Détection d'erreur au plus tôt dans les systèmes temps-réel : une approche basée sur la vérification en ligne

Thomas Robert

THESE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Institut National Polytechnique de Toulouse*
Discipline ou spécialité : *Systèmes Informatiques Critiques*

Présentée et soutenue par *Thomas ROBERT*
Le *24 juin 2009*

Titre : *Détection d'erreur au plus tôt dans les systèmes temps-réel une approche basée sur la vérification en ligne*

JURY
Isabelle PUAUT, Rapporteur, Professeur, Université de Rennes 1
Rachid GUERRAOUI, Rapporteur, Professeur, Ecole Polytechnique Fédérale de Lauasanne
Denis CLARAZ, Examinateur, Ingénieur expert, Continental Automotive France
Sébastien FAUCOU, Examinateur, Maître de Conférence, Université de Nantes 1 (IRCCyN)
François VERNADAT, Président, Professeur, INSA Toulouse (LAAS;CNRS)
Jean-Charles FABRE, Directeur de Thèse, Professeur, INP Toulouse (LAAS;CNRS)
Matthieu ROY, Co-Directeur de Thèse, Chargé de Recherche CNRS (LAAS;CNRS)

Ecole doctorale : *Ecole Doctorale Systèmes*
Unité de recherche : *LAAS; CNRS*
Directeur(s) de Thèse : *Jean-Charles FABRE, Matthieu Roy*

Remerciements

Les travaux présentés dans ce mémoire ont été réalisés au laboratoire d'Analyse et d'Architecture des Systèmes au sein de l'équipe Tolérance aux fautes et Sûreté de Fonctionnement informatique. Pour la qualité de vie que j'ai pu apprécier durant ces quatre années, je remercie Messieurs Mallik Ghallab et Raja Chatila, les directeurs successifs du laboratoire durant cette période, pour m'avoir accueilli dans leur laboratoire. Pour des raisons similaires, je tiens à remercier Monsieur Jean Arlat et Madame Karama Kanoun, qui ont dirigé le groupe TSF, pour m'avoir permis de m'intégrer dans l'équipe et ses activités de recherche.

Vu le chemin parcouru et les obstacles surmontés, je suis particulièrement reconnaissant envers Messieurs Jean-Charles Fabre et Matthieu Roy, respectivement Professeur à l'Institut National Polytechnique de Toulouse et Chargé de Recherches CNRS au LAAS-CNRS. Ils ont tous les deux acceptés d'être mes directeurs thèse. J'ai énormément apprécié leur disponibilité, et leur soutien tout au long de mes travaux de recherche et de la phase de rédaction. Je tiens à les remercier tout particulièrement pour leur enthousiasme, leurs conseils et l'ambiance de travail qu'ils ont su créer durant ces quatre ans, encore une fois merci.

J'ai été honoré que Monsieur François Vernadat, Professeur à l'Institut National des Sciences Appliquées de Toulouse, ait accepté de présider mon jury de thèse. J'en profite pour remercier également :

- Isabelle Puaut, Professeur à l'Université de Rennes I,
- Rachid Guerraoui, Professeur à l'École Polytechnique Fédérale de Lausanne,
- Sébastien Faucou, Maître de Conférence à l'Université de Nantes I,
- Denis Claraz, Ingénieur à Continental Automotive,
- Jean-Charles Fabre, Professeur à l'Institut National Polytechnique de Toulouse,
- Matthieu Roy, Chargé de Recherche CNRS au LAAS-CNRS,

pour l'honneur qu'ils m'ont fait en acceptant de participer à mon jury de thèse.

Je suis particulièrement reconnaissant envers Madame Isabelle PUAUT et Monsieur Rachid Guerraoui pour avoir accepté la charge de rapporteur.

Comme je l'ai souligné au début, le groupe de recherche TSF est une grande famille, je tiens donc à remercier l'ensemble de ses membres pour l'ambiance qu'ils ont su instaurer. J'ai en particulier bien aimé participer au réseau d'excellence européen ReSIST qui m'a permis de tisser des liens assez fort avec de jeunes chercheurs européens. Je n'oublie pas non plus l'ensemble des personnes travaillant dans les différents services au LAAS : logistique, magasin, administration... Toutes ces personnes tendent à rendre la vie au LAAS plus facile et plus confortable.

Sans les citer tous, je suis très reconnaissant envers les doctorants qui ont partagés comme moi l'épreuve de force que représente une thèse. Ceux dont je parle se reconnaîtront. Pour finir, je tiens à remercier ma famille, et mes amis qui m'ont soutenu et parfois un peu secoué lorsque je me laissais aller à ma tendance à voir le verre toujours à moitié vide, merci à tous.

Table des matières

Introduction

De manière historique, la notion d'application temps-réel est associée au contrôle de systèmes physiques et à l'acquisition de données à partir de capteurs matériels. La prise en charge de l'acquisition de données et de leur traitement par des applications logicielles se généralise et devient presque une règle. Les domaines d'application de ce type de logiciel se sont élargis et approfondis. L'évolution du logiciel dans l'automobile en est un exemple criant. Les systèmes d'aide à la conduite, ou même de gestion de la direction électronique ont évolué. Historiquement leur réalisation se faisait à l'aide de puces dédiées. Ces fonctions sont désormais assurées par un logiciel qui pilote en temps-réel le système physique. Désormais, les voitures contiennent des calculateurs puissants combinés avec des systèmes d'exploitation permettant de déployer des applications logicielles relativement complexes. Le standard AUTOSAR illustre les efforts de rationalisation des architectures logicielles embarquées dans les véhicules pour permettre le déploiement d'applications de plus en plus complexes.

