Approche formelle pour la spécialisation de systèmes d'exploitation temps réel

Kabland Toussaint Gautier Tigori, Jean-Luc Béchennec, Olivier Henri Roux

Approche formelle pour la spécialisation de systèmes d'exploitation temps réel

Kabland Toussaint Gautier Tigori[2,3], Jean-Luc Béchennec[1,3], and Olivier Henri Roux[2,3]

[1] CNRS
[2] École Centrale de Nantes
[3] IRCCyN UMR 6597 (Institut de Recherche en Communications et Cybernétique de Nantes), Nantes, FRANCE
Kabland-Toussaint.Tigori@irccyn.ec-nantes.fr, Jean-Luc.Bechennec@irccyn.ec-nantes.fr, olivier-h.roux@irccyn.ec-nantes.fr

Résumé

Le développement d'un système d'exploitation spécialisé pour une application consiste, à partir du code d'un système d'exploitation complet, à l'élaguer pour l'adapter aux fonctionnalités nécessaires à l'application. L'opération d'élagage est réalisée à la main au cas par cas. Cela rend très difficile la vérification et la certification d'un système d'exploitation car non seulement la phase de certification est à réaliser pour chaque spécialisation mais aussi le problème difficile et bien connu du fossé entre le modèle utilisé pour la vérification et le code est à étudier au cas par cas. Dans cet article, nous proposons une méthode pour l'automatisation complète de cet élagage au moyen d'une synthèse du système d'exploitation à partir d'un modèle formel embarquant le code du système d'exploitation. Ce modèle comporte en effet à la fois le flot de contrôle, les variables du système d'exploitation et les séquences d'instructions manipulant ces variables. L'élagage, formalisé par l'élimination des chemins infaisables du modèle, permet de ne conserver que la partie accessible (et donc utile) du modèle et donc du code. Cette méthode, non seulement accélère considérablement l'adaptation du système d'exploitation mais, de plus, garantie l'optimalité du code généré et l'absence de code mort. La vérification et la certification de toutes les fonctionnalités souhaitées du système d'exploitation comme l'absence de blocage, peuvent être également réalisées sur le modèle avant la génération de code. Nous utilisons un modèle du système d'exploitation temps réel Trampoline compatible OSEK et AUTOSAR, qui est reconnu comme fiable et largement utilisé aux niveaux académique et industriel.

1 Introduction

1.1 Contexte

Les systèmes embarqués sont de plus en plus utilisés dans de nombreux domaines. On les retrouve aussi bien dans les appareils électroménagers que dans l'industrie ou l'aéronautique et l'automobile. Dans le domaine automobile, les nouveaux véhicules intègrent environs 270 fonctions déployées dans 70 unités de commande électronique (ECU) [8] et ce nombre ne cessera de croître dans l'avenir. Pour limiter les coûts matériels, chaque unité de commande électronique dans lequel le logiciel embarqué doit être intégré dispose d'une puissance de calcul et d'une capacité mémoire très restreinte.

Le nombre croissant de fonctions à embarquer sur les micro-contrôleurs entraîne une plus grande complexité et rend nécessaire l'utilisation de systèmes d'exploitation temps réel. En effet, les systèmes d'exploitation temps réel facilitent l'intégration des fonctions tout en organisant l'accès des tâches aux ressources matérielles et en contrôlant l'exécution des tâches selon une politique d'ordonnancement fonctionnellement et temporellement déterministe. Ils facilitent également la portabilité des applications et leur réutilisation sur une gamme de véhicules.

Cependant, sur les petites cibles embarquées où la mémoire disponible est très limitée comme dans l'exemple des unités de commande électronique vu dans la section précédente, l'utilisation des systèmes d'exploitation engendre un coût additionnel en terme de mémoire. De plus, étant donné que d'une application à une autre les besoins diffèrent, une application ne requiert pas tous les services qu'offre un système d'exploitation temps réel. Ces systèmes doivent donc être développés de sorte à être hautement configurables. Cela permet à la fois de satisfaire les contraintes de mémoire auxquelles les petits systèmes embarqués font face, d'optimiser le code source du système d'exploitation correspondant et de garantir la stabilité du système. En effet, un système d'exploitation qui est parfaitement spécialisé en fonction des besoins de son application ne devrait contenir aucun code inutile (code mort). La présence de code mort au sein d'un système d'exploitation constitue un vecteur privilégié d'attaques mais aussi engendre une empreinte mémoire plus importante. En effet, en cas de configuration imparfaite, le code mort peut être exécuté et conduire le système dans un état incorrect.

Généralement, dans les systèmes embarqués temps réel critiques, les applications sont statiques, c'est-à-dire qu'à la compilation les exigences de l'application en termes de services, le nombre de tâches, les ressources, ... sont connus et fixés et aucun objet n'est créé durant l'exécution. Par conséquent, il est possible de spécialiser de façon statique le système d'exploitation en fonction des exigences des applications. Les systèmes automobiles font partie de ce type de système [1].

