La seule différence concerne l'utilisation du pointeur. Dans le code de Trampoline, `highest` est un pointeur vers la FIFO de la tâche de plus haute priorité. Dans le modèle, c'est une copie locale de la FIFO.



(a) `tpl_call_counter_tick` (b) `tpl_counter_tick`

(c) `tpl_raise_alarm1`

FIGURE 3.7 – Modèle du gestionnaire de compteur

Cette fonction est donc associée à une transition de l'automate `tpl_get_proc` et est exécutée de façon atomique lorsque la transition correspondante est traversée.

3.4.3 Modélisation des services

3.4.3.1 L'API

Trampoline dispose au total de 26 fonctions d'API qui permettent aux tâches de l'application d'accéder à ses services. Comme exemple, considérons l'API `TerminateTask` qui permet la terminaison d'une tâche en cours d'exécution. Sa description sous la forme d'un automate UPPAAL est représentée à la figure 3.9. À l'appel de cet automate, (`TerminateTask?`), il appelle aussitôt l'automate `tpl_terminate_task_service` pour exécuter le service demandé. La garde associée à la première transition de l'automate (`lock_kernel == 0`) permet de décrire le fait qu'un appel d'une fonction de l'API se fait toujours en mode utilisateur. La mise à jour de la variable `lock_kernel` (`lock_kernel := 1`) signifie qu'à l'appel d'une fonction de l'API, le processeur passe en mode noyau pour exécuter le service sollicité. Ainsi, lorsque la variable `lock_kernel` est égale à 1 (resp. à 0), cela signifie que l'exécution s'effectue en mode noyau (resp. en mode utilisateur)

```
tpl_proc_id tpl_get_proc_uppaal ()
{
 tpl_proc_id elected;
 tpl_proc_id
 highest[TASK_COUNT+ISR_COUNT+1];
 int[0,255] read_idx;

 highest=tpl_ready_list[tpl_h_prio].fifo;
 read_idx=tpl_fifo_rw[tpl_h_prio].read;
 elected = highest[read_idx];
 read_idx++;
 if (read_idx>=
  tpl_ready_list[tpl_h_prio].size)
  { read_idx=0; }
 tpl_fifo_rw[tpl_h_prio].read=read_idx;
 tpl_fifo_rw[tpl_h_prio].size--;
 while ((tpl_h_prio>=0) &&
 (tpl_fifo_rw[tpl_h_prio].size==0))
  { tpl_h_prio--; }
  return elected;
}
```

```
FUNC(VAR(tpl_proc_id, AUTOMATIC),
         OS_CODE) tpl_get_proc(void)
{
 VAR(tpl_proc_id, AUTOMATIC) elected;
 P2VAR(tpl_proc_id, AUTOMATIC,
 OS_APPL_DATA) highest;
 VAR(u8, AUTOMATIC) read_idx;
 highest=tpl_ready_list[tpl_h_prio].fifo;
 read_idx=tpl_fifo_rw[tpl_h_prio].read;
 elected=highest[read_idx];
 read_idx++;
 if (read_idx >=
  tpl_ready_list[tpl_h_prio].size)
 { read_idx = 0; }
 tpl_fifo_rw[tpl_h_prio].read=read_idx;
 tpl_fifo_rw[tpl_h_prio].size--;
 while ((tpl_h_prio >= 0) &&
 (tpl_fifo_rw[tpl_h_prio].size==0))
 { tpl_h_prio--; }
 return elected;
}
```

FIGURE 3.8 – Code de la fonction `tpl_get_proc` dans le modèle (gauche) et dans Trampoline (droite)

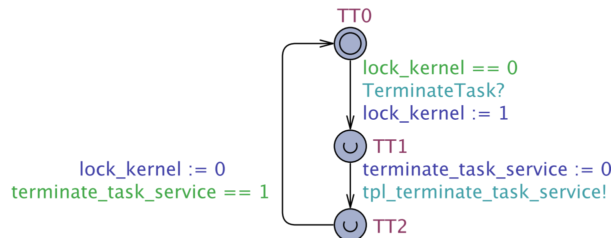


FIGURE 3.9 – Modèle de l'API TerminateTask

3.4.3.2 Les services

L'automate `tpl_terminate_task_service` est appelé suite à l'appel de l'automate `TerminateTask` (voir figure 3.10). Lors de son appel, nous vérifions d'abord si les interruptions ne sont pas désactivées (`CHECK_INTERRUPT_LOCK (result)`) et si le processus qui fait appel à ce service est bien une tâche (`CHECK_TASK_CALL_LEVEL_ERROR (result)`) car l'appel peut provenir d'une interruption de catégorie 2. Nous vérifions également si la tâche ne détient aucune ressource (`CHECK_RUNNING_OWNS_REZ_ERROR(result)`).

Si aucune erreur n'est détectée (`result == E_OK`), le compteur d'activation de la tâche est décrémenté (`tpl_kern.running.activate_count`–) puis l'automate modélisant la fonction du gestionnaire de tâches `tpl_terminate` est appelé. Ensuite, l'automate `tpl_get_proc` est appelé pour retirer de la liste des tâches prêtes celle ayant la priorité la plus élevée. Cette tâche est enfin mise en exécution grâce à l'automate `tpl_start`. Dans le cas où une erreur survient lors de la phase de vérification en début de la fonction, elle se termine immédiatement. La fonction `PROCESS_ERROR(result)` (écrite selon la syntaxe de UPPAAL) traite le résultat et le renvoie ensuite dans la variable `return` qui est retournée par l'automate `tpl_terminate_task_service`.

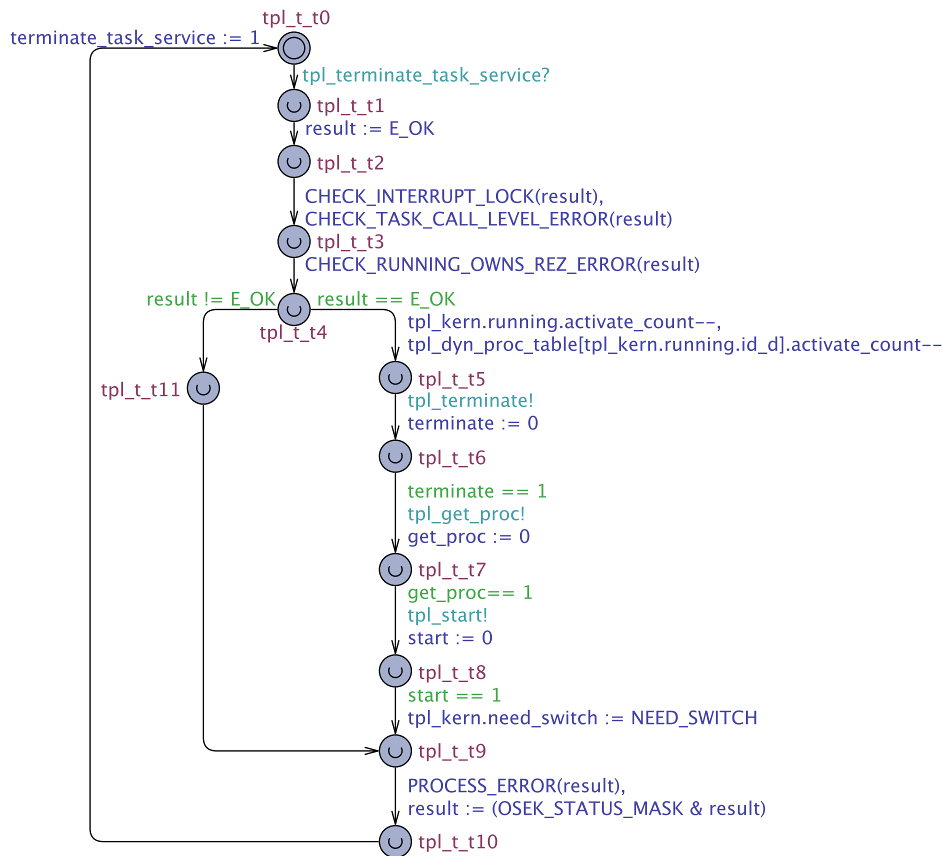


FIGURE 3.10 – Modèle du service de terminaison d'une tâche (`tpl_terminate_task_service`)

3.4.4 Modélisation de l'interruption matérielle liée au timer

Au moins une interruption matérielle est toujours utilisée par le système d'exploitation : il s'agit du *Timer*. Le *Timer* est modélisé par un automate temporisé (voir 3.11) ne contenant qu'un seul état. Une variable d'horloge appelée *timer* est utilisée pour franchir la transition chaque une unité de temps. Le franchissement d'une transition correspond au déclenchement d'une interruption matérielle. À chaque fois que l'interruption est déclenchée, l'automate `tpl_call_counter_tick` est appelé (voir figure 3.7) afin d'incrémenter le temps et de libérer tous les événements qui doivent se produire à cet instant.

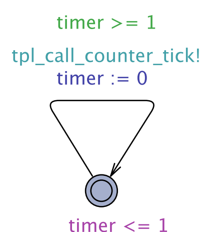


FIGURE 3.11 – Modèle du *Timer*

3.5 Modèle de Trampoline dédié aux architectures multicœur

Le modèle de Trampoline multicœur comporte 120 automates finis étendus, 1 automate temporisé et 67 fonctions décrites sous une syntaxe proche du langage C. La version multicœur de Trampoline est une suite de sa version monocœur. Des modules issus de la version monocœur ont été modifiés pour l'adapter au cas multicœur. La version de Trampoline multicœur a été développée de sorte à supporter aussi le cas monocœur. Il existe ainsi un ensemble de macros assurant le passage du multicœur au monocœur et inversement. Dans cette section, nous présentons brièvement le modèle de la version multicœur de Trampoline.

3.5.1 Modélisation du noyau

Le noyau est la partie ayant subi le plus de modifications pour le passage au cas multicœur. La structure de données (liste des processus prêts) manipulée par l'ordonnanceur a été complètement réécrite, les fonctions contenues dans le gestionnaire de tâches ont aussi été modifiées. La figure 3.12 montre un exemple de modification apportée à l'automate modélisant la fonction `tpl_terminate` du gestionnaire de tâches.

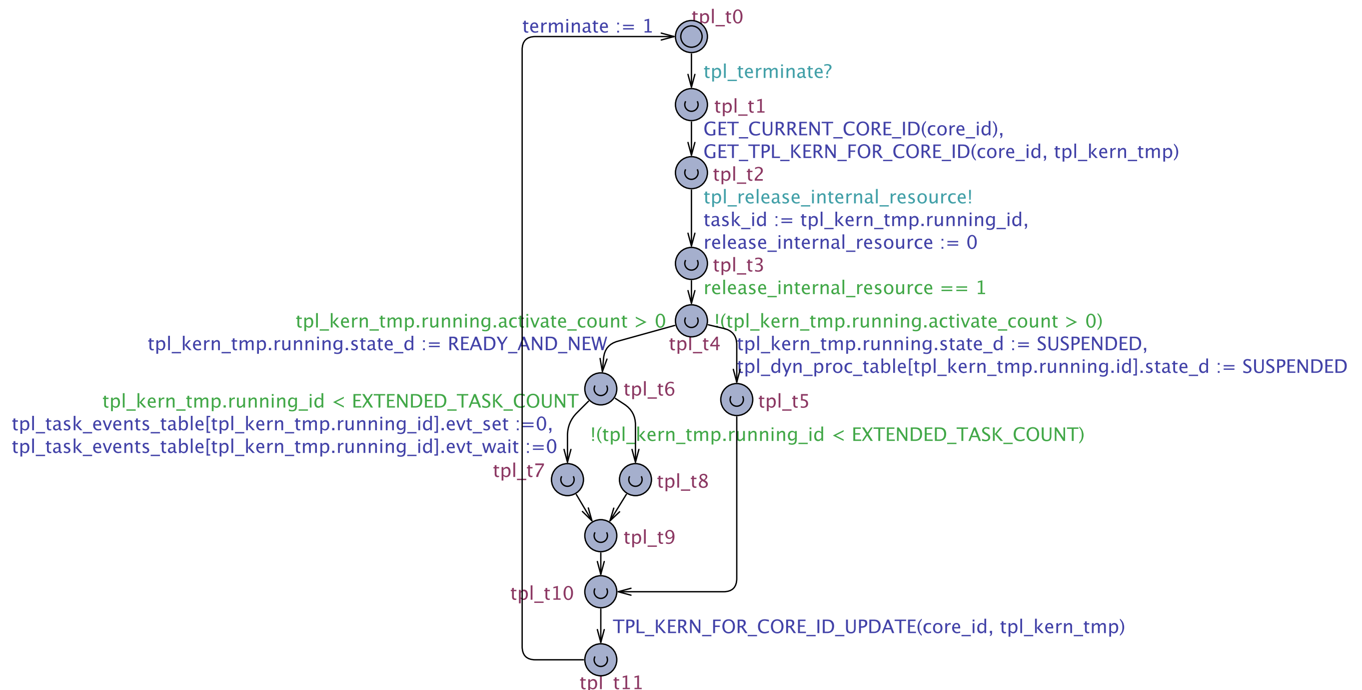


FIGURE 3.12 – Modèle de la fonction `tpl_terminate` dans le cas multicoeur

Le fonctionnement de cet automate reste le même que dans le cas monocœur (*cf.* figure 3.6) à l'exception des différentes macro qui ont été ajoutées. il s'agit de :

- `GET_CURRENT_CORE_ID (core_id)` : Cette macro permet de récupérer l'identifiant du cœur sur lequel le service est demandé. L'identifiant est alors inséré dans la variable (`core_id`). Cette macro a été écrite dans le modèle sous la syntaxe du langage C.
- `GET_TPL_KERN_FOR_CORE_ID()` : Dans la version monocœur de Trampoline, il existe une structure de données appelée `tpl_kern` qui enregistre des informations (par ex. identifiant, priorité etc.) sur la tâche détenant le processeur. Puisque dans la version multicœur, il y a autant de tâches en exécution que de cœurs, chaque cœur contient une structure de

données contenant des informations sur la tâche en exécution. Cette macro permet de récupérer la structure de données (`tpl_kern`) correspondant à un cœur donné.

— `TPL_KERN_FOR_CORE_ID_UPDATE()` : Cette fonction réalise la mise à jour des informations contenues dans la structure de données `tpl_kern` correspondant au cœur ayant pour identifiant `core_id`.

3.5.2 Modélisation des services

La figure 3.13 présente le modèle du service de la fonction `tpl_terminate_task_service` de la version multicoeur. Son fonctionnement est similaire à celui de sa version monocœur (*cf.* figure 3.10). Elle contient des macros d'adaptation au cas multicœur, à savoir la macro `GET_CURRENT_CORE_ID` et la macro `TPL_KERN` qui permet de retrouver directement en fonction de l'identifiant du cœur `core_id` la structure de donnée `tpl_kern` correspondante.

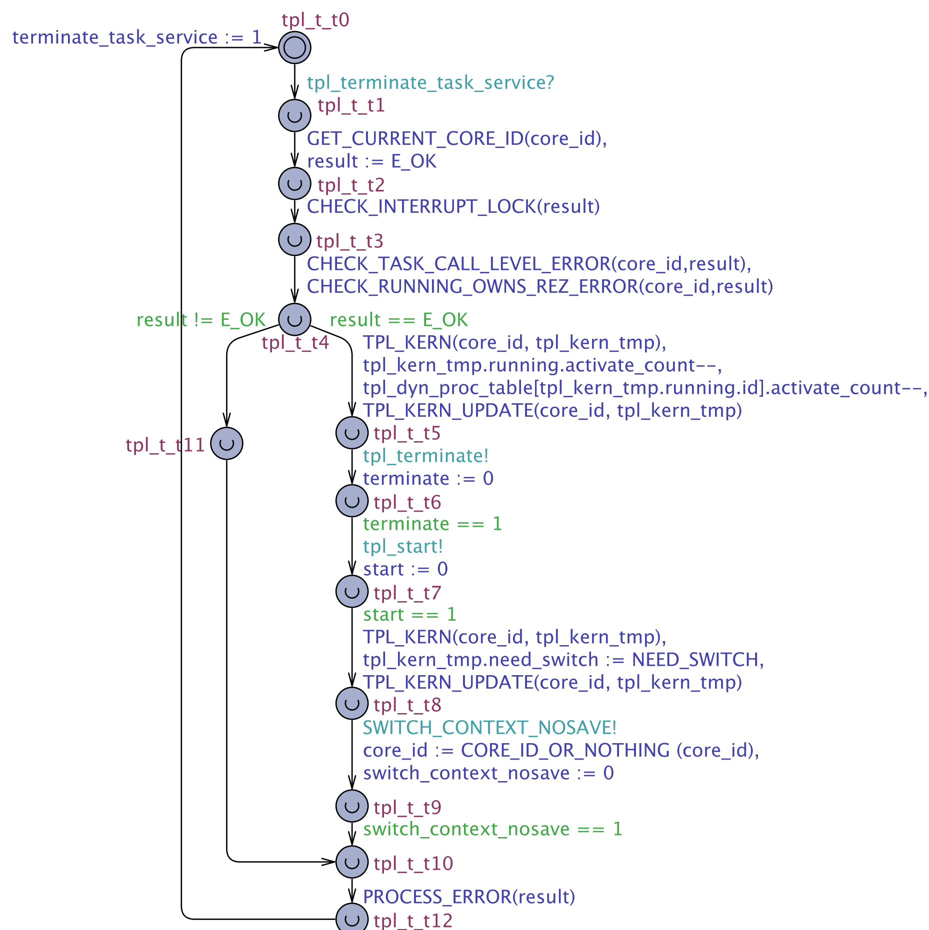


FIGURE 3.13 – Modèle de la fonction `tpl_terminate_task_service` dans le cas multicoeur

3.6 Modèle d'applications

L'application est caractérisée par un ensemble de tâches s'exécutant de façon concurrente. Les tâches interagissent avec le système d'exploitation au moyen des appels système. Dans Trampoline, une application est décrite d'une part dans un fichier OIL qui contient la déclaration

de tous les objets qu'elle manipule et d'autre part dans un fichier C contenant le code source de toutes les tâches qui la caractérisent. Pour décrire le modèle de l'application, nous extrayons du fichier OIL les instances des structures de données correspondantes comme les descripteurs de tâches et à partir du code source nous construisons un automate.