Ces logiciels ont souvent pour objectif de synchroniser l'évolution d'un système physique avec son environnement. Pour que le système puisse contrôler ou réagir aux modifications de son environnement, il doit être capable de respecter des contraintes temporelles variées : échéances, et définition de fréquences minimales ou maximales d'activations. Le service délivré par l'application se décrit en terme de comportement temps-réel. Or ce comportement doit être sûr de fonctionnement. Cela signifie que le logiciel doit satisfaire un certain nombre d'exigences liées à son comportement en opération : la précision et la correction de ses calculs, sa disponibilité, sa réactivité... Une grande partie des travaux réalisés en recherche dans les sciences et techniques de l'ingénierie du logiciel a été consacrée au développement de méthodes de conception, et de mécanismes destinés à assurer le bon fonctionnement du logiciel selon ces critères. Les systèmes pour lesquels les conséquences d'un dysfonctionnement sont inacceptables sont qualifiés de systèmes critiques.

Deux types d'approches complémentaires sont considérées pour concevoir ces systèmes et obtenir une confiance suffisante dans leur sûreté de fonctionnement. La première approche consiste à prévenir ou réduire la fréquence des causes de dysfonctionnement du système. Cela correspond à la réalisation d'un logiciel de qualité utilisé dans un environnement maîtrisé de telle sorte que le service puisse toujours être délivré. La seconde approche repose sur l'intégration au logiciel de mécanismes permettant de contrôler le système pour maîtriser les conséquences d'un dysfonctionnement du logiciel voire d'en prévenir l'occurrence dans la mesure du possible. Ainsi, on pourrait résumer les alternatives possibles, par d'un coté, la réalisation d'un logiciel zéro défaut dans lequel l'ensemble des causes susceptibles de déclencher un dysfonctionnement sont absentes, et d'un autre coté, par une conception permettant de tolérer les causes de dysfonctionnement en contrôlant l'exécution de l'application. L'adage "il vaut mieux prévenir que guérir" résume un principe élémentaire d'efficacité dans la production de logiciel. Il est préférable

dans la mesure du possible de prendre en compte autant de causes connues de dysfonctionnement que possible et de les prévenir au maximum. Cependant, il est extrêmement difficile, voire impossible de connaître et contrôler ces causes de manière exhaustive. En revanche, les manifestations de ces causes de défaillance peuvent être observées et analysées en ligne avant qu'elles ne déclenchent effectivement la défaillance du système. Dans ce cas précis, il est possible d'intercepter l'exécution de l'application avant que le système n'ait défailli pour tenter d'infléchir l'exécution de l'application pour en éviter la défaillance.

Une tendance générale dans le développement de logiciel passe par la définition et l'utilisation de modèles. Ces modèles servent à décrire le fonctionnement supposé ou espéré de l'implémentation de l'application. La vérification de modèle consiste à valider les propriétés comportementales d'un modèle de l'implémentation. Cette technique tombe dans la première catégorie d'approches pour la sûreté de fonctionnement du logiciel en permettant de raisonner sur l'implémentation de l'application dans ses phases amont de conception. Une évolution de ces méthodes de vérification consiste à réaliser certains contrôles en opération. Pour ce faire, un programme peut superviser l'exécution de l'application pour en extraire une information représentative de sa dynamique d'exécution. Cette information peut être utilisée après coup à des fins d'amélioration du logiciel ou d'évaluation de ses performances. Ces techniques ont un intérêt reconnu pour le test et la mise au point de l'application[1]. Cependant, ces moniteurs d'exécution nous intéressent pour ce qu'il peuvent apporter dans la détection en ligne d'erreurs et de dysfonctionnements dans l'exécution d'une l'application. En effet, il est possible de comparer en ligne le modèle de l'exécution courante, à la description des comportements attendus. Dès que le comportement observé se distingue des comportements attendus une erreur est signalée. Cette approche a déjà été éprouvée pour des modèles simples et faciles à traiter dans les années 1990. Le défaut de ces détecteurs résidait dans leur manque d'expressivité pour spécifier les comportements attendus. Des formalismes plus riches sont apparus à peu près à la même époque : logiques temporelles adaptées au temps-réel, systèmes à transitions intégrants des contraintes temporelles ... Ces formalismes ont atteint depuis un niveau suffisant de maturité pour qu'ils soient pris en considération comme outil de description des comportements attendus dans la vérification en ligne. Le domaine connaît depuis peu un regain d'intérêt avec l'apparition de cette nouvelle gamme de moniteurs comportementaux. Des algorithmes ont été proposés pour comparer en ligne une exécution à ces modèles. Plus le formalisme utilisé pour spécifier les comportements attendus est complexe, plus il devient difficile de comprendre l'algorithme d'analyse en ligne de l'exécution de l'application. La complexité des calculs réalisés dans les moniteurs utilisant les formalismes les plus puissants est telle qu'il existe des doutes sur les caractéristiques pra-

[1]La mise au point d'un logiciel consiste à rechercher et corriger des fautes de programmation ou de conception

tiques de ces moniteurs. Pour résumer, il devient raisonnable d'étudier la fiabilité de ces détecteurs. Pour cela, il sera nécessaire de répondre à trois questions clés :

- *A quoi correspond le signal d'erreur levé par ces moniteurs et comment peut-il être utilisé ?*
- *Quel est la latence entre l'occurrence de l'erreur et son signalement ?*
- *Étant donnée une description du comportement attendu de l'application, quel est le comportement du détecteur «parfait» en présence d'erreurs ?*

Des embryons de réflexion peuvent être trouvés dans la littérature décrivant les moniteurs. Cependant, ce type d'analyse n'a pas été menée à terme, à notre connaissance, dans le cadre des applications temps-réel. Ceci nous apparaît d'autant plus critique, si l'on envisage d'utiliser ces moniteurs pour superviser et contrôler l'application temps-réel.