1.2 Travaux connexes

Le besoin de spécialiser les systèmes d'exploitation temps réel en fonction des applications n'est pas nouveau. Plusieurs travaux de recherches [7, 12, 14] ont montré que les systèmes d'exploitation spécialisés sont avantageux en terme de performances et d'espace mémoire utilisé. Par conséquent, plusieurs méthodes de configuration d'un système d'exploitation ont été proposées. Dans [15] la programmation orientée aspect est utilisée pour le développement d'un système d'exploitation configurable. Cette approche consiste à encapsuler chaque fonctionnalité du système d'exploitation dans un *aspect*. Plus précisément, chaque *aspect* implémente une partie du système d'exploitation et constitue donc une fonctionnalité qui peut être incluse ou non dans le système d'exploitation en fonction de son utilité. Par contre, l'article ne mentionne aucun moyen pour la vérification du système d'exploitation configuré. Dans [2] une technique basée sur les *DSL* (*Domain Specific Language*) est utilisée pour la configuration de systèmes d'exploitation temps réel, mais l'auteur se limite à la configuration de l'ordonnanceur, néanmoins l'ordonnanceur configuré peut être vérifié [5]. Dans [6], une bibliothèque de composants logiciels appelée DREAMS est présentée. Ces composants peuvent être configurés selon les besoins de l'application par un configurateur appelé TEReCS. Le système d'exploitation est spécifié par trois graphes. Le premier, appelé *graphe de ressource*, décrit les composants du matériel (processeur). Le second, appelé *graphe de communication des processus*, décrit la communication des tâches de l'application. Enfin, un troisième graphe décrit l'ensemble des composants logiciels disponibles. Ces trois graphes sont composés afin d'obtenir un graphe qui décrit le système d'exploitation incluant les services requis par l'application. Le système d'exploitation est ensuite engendré à partir de cette composition.

1.3 Contributions

Nous proposons une méthode basée sur une approche formelle pour la synthèse automatique d'un système d'exploitation temps réel spécialisé pour une application. À partir d'un système d'exploitation existant, un modèle formel et exécutable a été 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 ou états inaccessibles. Ces chemins représentent en réalité les portions de code du système d'exploitation qui ne seront jamais exécutées pour cette application, c'est à dire le code mort. Tous ces chemins sont par la suite supprimés du modèle de façon automatique et cela conduit à un modèle spécialisé qui ne contient que les chemins faisables et donc que le code indispensable à l'exécution de l'application.

Cette méthode permet également la vérification du modèle spécialisé avant d'engendrer le code source correspondant.

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.

La suite de l'article est organisée comme suit: dans la section 2, nous définissons le formalisme utilisé pour établir nos modèles; la section 3 présente la modélisation d'un système d'exploitation temps réel compatible OSEK/VDX [11] et AUTOSAR puis le processus de spécialisation. La section 4 présente les outils implémentés pour la spécialisation du système d'exploitation temps réel. La section 5 présente une étude de cas et la dernière section conclut cet article et présente nos travaux futurs.

2 Définitions

2.1 Les automates finis

Généralement, les systèmes d'exploitation sont écrits dans un langage de programmation impératif tel que le C, C++, Ada, etc, où chaque fonctionnalité est décrite par une fonction contenant un ensemble d'instruction s'exécutant de façon séquentielle. Cependant, chaque fonction peut être formellement décrite par un automate fini $G = (S, \Sigma, \rightarrow, S_0)$ où S est un ensemble fini d'états, Σ est un ensemble non vide d'actions réalisables appelées alphabet, $\rightarrow \subseteq S \times \Sigma \times S$ représente la relation de transition et S_0 l'état initial. La relation de transition est écrite $p \xrightarrow{\sigma} q$ 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.

2.2 Les automates finis étendus avec des variables

Un automate fini étendu est une extension d'un simple automate fini avec un ensemble de variables. Dans ce cas, les transitions sont complétées avec des conditions et des actions sur ces variables. Les conditions sont aussi appelées *gardes* et les actions sont caractérisées par 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 action de mise à jour peut être effectuée sur les variables et des fonctions mettant à jour des variables peuvent également être appelées.

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] :

$$\gamma ::= \lambda \sim 0 \mid \gamma \wedge \gamma$$

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 1. (Automate fini étendu).

Un automate fini étendu F est un tuple $(S, s_0, X, \Sigma, \pi_0, \longrightarrow)$, où: 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. Nous considérerons les entiers relatifs de valeur absolue bornée ; Σ est un ensemble fini d'évènements ; $\pi_0 \in \mathcal{U}(X)$ est l'ensemble des affectations initiales sur X ; et $\longrightarrow \subseteq S \times \mathcal{G}(X) \times \Sigma \times \mathcal{U}(X) \times S$ est un ensemble fini de transitions. $t = \langle s, g, \sigma, \pi, s' \rangle \in \longrightarrow$ représente une transition allant de s à s' avec la garde g, l'évènement 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 évaluation 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 évaluation $v \in \mathbb{Z}$, nous notons $\varphi(v) \in \{\mathbf{true}, \mathbf{false}\}$, la vrai 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 évaluations v qui satisfont φ).
Pour $x \in X$, nous notons $v[x := k]$ la nouvelle évaluation v' tel 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 évaluation obtenue à partir de v en appliquant l'ensemble des mises à jour de π.