La figure 3.14 montre le principe de modélisation d'une tâche écrite sous Trampoline.

Le modèle de l'application est réalisé pour chaque système. Il doit décrire toutes les séquences possibles d'appels des fonctions de l'API du système d'exploitation qui sont utiles pour l'application. Afin d'éliminer certains comportements impossibles du système et ainsi réduire l'espace d'état, l'application est modélisée par des automates temporisés. Le modèle contient en effet une variable d'horloge x qui est mise à 0 quand la tâche est activée. Alors, chaque état contient un invariant de la forme $x_i <= WCRT_i$ où $WCRT_i$ désigne le pire temps de réponse de la tâche i. Comme énoncé précédemment (voir la section 3.3), cela est une sur-approximation du comportement réel de la tâche car elle décrit tous les comportements possibles qui permettraient à la tâche de se terminer avant son pire temps de réponse. Notons aussi que l'ordre d'exécution des tâches du modèle est cohérent grâce à la modélisation de la politique d'ordonnancement.

Le modèle de l'application décrit donc l'ensemble des appels système effectués par les tâches d'une application. Cela est suffisant comme information pour mener à bien la synthèse du système d'exploitation en fonction des besoins de l'application. Considérons maintenant le modèle des tâches 1 et 2 de l'application. La fonction `RunningTask()` est exprimée comme garde sur chacune des transitions du modèle. En fonction de l'ordonnancement, elle permet de n'exécuter qu'un seul automate à la fois. L'automate s'exécutant est toujours celui modélisant la tâche qui doit être en cours d'exécution. La variable `TaskPar` représente le paramètre attribué à la fonction `ActivateTask` lors de son appel.

Décrivons brièvement l'exécution de cette application :
La tâche 1 est activée au démarrage du système car elle est qualifiée comme AUTOSTART (*cf.* Description OIL), elle active ensuite la tâche 2 via la fonction de l'API `ActivateTask`. Suite à l'activation de la tâche 2, la tâche 1 est préemptée par celle-ci car elle présente une priorité plus basse. La tâche 2 commence alors son exécution puis se termine après l'appel de la fonction de l'API `TerminateTask`. Après la terminaison de la tâche 2, un réordonnancement se produit et la tâche 1 est remise en exécution jusqu'à se terminer après l'appel de la fonction `TerminateTask`.

Cet exemple relativement simple nous montre le niveau d'abstraction d'une application. Le modèle d'une application peut aussi contenir des branchements, des boucles et des tâches communicant entre elles via des variables partagées.

3.7 Propriétés du modèle

Soit X l'ensemble des variables du système complet composé du système d'exploitation et de l'application. Soit $OS_1, ..., OS_n$ l'ensemble des n automates finis étendus $OS_i = (S_i^{os}, s_{i0}^{os}, X, \Sigma, \pi_{i_0}^{os}, \longrightarrow_i^{os}$) modélisant les différents composants du système d'exploitation et *Timer* un automate fini étendu temporisé modélisant le *timer* (ou un ensemble de *timer*). *Timer* est un automate fini étendu temporisé mais pour être concis, nous l'assimilerons à un automate fini étendu $T = (S^T, s_0^T, X, \Sigma, \pi_0^T, \longrightarrow^T)$.

Soit $TA_1, ..., TA_k$ k automates finis étendus temporisés modélisant les tâches de l'application. Une fois de plus, pour plus de concision, nous les assimilerons à des automates finis étendus $A_i = (S_i^a, s_{i0}^a, X, \Sigma, \pi_{i_0}^a, \longrightarrow_i^a)$.

Le modèle complet est le produit $\mathcal{A}$=$(OS_1 \parallel ... \parallel OS_n \parallel Timer \parallel A_1 \parallel ... \parallel A_n)_f$ où les synchronisations sont réalisées comme défini dans la section 3.1.3 par les actions de synchronisation

```
APPMODE std {};

TASK Task1 {
    PRIORITY = 1;
    AUTOSTART = TRUE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
};

TASK Task2 {
    PRIORITY = 2;
    AUTOSTART = FALSE;
    ACTIVATION = 1;
    SCHEDULE = FULL;
};
```

Listing 3.5 – Description OIL de l'application

```
TASK(Task1){

    ActivateTask(Task2);
    .....
    TerminateTask()
};

TASK(Task2){
    .....
    TerminateTask()
};
```

Listing 3.6 – Code source de l'application

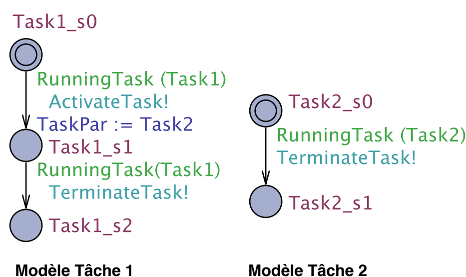


FIGURE 3.14 – Modélisation d'une application

offert par UPPAAL.

Un état s du modèle complet $\mathcal{A}$ est $s = (s_1^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a) \in S_1^{os} \times ... \times S_n^{os} \times S^T \times S_1^a ... \times S_m^a$.

L'état du pointeur de programme du système d'exploitation est abstrait par la projection $s^{os} = (s_1^{os}, ..., s_n^{os},)$ de l'état $(s_1^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a)$ sur $OS_1 \parallel ... \parallel OS_n$ et l'état du pointeur de programme d'une tâche i est abstrait par la projection s_i^a de l'état $(s_1^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a)$ sur l'automate A_i.

Définition 9. *(Configuration observable)*
Une configuration observable du modèle $\mathcal{A}$ est la paire $c = (s, v)$ où s est l'état du modèle complet et v est une valuation des variables $x \in X$ pour cet état.

Propriété 1. *Le modèle $OS_1 \parallel ... \parallel OS_n$ est un automate fini étendu borné.*

Démonstration. Les variables qui manipulent le système d'exploitation sont soient bornées (telle que la valeur de la priorité d'une tâche) soit de type énuméré (tel que l'état d'une tâche), elles sont équivalentes à des entiers dans un domaine fini puisqu'elles prennent leurs valeurs dans un ensemble fini de valeurs.

Le système d'exploitation et le modèle à base d'automates finis étendus ont le même ensemble de variables X, le modèle est par conséquent borné. □

Le système d'exploitation lui-même utilise six variables entières ou de type énuméré pour enregistrer les références d'une tâche en cours d'exécution. Les entiers sont bornés par le nombre de tâches dans l'application et les types énumérés sont restreints à quatre valeurs possibles. En

plus, un nombre fini de descripteurs d'objet est utilisé dont celui des tâches, des alarmes, des ressources etc.

Chaque champ dans ces descripteurs est soit un entier borné ou un type énuméré. Certains champs d'entiers sont utilisés pour enregistrer la liste des objets référencés. Ainsi, ils sont bornés par le nombre d'objets qu'ils référencient. Par exemple, la liste des ressources détenues par une tâche est bornée par le nombre de ressources. Les autres sont indirectement bornés par le nombre d'objets.

Par exemple, la priorité attribuée à une tâche est bornée par le nombre de tâches présentes dans l'application. Un descripteur de tâches utilise dix variables entières ou de types énumérés, un descripteur de ressource admet 5 champs et ainsi de suite.

Propriété 2. *L'accessibilité d'une configuration observable est décidable pour le modèle complet $\mathcal{A}$.*

Démonstration. Le modèle du système d'exploitation est un automate fini étendu et borné. Le modèle de la tâche est juste une séquence d'appels système et est un automate temporisé dont les états possèdent des invariants qui permettent de limiter le temps d'attente dans un état donné. Le modèle du *timer* est un automate fini étendu temporisé. L'accessibilité est décidable pour un automate fini étendu et pour un automate temporisé ayant des variables bornées. La complexité du problème d'accessibilité pour cette classe de modèle est PSPACE-complet. □

Proposition 2. *Le modèle complet (système d'exploitation + Application + Timer) contient tous les chemins qui pourraient être traversés durant l'exécution du programme du système d'exploitation pour cette application.*

Démonstration. Les variables et le code du système d'exploitation sont inclus dans le modèle formel. L'espace d'état du modèle complet abstrait l'espace d'état du système complet. Par l'atomicité des séquences d'instructions associées à une transition, toutes les configurations du système d'exploitation ne sont pas observables et donc ne sont pas compris dans l'espace d'état du modèle. Cependant, même si des valeurs intermédiaires des variables X dans une séquence d'instructions associée à une transition t ne sont pas dans l'espace d'état du modèle UPPAAL, les instructions associées à t sont exécutées de façon séquentielle (comme un programme réel du système d'exploitation) pour donner la configuration observable suivante. □

Notons que l'atomicité permet la concision du modèle et évite les explosions combinatoires du nombre d'état. Cependant, si l'on veut observer une configuration particulière dans une séquence d'instructions associée à une transition, il suffit juste d'ajouter un état supplémentaire dans l'automate à l'endroit ou l'observation est voulue.

3.8 Conclusion

Dans ce chapitre, nous avons présenté des règles de modélisation du système d'exploitation Trampoline à partir de modèles à états finis.

Basé sur ces règles, nous avons établi les modèles des versions monocoeur et multicoeur de Trampoline. Ces modèles embarquent en effet le code source de Trampoline et décrivent fidèlement son comportement.

L'objectif de notre thèse étant basé sur la spécialisation de systèmes d'exploitation en fonction des besoins de l'application, nous avons également décrit le principe de modélisation de l'application. Contrairement au système d'exploitation, le modèle de l'application ne décrit pas le comportement complet de l'application mais décrit plutôt l'ensemble des appels système que celle-ci effectue pour solliciter les services qui lui sont nécessaires.

À partir des modèles de l'application et du système d'exploitation, il est possible de réaliser la synthèse du système d'exploitation de sorte qu'elle puisse contenir les services requis par

l'application. Ainsi, dans le chapitre suivant, nous présenterons les différentes étapes permettant de réaliser cette synthèse.

Chapitre 4

Synthèse formelle de systèmes d'exploitation

Sommaire

4.1 Méthodologie **78**
4.1.1 Modélisation 79
4.1.2 Synthèse formelle 79
4.1.3 Génération de code 81
4.2 Outil développé **83**
4.2.1 Synthétiseur de modèle 83
4.2.2 Générateur de code 84
4.3 Étude de cas **85**
4.4 Conclusion **88**

Dans ce chapitre, nous présentons l'approche décrivant la génération de systèmes d'exploitation à partir de modèles à états finis. D'abord, nous présentons en détail chaque étape de l'approche ainsi que l'outil développé qui a servi à sa mise en œuvre. Enfin, nous montrons dans une étude de cas industriel l'applicabilité de l'approche proposée.

La synthèse logicielle consiste à générer l'implémentation d'un système à partir de sa spécification. Le principe est tel qu'une synthèse logicielle permet de générer une implémentation optimale dans un contexte donné. Cela peut être la synthèse de pilotes de périphériques [TBB+13], d'ordonnanceurs [BNO+04] ou encore d'applications temps réel [KKC+15].

Notre principal objectif concerne la synthèse formelle de systèmes d'exploitation spécifiques aux applications. L'approche proposée permet ainsi la génération d'un système d'exploitation optimisé du point de vue de sa taille. Dans ce système d'exploitation, seuls les services indispensables à l'application sont fournis. De plus, la quantité de code mort est réduite car seul le code du système d'exploitation qui est indispensable à l'exécution de l'application est pris en compte.

Cela a pour avantage de générer un système d'exploitation robuste car un code mort représente du code indésirable et pourrait conduire le système dans un état non souhaité dans le cas où celui-ci parviendrait à s'exécuter.

Un autre avantage de cette approche est celui de la génération d'un système d'exploitation qui est optimal du point de vue de l'empreinte mémoire qu'occupe celui-ci. Cela permet de satisfaire les contraintes de mémoire qui sont fréquentes dans les systèmes embarqués à ressources limitées.

La figure 4.1 situe notre approche dans la chaîne de compilation de Trampoline. À partir du modèle du système d'exploitation décrit sous UPPAAL et après analyse de celui-ci dans un composant logiciel que nous appellerons « synthétiseur de système d'exploitation », son code source optimisé est engendré. Cette approche vient remplacer l'ancienne technique de configuration de Trampoline qui était basée sur le pré-processeur C et la compilation conditionnelle. Notre approche réalise une configuration de type statique s'effectuant à la génération du code source du système d'exploitation (voir section 1.2.1).

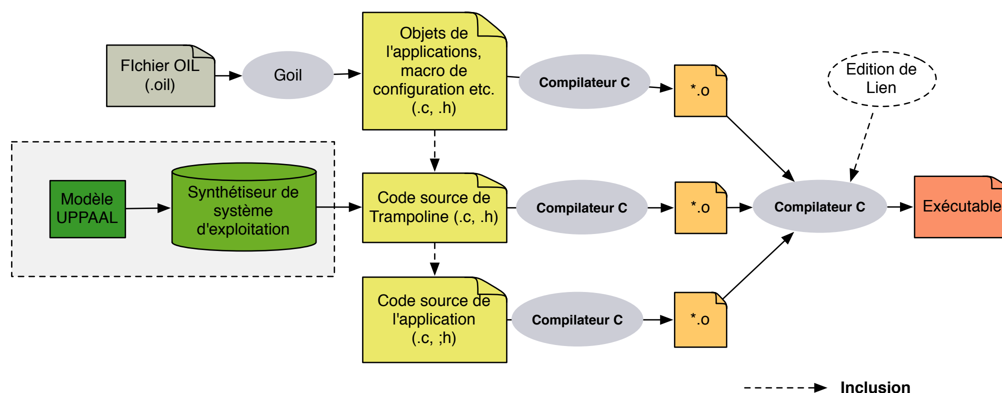


FIGURE 4.1 – Positionnement de notre approche dans la chaîne de compilation de Trampoline

4.1 Méthodologie

L'approche de synthèse de systèmes d'exploitation [TBR15b, TBFR16] s'effectue en trois grandes étapes que nous listons comme suit :

— La modélisation.
— La synthèse du modèle.
— La génération de code.

Par la suite, nous présentons en détail les différentes étapes de notre approche illustrée à la figue 4.2.

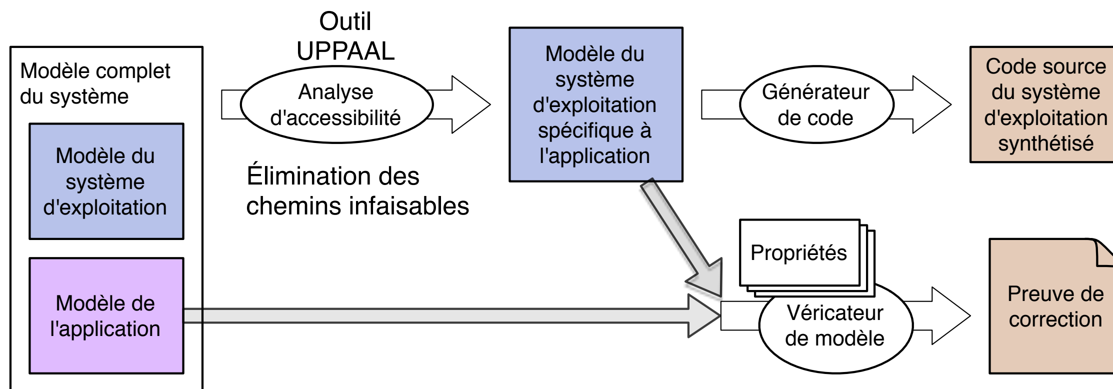


FIGURE 4.2 – Synthèse formelle du système d'exploitation

4.1.1 Modélisation

La modélisation constitue la première étape de notre approche. Elle est réalisée à partir du code source du système d'exploitation et de l'application.

Le modèle du système d'exploitation obtenu est constitué d'un réseau d'automates finis étendus (éventuellement temporisés) décrivant chacune de ses fonctions. Ces automates finis étendus embarquent le code source du système d'exploitation et grâce à leur structure, ils décrivent parfaitement le graphe de flot de contrôle du système d'exploitation.

Contrairement au modèle du système d'exploitation, le modèle de l'application ne décrit pas le comportement complet de l'application à développer. Notre objectif étant de réaliser la synthèse d'un système d'exploitation spécifique à l'application, le modèle de l'application décrit seulement les appels système qu'elle effectue. Ces appels système permettent l'interaction entre l'application et le système d'exploitation. Ainsi, à partir des appels système réalisés par une application, il est possible de connaitre les services nécessaires à son exécution.

Enfin, une fois la modélisation terminée, le modèle du système d'exploitation est composé avec celui de l'application pour former un modèle plus complet décrivant l'interaction entre l'application et le système d'exploitation. Ce modèle est ensuite analysé dans le but de réaliser la synthèse du système d'exploitation spécifique.

Notons que le modèle du système d'exploitation est construit une seule fois et réutilisé pour toute synthèse. Par contre, pour chaque synthèse, le modèle de la nouvelle application à développer doit être fourni dans le but de connaître ses besoins et de ne générer que le code nécessaire à son exécution.

4.1.2 Synthèse formelle

La synthèse formelle consiste à analyser d'abord le modèle du système complet (application + système d'exploitation) de manière à pouvoir ensuite le synthétiser en fonction de l'application. Le principe de synthèse est basé sur l'analyse d'accessibilité des états du modèle. Puisque le modèle traduit l'exécution de l'application sur le système d'exploitation, un état du modèle représente un état de l'exécution du système réel.

Par conséquent, un état accessible (resp. inaccessible) représente un état qui peut être traversé (resp. ne peut pas être traversé) lors de l'exécution du système.