Dans cette thèse, nous avons étudié les solutions existantes vis-à-vis d'une grille précise de critères permettant de déterminer les qualités et limites de ces détecteurs. Parmi ces critères, la latence de détection conditionne fortement l'efficacité de ces détecteurs. Il est donc crucial d'avoir une définition non ambigüe de cette latence. Nous avons centré notre travail sur la formalisation de la notion de détecteur au plus tôt et de sa concrétisation à travers la synthèse automatique de détecteur d'erreur pour des applications temps-réel. A travers ce terme, nous désignons un détecteur «glouton» signalant une erreur dès que l'on peut prouver à partir des observations que le système est condamné à subir une défaillance. Nous appellerons ce type de détecteur par la suite un détecteur au plus tôt. Un prototype a été proposé pour des applications temps-réel développées dans l'interface de programmation temps-réel Xenomai. Ce prototype permet d'illustrer l'intérêt du concept de détection au plus tôt et de sa formalisation. Ce travail souligne les correspondances entre certaines situations et concepts de la vérification à l'exécution avec ceux de la tolérance aux fautes. Ces éléments de correspondance ont servi de fondation à l'établissement d'un cadre clair pour comprendre comment utiliser ces moniteurs au sein de mécanismes de tolérance aux fautes pour des applications temps-réel critiques. Le manuscrit est organisé en quatre chapitres qui développent les points évoqués auparavant.

Le premier chapitre du manuscrit permet d'établir précisément le contexte dans lequel nous avons réalisé ces travaux. Ce chapitre présente les enjeux de la réalisation d'un détecteur d'erreur de comportement pour une application temps-réel critique. Dans un premier temps, nous rappellerons les enjeux et moyens classiques de conception et réalisation de systèmes temps réel critiques. Le rôle de zone de confinement d'erreur sera souligné. Par la suite, le rôle des vérifieurs en ligne sera clarifié vis-à-vis de la mise en place des ces zones de confinement. En particulier, nous nous intéresserons à leur capacité de détection d'erreur.

Le second chapitre correspond à une activité de formalisation de certains concepts clés dans l'analyse en ligne d'une exécution. Les traces d'événements temporisés serviront de pivot pour traduire la notion d'erreur et de symptôme d'erreur. Nous verrons en particulier

l'interaction forte qu'il y a entre la façon dont les observations sont délivrées et la façon dont les observations sont prises en compte pour vérifier le comportement de l'application. Ceci nous amènera à définir la notion de plus petit symptôme d'erreur comme étant la plus petite séquence d'observations permettant de décider de la présence d'une erreur dans l'état du système. Pour faire bref, la notion de détecteur causal d'erreur « au plus tôt » correspond à un vérifieur capable de signaler une erreur dès qu'un plus petit symptôme d'erreur est observé dans la trace d'exécution du système.

Le troisième chapitre s'attache à proposer un algorithme pour appliquer ces concepts à la détection d'erreurs via la vérification en ligne de comportements décrits à travers des automates temporisés de Rajeev Alur et David L. Dill [AD94]. Les analyses réalisées dans le chapitre 2, nous permettrons de guider la conception d'une méthode permettant de réaliser automatiquement un tel détecteur à partir d'un simple automate temporisé. La réalisation du détecteur repose sur un partage de la capture des plus petits symptômes d'erreur entre un mécanisme passif réactif et un mécanisme pro-actif. La coopération de ces deux phénomènes permet de capturer de manière exhaustive les plus petits symptômes d'erreur. Du point de vue technique, l'identification des plus petits symptômes d'erreur associés à un automate temporisé se fait à travers un modèle dérivé de l'automate couramment utilisé dans l'univers de la vérification de modèles.

Le quatrième chapitre s'attaque à la question du déploiement et de l'implémentation pratique de ce détecteur sur un support d'exécution concret. Nous avons proposé une architecture logicielle adaptée à la plate-forme de développement Xenomai. Cette section permet de finaliser la description du processus de synthèse automatique de détecteur de comportements erronés à partir de modèles formels. L'évaluation du coût d'intégration de ces détecteurs est réalisée pour permettre de mieux évaluer le rapport investissement/bénéfices d'un tel système de détection.

Le manuscrit s'achève par une synthèse des résultats obtenus et un ensemble de remarques sur la démarche suivie pour les obtenir, ses pièges et les ponts possibles avec d'autres domaines de recherche. Cette analyse permettra d'évoquer les pistes de recherche ouvertes par ces travaux.

Chapitre 1

Contexte et Problématique

Introduction

La conception d'applications temps-réel correspond à un cas particulier de système réactif. Nous verrons dans ce chapitre la terminologie et les concepts autour desquels s'articulent nos travaux. Cela comprend un panorama synthétique de la conception d'applications temps-réel, et des mécanismes et architectures classiques assurant la tolérance aux fautes. Ceci nous permettra de définir précisément le rôle de la vérification en ligne du point de vue de la sûreté de fonctionnement. Le domaine des vérifieurs en ligne étant très vaste, nous avons regroupé les approches existantes en fonction de la nature des vérifications réalisées en ligne. Cette réflexion servira de point de départ à la présentation des objectifs et enjeux des travaux présentés dans ce manuscrit.

1.1 Enjeux de la conception d'un système temps-réel critique

Lorsque l'on doit définir le comportement d'une application, il est important de comprendre la façon dont l'environnement et l'application interagissent. La nature des interactions entre ces deux entités et le rôle associé à chacune influencent énormément la façon dont elle est conçue et réalisée.