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

Définition 2. (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 $S_H = (Q, q_0, \rightarrow)$ avec $Q = S \times (\mathbb{Z})^X$, $q_0 = (s_0, \overline{0}[\pi_0])$ est l'état initial et $\rightarrow$ est la transition définie pour tout $\sigma \in \Sigma$ par $(s, v) \xrightarrow{\sigma} (s', v')$ avec $s \xrightarrow{g,\sigma,\pi} s'$, $g(v) = \mathbf{true}$, $v' = v[\pi]$.

Produit d'automates finis étendus. Il est pratique de décrire un système complexe comme un ensemble de sous systèmes interagissant entre eux. Cela facilite en effet la modélisation d'un tel système. 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. Soit $F_1, ..., F_n$ n automates finis étendus tel 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. Un état de $(F_1 \mid ... \mid F_n)_f$ est $(s_1, ..., s_n) \in S_1 \times ... \times S_n$.

1. Dans les figures, nous utiliserons == plus tôt que = pour être proche de la syntaxe des gardes exprimées selon l'outil UPPAAL.

Définition 3. (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 $(\Sigma \cup \{\bullet\})^n \to \Sigma$.

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:

— $S = S_1 \times ... \times S_n$, $\subseteq$

— $s_0 = (s_{10}, s_{20}, \cdots, s_{n0})$,

— $\pi_0 = \bigcup_{i=1 \to n} \pi_{i0}$

— $(s_1, s_2, \cdots, s_n) \xrightarrow{g,\sigma,\pi} (s'_1, s'_2, \cdots, s'_n)$ 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 $s_i = s'_i$, $g_i = \mathbf{true}$, $\pi_i = \emptyset$,

    — 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. $\forall \xrightarrow{g,\sigma,\pi} \in \longrightarrow$, $\pi \in \mathcal{U}(X)$ (cad $\forall x \in X$ si $(x := k) \in \pi$ alors $\forall k' \neq k$, $(x := k') \notin \pi$). Les automates que nous manipulons dans cet article respectent cette hypothèse quelque 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 cet article, 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 de l'article, nous omettrons la fonction de synchronisation et utiliserons la notation $\sigma?\sigma!$.

Définition 4. (Accessibilité)
Soit un automate fini qui a pour état initial s_0, un état s est dit accessible si et seulement si $s_0 \longrightarrow^* s$, autrement, s est inaccessible.

3 Modélisation et processus de spécialisation du système d'exploitation

OSEK/VDX[11] est un standard ouvert de l'industrie automobile. Il définit les API et le comportement de composants logiciels destinés à être déployés sur les calculateurs embarqués dans les véhicules. L'un de ces composants est un système d'exploitation temps réel. Le modèle que nous avons conçu est celui du système d'exploitation Trampoline [3] qui est une implémentation de la spécification OSEK/VDX OS et AUTOSAR OS. Cette spécification offre plusieurs services liés à la gestion des interruptions, la gestion des tâches basiques et étendues, la synchronisation des tâches étendues, la gestion des ressources pour réaliser les exclusions mutuelles, la gestion des alarmes qui permet de réaliser des traitements récurrents comme les tâches périodiques. Au total, 25 services sont disponibles.

3.1 Modélisation du système d'exploitation

Pour la conception de notre modèle, nous utilisons l'outil UPPAAL [4] qui permet la modélisation, la vérification et la validation des systèmes temps réel. Pour le moment nos modèles ne prennent pas en compte le temps car nous supposons que les fonctions du systèmes d'exploitation s'exécutent en temps nul. Par conséquent seules les propriétés fonctionnelles du système d'exploitation sont vérifiées. Le choix d'UPPAAL se justifie pour une prochaine extension où il s'agira de prendre en compte dans le modèle de l'application le temps d'exécution des tâches afin de pourvoir vérifier les propriétés temporelles de l'application ainsi que l'ordonnancement des tâches. Ainsi, pour cet article, les automates modélisant les services de l'OS ne permettent pas au temps de s'écouler (ce qui est facilement implémentable en UPPAAL avec l'attribut *urgent* équivalent à un invariant bloquant le temps).

Le code source de Trampoline est modélisé par un réseau d'automates finis étendus représentant le graphe de flot de contrôle et par un ensemble de variables de contrôle du système d'exploitation. L'outil UPPAAL permet d'associer à chaque transition un ensemble d'expressions impératives dont la syntaxe est proche de celle du langage C.

Toutes les fonctions du système d'exploitation sont modélisées par une combinaison d'automates finis étendus et d'expressions impératives dont la syntaxe est proche de celle du langage C. Par conséquent, l'ensemble des automates et des expressions impératives font partie intégrante du modèle complet.

Notre modélisation, est construite sur les 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 contrôles du système d'exploitation.
— Le code des expressions impératives associées aux actions (transitions) des automates, traduit fidèlement le code du système d'exploitation.
— Les actions et les conditions sur ces variables associées à chaque transition des automates sont les mêmes actions et les mêmes conditions que celles du programme du système d'exploitation.

Atomicité En UPPAAL, le code associé à une transition de l'automate est exécuté séquentiellement mais est considéré comme atomique. Ainsi, Supposons un automate à deux états e_1 et e_2 ; deux variables x et y valant initialement 0 ; une transition t entre l'état e_1 et l'état e_2 ; le code associé à la transition t est $x = 2$; $y = 3$. L'état pour lequel $x == 2$ et $y == 0$ n'est pas dans l'espace d'états du modèle.