La synthèse du modèle se définit alors par la suppression de l'ensemble des états inaccessibles

du modèle. Cela permet d'exclure du modèle l'ensemble des chemins infaisables et donc de ne conserver que les chemins qui seront parcourus lors de l'exécution du système.

Pour déterminer les états accessibles du modèle, nous effectuons une recherche d'accessibilité des états de chaque automate modélisant une fonction du système d'exploitation.

En se référant à la section 3.7, nous considérons $\{OS_1, ..., OS_n\}$ l'ensemble des n automates finis étendus $OS_i = (S_i^{os}, s_{i0}^{os}, X, \Sigma, \pi_{i_0}^{os}, \longrightarrow_i^{os})$ (pour i allant de 1 à n) modélisant les différents composants du système d'exploitation,

Timer un automate fini étendu $T = (S^T, s_0^T, X, \Sigma, \pi_0^T, \longrightarrow^T)$ modélisant le *timer*,

$TA_1, ..., TA_k$ k automates finis étendus $A_i = (S_i^a, s_{i0}^a, X, \Sigma, \pi_{i_0}^a, \longrightarrow_i^a)$ modélisant les tâches de l'application et $\mathcal{A}$ le modèle complet du produit tel que $\mathcal{A}$=$(OS_1 \parallel ... \parallel OS_n \parallel Timer \parallel A_1 \parallel ... \parallel A_n)_f$.

Un état s du modèle complet $\mathcal{A}$ est $s = (s_1^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a) \in S_1^{os} \times ... \times S_n^{os} \times S^T \times S_1^a ... \times S_m^a$.

Soit $Reach(\mathcal{A})$ l'ensemble des états accessibles de $\mathcal{A}$ défini par :

$Reach(\mathcal{A})$=$\{s = (s_1^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a) \mid s_0 \longrightarrow^* s\}$.

L'ensemble des n automates finis étendus modélisant les composants du système d'exploitation étant l'ensemble $\{OS_1, ..., OS_n\}$, nous définissons la projection des états accessibles de $\mathcal{A}$ sur un élément OS_i (i allant de 1 à n) par :

$Acc(OS_i)$=$\{s_i^{os} \mid \exists (s_1^{os}, ..., s_i^{os}, ..., s_n^{os}, s^T, s_1^a, ..., s_m^a) \in Reach(\mathcal{A})\}$.

Le modèle synthétisé est donc obtenu par la suppression des états inaccessibles de chaque automate du modèle du système d'exploitation.

Pour les automates finis, l'automate réduit à sa partie accessible est défini dans [CL09], et est obtenue en supprimant les états non accessibles et les transitions associées. Cette réduction est étendue sur les automates finis étendus de la manière suivante.

Définition 10. *Soit l'automate fini étendu $F = (S, s_0, X, \Sigma, \pi_0, \longrightarrow)$ et $Acc(F) \subseteq S$ l'ensemble des états accessibles de F. L'automate réduit à sa partie accessible est l'automate fini étendu $G = (Acc(F), s_0, X, \Sigma, \pi_0, \longrightarrow_G)$ avec $\longrightarrow_G$ la relation de transition $\longrightarrow$ restreinte à $Acc(F)$ de la manière suivante (voir définition 2 dans 3.1.2) :*

$\langle s, g, \sigma, \pi, s' \rangle \in \longrightarrow_G$ ssi $s \in Acc(F)$, $s' \in Acc(F)$ et $\langle s, g, \sigma, \pi, s' \rangle \in \longrightarrow$.

L'algorithme 1 a été implémenté et appliqué sur chaque automate pour réaliser la synthèse

du modèle.

Algorithme 1 : Synthèse du modèle

début
  pour *Automate fini étendu* $F \in \{OS_1, ..., OS_n\}$ **faire**
    EtatsAccessibles = $\emptyset$
    EtatsAutomate$F$= $\{S_0, S_1, ...S_n\}$
    EtatsAutomate$G$= $\emptyset$
    TransitionAutomate$F$ = $\{T_0, T_1, ...T_n\}$
    TransitionAutomate$G$ = $\emptyset$
    pour $etat \in EtatsAutomateF$ **faire**
      si *etat est accessible* **alors**
        Ajouter etat à EtatsAccessibles ;
    EtatsAutomate$G$:= EtatsAccessibles ;
    pour $transition \in TransitionAutomateF$ **faire**
      si $source(transition) \in EtatsAutomateG$ ***et*** $cible(transition) \in EtatsAutomateG$ **alors**
        Ajouter transition à TransitionAutomate$G$;

Suite à l'analyse d'accessibilité, le modèle du système obtenu ne décrit que le code nécessaire à l'exécution de l'application. Ce modèle sert d'une part à la génération du code source du système d'exploitation et d'autre part à la vérification afin de s'assurer que le système d'exploitation est correct du point de vue fonctionnel.

Un exemple de synthèse de modèle est montré sur la figure 4.3. Dans cet exemple, nous considérons la fonction `tpl_terminate` qui a été décrite à la section 3.10.

Dans cet exemple, nous supposons qu'il n'existe aucune tâche étendue qui effectue l'appel de cette fonction. Par conséquent la condition `tpl_kern.running_id < EXTENDED_TASK_COUNT` est toujours fausse quelle que soit l'exécution du système d'exploitation. De ce point de vue, l'état `tpl_t5` n'est jamais traversé durant l'exécution de cette fonction.

Après le test d'accessibilité des états du modèle de la fonction `tpl_terminate`, l'état `tpl_t5` est déclaré comme inaccessible et est alors supprimé du modèle initial. Le modèle final ne contient alors pas l'état `tpl_t5`.

À partir de cet exemple, nous pouvons remarquer que l'analyse d'accessibilité permet d'identifier le code qui a été réellement exécuté (le code utile).

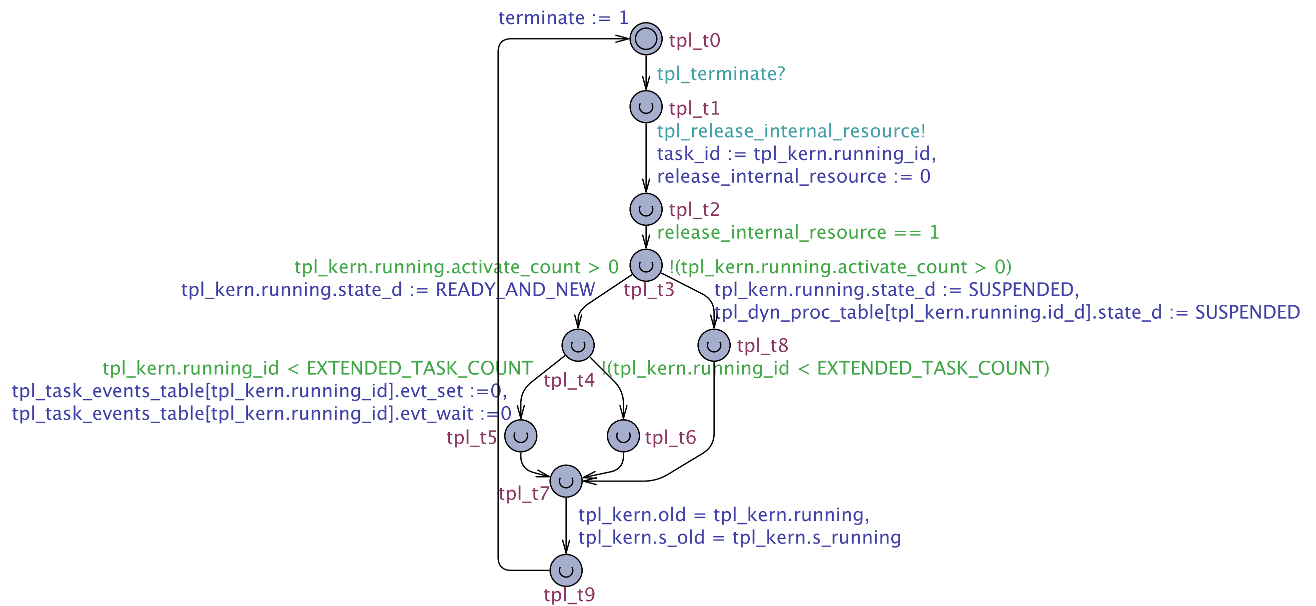
4.1.3 Génération de code

La génération de code constitue la dernière étape de l'approche. Le code du système d'exploitation est généré suite à son modèle synthétisé. Selon le principe de la modélisation, chaque automate décrit une fonction du système d'exploitation et l'ensemble des transitions d'un automate embarque le code source de cette fonction.

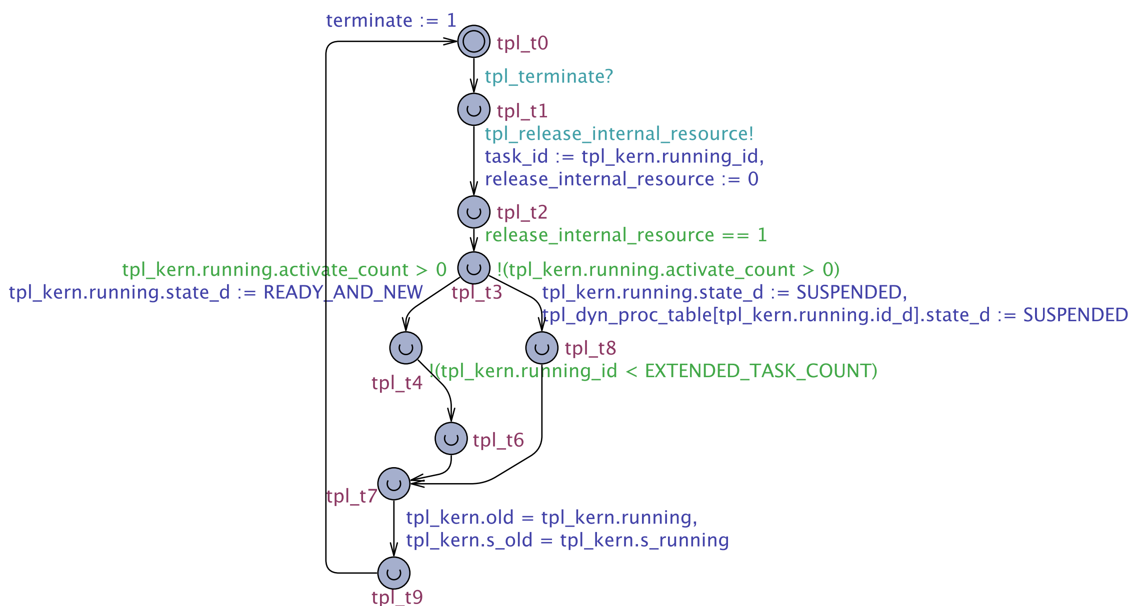
De ce fait, le générateur de code parcourt chaque automate du modèle puis lit et traduit les informations contenues dans chacune des transitions en code C.

4.1.3.1 Règles de traduction

Pour la génération de code à partir d'un automate modélisant une fonction du système d'exploitation, nous avons défini les règles suivantes :



(a) Modèle initial



(b) Modèle réduit

FIGURE 4.3 – Synthèse de modèle

— Une action de synchronisation suivie d'un "?" est remplacée par une déclaration de fonction en C.
— Une action de synchronisation suivie d'un "!" est remplacée par un appel de fonction où le nom de l'action correspond à la fonction appelée.
— Une action de mise à jour sur les variables du système d'exploitation est remplacée par du code C.
— Si un état admet une transition entrante et deux transitions sortantes et qu'une garde (sur

les variables du système d'exploitation) est associée à l'une de ses transitions sortantes alors cet état est remplacé par une instruction `if`.

— Si un état admet deux transitions entrantes et deux transitions sortantes et qu'une garde (sur les variables du système d'exploitation) est associée à l'une de ses transitions sortantes alors cet état est remplacé soit par une instruction `while` ou une instruction `for`. La nature de l'instruction est déterminée suite au nom associé à cet état.
— Le code associé à une fonction écrite sous la syntaxe de UPPAAL décrivant une fonction de Trampoline est automatiquement généré.
— Si l'automate ne fait pas partie du modèle synthétisé alors le code source correspondant à la fonction qu'il modélise n'est pas généré.
— Les variables du système d'exploitation liées à l'automate sont automatiquement générées.

La figure 4.4 montre un exemple du code généré suite au modèle réduit de la fonction `tpl_terminate` (*cf.* figure 4.3)

```
FUNC(void, OS_CODE) tpl_terminate(void)
{
  CALL_POST_TASK_HOOK()
  tpl_release_internal_resource((tpl_proc_id)tpl_kern.running_id);
  if (tpl_kern.running->activate_count > 0)
  {
    tpl_kern.running->state = READY_AND_NEW;
  }
  else
  {
    tpl_kern.running->state = SUSPENDED;
  }
  tpl_kern.old = tpl_kern.running;
  tpl_kern.s_old = tpl_kern.s_running;
}
```

FIGURE 4.4 – Code généré suite au modèle réduit de la fonction `tpl_terminate`

Contrairement au code initial de la fonction `tpl_terminate` (*cf.* figure 3.6), le code généré suite à son modèle synthétisé est beaucoup plus réduit. En se basant sur l'hypothèse faite à la section 4.1.2, cette réduction se caractérise par le fait que la condition `tpl_kern.running_id < EXTENDED_TASK_COUNT` est toujours fausse. Cela implique la non exécution du code à l'intérieur de cette condition. Ce code devient donc inutile et représente du code mort.

Ainsi, dans la fonction générée il n'y apparaît que le code nécessaire à l'exécution de l'application. Un autre point important concerne l'absence de directives de préprocesseur (`#ifdef`, `#endif`, etc...) dans le code de la fonction générée.

4.2 Outil développé

Un outil de synthèse de système d'exploitation implémenté en Python a été mis au point pour la mise en œuvre de l'approche proposée. Il est constitué de deux parties dont un synthétiseur de modèle et un générateur de code. La figure 4.5 décrit l'architecture générale de l'outil développé.

4.2.1 Synthétiseur de modèle

Le synthétiseur de modèle est formé de plusieurs modules dont :

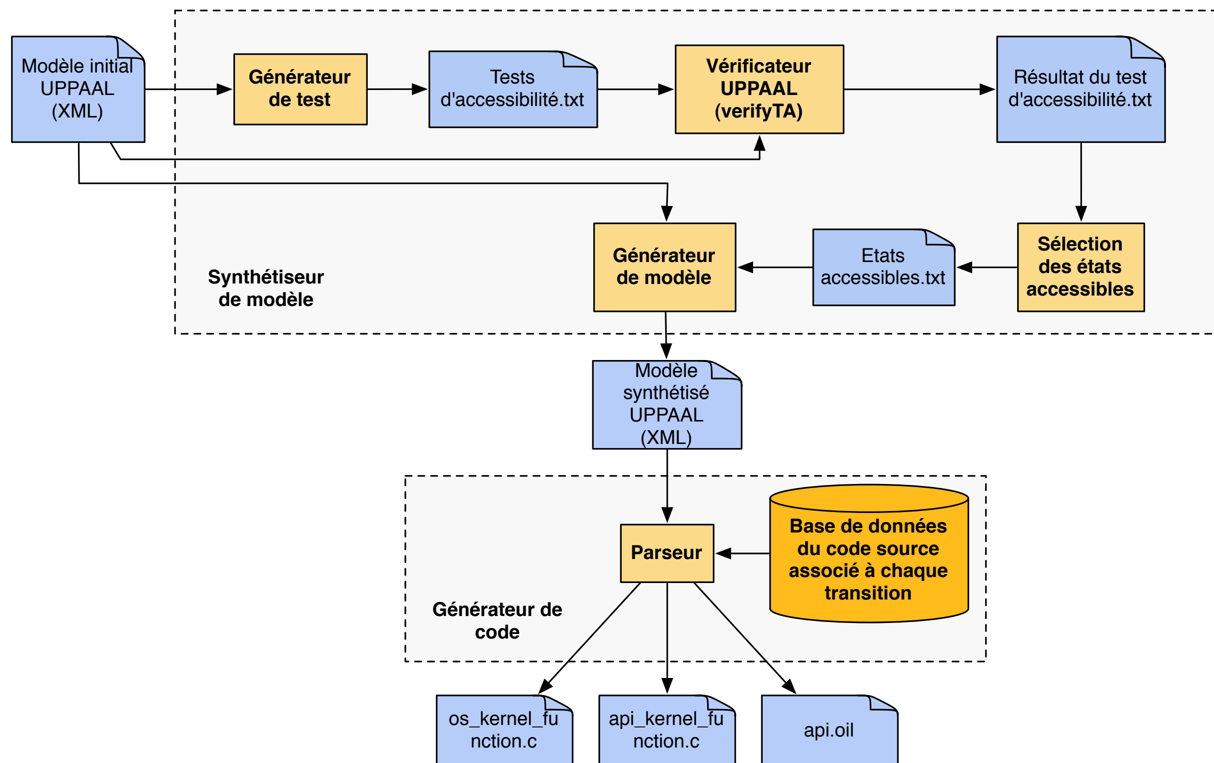


FIGURE 4.5 – Architecture du synthétiseur du système d'exploitation

Le générateur de tests : Le générateur de tests contient un analyseur de fichiers XML qui lit le modèle et génère les tests d'accessibilité des états du modèle. Ces tests sont écrits sous forme de propriétés CTL sous une syntaxe propre à l'outil UPPAAL.

Le vérificateur de modèles : Le vérificateur de modèles est basé sur le *model-checker* de l'outil UPPAAL appelé *VerifyTA*. Il prend en son entrée les différents tests d'accessibilité ainsi que le modèle initial puis génère un verdict sur chaque test d'accessibilité effectué sur les états du modèle.

Générateur des états accessibles : Le générateur d'états accessibles utilise les informations fournies par le vérificateur de modèles puis génère dans un fichier tous les états accessibles du modèle initial.