1.1.1 Etat et comportement d'une application

Un exemple typique d'application «temps-réel» consiste à capturer et traiter en temps-réel des mesures de divers capteurs afin de contrôler un système physique. La fonction de l'application est de «réagir» aux variations de l'environnement. La première étape de la description des interactions entre l'environnement et l'application revient à décider qui doit mener la danse : l'application ou l'environnement. Par exemple, si l'on considère un outil de calcul numérique haute précision permettant de calculer l'inverse de matrices de grande taille, alors la précision numérique du résultat est la qualité requise principale pour ce type d'application. L'utilisateur ou l'environnement de cette application déclenche l'exécution de l'application mais va attendre aussi longtemps que nécessaire le résultat du calcul. Ce type d'application sera désigné sous le terme d'application conversationnelle puisque l'utilisateur ou l'environnement attend volontairement que l'application délivre sa fonction. En revanche, si l'on considère l'exemple d'un service de signalisation des trains s'arrêtant aux différents quais d'une gare, ce système est dépendant de l'activité de son environnement. Un affichage qui ne prend pas en compte suffisamment vite l'arrivée et le départ des trains n'est pas satisfaisant du point de vue d'un voyageur.

Les systèmes réactifs voient leur activité guidée par leur environnement. Leur comportement se définit du coup sous la forme de règles ou équations qui associent un stimuli à une réaction instantanée, [Ber89]. Ceci constitue le modèle le plus simple d'un système réactif. D'un point de vue réaliste, le système peut avoir une dynamique relativement complexe entre un stimuli et la réaction désirée. Les applications temps-réel sont une classe particulière de systèmes réactifs pour lesquels la réaction à un stimuli doit être fournie dans une plage temporelle précise. Toute application temps-réel réaliste contient une partie «réactive» et une partie «conversationnelle». La description de la fonction de l'application peut alors se faire en décrivant son comportement. Ceci requiert la définition de l'état du système et de sa dynamique.

1.1.1.1 État et comportement d'une application temps-réel

L'état de l'application représente l'ensemble de l'information caractérisant l'application en un instant de son exécution. La variation de cette information au cours du temps constitue le comportement de l'application. Suivant la nature de l'état du système, il existe divers moyens pour décrire l'enchaînement des états. Dans le cas de l'état d'une application logicielle, la variation de l'état est discontinue par nature. Chaque fois que l'état de l'application varie la fonction représentant l'état du système en fonction du temps fait un «saut». Ainsi, l'état du système peut être représenté au cours du temps par une fonction constante par morceaux. De manière plus classique, la dynamique du système est vue comme une succession d'un certain nombre d'étapes où l'état du système est stable puis change de manière instantanée de valeur.

Chaque discontinuité représente un changement d'état et sera appelée transition. Un événement peut être vu comme la cause ou la manifestation d'une telle transition. Plus généralement, un événement est un moyen simple pour identifier un instant, un point de passage dans l'exécution de l'application. La notion d'événement permet de nommer et dater les transitions lors de l'exécution de l'application.

1.1.1.2 Interactions environnement/application

L'implémentation de l'application définit un ensemble d'états de départ et les actions à réaliser pour obtenir une séquence de transitions permettant d'obtenir la fonction désirée en collaboration et/ou en dépit de l'environnement. L'interaction entre l'environnement et l'application fait partie intégrante de la conception de l'application. En effet, l'environnement est régi par des lois qui permettent d'utiliser le comportement attendu de l'environnement pour s'assurer que l'application délivre correctement son service. Ainsi, l'interaction entre l'application et l'environnement est bidirectionnelle et se fait à travers une interface. Cette interface caractérise la fraction commune de leurs états respectifs.

Puisque les définitions que nous considérons sont centrées autour de l'application, cette fraction d'état partagée entre l'environnement et l'application sera appelée «état externe» de l'application. Les échanges de données entre l'environnement et l'application se font grâce aux valeurs prises par cet état externe au cours du temps.

La réalisation de l'implémentation repose sur des hypothèses concernant les comportements raisonnables de l'environnement (vitesse d'évolution, lois physiques, phénomènes vraisemblables...). Ces hypothèses permettent de définir la nature des transitions que l'environnement peut provoquer sur l'état externe de l'application. Ces changements d'états constituent le comportement normal de l'environnement. Par exemple, l'effet de la gravité ou des frottements de l'air sur la carlingue de l'avion constituent des influences normales de l'environnement sur l'avion. En revanche, l'impact d'un éclair sur sa carlingue ne fait pas partie des influences «normales» de l'environnement. Cette séparation ne signifie pas que la seconde interaction entre l'environnement et le système ne sera jamais étudiée mais qu'elle n'entre pas directement en ligne de compte pour assurer la fonction de l'avion. Cette notion d'interactions normales nous permet d'être plus précis lors de la définition d'un état correct d'une implémentation de l'application [ALRL04].

1.1.1.3 Etats corrects et application zéro défauts

La conception du système est correcte du point de vue fonctionnel, lorsque l'application peut toujours assurer sa fonction lorsqu'elle est exécutée dans un environnement «normal». Cela revient à définir un contexte d'utilisation idéalisé, et à s'assurer que dans ce contexte la fonction sera toujours satisfaite. Nous dirons alors que l'application est bien conçue (du point de vue fonctionnel). Tout le problème réside dans la sélection des interactions normales.

Supposons que l'on ait une application bien conçue. Lorsqu'une application bien conçue du point de vue fonctionnel est exécuté dans un environnement «normal», alors tous les états occupés par l'application durant une telle exécution sont dits corrects. La propriété principale d'un état correct est que l'implémentation de l'application peut poursuivre son exécution à partir de cet état et assurer sa fonction. Cette vision d'une application et de sa conception peut sembler utopique mais correspond à ce que l'on pourrait appeler le logiciel zéro-défaut. En effet cette vision suppose que tout a été prévu, et correctement pris en compte. Ce type de conception repose aussi sur une perception souvent tronquée de l'environnement où l'on ne considère qu'une portion de ses interactions possibles avec l'application. La conséquence directe est que si l'environnement peut «agresser» l'application en la plaçant dans un état qui ne soit pas correct, alors tout peut arriver et en particulier le pire. La vision zéro défaut est souvent utilisée dans un contexte où l'environnement d'utilisation de l'application est aménagé, «dompté», pour s'assurer qu'il se comportera tel que prévu lors de la conception de l'application.