Afin de garantir que le code du système d'exploitation respecte l'atomicité du modèle, nous imposons les règles suivantes :

— Le code associé à une transition correspond à du code non interruptible du système d'exploitation, ce qui garantie l'atomicité.
— Si deux transitions de l'automate sont synchrones (au sens du produit synchronisé) alors l'une de ces deux transitions n'a pas de code associé. Le code du système d'exploitation correspondant au franchissement simultanée de ces deux transitions est donc non interruptible et ainsi respecte l'atomicité.

Ainsi, les variables et le code du système d'exploitation sont embarqués dans le modèle formel. Modulo cette atomicité, les états du modèle et du système d'exploitation sont les mêmes. Un état du programme du système d'exploitation est atteignable si et seulement si il est atteignable au niveau du modèle et le modèle contient tous les chemins qui pourraient être traversés par le programme du système d'exploitation durant son exécution.

Notre dernière hypothèse est que toutes les variables sont entières et bornées ce qui implique

que l'espace d'états du modèle est fini. Nous obtenons ainsi le théorème suivant :

Théorème 3.1. *Notre modèle complet (OS+application) contient tous (et seulement tous) les chemins qui pourraient être traversés par le programme du système d'exploitation pendant son exécution pour cette application.*

Trampoline est principalement écrit en C et contient 174 fonctions pour un total de 4530 lignes de code. Le modèle de Trampoline est composé de 75 automates finis étendus qui décrivent tous les services de Trampoline et toutes les fonctions de son noyau. Dans le modèle, chaque fonction de Trampoline est décrite soit par un automate soit par une fonction. Une présentation exhaustive n'est pas envisageable faute de place. Dans cette partie nous présenterons quelques modèles de fonctions liés à la gestion des tâches.

3.1.1 Modèle du service d'activation d'une tâche

Lorsque l'application sollicite l'activation d'une tâche, elle fait appel à la fonction de l'API [2] `ActivateTask`. La figure 1(a) représente le modèle de cette fonction. Quand cette fonction est appelée, elle appelle à son tour la fonction `tpl_activate_task_service` (`tpl_activate_task_service!`). La variable `lock_kernel` est utilisée pour éviter des appels simultanés de service car cela n'est pas permis dans Trampoline. Cette variable est également utilisée dans Trampoline pour la même raison. Lorsque cette variable est à 0, cela signifie que le noyau est déverrouillé et qu'un appel de service peut être effectué. Si le noyau est verrouillé, elle passe à 1 et aucun autre service ne peut être appelé.

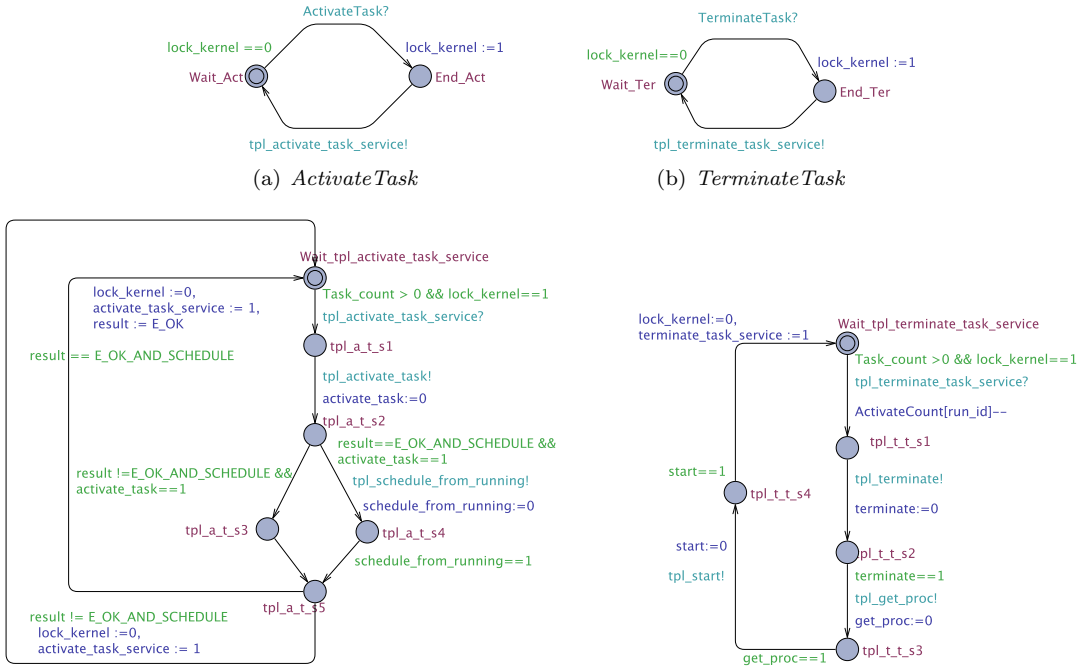
Quand la fonction `tpl_activate_task_service` est appelée, voir figure 1(c), le noyau est verrouillé puis la fonction de bas niveau `tpl_activate_task` est appelée. Pour décrire l'appel de fonction, une variable est associée à chaque automate du modèle. Cette variable permet la synchronisation de l'exécution de l'automate appelant et de l'automate appelé. Ainsi quand un automate appelle un autre automate, il met la variable associée à l'automate appelé à 0 puis reste bloqué (car la variable est exprimée comme étant une garde sur la transition suivante de l'automate appelant) jusqu'à ce que l'automate appelé termine son exécution. A la fin de l'exécution de l'automate appelé, la variable qui lui est associée est mise à 1. Ensuite, l'automate appelant reprend son exécution à partir de l'état où il était bloqué. Cette règle a été établie afin de permettre une exécution séquentielle des automates du modèle et de décrire fidèlement le comportement du système d'exploitation. Une fois appelée, la fonction `tpl_activate_task` active la tâche correspondante et l'insère dans la liste des tâches prêtes. Cette liste est modélisée par un tableau trié par ordre de priorité. Si la fonction retourne `E_OK_AND_SCHEDULE`, un ré-ordonnancement est requis. Dans le cas contraire, la tâche est simplement activée. Si un ré-ordonnancement est nécessaire, la fonction `tpl_schedule_from_running` est appelée, ainsi si la tâche en cours d'exécution a une priorité inférieure à celle qui est nouvellement activée, elle est préemptée. À la fin de l'exécution de `tpl_activate_task_service` le noyau se déverrouille (`lock_kernel := 0`) et un autre service peut être appelé.