Générateur de modèles : Le générateur de modèles prend en entrée le modèle initial et l'ensemble des états accessibles issus du générateur des états accessibles. Il construit alors un nouveau modèle qui ne contient que l'ensemble des états accessibles du modèle initial.

4.2.2 Générateur de code

Le générateur de code est constitué d'un analyseur de fichiers XML qui lit le modèle synthétisé et d'une base de données qui contient toutes les informations concernant le code source en C associé à chaque transition du modèle synthétisé. Il fait donc une correspondance, entre

les informations contenues dans chaque transition du modèle et leur équivalence en code C puis génère le code complet du système d'exploitation en 3 fichiers dont :
- `os_kernel_function.c` qui contient toutes les fonctions du noyau du système d'exploitation Trampoline.
- `api_kernel_function.c` qui contient tous les services de Trampoline nécessaires à l'application.
- `api.oil` qui décrit l'ensemble des fonctions de l'API qui ont été invoquées par l'application durant son exécution.

4.3 Étude de cas

Comme application de l'approche de synthèse de systèmes d'exploitation, nous nous intéressons à l'étude d'un cas industriel. Le système mis en exergue est une application d'assistance à la conduite appelée *ADAS* (*Advanced Driver Assistance System*) s'exécutant sur une unité de contrôle électronique (ECU).

Cette application a été développée lors du projet *RESPECTED* et est composé de trois sous systèmes dont :
- Un régulateur de vitesse adaptatif ou *ACC* (*Adaptative Cruise Control*).
- Un système de freinage d'urgence autonome ou *EAB* (*Emergency Autonomous Braking*).
- Un logiciel de base ou *BSW* (*Basic Software*).

Le régulateur de vitesse adaptatif permet de réguler la vitesse du véhicule en fonction d'une consigne de vitesse, d'éventuels obstacles et de véhicules à faible vitesse se trouvant en avant de celui-ci.

Le système de freinage d'urgence autonome permet l'arrêt du véhicule lorsqu'un obstacle est détecté à une portée très courte sur la route.

Le logiciel de base gère le démarrage et l'arrêt de l'unité de contrôle électronique sur laquelle l'application *ADAS* s'exécute ainsi que la communication.

Plusieurs types d'information dont la vitesse réelle du véhicule, la vitesse et la portée des obstacles sont reçus par un réseau de *CAN* à bord du véhicule. Ces informations proviennent de l'unité de commande du moteur et de l'unité de contrôle du véhicule ou *BCM* (*Body Control Module*).

Le régulateur de vitesse adaptatif est composé de 5 tâches :
- `TASK_Chart` calcule une variable d'état en fonction de l'activation du freinage d'urgence.
- `TASK_Aux2` ajuste l'accélération et les commandes de freinage en fonction de la variable d'état.
- `TASK_ACC_IDM` contient l'algorithme principal du système *ACC* qui est basé sur un modèle de conduite intelligente (*IDM*) [THH00].
- `TASK_ACC_Aux3` effectue la mise en forme des variables d'entrée pour l*'IDM*.
- `TASK_ACC_Aux4` effectue la mise en forme de l'accélération et des commandes de freinage calculées par l' *'IDM*.

Le système de freinage d'urgence autonome est composé d'une seule tâche appelée `TASK_EAB` qui implémente la fonction de freinage d'urgence. Les informations à sa sortie sont délivrées au régulateur de vitesse adaptatif afin de stopper la régulation de vitesse quand le freinage d'urgence est activé. Elles sont aussi envoyées au réseau de *CAN* pour l'unité de contrôle électronique de freinage.

Le logiciel de base est composé de 4 tâches :

— `TASK_Data` effectue la mise en forme des informations entrantes provenant des autres unités de contrôle électronique.
— `TASK_Aux1` réalise la mise en forme des informations sortantes destinées aux autres unités de contrôle électronique.
— `TASK_ComRx` reçoit les informations provenant du réseau de *CAN*.
— `TASK_ComTx` envoie des informations au réseau de *CAN*.

Dans le système de développement *AUTOSAR*, le squelette des tâches est généré avec les appels de services nécessaires en fonction de la spécification tandis que la partie fonctionnelle des tâches est fournie séparément comme un ensemble de fonctions appelées *runnables*.

Ce squelette est appelé environnement d'exécution ou *RTE* (*Run Time Evnivronment*). Les *runnables* sont écrits à la main ou générés à partir de modèles issus de Matlab par exemple.

L'environnement d'exécution utilisé dans notre cas est issu du générateur d'environnements d'exécution fourni par *Dassault Systems*.

Nous nous intéressons aux appels de services afin de pourvoir générer un système d'exploitation qui ne contient que les services nécessaires à l'application. Par conséquent, nous pouvons abstraire les *runnables*.

Toutes les tâches de l'application sont périodiques et étendues et elles présentent toutes une structure similaire. En effet, chaque tâche implémente une boucle infinie où la tâche est en attente de l'occurrence d'un événement, puis une fois l'événement produit, elle retourne au début de la boucle en attente d'un nouvel événement.

Les événements sont produits par des alarmes afin de rendre leur occurrence périodique. Puis à chaque tâche est associé un événement afin que celle-ci puisse s'activer lors de l'occurrence de son événement correspondant. La périodicité d'occurrence des événements du système est de 10 ou 100 ms. La communication entre tâches est faite par le biais de variables globales et les sections critiques sont protégées par la désactivation d'interruptions de sorte que le déclenchement d'une interruption ne préempte en aucun cas un programme critique.

L'ensemble des tâches de l'application *ADAS* invoque seulement 9 fonctions de l'API du système d'exploitation Trampoline.

La priorité de chaque tâche est attribuée en fonction de l'ordonnancement *Rate Monotonic* et en fonction de la dépendance de données entre chaque tâche. Les dates de démarrage, les priorités et les périodes des tâches de l'application sont données dans le tableau 4.1.

TABLE 4.1 – Priorité, démarrage et période des tâches de l'application ADAS.

Tâches	Priorité	Démarrage (ms)	Période (ms)
TASK_Chart	7	100	100
TASK_Aux2	8	100	100
TASK_ACC_IDM	12	11	10
TASK_ACC_Aux3	11	10	10
TASK_ACC_Aux4	13	13	10
TASK_EAB	14	10	10
TASK_Data	10	100	100
TASK_Aux1	9	100	100
TASK_ComRx	17	15	10
TASK_ComTx	16	15	10

Comme exemple, la figure 4.6 montre le modèle de la tâche `TASK_ACC_IDM`. Ce modèle est un automate temporisé où la variable d'horloge `response_time` représente le pire temps de réponse

de la tâche (10 ms).

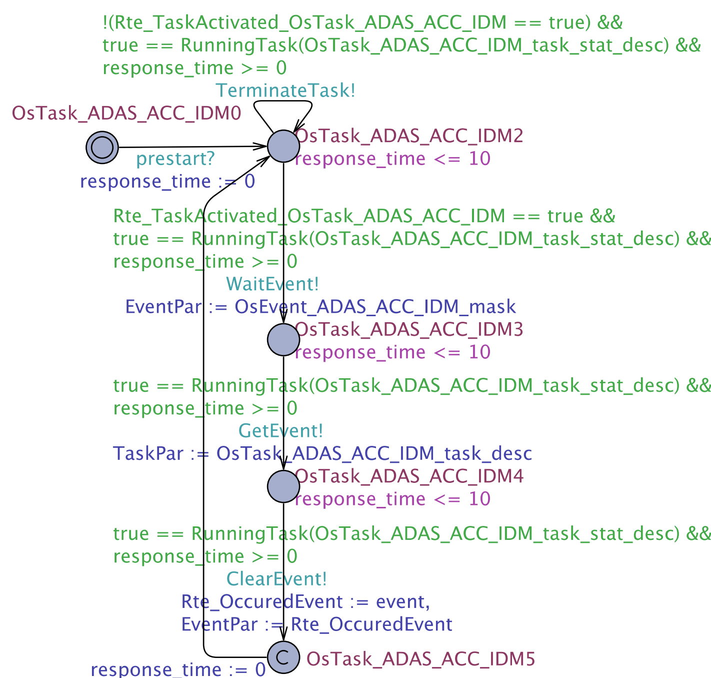


FIGURE 4.6 – Modèle de la tâche `TASK_ACC_IDM`

Cette tâche fait appel à 4 services du système d'exploitation Trampoline. Son activation et sa désactivation sont effectuées par d'autres fonctions de l'application. En effet, la variable `Rte_TaskActivated_OsTask_ADAS_ACC_IDM` permet de savoir si la tâche est activée (`true`) ou non (`false`). La fonction `RunningTask()` retourne vrai (`true`) si la tâche est bien activée et faux (`false`) dans le cas contraire.

Quand la tâche est activée (`Rte_TaskActivated_OsTask_ADAS_ACC_IDM == true`), elle est en attente de l'événement `OsEvent_ADAS_ACC_IDM` qui est produit par une alarme. Une fois l'événement produit, elle fait appel aux fonctions de l'API `GetEvent` et `ClearEvent` puis exécute le code correspondant aux *runnables* qui sont abstraits dans le modèle.

En cas de désactivation de la tâche (`!(Rte_TaskActivated_OsTask_ADAS_ACC_IDM == true)`), elle appelle la fonction de l'API `TerminateTask` afin de terminer son exécution.

L'application *ADAS* a été modélisée par un réseau d'automates finis étendus temporisés. Du modèle obtenu, nous avons appliqué la synthèse du système d'exploitation Trampoline. Le code généré a été compilé pour un micro-contrôleur NXP LCP2294 avec un processeur ARM7TDMI 32 bit.

L'analyse d'accessibilité a été réalisée avec l'outil UPPAAL (version 4.1.20) sur un ordinateur ayant un processeur Intel Core i5 tournant à 2.56 GHz et une mémoire RAM de 4Gb.

Nous avons testé l'accessibilité de 836 états du système d'exploitation. Parmi ces états, 541 états ont été marqués accessible et 295 états marqués inaccessible.

Durant l'analyse d'accessibilité, le temps nécessaire pour identifier un état accessible se situe entre 0,008 et 5,268 secondes consommant entre 18,9 et 312 Mb de mémoire tandis que le temps nécessaire pour identifier un état inaccessible se situe entre 86 et 91,2 secondes consommant entre 1,1 et 1,9 Gb de mémoire. Cette différence est raisonnable car pour identifier un état inaccessible, il est indispensable d'explorer la totalité de l'espace d'état du modèle et cela prend

un temps considérable.

Le tableau donne les informations concernant la synthèse de Trampoline pour l'application ADAS.

TABLE 4.2 – Résultat de la synthèse du système d'exploitation

	RTOS synthétisé	**RTOS initial**
Taille du code	4.5 kB	7.7 kB
Nombre de lignes de code	1287	1866

Chaque valeur dans ce tableau représente la taille du code compilé du système d'exploitation synthétisé à partir du modèle (colonne 2) et la taille du code du système d'exploitation initial (colonne 3). Le jeu d'instructions de la cible est celui d'une architecture ARM 32 bits.

Le code source initial de Trampoline comporte 1866 lignes de code et correspond principalement à la partie configurable de Trampoline, c'est à dire l'ensemble des services et des fonctions du noyau. Après la synthèse de Trampoline, nous remarquons une réduction très importante du nombre de lignes de code (1287) et donc de la mémoire occupée (4.5 kB) par Trampoline.

4.4 Conclusion

Dans ce chapitre, nous avons présenté l'approche de synthèse de systèmes d'exploitation. Cette approche étant basée sur le modèle, elle permet la génération d'un système d'exploitation spécifique aux besoins de l'application à partir de la composition des modèles de l'application et du système d'exploitation.

Elle présente en effet un avantage majeur qui est celui de la génération d'un système d'exploitation ne contenant aucun code mort et comportant le strict minimum en terme de code nécessaire à l'exécution de l'application.

En éliminant le code mort, cette approche permet également d'éliminer du système d'exploitation des comportements indésirables qui pourraient survenir à cause d'une erreur de configuration ou d'un *bug*.

Nous notons alors une forte optimisation de la taille du code source du système d'exploitation qui vient résoudre un problème que l'on rencontre dans les systèmes embarqués à ressources limitées : celui de la mémoire disponible puisque de tels systèmes sont très contraignants en terme de mémoire disponible.

Nous avons également présenté l'outil développé pour la mise en œuvre de l'approche. Enfin, dans une étude de cas, nous avons montré une application de l'approche à un cas industriel. En effet, des résultats très intéressants ont été obtenus. Mais une grande question se pose : Le système d'exploitation généré est-il fiable ?

Pour répondre à cette question dont nous ignorons actuellement la réponse, nous présentons dans le chapitre suivant les différents techniques qui ont été mises en place pour s'assurer de la correction sur le plan fonctionnel du système d'exploitation généré.

Chapitre 5

Vérifications formelles

Sommaire

5.1 Les techniques de vérification formelle **90**
5.1.1 Model Checking 90
5.1.2 Analyse statique 91
5.1.3 Test 91
5.2 Vérification formelle de la conformité OSEK/VDX **92**
5.2.1 Cas de test 93
5.2.2 Séquence de test 93
5.2.3 Suite de test 94
5.2.4 Exemple 94
5.3 Vérification formelle de la conformité OSEK/VDX à partir d'observateurs génériques **97**
5.3.1 Présentation de l'approche 97
5.3.2 Observateurs 98
5.3.3 Étude de cas 102
5.4 Conclusion **105**

Dans ce chapitre, nous présentons tout d'abord quelques techniques de vérification de systèmes informatiques telles que *le model checking*, *l'analyse statique* et *le test formel*. Ensuite, nous présentons deux approches permettant de déterminer la conformité de systèmes d'exploitation OSEK/VDX à partir de leurs modèles formels.

Dans la première approche, la suite de test issue de la spécification OSEK/VDX est traduite en des automates finis étendus qui sont composés au modèle du système d'exploitation afin de tester son comportement. La conformité du système d'exploitation est ainsi déterminée à partir de son modèle. Pour cette approche, il faut à chaque fois concevoir une nouvelle suite de test adaptée au système d'exploitation configuré car les services fournis par le système d'exploitation peuvent varier d'une configuration à une autre et une suite de test est liée aux services du système d'exploitation. Cela exige en effet plus d'efforts et de temps pour la détermination de la conformité OSEK/VDX en utilisant cette approche.

Pour cela, nous présentons une seconde approche qui permet une traduction des cas de tests en observateurs en utilisant des automates finis étendus. Les modèles d'observateurs obtenus sont composés au modèle du système complet (système d'exploitation + application) afin d'observer le comportement du système d'exploitation et de déterminer sa conformité en fonction de la spécification OSEK/VDX.

Suite à une configuration, un système d'exploitation fournit un certain nombre de services nécessaires à l'application. Par conséquent, les cas de tests issus de la spécification OSEK/VDX ne sont pas tous utiles à vérifier. Seuls ceux qui sont liés aux services fournis par le système

d'exploitation configuré sont nécessaires à vérifier. Cette approche a ainsi l'avantage de permettre une sélection automatique des cas de tests applicables au système d'exploitation configuré.

De plus, les observateurs étant génériques, ils sont utilisés pour toutes configurations. Cette approche facilite alors la vérification de la conformité d'un système d'exploitation à partir de son modèle en fonction de la spécification OSEK/VDX.

5.1 Les techniques de vérification formelle

5.1.1 Model Checking

Introduit par les travaux de Clarke et Emerson [CE82] puis de Queille et Sifakis [QS82], la vérification de modèles ou *model checking* dans la littérature anglaise est une technique permettant de vérifier automatiquement si le modèle d'un système satisfait bien un ensemble de propriétés.

Les modèles sont la plupart du temps exprimés par des automates à états finis qui consistent en un nombre fini d'états et de transitions où les états contiennent les informations relatives aux valeurs actuelles des variables du système (eg. Compteur de programmes) et les transitions décrivent la manière dont le système évolue d'un état à un autre. La figure 5.1 présente le principe du *model checking*.

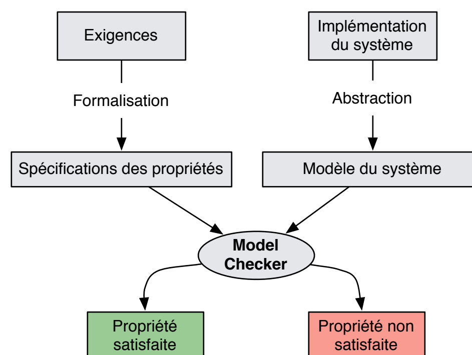


FIGURE 5.1 – Processus du *model checking*

Pour réaliser un *model checking*, dans un premier temps, le système réel est modélisé. Ensuite, l'ensemble des exigences ou des propriétés souhaitées sont formalisées dans un langage de spécification de propriétés comme le *LTL (Linear Temporal Logic)* [Pnu77] ou encore le *CTL (Computation Tree Logic)* [CES86]. Les propriétés spécifiées formellement sont par la suite vérifiées sur le modèle dans le but de savoir si elles sont bien satisfaites ou non.

Le modèle du système décrit comment le système réel se comporte tandis que les propriétés traduisent la manière dont le système doit se comporter.

Le *model checking* permet la vérification de plusieurs types de propriétés telles que :

- Les propriétés **fonctionnelles** : le système fait-il correctement ce qu'il est supposé faire ?
- Les propriétés **d'accessibilité** : est-il possible que le système atteigne un certain état ?
- Les propriétés de **sûreté** : quelque chose de mauvais ne se produira jamais.
- Les propriétés de **vivacité** : quelque chose de bon se produira immanquablement.
- Les propriétés **d'équité** : sous certaines conditions, un événement se produira infiniment souvent.

— Les propriétés **d'absence de blocage** : le système ne se retrouve jamais dans un état dans lequel il n'a plus la possibilité d'avancer.
— Les propriétés **temporelles** : Le système agira-t-il à temps ?

Plusieurs outils de *model checking* ont été mis en œuvre parmi lesquels nous pouvons citer *UPPAAL* [BLL+95] pour la vérification de systèmes décrits par des automates temporisés, *ROMEO* [GLM+05] pour la vérification de systèmes modélisés à partir de réseaux de Petri temporels et *Spin* [Hol04] utilisé pour la vérification de systèmes modélisés à partir d'automates finis.