1.1.2 Contraintes comportementales temps-réel

L'environnement d'une application temps-réel possède sa propre dynamique. Cela signifie qu'il peut évoluer indépendamment de l'application logicielle. L'exécution de l'application et l'évolution de l'environnement peuvent être vues comme deux activités réalisées en parallèle et en constante interaction. Un système temps-réel doit essentiellement s'assurer à travers des contraintes temporelles définies par les concepteurs qu'il est correctement synchronisé avec son environnement, [SR90], ni trop rapide, ni trop lent. Les contraintes temporelles de l'application sont dans la majorité des cas dérivées des propriétés physiques du système ainsi que de la fonction qu'il doit assurer.

Lorsque le comportement correct de l'application suppose que ces contraintes doivent absolument être respectées, le système est dit temps-réel «dur». Si le non-respect de ces contraintes n'entraîne pas une mise en défaut de la fonction du système mais seulement des ses performances, le système est dit temps-réel «mou». Dans l'ensemble de ce manuscrit, le terme «exigence» servira à décrire la définition d'un besoin lié à un service. La fonction de l'application détermine les exigences que l'application doit satisfaire à son interface avec son environnement. La spécification de l'application est un document censé faire la synthèse de ces exigences. Ceci nous permet de séparer la spécification, de la définition des comportements effectivement requis ou souhaités. Le terme «comportement correct» ou «exécution correcte» désignera une succession d'état corrects assurant la fonction de l'application.

1.1.3 Plate-forme d'exécution et prédictibilité

La prédictibilité est la capacité à prédire le comportement d'un système en opération. Le terme plate-forme d'exécution désigne l'architecture matérielle et logicielle sur laquelle est déployée l'application pour constituer un «système informatique» délivrant la fonction de l'application. Les exigences temporelles issues de l'analyse de la fonction de l'application sont incluses dans celles concernant le système complet. Le matériel utilisé pour exécuter une application temps-réel peut contenir plusieurs sites de calcul et d'acquisition des données. Dans ce cas l'architecture du support d'exécution est distribuée.

Chaque site est organisé en quatre couches supposées : un matériel, un système d'exploitation, un intergiciel (optionel), et pour finir l'application.

- Le matériel assure la connexion entre le monde physique et le logiciel via des périphériques d'acquisition de données, et des ports de contrôle. Il fournit aussi les ressources nécessaires à l'exécution des couches supérieures du système informatique : des unités de calcul, des horloges, des espace de stockage de données...
- Le système d'exploitation organise l'exécution des différentes activités de calcul hébergées par la plate-forme d'exécution. Cette couche est supposé assurer le partage

des ressources matérielles entre les différentes activités exécutées dans l'application.
- L'intergiciel offre une couche de services supplémentaires facilitant la programmation d'un type particulier d'application logicielle. Ces fonctionnalités sont réalisées à partir des services offerts par le système d'exploitation. Cette couche est ajoutée lorsque le développement de l'application repose sur des services de communication et d'exécution nettement plus évolués que ceux offerts par le système d'exploitation : communication de groupe, objets partagés sur le réseau, exécution tolérante aux fautes par crash ...

Pour maîtriser le comportement de l'application temps-réel, il est nécessaire de maîtriser celui des trois couches inférieures. La théorie de l'ordonnancement permet de structurer le comportement de l'application pour que le système d'exploitation puisse correctement attribuer les ressources matérielles au cours du temps. L'ordonnancement d'une application passe par la traduction d'un problème de contraintes temporelles sur un ensemble de tâches, en un problème de partage de ressources au cours du temps.

Les unités de calcul sont les premières ressources concernées par l'ordonnancement. Les deux facettes de cette théorie sont la définition des problèmes d'ordonnancement, et leur résolution. Une analyse précise des propriétés de chaque solution est nécessaire pour permettre d'établir la connexion entre des contraintes temporelles, et le partage d'un processeur. Les contraintes temps-réel d'une application induisent des contraintes temporelles sur le début et la fin des différentes tâches exécutées par l'application. Pour s'exécuter toute tâche va «consommer» une ressource de calcul. L'ordonnancement permet de décider quelles tâches sont exécutées et quelles tâches doivent ou peuvent attendre. Cette décision peut aboutir à un ordre d'exécution prédéterminé, ou à un algorithme en ligne qui détermine à la volée quelles tâches doivent être exécutées. Le second cas est nécessaire lorsque la nature des tâches à exécuter varie. L'étude réalisée dans [SAÅ+04] permet d'avoir une vision synthétique des problèmes et des solutions concernant l'attribution du processeur vis-à-vis d'un ensemble de tâches. L'algorithme d'ordonnancement, celui qui attribue à l'exécution le ou les processeurs aux différentes tâches, est intégré aux deux couches du logiciel dont l'application se sert pour s'exécuter : le système d'exploitation et l'intergiciel (lorsqu'il est disponible). Deux questions récurrentes se posent lors de l'ordonnancement d'un système, que ce dernier soit fait hors ou en-ligne.

- Le problème d'ordonnancement a-t-il une solution ? Cela revient à savoir si l'on a suffisamment de ressources de calcul pour satisfaire l'ensemble des contraintes associées à l'exécution des tâches. Lorsque ce n'est pas le cas on dit le système non ordonnançable.
- Quel est le comportement de l'application lorsque le problème d'ordonnancement ne peut être résolu ? Indépendamment de la cause de ce dysfonctionnement de l'application, il est important de comprendre comment l'application poursuit son exé-

cution.
La théorie de l'ordonnancement repose sur de nombreuses hypothèses faites sur le comportement du support d'exécution, le comportement de l'application, le comportement de l'environnement. Ce composant d'un système temps-réel est en quelques sorte le cœur et le point faible de l'application.