3.1.2 Modèle du service de terminaison d'une tâche

Quand une tâche termine explicitement son exécution, elle fait appel à la fonction d'API `TerminateTask`, voir figure 1(b). Cette fonction appelle ensuite la fonction `tpl_terminate_task_service` (`tpl_terminate_task_service!`).

Lorsque la fonction `tpl_terminate_task_service` est appelée, voir figure 1(d) elle décrémente le compteur d'activation de la tâche correspondante (`ActivateCount[run_id]--`) et appelle la fonction

2. Application Programming Interface: ensemble de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.



(a) *ActivateTask* (b) *TerminateTask*

(c) *tpl_activate_task_service* (d) *tpl_terminate_task_service*

FIGURE 1 – API et service d'activation et de terminaison d'une tâche.

`tpl_terminate` qui, termine l'exécution de la tâche. La fonction `tpl_get_proc` est ensuite appelée pour extraire de la liste des tâches prêtes la tâche ayant la plus haute priorité. Cette tâche est alors ordonnancée grâce à l'appel de la fonction `tpl_start`.

3.1.3 Modèle de l'ordonnanceur

Trampoline utilise un ordonnanceur à priorités fixes pour le contrôle de l'exécution des tâches. L'ordonnanceur manipule la variable `run_id`, qui mémorise l'identifiant de la tâche en cours d'exécution, et la liste des tâches prêtes. Cette liste est une liste de tables FIFO indexées par niveau de priorité. Chaque table FIFO contient les tâches prêtes en fonction de leur ordre d'activation. Ainsi la tâche ayant l'ordre d'activation le plus ancien sera en tête de liste. Suivant cette structure, à chaque ré-ordonnancement, la tâche prête ayant la plus haute priorité est extraite de la liste des tâches prêtes puis exécutée par la processeur. Dans UPPAAL, nous utilisons un tableau trié par ordre de priorité pour la description de cette liste. Des fonctions impératives écrites dans le langage intégré à UPPAAL sont utilisées pour la gestion de cette liste à savoir `tpl_put_preempted_proc` qui insère une tâche préemptée dans la liste, `tpl_put_new_proc` qui insère une tâche nouvellement activée dans la liste et `tpl_get_proc` qui extrait la tâche prête de plus haute priorité de la liste [3].

3.2 Processus de spécialisation

La spécialisation d'un système d'exploitation en fonction d'une application consiste, à partir du système d'exploitation complet, en la suppression de tout le code inutile pour son exécution

3. Il existe toujours au moins une tâche active : la tâche *idle*

vis à vis de l'application: Les services qui ne sont jamais utilisés et le code qui n'est jamais exécuté au sein des services utiles sont supprimés. Cela revient à extraire du modèle, l'ensemble des chemins infaisables ou les états inaccessibles. Un modèle qui contient le strict nécessaire est donc obtenu. Trois étapes sont présentées ci-dessous pour la réalisation de cette spécialisation. Elle est également illustrée à la figure 4.

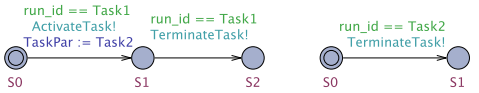
3.2.1 Modélisation

Dans cette première étape, l'application est modélisée et son modèle est synchronisé avec celui du système d'exploitation afin de former un système complet qui décrit le déploiement de l'application sur le système d'exploitation correspondant. À partir du modèle obtenu, il est possible de déterminer l'ensemble des services utilisés par l'application. Par exemple, considérons le modèle d'une application basique contenant deux tâches:

```
TASK(Tache1)
{
  ActivateTask(Tache2);
  Comp1();
  TerminateTask();
}
```

```
TASK(Tache2)
{
  Comp2();
  TerminateTask();
}
```

FIGURE 2 – Code source d'une application basique. La tâche 1 active la tâche 2, effectue un calcul et se termine. La tâche 2 effectue un calcul et se termine.