En ce qui concerne la vérification de systèmes d'exploitation, [Cho11] applique *le model checking* sur le système d'exploitation Trampoline. Le code du noyau de Trampoline est converti automatiquement en *PROMELA* qui est le langage de modélisation supporté par l'outil *Spin*. En revanche, les auteurs ne modélisent pas entièrement le système d'exploitation. Ils se focalisent sur un certain nombre de fonctions leur permettant de vérifier quelques propriétés de sûreté sur Trampoline. [BPSV94] utilise les automates temporisés pour la modélisation et la vérification du système d'exploitation PATHOS.

Concernant les applications temps réel, [WH08] propose une approche de vérification d'applications OSEK/VDX en utilisant le *model checking*. Les auteurs décrivent le modèle de l'application OSEK/VDX puis une partie du système d'exploitation OSEK/VDX en utilisant des automates temporisés. Ils effectuent ensuite la vérification de propriétés temporelles spécifiées en langage *TCTL*(*Temporal Computation Tree Logic*) grâce à l'outil *UPPAAL*. En se basant sur un ordonnancement non-préemptif, ils sont en mesure de vérifier le pire temps d'exécution des tâches de l'application.

5.1.2 Analyse statique

L'analyse statique est une technique permettant de déterminer automatiquement des informations sur le comportement d'un programme sans réellement l'exécuter. Ce type de technique est utilisé dans l'optimisation de compilateurs mais aussi dans la vérification de programmes. Suite à l'analyse statique, il est possible de déterminer au sein d'un programme, la présence d'erreurs comme une division par zéro, le débordement d'un tableau etc.

L'analyse statique permet aussi de garantir la terminaison d'un programme. Dans la littérature, il existe plusieurs travaux de recherches basés sur l'analyse statique pour la vérification de programmes ainsi que plusieurs outils d'analyse statique. Dans [AHM+08] les auteurs ont développé un outil appelé *FindBugs* qui permet de détecter des erreurs dans des programmes écrits en Java. L'outil LINT [Joh77] développé par *Bell Labs* est utilisé pour la vérification de programmes écrits en C.

5.1.3 Test

Le test est une opération permettant de vérifier la correction de l'implémentation d'un système. Les tests sont appliqués sur l'implémentation dans un environnement contrôlé et les sorties ou les états du système sont observés. Un verdict de la correction fonctionnelle du système est ainsi établi. Les critères de correction sont issus d'une spécification qui contient toutes les exigences concernant le fonctionnement du système et qui constitue ainsi la base de toute activité de test. Les tests apparaissent ainsi comme un moyen d'améliorer la qualité des systèmes informatiques et doivent donc faire partie de leur cycle de développement.

Par contre, plusieurs problèmes peuvent intervenir dans le processus de test quand la spécification est établie de manière informelle car ce type de spécification est souvent incomplet et ambiguë. Par conséquent, baser l'activité de test sur une spécification formelle constitue un

avantage puisqu'elle est précise, complète et décrit correctement le système à tester dans un langage de spécification bénéficiant d'une syntaxe concise.

Le test issu d'une spécification formelle est appelé *test formel*. À partir de la spécification, des algorithmes sont utilisés pour la génération de cas de tests de manière à tester le système sous test. Si le système sous test constitue une boite noire alors le test est appelé *test de conformité* [Tre92]. La figure 5.2 montre le processus de test de conformité.

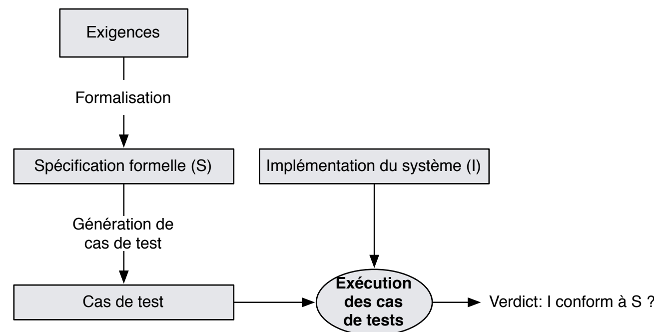


FIGURE 5.2 – Test de conformité

Il existe ainsi plusieurs outils de génération de test dont TGV [JJ05], TorX [BB05] et STG [CJRZ02] où les spécifications sont décrites sous la forme de systèmes de transitions et qui permettent d'établir le test de conformité de systèmes réactifs et temps réel.

Concernant le test de conformité de systèmes d'exploitation, [CA11] présente une méthode permettant de générer des cas de tests à partir de la spécification OSEK/VDX. En effet, la spécification OSEK/VDX traduit un certain nombre d'exigences du point de vue du fonctionnement du système d'exploitation. Ces exigences sont ensuite formalisées en *Notation Z* [SA92] pour enfin générer un ensemble de cas de tests à travers un outil appelé *TGT* pour établir le test de conformité OSEK/VDX. Cette approche permet de générer de façon exhaustive des tests mais les auteurs se sont intéressés essentiellement aux fonctionnalités du système d'exploitation liées à la gestion des tâches. Le test de conformité de toutes les fonctionnalités du système d'exploitation n'est pas réalisé.

Toujours dans le même contexte, [FKDO12] décrit une approche pour la génération de cas de tests à partir du modèle d'un système d'exploitation AUTOSAR dans le but de déterminer sa conformité au standard AUTOSAR. Les cas de tests sont par la suite traduits en programme C puis exécutés sur le système d'exploitation réel afin d'observer son comportement.

5.2 Vérification formelle de la conformité OSEK/VDX

Un test de conformité OSEK/VDX consiste à appliquer une suite de test sur le système d'exploitation afin d'observer son comportement, chaque test se concluant par un échec ou un succès. Historiquement, cette suite de test a été conçue dans le cadre du projet européen *MODISTARC* [Joh98]. La réalisation du test de conformité OSEK/VDX peut s'appuyer sur un premier document [OSE99] qui décrit 250 cas de tests environ et un second document [Gro99] décrivant la procédure de test. Dans la procédure de test, il existe 37 séquences de tests dont certaines séquences enchaînent jusqu'à une vingtaine de cas de tests.

5.2.1 Cas de test

Dans le cas du test de conformité OSEK/VDX, un cas de test est un appel de service du système d'exploitation. Le verdict du cas de test est déterminé en fonction de la réaction du système d'exploitation à l'appel de service. Lorsque le service est effectué selon les exigences de la spécification OSEK/VDX, le cas de test est une réussite. Dans le cas contraire celui-ci est un échec. Le tableau 5.1 montre un exemple de cas de tests appliqués au cours du test de conformité OSEK/VDX.

Table 5.1 – Cas de test

Cas	Action	résultat attendu
12	Appeler `ActivateTask()` à partir d'une tâche non-préemptable sur une tâche basique *prête* qui n'a pas encore atteint son compteur d'activation maximal.	La tâche en cours d'exécution n'est pas préemptée. La tâche activée est mise en attente dans la liste des tâches prêtes et le service retourne `E_OK`.
17	Appeler `ActivateTask()` à partir d'une tâche non-préemptable sur une tâche basique en cours d'exécution qui n'a pas encore atteint son compteur d'activation maximal.	La tâche en cours d'exécution n'est pas préemptée. La tâche activée est mise en attente dans la liste des tâches prêtes et le service retourne `E_OK`.
30	Appeler `ChainTask()` à partir d'une tâche non-préemptable sur une tâche basique *prête* qui n'a pas encore atteint son compteur d'activation maximal.	La tâche en cours d'exécution est terminée, la tâche activée est mise en attente dans la liste des tâches prêtes et la tâche prête ayant la plus haute priorité est mise en exécution.

5.2.2 Séquence de test

Une séquence de test est un ensemble de cas de tests. Pour le test de conformité OSEK/VDX, une séquence de test est caractérisée par une application ne contenant que des appels de services. La séquence de test permet ainsi d'effectuer successivement un ensemble d'appel de services. Une séquence de test est une réussite lorsque tous les cas de tests qui la composent s'effectuent avec succès. Le tableau 5.2 montre un exemple de séquence de test regroupant les cas de tests 12, 17 et 30 du tableau 5.1.

Table 5.2 – Séquence de test

Tâche en exécution	Services appelés	résultat retournés	cas de test
t1	`ActivateTask(t2)`	`E_OK`	
t1	`ActivateTask(t2)`	`E_OK`	12
t1	`Schedule()`	`E_OK`	
t2	`TerminateTask()`		
t2	`TerminateTask()`		
t1	`ActivateTask(t1)`	`E_OK`	17
t1	`ActivateTask(t3)`	`E_OK`	
t1	`ChainTask(t3)`		30
t3	`TerminateTask()`		
t3	`TerminateTask()`		
t1	`ActivateTask(t1)`	`E_OK`	
t1	`TerminateTask()`		
t1	`TerminateTask()`		

Description de la séquence de test Tout d'abord, cette séquence de test est une application constituée de 3 tâches T_1, T_2 et T_3 tel que `Priorité(`T_1`)`<`Priorité(`T_2`)`<`Priorité(`T_3`)`. Le compteur d'activation maximum [1] de chaque tâche est égal à 2. Pour cette séquence, nous considérons un ordonnancement non-préemptif.

Au démarrage du système d'exploitation, la tâche T_1 est activée et mise en exécution. Elle active une première fois la tâche T_2 puis une seconde fois. L'ordonnancement étant non-préemptif, et bien que T_2 soit de priorité supérieur, elle ne préempte pas T_1 et le résultat retourné suite à l'appel des deux services est `E_OK` signifiant que ceux-ci ont été exécutés correctement.

Ensuite, elle fait appel au service `Schedule()` qui provoque un ré-ordonnancement. La tâche T_2 qui avait été activée deux fois (deux jobs) auparavant préempte donc la tâche T_1, s'exécute deux fois puis fait appel 2 fois au service `TerminateTask()` dans le but de terminer son exécution. Une fois la tâche T_2 terminée, la tâche T_1 est remise en exécution puis cette fois s'active elle même et ensuite active la tâche T_3 qui ne la préempte pas. La tâche T_1 fait ensuite appel au service `ChainTask()` qui permet à la fois de terminer son exécution et d'activer une fois de plus la tâche T_3. Après cela, la tâche T_3 qui avait été activée 2 fois précédemment se termine à deux reprises via le service `TerminateTask()`. Enfin la tâche T_1 reprend son exécution, s'active une fois de plus et se termine à deux reprises.

5.2.3 Suite de test

Une suite de test est un ensemble de séquence de tests. Dans le cas de la conformité OSEK/VDX, elle regroupe un ensemble d'applications exécutées sur le système d'exploitation. Suite à cela, la conformité du système d'exploitation par rapport au standard OSEK/VDX est déterminée.

Pour adapter le test de conformité OSEK/VDX au processus de synthèse formelle du système d'exploitation, la suite de test est décrite de façon formelle et appliquée au modèle du système d'exploitation synthétisé [TBFR15, TBR15a]. Pour cela, le modèle de chaque séquence de test qui constitue la suite de test est formé puis composé au modèle du système d'exploitation synthétisé. A partir du comportement du modèle du système d'exploitation face aux séquences de test, il est ainsi possible d'en déduire la conformité du système d'exploitation synthétisé.

5.2.4 Exemple

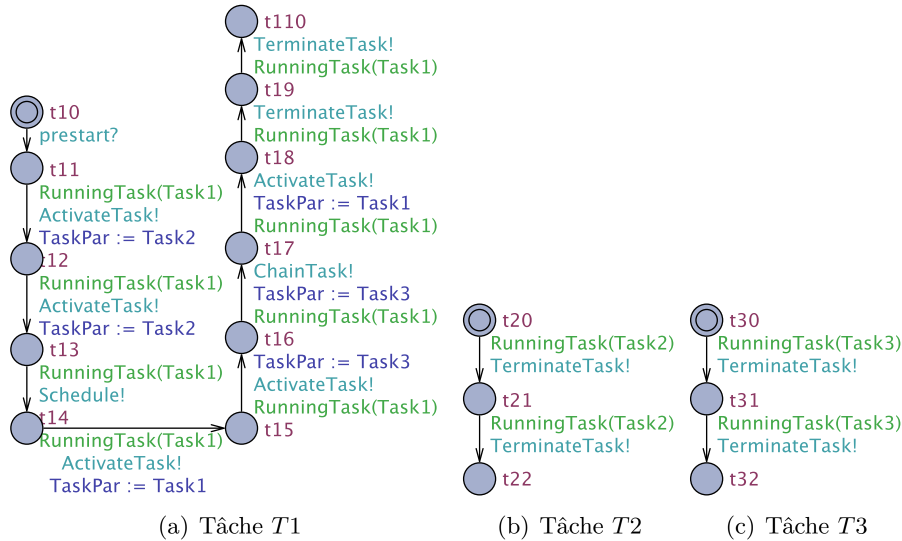
Ici, un exemple de modèle de séquence de test est proposé ainsi que la manière dont il est utilisé pour tester le modèle du système d'exploitation. Considérons la figure 5.3 présentant le modèle de la séquence de test décrite à la section 5.2.2.

Le modèle est la traduction en automates finis étendus de la séquence de test. Pour savoir si le modèle du système d'exploitation réagit correctement face au modèle de la séquence de test, un deuxième modèle est construit : il s'agit du modèle de l'observateur de la séquence de test.

L'observateur de la séquence de test retrace l'exécution attendue de la séquence de test puis permet d'émettre un verdict. Le modèle de l'observateur de l'exécution de la séquence de test représentée au tableau 5.2.2 est montré à la figure 5.4.

À chaque appel de service, l'observateur progresse jusqu'à atteindre son état final `success` si la séquence de test s'exécute comme convenu. Dans le cas contraire, c'est l'état `Fail` qui est atteint. Par exemple si à l'appel du premier service de la séquence de test `ActivateTask` le

1. Le compteur d'activation maximum détermine le nombre d'activations mémorisables d'une tâche. Par exemple si une tâche admet un compteur d'activation maximum égal à n, alors, cette tâche peut s'activer n fois jusqu'à la fin de son exécution



(a) Tâche $T1$ (b) Tâche $T2$ (c) Tâche $T3$

FIGURE 5.3 – Modèle de séquence de test

résultat retourné est différent de `E_OK`, alors le service ne s'est pas exécuté comme prévu et l'état `Fail` est atteint.

La variable `sequence` permet de retrouver à quel stade de l'exécution de la séquence de test une erreur s'est produite. Pour réaliser cette vérification à partir de l'observateur de l'exécution de la séquence de test, deux types de propriétés écrites en formule CTL peuvent être vérifiées. D'une part, nous vérifions une propriété de vivacité exprimant que tous les chemins du modèle complet mènent fatalement à l'état `success` (`AF Observer.success`) en d'autres termes, quelle que soit l'exécution du système d'exploitation la séquence de test s'exécute avec succès. D'autre part, nous vérifions une propriété d'accessibilité exprimant qu'un chemin du modèle conduit l'observateur vers l'état `Fail` (`EF Observer.Fail`). Dans ce cas, si la propriété d'accessibilité est vérifiée nous pouvons déduire que le test est un échec.

Pour réaliser la vérification de la conformité du système d'exploitation, le modèle de la suite de test est établi. Ce modèle comprend l'ensemble des modèles des séquences de tests qui la constituent puis à chaque séquence de test est associé un modèle d'observateur qui observe son exécution. Alors, pour chaque système d'exploitation synthétisé, un modèle de suite de test adapté aux services qu'il prend en charge est établi. Enfin, en composant le modèle de chaque séquence de test à celui du système d'exploitation synthétisé, sa conformité est déterminée.

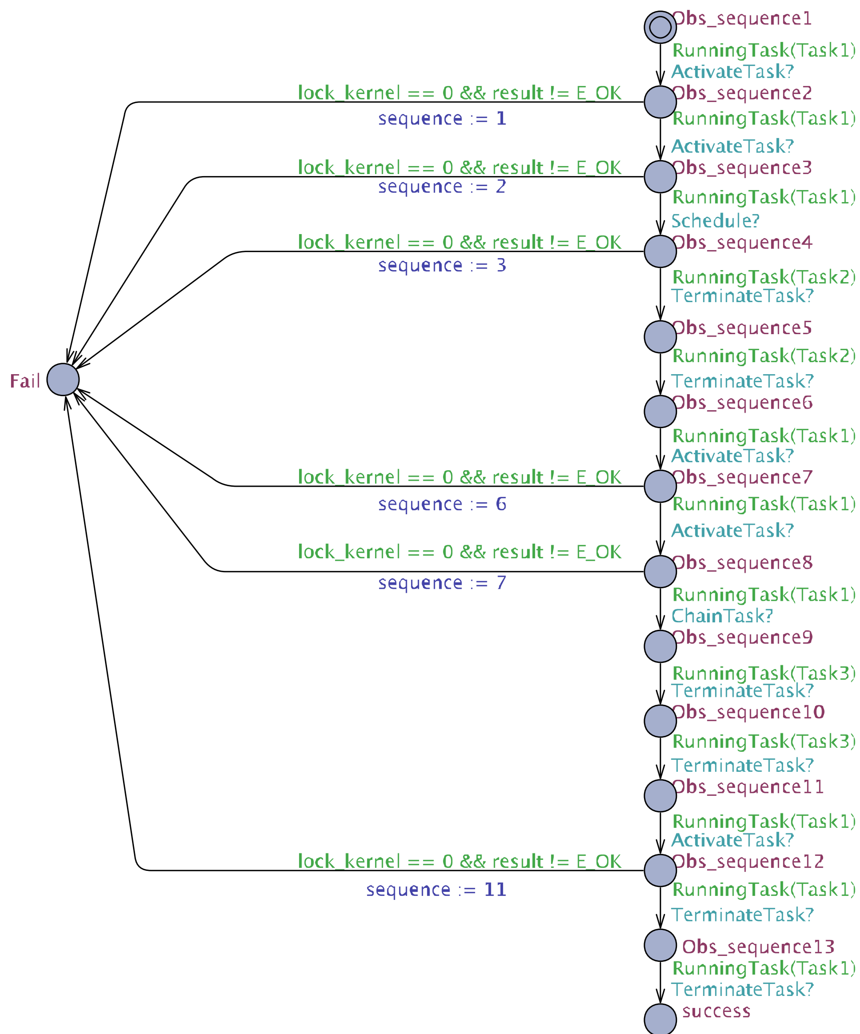


FIGURE 5.4 – Modèle d'un observateur de séquence de test

5.3 Vérification formelle de la conformité OSEK/VDX à partir d'observateurs génériques

5.3.1 Présentation de l'approche

Dans la première approche, il faut à chaque fois concevoir une nouvelle suite de test pour chaque synthèse car les services fournis par le système d'exploitation peuvent varier d'une synthèse à une autre et une suite de test est liée aux services d'un système d'exploitation. Cela exige en effet plus d'efforts et de temps pour la détermination de la conformité OSEK/VDX en utilisant cette approche.