1.1.4 La sûreté de fonctionnement du logiciel

La notion de sûreté de fonctionnement a été développée pour traiter un large éventail de questions liées à l'évaluation et la maîtrise des conséquences des dysfonctionnements d'une application logicielle. Une terminologie précise a été proposée dans [ALRL04] pour décrire, analyser et résoudre ces questions. Cette sous-section synthétise le contenu de l'étude [ALRL04] de la terminologie et des concepts liés à la sûreté de fonctionnement.

1.1.4.1 Concepts élémentaires

L'analyse de la sûreté de fonctionnement possède plusieurs facettes appelées attributs. Chaque attribut caractérise l'un des aspects permettant d'obtenir la propriété globale de fonctionnement sûr. Parmi l'ensemble des facettes possibles, trois d'entre elles ont attiré notre attention : la disponibilité, la fiabilité, et la sécurité-innocuité.

- La disponibilité caractérise la capacité du système à pouvoir assurer sa fonction. Cela revient à évaluer le rapport entre le temps pendant lequel le service peut être délivré, et le temps pendant lequel le service ne peut absolument pas être délivré.
- La fiabilité caractérise la capacité à délivrer le service de manière continue lorsque le service est disponible. Cela correspond à évaluer l'intervalle de temps pendant lequel le service peut être correctement délivré.
- La sécurité innocuité caractérise l'absence de conséquences catastrophiques à l'issu d'un dysfonctionnement.

Pour raisonner sur ces aspects, trois termes ont été définis pour décrire la dynamique du système lorsqu'il satisfait ou non sa fonction.

La défaillance : Si le comportement de l'application interdit la satisfaction des exigences induites par sa fonction, le système est dit défaillant. La défaillance du système correspond à l'instant à partir duquel les exigences et le comportement de l'application rentrent en conflit. Si la spécification du système caractérise totalement sa fonction, le système est défaillant lorsqu'il ne respecte pas sa spécification. Dans tous les cas, la défaillance du système est un événement. La défaillance du système correspond donc à un changement de l'état externe de l'application induisant la violation d'une exigence liée à sa fonction.

L'erreur La notion d'erreur se définit par rapport à la notion d'état correct précédemment définie. Une erreur est une fraction ou la totalité d'un état de l'application qui n'est pas correct. Si cet état n'appartient pas aux états corrects, il est a priori impossible de dire si le comportement de l'application à partir de cet état remplira ou non sa fonction. Ainsi, la notion d'erreur est associée à un état pouvant causer la défaillance de l'application puisqu'a priori seuls les états corrects permettent d'affirmer que la fonction sera délivrée.

La faute Le terme de faute sert à désigner la cause de l'erreur. Une erreur peut être le résultat de plusieurs fautes. Ainsi, la ou les fautes à l'origine d'une erreur sont les éléments permettant d'expliquer la transition de l'application d'un état correct vers un état erroné. Un cas particulier doit être introduit : les fautes de conception amenant à une définition incorrecte des états corrects. Tout type de faute peut être ramenée à une faute de conception : cela revient à admettre que la démarche de réalisation de l'application a été réalisée de manière incorrecte. Cependant, cette vision n'a pas d'intérêt puisque l'intérêt d'identifier des fautes au delà du processus de conception réside dans la mise en place de méthodes permettant de contrôler leur conséquences en opération ou même de les rendre inoffensives.

1.1.4.2 Vision système de systèmes

Dans une application réaliste l'application est conçue comme un système de systèmes obtenu par l'assemblage de plusieurs composants délivrant des services élémentaires combinés pour obtenir la fonction de l'application. L'état de l'application se décompose entre ses différents composants et l'état des interactions entre ces composants (contenu des canaux de communication). Cette structure est potentiellement récursive et permet d'organiser l'application en services internes qui collaborent pour satisfaire la fonction globale de l'application. L'exécution de chaque composant peut dépendre du bon fonctionnement d'autres composants de l'application. Cette vision permet d'illustrer le fait que la défaillance d'un service interne peut déclencher une erreur dans un autre composant de l'application. Ce raisonnement permet d'expliquer comment une erreur d'un composant interne à l'application se transforme en défaillance de ce composant. Cette défaillance peut à son tour devenir une faute pour un autre composant de l'application et causer une erreur dans l'état de ce dernier composant. Ce phénomène est appelé la propagation de l'erreur dans l'application. Par abus de langage, il est souvent dit qu'un état contient une erreur lorsque l'état d'un composant de l'application est erroné. Dans ce cas précis, l'erreur ne touche qu'une portion de l'état de l'application, et est contenue dans l'état de l'application.

1.1.4.3 Moyens de la sûreté de fonctionnement

Quatre points peuvent être étudiés pour analyser et maîtriser les différents attributs de la sûreté de fonctionnement.

- La prévention des fautes : cette activité a pour objectif la mise en place de moyens permettant d'empêcher l'occurrence des fautes.
- L'élimination des fautes : cette activité a pour objectif de diminuer la fréquence et la sévérité des conséquences des fautes.
- La tolérance aux fautes : cette activité a pour objectif d'éviter l'occurrence des défaillances, en présence des fautes et des erreurs associées.
- La prévision des fautes : cette activité a pour but de prévoir la nature, la fréquence et la sévérité des conséquences potentiellement indésirables des fautes.

Pour concevoir une application sûre de fonctionnement, il est fortement recommandé d'utiliser une combinaison de ces quatre moyens pour obtenir un niveau de confiance raisonnable dans l'application. Dans ce paysage de la tolérance aux fautes a un rôle à part puisqu'elle modifie le déroulement de l'exécution de l'application.