(a) Modèle de la tâche 1

(b) Modèle de la tâche 2

FIGURE 3 – Modèle de l'application

Le modèle, présenté à la figure 3, décrit comment les tâches de l'application réalisent les appels de services du système d'exploitation. Pour cet exemple, nous supposerons que la tâche 2 a une priorité plus élevée que la tâche 1. Au démarrage du système d'exploitation, la tâche 1 est activée et ordonnancée. Quand cette tâche débute son exécution, la variable `run_id` définie dans le modèle, contient l'identifiant de la tâche en exécution. La tâche 1 effectue alors un appel de service du système d'exploitation via la fonction d'API `ActivateTask` pour l'activation de la tâche 2. la variable `TaskPar` enregistre l'identifiant de la tâche à activer. La tâche 2 est ensuite insérée dans la liste des tâches prêtes et un ré-ordonnancement est effectué. Puisque la tâche 2 a une priorité plus élevée que celle de la tâche 1, elle la préempte pour commencer son exécution et la variable `run_id` enregistre son identifiant. Quand la tâche 2 s'exécute, elle effectue un calcul (`comp2`) et termine en effectuant un appel à la fonction d'API `TerminateTask`. Enfin, la tâche 1 reprend son exécution puis se termine.

3.2.2 Analyse d'accessibilité

La formalisation de la spécialisation du système d'exploitation constitue la seconde étape. À partir du modèle complet obtenu lors de la première étape, une recherche d'accessibilité est effectuée. Les états inaccessibles sont marqués et supprimés du modèle. Par conséquent, un modèle spécialisé ne contenant que les états accessibles et les chemins faisables est obtenu. Une vérification formelle peut être réalisée sur le modèle spécialisé avant la génération du code source correspondant.

3.2.3 Génération de code

La génération du code source du système d'exploitation spécialisé à partir de son modèle est la dernière étape. Pour chaque modèle de service du système d'exploitation spécialisé, le code source correspondant est engendré. Ainsi après cette étape, le code source du système d'exploitation spécialisé complet est engendré et peut être compilé.

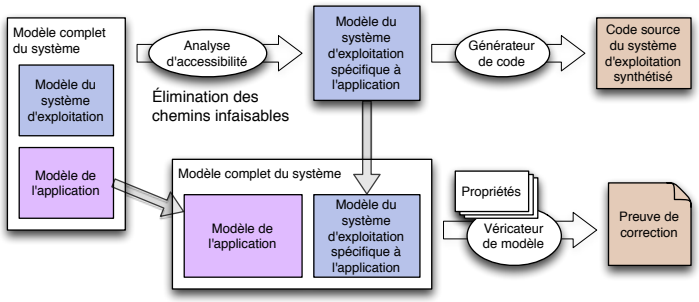


FIGURE 4 – Processus de spécialisation du système d'exploitation

4 Implémentation avec UPPAAL

Dans cette section, nous présentons l'outil développé pour la spécialisation du système d'exploitation ainsi que le processus de vérification mis en œuvre avec UPPAAL.

4.1 Outil de spécialisation et générateur de code

Deux outils ont été développés dans le langage Python pour réaliser la spécialisation du modèle et engendrer le code source du système d'exploitation. La figure 5 montre l'architecture de ces deux outils.

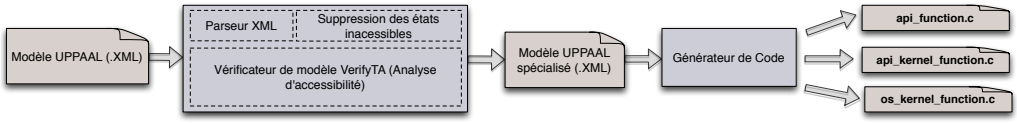


FIGURE 5 – Outillage de synthèse du modèle et de génération de code

Outil de synthèse du modèle

L'outil de synthèse utilise `VerifyTA`, le vérificateur de modèle d'UPPAAL pour réaliser la synthèse du modèle du système d'exploitation. Cet outil prend en entrée le modèle du système d'exploitation selon UPPAAL et génère en sa sortie le modèle synthétisé du système d'exploitation.

Cet outil comprend:

— Un parseur XML qui permet la lecture du modèle au format UPPAAL.

— Le vérificateur de modèle d'UPPAAL VerifyTA pour effectuer la recherche d'accessibilité de tous les états du modèle d'entrée. En effet VerifyTA prend en entrée un fichier qui est généré par un script et qui contient tous les tests d'accessibilité des états du système d'exploitation, puis génère en sa sortie le verdict de chaque test d'accessibilité.

— Un bloc de traitement qui récupère les résultats de l'analyse d'accessibilité du modèle et qui engendre le modèle de sortie qui ne contient que l'ensemble des états accessibles du modèle.

Générateur de code

Le générateur de code engendre le code source en C du système d'exploitation en fonction de son modèle UPPAAL.

Le générateur de code prend en entrée le modèle au format UPPAAL et génère des fichiers C. Le fichier `api_function.c` contient le code source des fonctions d'API, le fichier `kernel_api_function.c` contient le code source des services et le fichier `os_kernel_function.c` contient le code source des fonctions du noyau. Toutes ces fonctions sont celles qui sont nécessaires durant l'exécution de l'application.