Nous présentons pour cela une seconde approche plus simple et efficace. Cette approche permet de vérifier directement lors de l'exécution du modèle complet (système d'exploitation et application) certaines propriétés du système d'exploitation liées à la spécification OSEK/VDX qui sont la traduction directe des cas de tests.

Elle permet en effet de conclure à la conformité du système d'exploitation selon la spécification OSEK/VDX pour tout modèle d'application. De plus, cette méthode n'est pas seulement limitée au cas du système d'exploitation Trampoline. Elle peut être appliquée sur tout système d'exploitation implémenté à partir de la spécification OSEK/VDX et modélisé par des automates finis étendus.

La figure 5.5 montre le processus de vérification que nous avons développé. En fonction d'un plan de test [OSE99] qui contient l'ensemble des cas de tests et qui est dérivé de la spécification OSEK/VDX, tous les cas de tests (implicitement des propriétés) sont traduits en des observateurs.

Ces observateurs sont ensuite insérés dans le modèle complet composé du modèle du système d'exploitation et de l'application afin d'observer le comportement du système d'exploitation. Des propriétés d'accessibilité sont ensuite testées sur les états des observateurs pour vérifier que le système d'exploitation se comporte selon les exigences de la spécification OSEK/VDX lors de son interaction avec une application particulière.

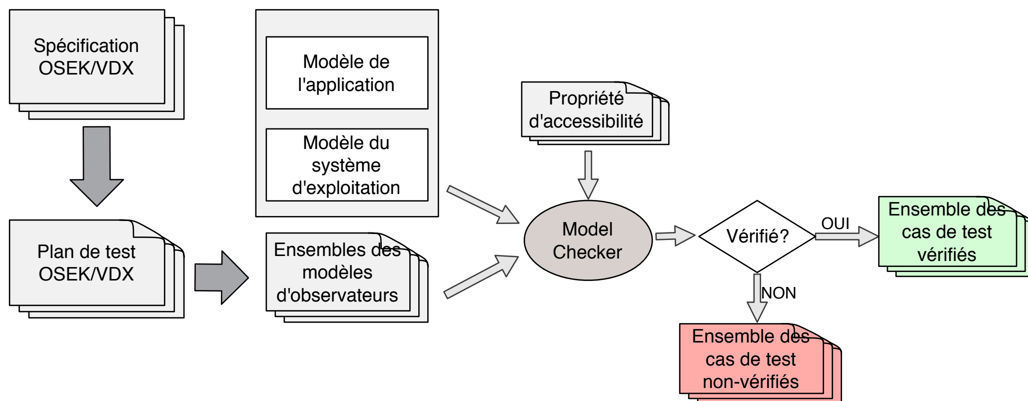


FIGURE 5.5 – Processus de vérification formelle

5.3.2 Observateurs

5.3.2.1 Observateur générique

Un observateur est par définition un automate fini étendu comportant des états *engagés*[2] ou *committed states* (voir section 3.3.1.2).

La figure 5.6 montre la forme générique d'un observateur où la pré-condition définit l'état dans lequel se trouve le système d'exploitation avant l'appel d'un de ses services. Cette pré-condition est exprimée par une garde.

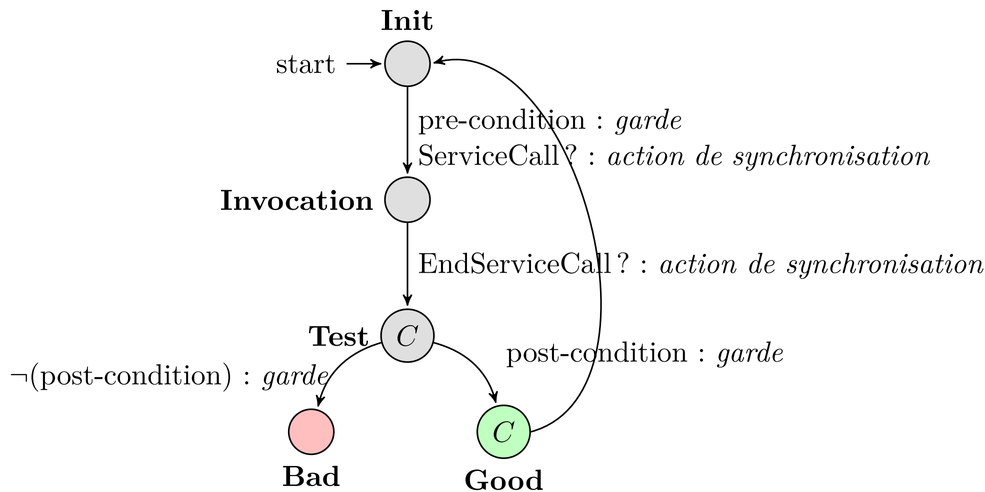


FIGURE 5.6 – Observateur générique *Obs*(`t`) pour un cas de test (propriété) `t`

Quand un service du système d'exploitation est appelé, le système d'exploitation le signale aux observateurs au moyen d'actions de synchronisation de type *broadcast* : `ServiceCall` (voir section 3.3.1.2) et à la fin de l'exécution du service demandé, le système d'exploitation utilise l'action `EndServiceCall` pour le signaler aux observateurs.

La post-condition représente l'état espéré du système d'exploitation lors de l'exécution du service demandé. Elle est aussi exprimée sous la forme d'une garde et si elle est satisfaite, l'état `Good` de l'observateur est atteint et le système d'exploitation satisfait bien la propriété correspondant à l'observateur. Dans le cas contraire, l'état `Bad` de l'observateur est atteint et la propriété traduite par celui-ci n'est pas vérifiée par le système d'exploitation.

Puisque le comportement du modèle est observé au moyen d'actions de synchronisation de type *broadcast*, il n'est pas impacté par les observateurs. En se référant à la figure 3.13 de la section 3.5.2, la figure 5.7 montre la modification apportée au modèle (principalement les services) du système d'exploitation afin de l'adapter à la procédure de vérification via les observateurs.

Dans ce modèle de fonction, deux actions de synchronisation ont été ajoutées : il s'agit des actions `TerminateTaskService` et `EndServiceCall`. L'action `TerminateTaskService` permet de notifier aux observateurs que le service de terminaison de tâches a été invoqué. Cette action représente l'action `ServiceCall` de la forme générique des observateurs. L'action `EndServiceCall` notifie la fin de l'exécution des services demandés. Tous les autres automates du modèle modélisant les services du système d'exploitation ont été modifiés de façon similaire.

2. Pour des raisons d'implémentation avec l'outil UPPAAL, nous ajoutons des *committed states* au modèle de l'observateur

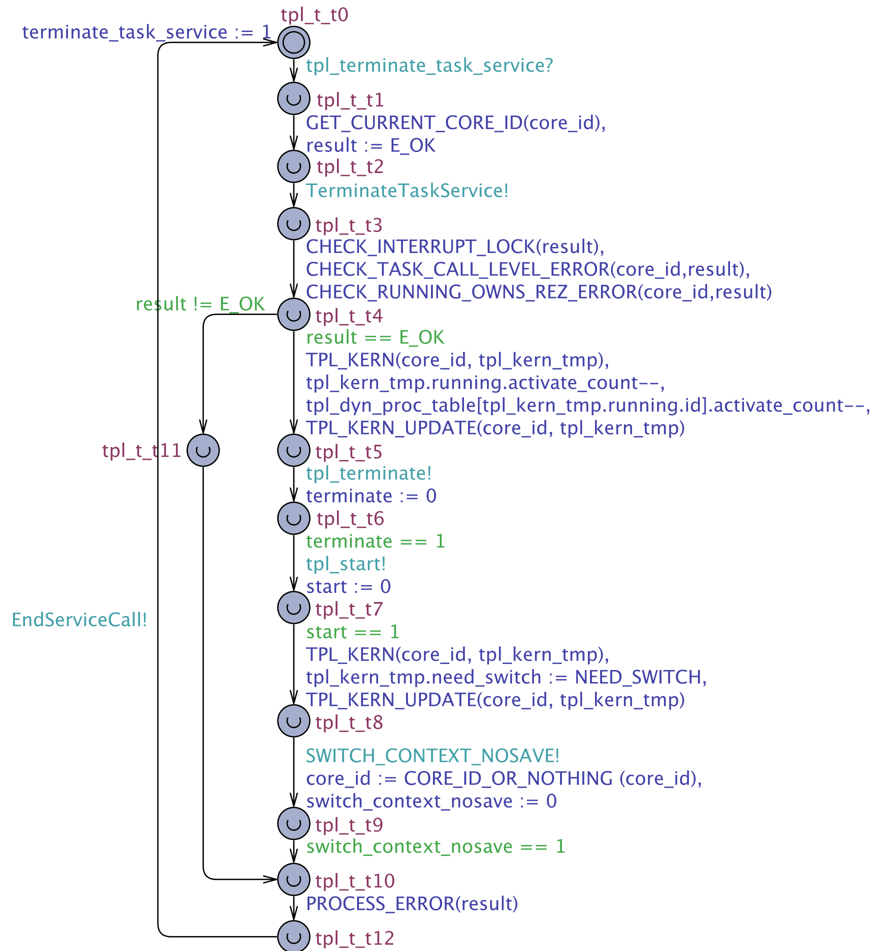


FIGURE 5.7 – Modification apportée au modèle initial de la fonction `tpl_terminate_task_service`

5.3.2.2 Propriété des observateurs

Les observateurs sont la traduction en automate des différentes propriétés ou cas de tests issus du plan de test OSEK/VDX. Les propriétés ne sont pas toutes utiles à vérifier vis à vis d'une application. En effet, pour une application qui nécessite un certain nombre de services, seules les propriétés qui sont associées à ces services doivent être prises en compte. Soit les définitions suivantes :

Définition 11. *Une propriété* $\mathtt{t} = \langle \mathtt{pre\text{-}condition,ServiceCall,post\text{-}condition} \rangle$ *est définie par une condition de déclenchement (pre-condition), un appel de service (ServiceCall) et une condition à satisfaire (post-condition).*

Définition 12. *Une propriété* $\mathtt{t} = \langle \mathtt{pre\text{-}condition,ServiceCall,post\text{-}condition} \rangle$ *est sensibilisée si la pré-condition est vraie lorsque l'appel de service* `ServiceCall` *se produit.*

Soit $Obs(\mathtt{t})$ l'observateur représenté à la figure 5.6 de la propriété `t` associé à la propriété $\mathtt{t} = \langle \mathtt{pre\text{-}condition,ServiceCall,post\text{-}condition} \rangle$. Nous introduisons ainsi les différentes propositions :

Proposition 3. *L'état* `Invocation` *de l'observateur* $Obs(\mathtt{t})$ *est atteignable si et seulement si la propriété qui lui est associée est sensibilisée.*

Démonstration. Le modèle du système d'exploitation décrit parfaitement son comportement réel (*cf.* proposition 2 de la section 3.7). L'observateur *Obs*(`t`) reste dans son état initial (`init`) jusqu'à ce que l'action `ServiceCall` se produise et que la pré-condition soit vraie. Après cela, l'observateur passe dans l'état `Invocation`. □

Proposition 4. *L'état* `Bad` *de l'observateur Obs*(`t`) *est atteignable si et seulement si la propriété* `t` *qu'elle traduit est sensibilisée et que la post-condition n'est pas satisfaite.*

Démonstration. Tout d'abord, la proposition 3 affirme que l'état `Invocation` de l'observateur *Obs*(`t`) est accessible dès que la propriété `t` est sensibilisée. De plus, dans chaque modèle de service du système d'exploitation, l'action `ServiceCall` est toujours suivie de l'action `EndServiceCall` et deux appels successifs de services (`ServiceCall`) sont toujours séparés par l'action `EndServiceCall`. Aussitôt que le système d'exploitation invoque l'action `EndServiceCall`, l'observateur passe à l'état `Test`. Après l'invocation de `EndServiceCall`, le modèle du système d'exploitation est dans un état non engagé. Ainsi, le modèle du système d'exploitation reste dans son état courant et l'observateur qui se trouve à cet instant à l'état `Test`(état engagé) le quitte immédiatement. En fonction de la post-condition qui est au préalablement définie, l'observateur part dans l'état `Bad` ou `Good`. Puisque l'état `Good` est aussi engagé, un autre appel de service ne peut être effectué par le système d'exploitation jusqu'à ce que l'observateur revienne à son état initial afin d'être capable d'observer à nouveau la propriété qu'elle traduit si elle est une fois de plus sensibilisée. Ainsi, soit la post-condition est fausse et l'observateur part dans l'état `Bad` ou elle est vraie et l'observateur est prêt à observer la prochaine sensibilisation de la propriété `t` qu'il traduit. □

5.3.2.3 Observateur de conformité OSEK/VDX

Pour la vérification de la conformité du système d'exploitation au standard OSEK/VDX, 139 cas de tests ou propriétés issus du plan de test OSEK/VDX ont été traduits en observateurs. Les observateurs obtenus suivent le modèle de l'observateur générique de la figure 5.4. La figure 5.8 montre l'exemple de l'observateur décrivant le cas de test 12 du tableau 5.2.2.

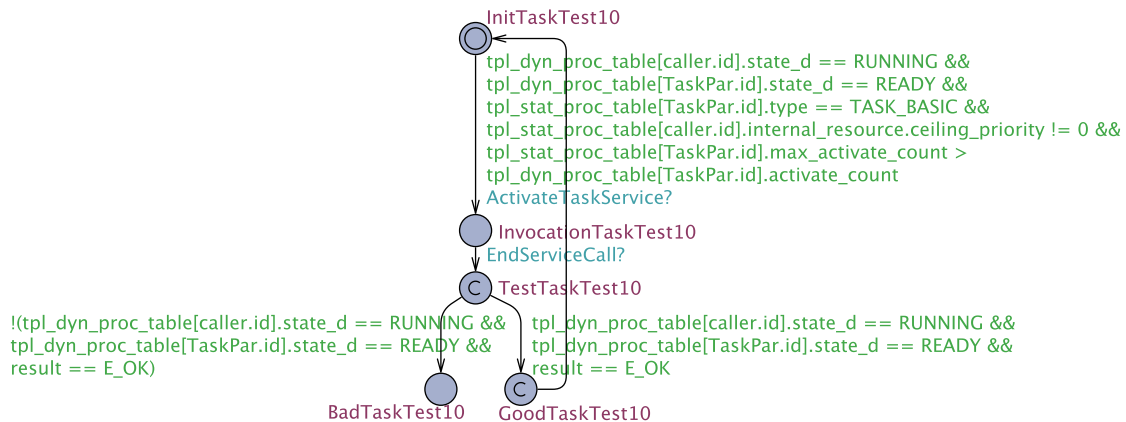


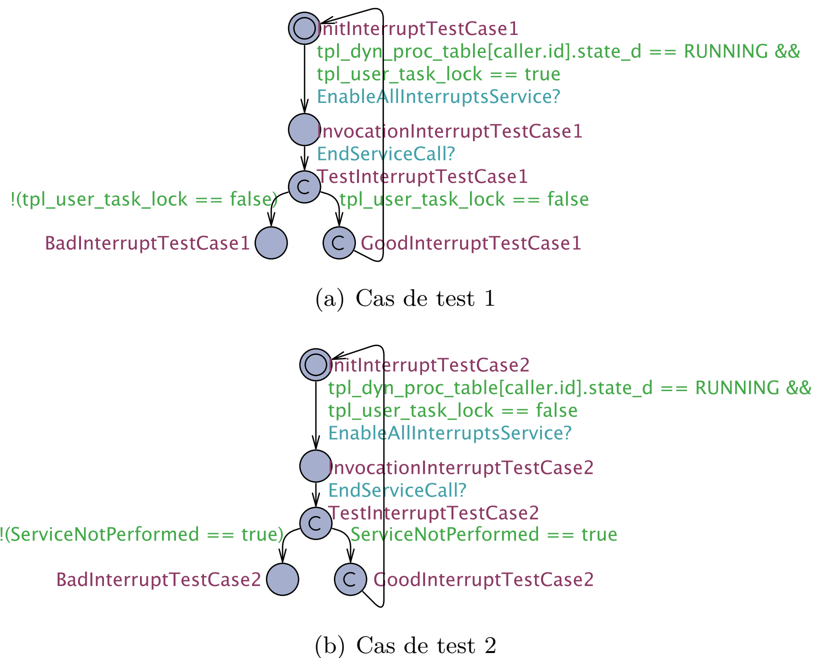
FIGURE 5.8 – Exemple d'un modèle d'observateur

Pour ce modèle d'observateur, la variable `caller.id` contient l'identifiant de la tâche qui effectue l'appel du service et la variable `TaskPar.id` contient l'identifiant de la tâche pour laquelle le service est appelé. Ainsi, la lecture de cet observateur se fait comme suit.
Si une tâche non préemptable (`tpl_stat_proc_table[caller_id].internal_resource.ceiling_priority != 0`) en cours d'exécution (`tpl_dyn_proc_table[caller.id].state_d == RUNNING`) fait appel au service `ActivateTask` (`ActivateTaskService?`) sur une tâche basique (`tpl_stat_proc_table[TaskPar.id].type == TASK_BASIC`) et prête (`tpl_stat_proc_table [TaskPar.id].state_d == READY`) et qui n'a pas encore atteint son compteur d'activation (...`max_activate_count` > ...`activate_count`), la tâche en

cours d'exécution n'est pas préemptée (`tpl_dyn_proc_table[caller.id].state_d == RUNNING`), la tâche activée est mise en attente dans la liste des tâches prêtes (`tpl_stat_proc_table [TaskPar.id].state_d == READY`) et le service retourne `E_OK` (`result == E_OK`).