1.1.5 Applications temps-réel tolérantes aux fautes

Le contrôle et la supervision de grandes infrastructures de production, ou de systèmes automatisés de grande taille est par nature temps-réel et critique. Nous allons tout d'abord définir les objectifs précis et les moyens de la tolérance aux fautes. Nous verrons par la suite comment la tolérance aux fautes est mise en place pour des support d'exécution conçus pour en faciliter la mise en place, puis pour des supports d'exécution génériques. Ces deux points de vue nous permettrons de faire ressortir l'importance d'une famille de mécanismes de détection d'erreur : les vérifieurs en ligne.

1.1.5.1 Concepts clés de la tolérance aux fautes

La tolérance aux fautes repose sur une organisation de l'état de l'application et le contrôle de son exécution de telle sorte que l'application puisse délivrer sa fonction en dépit de l'occurrence de fautes. Cela revient à permettre au système de délivrer son service même lorsque ce dernier se trouve dans un état erroné (contraire de correct).

Pour atteindre cet objectif, deux activités sont mises en place : la détection d'erreur et le rétablissement du système dans un état correct

- La détection d'erreur est en charge de surveiller l'exécution de l'application et de signaler la présence d'une erreur dans son état. Le signalement de l'erreur peut servir de déclencheur au rétablissement du système dans un état correct.
- Le rétablissement du système dans un état correct correspond à un ensemble d'action permettant de mener l'application d'un état erroné à un état correct.

Ces deux activités peuvent être réalisées selon des philosophies relativement éloignées.

Stratégies de détection d'erreurs

La fonction de détection est la première étape du processus permettant de tolérer une faute. On peut distinguer deux stratégies[1] pour réaliser la détection

Détection «basée redondance» : la détection peut être réalisée à travers la création d'une situation où l'information utile de l'état est mémorisée de manière redondante. Une erreur peut être détectée lorsqu'elle brise cette corrélation entre les différentes parties de l'état du système. Dans ce cas la détection repose sur l'existence de redondances dans l'état du système et d'une fonction de comparaison.

Détection «basée test de vraisemblance» : la détection d'une erreur peut être réalisée sans directement utiliser de redondances dans l'état du système. Ce type de mécanismes de détection repose sur l'utilisation d'un oracle fournissant une réponse par oui ou non concernant la validité d'une exécution. Par exemple, vérifier que le résultat de l'opération racine carrée est bien positif est en soit un test de vraisemblance. Une erreur est détecté dès le moment où l'oracle est en désaccord avec les observations.

Stratégies de rétablissement du système dans un état correct

Les méthodes de rétablissement du système dans un état correct peuvent être classées selon trois classes qui elles aussi seront distinguée en fonction de leur utilisation ou non de redondances dans l'information contenue dans l'état du système.

Le «recouvrement arrière» désigne un ensemble de méthodes de rétablissement du système reposant sur des sauvegardes régulières de l'état du système. Dès qu'une erreur est détectée, le dernier état sauvegardé est rechargé pour permettre le rétablissement du système dans un état supposé correct.

Le «recouvrement avant» correspond à un ensemble de méthodes de rétablissement de l'état du système qui remplacent l'état erroné par un état déterminé à partir de la nature de l'erreur et non de son passé. Cet état est supposé exempt d'erreur. Ces méthodes sont aussi appelées «recouvrements par pousuite d'exécution» car l'état de substitution utilisé correspond souvent à un état permettant de dépasser l'étape de l'exécution où l'erreur a été détectée (pas de ré-éxécution).

La «compensation d'erreur» désigne un ensemble de méthodes de rétablissement de l'état du système qui reposent sur la redondance de l'information contenue dans l'état du système. L'erreur est censée ne pouvoir toucher qu'une fraction de l'état du système. La redondance de l'information doit permettre d'identifier la portion

[1] la distinction principale entre ces deux approches réside dans la nature des objets comparés

erronée de l'état de l'application, et de la corriger en la remplaçant grâce à la fraction d'information correcte contenue dans l'état.

Propagation et Zones de confinement des erreurs : Les modèles de fautes les plus fréquemment considérés sont les fautes du matériel ayant un impact direct sur le logiciel, ainsi que les fautes du logiciel déclenchant une défaillance du matériel ou des couches inférieures du support d'exécution. La conséquence directe de telles fautes en l'absence de mécanismes de tolérance aux fautes est la défaillance d'un site de calcul (incluant le matériel et le logiciel). Les mécanismes permettant de tolérer ces fautes et leurs conséquences reposent essentiellement sur des architectures distribuées permettant la mise en place de stratégies de réplication [ALRL04]. Ce modèle est pratique puisqu'il permet de traiter certaines conséquences extrêmes des fautes touchant directement l'application. Ces conséquences extrêmes correspondent au cas où une faute dans l'application génère une erreur qui se propage dans les couches inférieures du système. Bien que ces modèles soient assez polyvalents, il est tout de même important d'être capable de confiner les erreurs pour éviter qu'elles n'influencent le reste du système.

En décomposant l'application en sous-services, chacun muni de ses propres mécanismes de détection d'erreur, il est possible d'observer l'apparition des erreurs au sein des composants. Cette détection est le premier pas vers le confinement des erreurs au sein d'un composant. Ce confinement est à la base de la conception d'applications tolérantes aux fautes de taille raisonnable. La nature des mécanismes de tolérance aux fautes pouvant être déployés dépend du matériel, et des services offerts par le système d'exploitation. Cependant, la détection des erreurs et le confinement des erreurs peut être adapté aux moyen disponibles pour créer des composants «auto-testables» en quelque sorte.

1.1.5.2 Développement à partir de supports d'exécution tolérants aux fautes

Nous avons déjà souligné le fait que dans de nombreux cas, la mise en place de la tolérance aux fautes se traduit par une implémentation distribuée de l'application. Comme l'application est soumise à des contraintes temporelles, il faut donc adapter la politique d'ordonnancement à cette contrainte de distribution qui rajoute des ressources (unités de calcul) mais change l'ensemble de tâches à ordonnancer et rajoute des contraintes comportementales (synchronisation, délais de communication...). Cette transformation de l'application augmente nettement la complexité du problème d'ordonnancement.