4.2 Vérification et certification

Le consortium OSEK/VDX a mandaté la société MBTech GmbH pour prendre en charge le processus de certification OSEK/VDX. La certification consiste à exécuter une suite de tests, une série d'applications OSEK/VDX, chaque test se concluant par un succès ou un échec. Historiquement, cette suite de tests a été conçue dans le cadre du projet européen MODISTARC [13]. Concernant le système d'exploitation, un premier document [9] décrit 250 cas de test environ. Chaque cas de test correspond typiquement à un appel de service dans des circonstances précises et à sa conséquence. Par exemple, les cas numéros 2 et 34 sont les suivants :

Cas	Situation testée	résultat attendu
2	Call ActivateTask() from non-preemptive task on *suspended* basic task	No preemption of *running* task. Activated task becomes *ready*. Service returns E_OK
34	Call Schedule() from task.	*Ready* task with highest priority is executed. Service returns E_OK

Un second document [10] décrit la procédure de test. Il s'agit de 37 séquences de test qui enchaînent jusqu'à une vingtaine de cas de test.

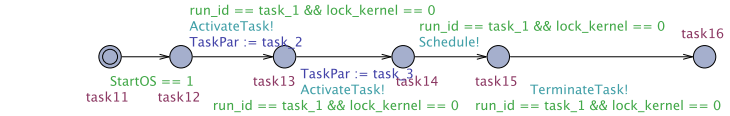
En se fondant sur cette spécification, il est possible de construire, d'une part, un modèle d'application pour chaque séquence de test, et d'autre part, un ou plusieurs observateurs permettant de vérifier les propriétés de chaque cas de test présent dans la séquence de test modélisée. Par exemple, la séquence de test 2 est une application de 3 tâches, t_1 (priorité 1), t_2 (priorité 2) et t_3 (priorité 3). La tâche t_1 est *ready* au démarrage du système d'exploitation et, en activant t_2 puis t_3, effectue 2 fois le cas de test numéro 2. Ensuite, en appelant `Schedule()`, elle effectue le cas de test numéro 34. Le résultat attendu est que t_2 et t_3 deviennent *ready* sans préempter t_1. Puis, lors de l'appel à `Schedule()`, t_3, qui a la plus forte priorité, s'exécute, puis t_2. Enfin, t_1 continue son exécution. La figure 6 présente le modèle de ces 3 tâches.

La séquence d'exécution attendue est modélisée grâce à un observateur présenté à la figure 7 et dont chaque transition est synchronisée avec les transitions des tâches. Si la séquence attendue est respectée, l'observateur atteint son état final.

La propriété suivante est alors vérifiée: *AF obs.success*. En d'autres termes ≪ Tous les chemins mènent fatalement à l'état *success* de l'observateur ≫.

Sur une version complète de notre OS (avec un modèle de l'application appelant tous les services) nous pouvons par cette approche appliquer l'ensemble des cas de test et remonter le processus de certification OSEK/VDX sur notre modèle.

Pour les versions spécialisées de notre OS, nous pouvons au cas par cas certifier l'ensemble des propriétés impliquées par les services utilisés pour cette spécialisation.



(a) t_1

(b) t_2 (c) t_3

FIGURE 6 – Modèle des tâches de la séquence de test 2.

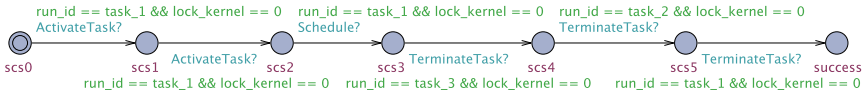


FIGURE 7 – Observateur du modèle.

5 Étude de cas

Dans cette partie, nous présentons la spécialisation de Trampoline à travers deux exemples d'applications.

5.1 Mise en pratique sur Trampoline

Application *Server*

Cette application utilise au total 7 fonctions d'API pour son exécution et contient 4 tâches définies comme suit:

— La tâche 1 est une tâche périodique ayant une priorité égale à 2. Elle est activée périodiquement par l'intermédiaire d'une alarme. Elle active ensuite les tâches 2, 3 et 4 et se met en attente des évènements ev1, ev2 et ev3. Quand l'un de ces évènements est reçu, la tâche 1 préempte la tâche émettrice, puisqu'elle est plus prioritaire, et active ensuite la tâche qui a envoyé l'évènement.

— Les tâches 2, 3 et 4 sont apériodiques et ont toutes des priorités égales à 1. Une fois activées, elles envoient respectivement l'évenement ev1, ev2 ou ev3 et terminent leurs exécutions.

Application *Periodic*

Cette application utilise au total 2 fonctions d'API pour son exécution et contient 2 tâches définies comme suit:

— La tâche 1 est une tâche périodique ayant une priorité égale à 1. Elle est activée par une alarme, effectue un calcul et termine son exécution.

— La tâche 2 a une priorité égale à 2 et est activée une seule fois par une alarme. Une fois activée, elle annule l'alarme responsable de l'activation de la tâche 1 et termine ensuite son exécution.

Une opération de spécialisation de Trampoline a été faite pour ces deux applications. Le code source généré a été compilé pour un microcontrôleur NXP LCP2294 avec un cœur ARM7TDMI de 32 bits. Le tableau 1 compare les tailles de code binaire pour les deux applications. Chaque valeur dans le tableau correspond à la taille du système d'exploitation compilé à partir de son modèle (colonne 2) et le code original de Trampoline compilé (colonne 3). Le jeu d'instructions de la cible est le jeu d'instructions de l'architecture ARM 32 bits.