Il existe certains cas de tests où le service appelé ne produit aucun effet sur les variables du système d'exploitation. Dans ce cas, il est difficile d'établir une post-condition qui permettrait de savoir si le cas de test s'est exécuté correctement. Ce type de cas de test intervient fréquemment pour la gestion des interruptions. Par exemple, si toutes les interruptions sont activées et qu'on fait appel au service permettant l'activation de ces interruptions, le service ne sera donc pas effectué et n'aura aucun effet sur les variables du système d'exploitation. Pour résoudre ce problème, nous ajoutons des variables supplémentaires au modèle du système d'exploitation qui n'influenceront en aucun cas son comportement.

Considérons maintenant la figure 5.9. Le cas de test 1 signifie que si toutes les interruptions ont été auparavant désactivées(`tpl_user_task_lock == true`), l'appel du service `EnableAllInterrupts` par une tâche entraîne leur activation. Ici, la post-condition peut être bien établie car ce cas de test entraîne une modification des variables du système d'exploitation. Par contre, le cas de test 2 signifie que lorsqu'aucune interruption n'a été désactivée auparavant et que le service `EnableAllInterrupts` est appelé par une tâche, le service n'est pas exécuté et ne produit donc aucun effet sur les variables du système d'exploitation. Ici, la post-condition est inexistante. Pour cela, une variable `ServiceNotPerformed` a été ajoutée au modèle du service d'activation d'interruptions. Cette variable qui est de type booléen permet ainsi d'établir la post-condition et lorsqu'elle est égale à `true`, cela signifie que le service appelé n'a pas été exécuté.



(a) Cas de test 1

(b) Cas de test 2

FIGURE 5.9 – Exemple de cas de tests liés aux interruptions

Tous les observateurs obtenus sont par la suite insérés dans le modèle complet (modèle de l'application et du système d'exploitation). Ainsi, par analyse de l'accessibilité des états `Invocation` et `bad` (`EF Obs.Bad` et `EF Obs.Invocation`) de chaque observateur, il est possible non seulement d'identifier l'ensemble des propriétés ou cas de tests applicables au modèle complet mais éga-

lement de savoir si ces propriétés sont satisfaites ou non. En plus de la vérification formelle du système d'exploitation, cette approche permet aux développeurs d'applications d'identifier automatiquement pour une application donnée, les propriétés qui sont pertinentes à vérifier pour prouver la conformité OSEK/VDX du système d'exploitation.

5.3.3 Étude de cas

Cette étude concerne un cas industriel proposé par le groupe THALES [3]. Le système à étudier est un système de suivi par vidéo aérienne qui est utilisé pour la surveillance intelligente et dans des applications de sécurité tactique.

Ce projet a été proposé pour le challenge *FMTV*[4] 2015. Ce défi a pour but de fournir des idées à propos de l'utilisation des méthodes formelles pour vérifier les propriétés temporelles des systèmes embarqués. Mais notre principal but concerne plutôt la vérification des propriétés fonctionnelles de systèmes d'exploitation. Par conséquent, les aspects temporels ne sont pas pris en compte.

Le système considéré est composé de deux grandes fonctions. D'une part une fonction de traitement des images vidéos et d'autre part une fonction de contrôle de caméras. Nous nous intéressons à la fonction de contrôle de caméras qui a pour but de contrôler la position d'une caméra en fonction de la trajectoire de l'avion. La caméra doit ainsi toujours se focaliser sur la cible quelle que soit la trajectoire prise par l'avion. Cette fonction contient trois sous-fonctions dont :

— Une sous-fonction de contrôle du suivi (`Tâche 1`) qui prend des informations périodiquement via des capteurs sur la trajectoire de l'avion.
— Une sous-fonction de prédiction de la position de la cible (`Tâche 2`) qui reçoit les données concernant la vitesse de l'avion, sa position et sa direction à partir de la sous-fonction de contrôle du suivi et effectue la prédiction de mouvement de l'objet suivi.
— Une sous-fonction de contrôle de caméras (`Tâche 3`) qui reçoit les données de la position de l'objet suivi à partir de la fonction de contrôle de suivi, puis calcule un nouvel angle pour la caméra à partir de la position de l'avion, sa vitesse, sa direction ainsi que la prédiction du mouvement de l'objet suivi.

Chaque sous-fonction est considérée comme une tâche et nous supposons dans ce cas que `Priorité(Tâche 1) > Priorité(Tâche 2) > Priorité(Tâche 3)`.

Le modèle de la fonction de contrôle et du suivi est représenté à la figure 5.10. Pour la conception du modèle de l'application, nous nous sommes intéressé à l'ensemble des appels de services qu'elle effectue et non à son comportement complet.

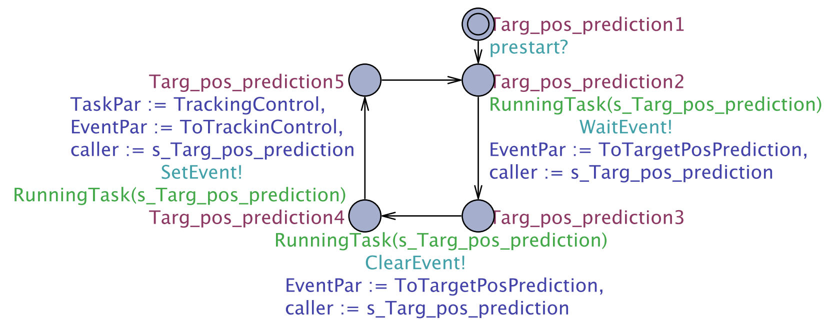
Puisque les différentes tâches de l'application s'exécutent en se synchronisant entre elles, nous utilisons les services de Trampoline liés à la gestion des événements qui assurent la communication et la synchronisation des tâches. Nous rappelons aussi que nous ne nous intéressons pas aux propriétés temporelles du système d'exploitation mais plutôt à ses propriétés fonctionnelles afin de prouver formellement que pour cette application, le système d'exploitation s'exécute correctement selon la spécification OSEK/VDX.

Toutes les tâches sont étendues et périodiques. Elles sont implémentées dans des boucles. Initialement, elle sont en attente d'événements se produisant périodiquement. Puis lorsque l'événement qu'elles attendent se produit, elle commence leurs exécutions et reviennent en début de boucle pour être en attente d'une prochaine occurrence d'événements.

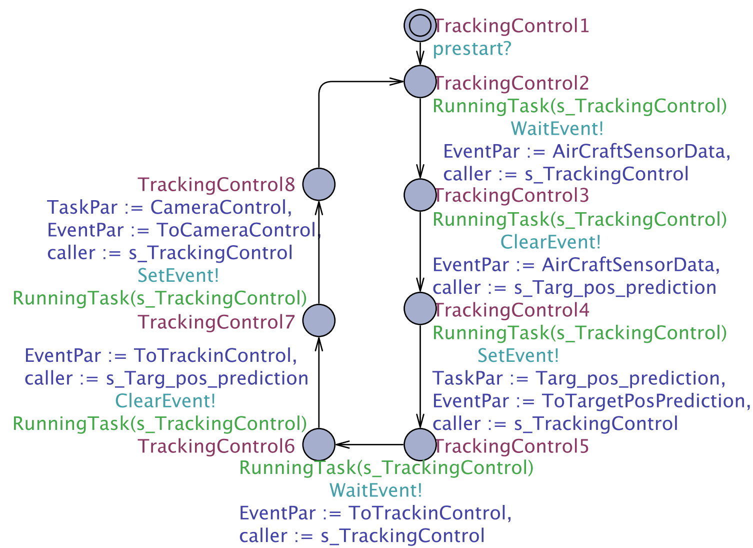
3. http://waters2015.inria.fr/challenge/
4. Formal Methods For Timing Verification

Lorsque nous prenons l'exemple de la tâche de contrôle du suivi. Elle est initialement en attente de l'occurrence de l'événement `AirCraftSensorData` (Cet événement est vu comme une information provenant du capteur de l'avion). Une fois l'événement produit, cette tâche fait appel au service `ClearEvent` pour effacer l'événement reçu. Ensuite, cette tâche produit l'événement `ToTargetPosPrediction` via le service `SetEvent` destiné à la tâche de prédiction de la position de la cible (cet événement représente l'information que la tâche de contrôle du suivi envoi à la tâche de prédiction de la position de la cible concernant la vitesse, la position et la direction de l'avion). Elle suspend ensuite son exécution en attente de l'événement `ToTrackingControl` devant provenir de la tâche de prédiction de la position de la cible. Lorsque l'événement est produit, la tâche fait appel au service `ClearEvent` puis produit l'événement `ToCameraControl` destiné à la tâche de contrôle de caméra (cet événement représente les données de la position de l'objet suivi que la tâche de contrôle de suivi envoie à la tâche de contrôle de la caméra). Une fois son exécution terminée, elle revient en début de boucle puis se remet en attente d'un nouvel événement.

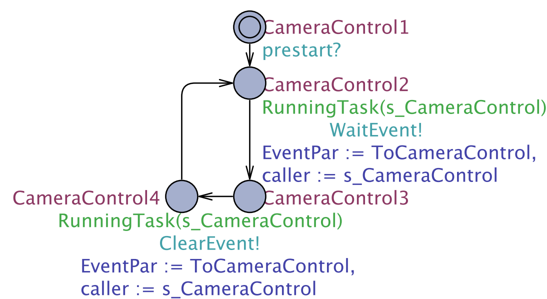
Résultat Parmi les 139 observateurs insérés dans le modèle du système complet, cinq observateurs ont été invoqués pour cette application. Ces observateurs traduisent les propriétés pertinentes à vérifier. La vérification a été faite à partir de l'analyse d'accessibilité des états `Bad` de chaque observateur. Pour chaque vérification, le vérificateur de UPPAAL prend 8 secondes sur un PC ayant un processeur Intel Core i5 tournant à une vitesse de 2.5GHz et 4Go de RAM. Les observateurs n'ont observé aucun comportement inattendu du système d'exploitation pour cette application. Cela permet de conclure que le modèle du système d'exploitation respecte bien les exigences de la spécification OSEK/VDX.



(a) Prédiction de la position de la cible



(b) Contrôle du suivi



(c) Contrôle de la caméra

FIGURE 5.10 – Modèle du système de contrôle de caméra et du suivi

5.4 Conclusion

Dans ce chapitre, nous avons d'abord présenté quelques techniques de vérifications formelles tels que le *model-checking*, l'analyse statique et le test formel ainsi que quelques approches basées sur ces techniques pour la vérification de systèmes informatiques.

Nous avons ensuite proposé deux approches de vérification de la conformité d'un système d'exploitation selon la spécification OSEK/VDX.

La première approche suit le processus standard du test de conformité OSEK/VDX c'est-à-dire que le test est établi à partir de l'exécution d'une suite de test sur le système d'exploitation. Puisque cette approche est basée sur le modèle, la suite de test est alors modélisée puis son modèle est composé au modèle du système d'exploitation afin de déterminer sa conformité par rapport au standard OSEK/VDX.

Compte tenu de l'objectif de la thèse qui concerne la synthèse formelle de systèmes d'exploitation en fonction d'une application, cette approche permet de vérifier pour chaque application la conformité du système d'exploitation selon la spécification OSEK/VDX. Ainsi, pour chaque synthèse du système d'exploitation, un modèle de suite de test est établi puis composé au modèle du système d'exploitation synthétisé. Cela demande en effet plus d'efforts dans la composition du modèle de la suite de test car il faut toujours tenir compte de l'ensemble des services pris en compte dans le système d'exploitation synthétisé.

Pour pallier à ce problème, une deuxième approche a été proposée : celle de la vérification de la conformité du système d'exploitation à partir d'observateurs. Dans cette approche, tous les cas de tests ou propriétés issus de la spécification OSEK/VDX sont traduits en observateurs. Ces observateurs ont pour but d'observer le comportement du système d'exploitation indépendamment de toute application. Lorsque la propriété qu'ils traduisent est bien observée et respectée par le système d'exploitation lors de son exécution, ils signalent que la propriété est satisfaite. Cette approche à pour avantage de déterminer automatiquement pour toute application, l'ensemble des propriétés utiles à vérifier sur le système d'exploitation afin de savoir si celles-ci sont satisfaites ou non. Elle accélère aussi le processus de vérification de la conformité du système d'exploitation selon la spécification OSEK/VDX.

Conclusion générale et perspectives

Bilan

Le développement des systèmes embarqués est une tâche ardue pour laquelle il faut prendre en compte les aspects temps réel, la concurrence et la répartition des fonctions. De plus, ces systèmes présentent un certain nombre de contraintes telles que la mémoire disponible, la consommation en énergie ou encore la puissance de calcul.

La complexité grandissante des supports d'exécution (e.g. micro-contrôleurs multi-cœur) ainsi que des applications qui exigent de plus en plus de fonctionnalités ne fait qu'accroitre la difficulté du développement de ces systèmes. Pour gérer cette complexité, l'utilisation des systèmes d'exploitation se révèle incontournable mais ceux-ci dans bien des cas entraînent des couts additionnels en mémoire et des risques liés à la sureté de fonctionnement. Pour cela, les systèmes d'exploitation doivent être conçus de manière à être configurable vis-à-vis de l'application (n'embarquer que les fonctionnalités nécessaires à l'exécution de l'application) ce qui permettrait d'optimiser l'empreinte mémoire occupée par ceux-ci afin de pouvoir gérer la complexité des systèmes embarqués tout en satisfaisant les contraintes de mémoire.

Ainsi, dans ce manuscrit nous avons tout d'abord présenté les différentes techniques de développement de systèmes d'exploitation configurables. Ces différentes techniques sont basées sur des paradigmes de développement tels que le langage orienté objet, le langage orienté aspect etc. Le but principal de ces approches repose sur la décomposition du système d'exploitation en modules de manière à pouvoir charger ou non des modules renfermant les fonctionnalités nécessaires à l'application. Cependant, ces techniques n'adressent pas le problème du code mort qui peut parfois être source de défaillance.

Dans ce cadre, nous avons proposé une approche basée sur les méthodes formelles permettant la génération de systèmes d'exploitation temps-réel statiques spécifiques aux besoins de l'application en terme de service. Dans cette approche, le modèle formel du système d'exploitation embarquant son flot de contrôle, ses variables et les séquences d'instructions manipulant ces variables ainsi qu'un modèle de l'application sont conçus. Ces deux modèles sont composés pour en former un plus complet qui décrit le déploiement de l'application sur le système d'exploitation. Ensuite, par analyse d'accessibilité des états du système d'exploitation, tous les chemins infaisables et états inaccessibles du modèle (et donc du système d'exploitation) sont supprimés. Ainsi, à partir de ce modèle spécialisé, le code en C correspondant est engendré au moyen d'un générateur de code.

Cette approche présente plusieurs avantages dont :

- La génération d'un système d'exploitation qui ne contient aucun code mort du point de vue des services qu'il fournit car suite à l'analyse d'accessibilité faite sur son modèle, seul le code nécessaire à l'exécution de l'application est retenu.
- La génération d'un système d'exploitation vérifié fonctionnellement à partir de son modèle grâce à la technique du *model checking*.

— La vérification de la conformité du système d'exploitation à la spécification qu'il implémente.

Cette approche a été appliquée sur le système d'exploitation Trampoline utilisé dans les systèmes embarqués automobiles à ressources limitées grâce à UPPAAL un outil de modélisation et de vérification des systèmes temps réel et d'un outil que nous avons implémenté en Python. Les différentes études de cas présentées ont montré la faisabilité de cette approche tout en présentant des résultats intéressants sur la puissance d'optimisation du code source et sur les moyens de vérification du système d'exploitation configuré. Par contre, cette approche ne peut s'appliquer que sur les systèmes d'exploitation statiques où tous les besoins de l'application sont connus à l'avance

Perspectives

La synthèse du système d'exploitation est actuellement faite sur la version monocœur de Trampoline. En ce qui concerne la version multicœur, pour l'instant, Trampoline utilise un système de verrouillage global du noyau qui interdit à plus d'un cœur d'exécuter le code du système d'exploitation à un instant donné ou encore de traiter simultanément des interruptions. Cette caractéristique permet de modéliser Trampoline par un seul ensemble d'automates quel que soit le nombre de cœurs car le niveau de parallélisme dans Trampoline n'est pas total.

Prochainement, concernant la version multicœur de Trampoline, nous avons pour objectif d'autoriser l'exécution simultanée du code du noyau sur des cœurs différents. Cette concurrence particulière n'est pas descriptible par des automates finis étendus. Nous projetons donc de baser notre modélisation sur les réseaux de pétri temporels [Mer74] permettant la description de systèmes concurrents. Il serait nécessaire de faire une extension des réseaux de pétri avec des couleurs dans l'outil Roméo [LRST09] où une couleur sera attribuée à chaque cœur de manière à bien décrire le parallélisme souhaité.

Une seconde perspective serait d'insérer du temps précis (pire temps d'exécution des tâches de l'application) dans les modèles d'applications afin de pouvoir vérifier l'ordonnancement des tâches vis-à-vis d'un ordonnanceur quelconque et de pouvoir le valider formellement.

Enfin, une troisième perspective serait d'utiliser un langage DSL afin de faciliter la modélisation du système d'exploitation. Le langage DSL permettra non seulement de générer automatiquement des modèles du système d'exploitation à partir de spécification qu'il décrit mais aussi de mettre en œuvre une génération de code plus efficace.

Publications

Publications dans des revues internationales à comité de lecture

— Toussaint Tigori, Jean-Luc Béchennec, Sébastien Faucou and Olivier-Henri Roux. Formal model-based synthesis of application specific static RTOS in *ACM Transaction on Embedded Computing Systems* à paraître, 2016.

Communication à des congrès internationaux à comité de lecture et actes publiés

— Toussaint Tigori, Jean-Luc Béchennec and Olivier-Henri Roux. "Formal Synthesis of Optimal RTOS" in *17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conf on Embedded Software and Systems*, HPCC-CSS-ICESS '15, pages 977–983, Washington, DC, USA, 2015. IEEE Computer Society.

— Toussaint Tigori, Jean-Luc Béchennec, Sébastien Faucou and Olivier-Henri Roux. Using formal methods for the development of safe application-specific RTOS for automotive systems In *Matthieu Roy, editor, CARS 2015 - Critical Automotive applications : Robustness and Safety,* Paris, France, September 2015.

Communication à des congrès nationaux à comité de lecture et actes publiés

— Toussaint Tigori, Jean-Luc Béchennec and Olivier-Henri Roux. Approche formelle pour la spécialisation de systèmes d'exploitation temps réel. In Stephan Merz and Jean-François Pétin, editors, *Modélisation des Systèmes Réactifs (MSR 2015),* Nancy, France, November 2015.

Bibliographie

[ABB+93] H. Assenmacher, T. Breitbach, P. Buhler, V. Hubsch, and R. Schwarz. The panda system architecture-a pico-kernel approach. In *Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of*, pages 470–476, Sep 1993.

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2) :183–235, 1994.

[AHM+08] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5) :22–29, Sept 2008.

[AUT13] AUTOSAR Release AUTOSAR. 4.1, specification of operating system, 2013.

[Bau99] Lothar Baum. Towards generating customized run-time platforms from generic components. In *Proc. of the 11th Conference on Advanced Information Systems Engineering (CAISE'99), 6th DC*, 1999.

[BB05] Henrik Bohnenkamp and Axel Belinfante. Timed testing with torx. In *International Symposium on Formal Methods*, pages 173–188. Springer, 2005.

[BBF15] Jean-Luc Béchennec, Mikaël Briday, and Sébastien Faucou. Système d'exploitation temps réel pour l'informatique embarquée. In *ETR'2015, Ecole d'été temps réel*, Rennes, France, Août 2015.

[BBFT06] Jean-Luc Béchennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline an open source implementation of the OSEK/VDX RTOS specification. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2006.

[BC09] A Balogh and Z Csörnyei. Subject-oriented operating system development based on system predicate classes. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 117–139, 2009.

[BCE+95] Brian N Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. Spin—an extensible microkernel for application-specific operating system services. *ACM SIGOPS Operating Systems Review*, 29(1) :74–77, 1995.

[BDL+01] Gerd Behrmann, Alexandre David, Kim G. Larsen, Oliver Möller, Paul Pettersson, and Wang Yi. Uppaal - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control.* IEEE Computer Society Press, 2001.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems : 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[Beu97] Danilo Beuche. An approach for managing highly configurable operating systems. In *Object-Oriented Technologys*, pages 531–536. Springer, 1997.

[BLL+95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[BLL+96] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 431–434. Springer–Verlag, 1996.

[BM02] Luciano Porto Barreto and Gilles Muller. Bossa : a language-based approach to the design of real-time schedulers. In *Proceedings of the 10th International Conference on Real-Time Systems (RTS'2002)*, pages 19–31, Paris, France, March 2002.

[BNO+04] R. Barreto, M. Neves, M. Oliveira, P. Maciel, E. Tavares, and R. Lima. A formal software synthesis approach for embedded hard real-time systems. In *Integrated Circuits and Systems Design, 2004. SBCCI 2004. 17th Symposium on*, pages 163–168, Sept 2004.

[BPSV94] Felice Balarin, Karl Petty, AL Sangiovanni, and Pravin Varaiya. Formal verification of the patho real-time operating systems. In *IEEE Conference on Decision and Control*, volume 3, pages 2459–2459, 1994.

[BRU+03] Ron Brightwell, Rolf Riesen, Keith Underwood, Trammell B Hudson, Patrick Bridges, and Arthur B Maccabe. A performance comparison of linux and a lightweight kernel. In *IEEE International Conference on Cluster Computing*, pages 251–258. IEEE, 2003.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. *SIGOPS Oper. Syst. Rev.*, 29(5) :267–283, December 1995.

[CA11] Jiang Chen and Toshiaki Aoki. Conformance testing for osek/vdx operating system using model checking. In *Software Engineering Conference (APSEC), 18th Asia Pacific*, pages 274–281, 2011.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CE04] Christopher L Conway and Stephen A Edwards. Ndl : a domain-specific language for device drivers. In *ACM Sigplan Notices*, volume 39, pages 30–36. ACM, 2004.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263, April 1986.

[Che84] David R. Cheriton. The v kernel : A software base for distributed systems. *Ieee Software*, 1(2) :19, 1984.

[Cho11] Yunja Choi. Safety analysis of trampoline OS using model checking : An experience report. In *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, pages 200–209, 2011.

[CJMR91] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design, 1991.

[CJRZ02] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. Stg : A symbolic test generation tool. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475. Springer, 2002.

[CKF00] Yvonne Coady, Gregor Kiczales, and Michael Feeley. Exploring an aspect-oriented approach to operating system code. In *Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Minneapolis, Minnesota, USA*, 2000.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 88–98. ACM, 2001.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*. Springer, 2012.

[CL09] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[Cop99] James O Coplien. *Multi-paradigm Design for C++*. Addison-Wesley, 1999.

[Dra93] R. Draves. The case for run-time replaceable kernel modules. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 160–164, Oct 1993.

[Dru93] Peter Druschel. Efficient support for incremental customization of os services. In *In Proceedings of the Third International Workshop on Object Orientation in Operating Systems*. Citeseer, 1993.

[DSS90] Sean Dorward, Ravi Sethi, and Jonathan E Shopiro. Adding new code to a running c++ program. In *C++ Conference*, pages 279–292, 1990.

[DuB96] Paul DuBois. *Software Portability with imake*. " O'Reilly Media, Inc.", 1996.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit : A substrate for kernel and language research. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 38–51. ACM, 1997.

[FKDO12] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for autosar multicore rtos. In *Software Testing, Verification and Validation, IEEE Fifth International Conference on*, pages 251–259, 2012.

[Frö01] Antônio Augusto Medeiros Fröhlich. *Application Oriented Operating Systems*, volume 1. GMD-forschungszentrum informationstechnik Sankt Augustin, 2001.

[FSP99] Antônio Augusto Fröhlich and Wolfaang Schröder-Preikschat. Tailor-made operating systems for embedded parallel applications. In *Parallel and Distributed Processing*, pages 1361–1373. Springer, 1999.

[G+05] OSEK Group et al. Osek/vdx operating system specification. *site web : http ://www. osek-vdx. org*, 2005.

[GLM+05] Guillaume Gardey, Didier Lime, Morgan Magnin, et al. Romeo : A tool for analyzing time petri nets. In *International Conference on Computer Aided Verification*, pages 418–423. Springer, 2005.

[GoEE89] W Morven Gentleman and National Research Council Canada. Division of Electrical Engineering. *Managing configurability in multi-installation realtime programs.* Citeseer, 1989.

[Gro99] OSEK Group. Osek/vdx os test procedure version 2.0. Technical report, OSEK Group, April 1999.

[HF98] Johannes Helander and Alessandro Forin. Mmlite : a highly componentized system architecture. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103. ACM, 1998.

[HO93] William Harrison and Harold Ossher. *Subject-oriented programming : a critique of pure objects*, volume 28. ACM, 1993.

[Hol04] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *ACM SIGOPS operating systems review*, volume 34, pages 93–104. ACM, 2000.

[JJ05] Claude Jard and Thierry Jéron. Tgv : theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4) :297–315, 2005.

[Joh77] Stephen C Johnson. *Lint, a C program checker.* Citeseer, 1977.

[Joh98] D. John. Osek/vdx conformance testing - modistarc. *IET Conference Proceedings*, pages 7–7(1), January 1998.

[KFG+93] Hermann Kopetz, Gerhard Fohler, Günter Grünsteidl, Heinz Kantz, Gustav Pospischil, Peter Puschner, Johannes Reisinger, Ralf Schlatterbeck, Werner Schütz, Alexander Vrchoticky, et al. Real-time system development : The programming model of mars. In *Autonomous Decentralized Systems, 1993. Proceedings. ISADS 93., International Symposium on*, pages 290–299. IEEE, 1993.

[KKC+15] Jinhyun Kim, Inhye Kang, Jin-Young Choi, Insup Lee, and Sungwon Kang. Formal synthesis of application and platform behaviors of embedded software systems. *Software & Systems Modeling*, 14(2) :839–859, 2015.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97—Object-oriented programming*, pages 220–242. Springer, 1997.

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. *Tools for the development of application-specific virtual memory management*, volume 28. ACM, 1993.

[KTD+13] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *20th Network and Distributed System Security Symposium (NDSS '13)*, 2013.

[Lad03] R. Laddad. Aspect-oriented programming will improve quality. *IEEE Software*, 20(6) :90–91, Nov 2003.

[LGS04] Daniel Lohmann, Wasif Gilani, and Olaf Spinczyk. On adaptable aspect-oriented operating systems. In *Proceedings of the 2004 ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS'04)*, pages 47–52, 2004.

[LHSP+09] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. Ciao : An aspect-oriented operating-system family for resource-constrained embedded systems. In *USENIX Annual Technical Conference*, 2009.

[LHSPS11] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Aspect-aware operating system development. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pages 69–80. ACM, 2011.

[LRST09] Didier Lime, Olivier H Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo : A parametric model-checker for petri nets with stopwatches. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 54–57. Springer, 2009.

[Mah11] Hatem Mahbouli. *Modélisation de programmes C en expressions régulières.* PhD thesis, Université Laval, 2011.

[MED06] David MacKenzie, Ben Elliston, and Akim Demaille. Gnu autoconf, creating automatic configuration scripts. *Free Software Foundation*, 2006.

[Mer74] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems.* PhD thesis, 1974. AAI7511026.

[Mos97] David Mosberger. *Scout : A path-based operating system.* PhD thesis, Citeseer, 1997.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil : An idl for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 2–2. USENIX Association, 2000.

[MSGSP02] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the pure operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, pages 49–54, 2002.

[OKH+95] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '95, pages 235–250, New York, NY, USA, 1995. ACM.

[OSE99] OSEK Group. OSEK/VDX OS test plan version 2.0, April 1999.

[Par78] David L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[PBC+97] Calton Pu, Andrew P Black, Crispin Cowan, Jonathan Walpole, and Charles Consel. Microlanguages for operating system specialization. In *SIGPLAN, Workshop on Domain-Specific Languages*, January 1997.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. 1995.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[RAA+91] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the chorus distributed operating systems. *Computing Systems*, 1 :39–69, 1991.

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit : Component composition for systems software. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, Berkeley, CA, USA, 2000. USENIX Association.

[RT78] OM Ritchie and Keith Thompson. The unix time-sharing system. *Bell System Technical Journal, The*, 57(6) :1905–1929, 1978.

[SA92] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[SESS96] Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. Dealing with disaster : Surviving misbehaved kernel extensions. In *OSDI*, pages 213–227, 1996.

[SL04] Olaf Spinczyk and Daniel Lohmann. Using aop to develop architectural-neutral operating system components. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 34. ACM, 2004.

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of aspectc++. *Knowledge-Based Systems*, 20(7) :636–651, 2007.

[SP94] Wolfgang Schröder-Preikschat. Suprenum and genesis peace — a software backplane for parallel computing. *Parallel Computing*, 20(10) :1471 – 1485, 1994.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9) :1175–1185, 1990.

[SS95] C. Small and M. Seltzer. Structuring the kernel as a toolkit of extensible, reusable components. In *Object-Orientation in Operating Systems, 1995., Fourth International Workshop on*, pages 134–137, Aug 1995.

[SSPSS99] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design rationale of the pure object-oriented embedded operating system. In *Distributed and Parallel Embedded Systems*, pages 231–240. Springer, 1999.

[TBB+13] Julien Tanguy, Jean-Luc Béchennec, Mikaël Briday, Sébastien Dubé, and Olivier Roux. Device driver synthesis for embedded systems. In *IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA), 2013*, pages 1–8. IEEE, 2013.

[TBFR15] Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, Sébastien Faucou, and Olivier Roux. Using formal methods for the development of safe application-specific RTOS for automotive systems. In Matthieu Roy, editor, *CARS 2015 - Critical Automotive applications : Robustness & Safety*, Paris, France, September 2015.

[TBFR16] Toussaint Tigori, Jean-Luc Bechennec, Sebastien Faucou, and Olivier H. Roux. Formal model-based synthesis of application specific static RTOS. *ACM Transactions on Embedded Computing Systems*, 2016. To appear, ACM.

[TBR15a] Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, and Olivier Henri Roux. Approche formelle pour la spécialisation de systèmes d'exploitation temps réel. In Stephan Merz and Jean-François Pétin, editors, *Modélisation des Systèmes Réactifs (MSR 2015)*, Nancy, France, November 2015.

[TBR15b] Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, and Olivier Henri Roux. Formal synthesis of optimal rtos. In *Proceedings of the 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conf on Embedded Software and Systems*, HPCC-CSS-ICESS '15, pages 977–983, Washington, DC, USA, 2015. IEEE Computer Society.

[THH00] Martin Treiber, Ansgar Hennecke, and Dirk Helbing. Congested traffic states in empirical observations and microscopic simulations. *Phys. Rev. E*, 62 :1805–1824, Aug 2000.

[Tho00] Gary Thomas. ecos : An operating system for embedded systems. *Dr. Dobb's journal*, 25(1), 2000.

[TKH+12] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Automatic os kernel tcb reduction by leveraging compile-time configurability. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability*, HotDep'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[Tre92] Gerrit Jan Tretmans. A formal approach to conformance testing. 1992.

[WH08] Libor Waszniowski and Zdeněk Hanzálek. Formal verification of multitasking applications based on timed automata model. *Real-Time Systems*, 38(1) :39–65, 2008.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

UNIVERSITE BRETAGNE LOIRE

ECN Centrale Nantes

Thèse de Doctorat

Kabland Toussaint Gautier TIGORI

Méthode de génération d'exécutifs temps-réel

Process of real-time operating systems generation

Résumé

Dans les systèmes embarqués, la spécialisation ou la configuration des systèmes d'exploitation temps réel en fonction des besoins de l'application consiste en la suppression des services inutiles du système d'exploitation. Cela a non seulement pour but d'optimiser l'empreinte mémoire occupée par le système d'exploitation temps réel, afin de satisfaire les contraintes de mémoire auxquelles les systèmes embarqués font face, mais aussi de réduire la quantité de code mort au sein du système d'exploitation temps réel afin d'améliorer la sureté de fonctionnement.
Dans nos travaux de thèse, nous nous intéressons à l'utilisation des méthodes formelles dans le processus de spécialisation des systèmes d'exploitation temps réel.
Une difficulté majeure dans l'utilisation de modèles formels est la distance entre le modèle et le système modélisé. Nous proposons donc de modéliser le système d'exploitation de telle sorte que le modèle embarque son code source et les structures de données manipulées. Nous utilisons à cet effet un modèle à états finis (éventuellement temporisé) augmenté par des variables discrètes et des séquences d'instructions, considérées comme atomiques, manipulant ces variables.
À partir du modèle du système d'exploitation et d'un modèle de l'application visée, l'ensemble des états accessibles du modèle du système d'exploitation traduisant le code effectivement utilisé lors de l'exécution de l'application est calculé. Le code source du système d'exploitation spécialisé est extrait de ce modèle élagué.
L'ensemble de la démarche exposée est mise en œuvre avec Trampoline, un système d'exploitation temps réel basée sur les standards OSEK/VDX et AUTOSAR.
Cette technique de spécialisation garantit l'absence de code mort, minimise l'empreinte mémoire et fournit un modèle formel du système d'exploitation utilisable dans une étape ultérieure de model-checking. Dans ce cadre, nous proposons une technique automatique de vérification formelle de la conformité aux standards OSEK/VDX et AUTOSAR à l'aide d'observateurs génériques.

Abstract

In embedded systems, specialization or configuration of real-time operating systems according to the application requirements consists to remove the operating system services that are not needed by the application. This operation allows both to optimize the memory footprint occupied by the real-time operating system in order to meet the memory constraints in embedded systems and to reduce the amount of dead code inside the real-time operating system in order to improve its reliability.
In this thesis, we focus on the use of formal methods to specialize real-time operating systems according applications.
One major difficulty using formal models is the gap between the system model and its implementation. Thus, we propose to model the operating system so that the model embeds its source code and manipulated data structures. For this purpose, we use finite state model (possibly timed model) with discrete variables and sequences of instructions which are considered as atomic manipulating these variables.
From the operating system model and an application model, the set of reachable states of the operating system model describing the code needed during application execution is computed. Thus, the source code of the specialized operating system is extracted from the pruned model.
The overall approach is implemented with Trampoline, a real-time operating system based on OSEK/VDX and AUTOSAR standards.
This specialization technique ensures the absence of dead code, minimizes the memory footprint and provides a formal model of the operating system used in a last step to check its behavior by using model checking. In this context, we propose an automatic formal verification technique that allows to check the operating systems according OSEK/VDX and AUTOSAR standards using generic observers.

Mots clés

Systèmes d'exploitation temps-réel, systèmes embarqués, méthodes formelles, modèles à états finis, vérification formelle.

Key Words

Real time operating systems, embedded systems, formal methods, finite state model, formal verification.