TABLE 1 – Comparaison des tailles des binaires du système d'exploitation entre le code engendré à partir du modèle spécialisé et le code original de Trampoline

Application	Code de Trampoline spécialisé	Code original de Trampoline
Server	9.5 ko	14.8 ko
Periodic	7.6 ko	14.8 ko

A partir de ce tableau, nous remarquons qu'une optimisation très importante a été faite sur le code de Trampoline en fonction des deux applications. Pour l'application *server*, l'analyse d'accessibilité a pris au total 216,5 secondes tandis que le temps écoulé pour celui de l'application *periodic* n'a pris que 14,2 secondes. Cette différence se caractérise par l'ordre de grandeur des différentes applications car plus l'application est grande, plus l'espace d'états est grand et plus l'exploration de l'espace d'états prend du temps.

5.2 Résultat de la vérification

Quelques vérifications ont été effectuées, à titre d'exemple, avant la génération du code source afin de garantir certaines propriétés. La vérification a été établie pour les deux applications précédentes. Les propriétés ont été spécifiées en CTL (Computation Tree Logic). Rappelons que les automates UPPAAL modélisant les services de l'OS ne laissent pas le temps s'écouler afin de correspondre à un modèle non temporisé pour les propriétés AF et AG. Le tableau 2 montre le résultat obtenu suite à cette vérification.

TABLE 2 – résultats de la vérification

	Server	Periodic	Spécification des propriétés
P1	True	True	AG not deadlock
P2	False	False	EF terminate.tpl_4 and (State[run_id]!= SUSPENDED and State[run_id]!= READY_AND_NEW)
P3	True	True	(State[i] == READY_AND_NEW) $\longrightarrow$ run_id == i

La propriété 1 signifie qu'aucun blocage n'est présent dans le système. Cette propriété est vérifiée pour les deux applications. Par conséquent nous pouvons garantir que le système peut s'exécuter indéfiniment sans blocage.

La propriété 2 indique que lorsqu'une tâche termine son exécution en faisant appel à la fonction d'API `TerminateTask` elle passe dans un état autre que les états `SUSPENDED` et `READY_AND_NEW`. le résultat nous montre que cette propriété est fausse, ainsi nous pouvons conclure que lorsqu'une tâche termine son exécution, elle passe soit dans l'état `SUSPENDED`, soit dans l'état `READY_AND_NEW`. Ce dernier cas se produit lorsqu'une tâche est activée de nouveau alors que sont exécution actuelle n'est pas terminée.

La propriété 3 signifie que quand une tâche est prête (`READY`), elle est toujours exécutée par le processeur et nous remarquons que cette propriété est bien vérifiée pour les deux applications.

6 Conclusion

Dans cet article, nous avons proposé une méthode pour la génération automatique de systèmes d'exploitation spécialisés à partir d'une modélisation formelle par un produit d'automates finis étendus avec des variables. Cette approche consiste à établir à la fois un modèle formel du système d'exploitation embarquant son flot de contrôle, ses variables et les séquences d'instructions manipulant ces variables, et de l'application. Ces deux modèles sont composés

pour en former un plus complet qui décrit le déploiement de l'application sur le système d'exploitation. En effectuant une analyse d'accessibilité, tous les chemins infaisables et états inaccessibles du modèle (et donc du système d'exploitation) sont supprimés.

À partir de ce modèle spécialisé, le code en C correspondant est engendré au moyen d'un générateur de code. Pour la réalisation de notre approche, nous utilisons UPPAAL qui permet la modélisation, la vérification et la validation des systèmes temps réel. Cette approche permet la vérification, voire la certification, de la correction du système d'exploitation généré à partir de son modèle. Comme application, nous avons choisi le système d'exploitation Trampoline et des résultats encourageants ont été obtenus.

Les travaux futurs visent à étendre notre méthode sur le modèle de la version multi-cœur de Trampoline.

Références

[1] GbR AUTOSAR. Specification of operating system. *V3*, 1:R3, 2009.

[2] Luciano Porto Barreto, Gilles Muller, et al. Bossa: A dsl framework for application-specific scheduling policies. 2001.

[3] Jean-Luc Bechennec, Mikaël Briday, Sébastien Faucou, and Yvon Trinquet. Trampoline an open-source implementation of the osek/vdx rtos specification. 2006.

[4] 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.

[5] Jean-Paul Bodeveix, Mamoun Filali, Julia L Lawall, and Gilles Muller. Automatic verification of bossa scheduler properties. *Electronic Notes in Theoretical Computer Science*, 185:17–32, 2007.

[6] Carsten Boke, Marcelo Gotz, Tales Heimfarth, D El Kebbe, FJ Rammig, and S Rips. (re-) configurable real-time operating systems and their applications. In *Object-Oriented Real-Time Dependable Systems, 2003.(WORDS 2003). Proceedings of the Eighth International Workshop on*, pages 148–155. IEEE, 2003.

[7] 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 *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 251–258. IEEE, 2003.

[8] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, Feb 2007.

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

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

[11] OSEK Group et al. Osek/vdx operating system specification, 2009.

[12] 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.

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

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

[15] 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.