La mise au point de ces systèmes repose sur l'utilisation d'un matériel et d'un système d'exploitation permettant de mettre en place les architectures logicielles de réplication. Cette méthode de conception est donc transverse à l'ensemble du système et touche chaque niveau. Cette démarche de conception se retrouve dans des cadres de développement (architectures, méthodologies ...) de systèmes distribués temps-réel tolérants aux

fautes : MARS et son évolution en TTA [KDK+88, KB03], G.U.A.R.D.S. [PABD+99], FT-RT MACH [EKM+99], Hades [ACCP98]...

Ces méthodes suivent toutes un fil conducteur assez classique. Premièrement, il faut déterminer, en fonction d'un besoin en terme de fautes tolérées, l'architecture logicielle de tolérance aux fautes. Ce choix impose l'architecture matérielle permettant son exécution. Pour finir, l'ordonnancement de l'application tolérante aux fautes doit être réalisée sur l'architecture physique. La quantité d'algorithmes disponibles pour ordonnancer les stratégies de tolérance aux fautes est relativement importante, [SAÅ+04]. Un point important dans la réalisation de ces systèmes tient à leur capacité à interrompre leur exécution le plus tôt possible en présence d'une erreur.

1.1.5.3 Développement à partir de support génériques

Le développement d'application temps-réel est en train de connaître une véritable révolution à travers l'émergence de «systèmes génériques» utilisés par un public de plus en plus large pour développer des applications temps-réel. Malgré une organisation en niches du domaine avec une multitude de systèmes dédiés à des classes de matériels précis, certains éléments architecturaux sont partagés par un grand nombre de systèmes d'exploitation temps-réel.

Le noyau constitue la partie active du système d'exploitation en charge de gérer l'activité de ce dernier. L'architecture à base de micro-noyaux a convaincu par son efficacité, [Lie95]. Un micro-noyau est une couche logicielle offrant un ensemble réduit de services permettant d'organiser l'activité des tâches du système. Le propre d'un mirconoyau est de répondre aux besoins d'une niche applicative. En offrant moins de services, il est plus facile de les vérifier et de les rendre prédictibles tant au niveau fonctionnel que temporel. Un micro noyau est dit temps-réel générique lorsque les services du système sont spécifiquement dédiés à la programmation d'applications temps-réel sans a priori sur la sémantique de l'application, ou la cible matérielle visée. Plusieurs micro noyaux temps-réels «génériques» ont vu le jour. Parmi les plus connus, on trouve VxWorks et RT-Linux [Win08], LynxOS series [Lyn08], ChorusOS [Mic], Xenomai et RTAI [Xen, dIA05]. Ces noyaux proposent tous une des politiques d'ordonnancement les plus classiques : l'ordonnancement à priorités fixes. Les protocoles de communications disponibles sont des plus rudimentaires et n'ont pas a priori de capacités spéciales de tolérance aux fautes. Malgré l'écart qu'il existe entre ces plateformes d'exécution et celles intégrant «nativement» la tolérance aux fautes, il est possible de réaliser des mécanismes permettant au moins de limiter la propagation des erreurs dans de telles application. Puisqu'a priori la distribution des traitements n'est pas disponible, il faut mettre en place des solutions alternatives pour assurer le confinement des erreurs.

L'objectif dans ce type de systèmes «génériques» est de confiner au maximum les

erreurs dans la couche applicative pour en empêcher la propagation au niveau du système d'exploitation, ou du matériel. Ce confinement peut être réalisé en rendant les différents éléments de l'application auto-testables. Dans le cadre d'une application temps-réel, cela revient à surveiller de manière continue le comportement de l'application. Dans le cadre de la sûreté de fonctionnement, les vérifieurs en ligne désignent usuellement les détecteurs d'erreur conçus à partir d'un test de vraisemblance.

1.2 Vérification à l'exécution

La vérification à l'exécution, *runtime verification*, désigne une technologie permettant de réaliser des détecteurs implémentant un test de vraisemblance. De manière usuelle de tels détecteurs sont réalisé de manière ad hoc. Le but est de trouver un compromis entre le coût de la comparaison entre l'exécution et l'oracle, et l'efficacité du système de détection. Plus récemment, la communauté des méthodes formelles s'est ré-appropriée ce terme et lui a donné un sens plus précis, plus technique. De leur point de vue la vérification en ligne consiste à évaluer un problème de vérification classique grâce à un ensemble de données collectées en ligne, [HR02b]. Lorsque cette vérification est réalisée en ligne, elle peut servir à décider quand signaler une erreur. En pratique, ceci correspond simplement à la formalisation de la notion de test de vraisemblance. Nous nous sommes intéressés aux caractéristiques de ces systèmes en terme détection et de confinement d'erreur.

1.2.1 Principe de la vérification à l'exécution

Les détecteurs par test de vraisemblance ont fait relativement tôt partie intégrante du logiciel. L'intégration de manière ad hoc d'assertions exécutables a été proposée relativement tôt pour permettre l'amélioration de la qualité du logiciel, [Ros95]. À l'utilisation, l'insertion manuelle des assertions s'est avérée fastidieuse malgré la richesse des propriétés pouvant être vérifiées. Nous avons déjà indiqué que le comportement de l'application avait deux dimensions : une temporelle et une structurelle. Dans sa version la plus générale, la vérification à l'exécution vérifie ces deux aspects conjointement lorsque les propriétés instantanées à vérifier sur l'état de l'application varient au cours du temps. Cependant, spécifier correctement le comportement de l'application dans ces conditions est relativement difficile. Il est fréquent de distinguer sur chaque dimension les différents types d'éléments de description.

1.2.1.1 Raffinement des dimensions du comportement de l'application

Sur la dimension temporelle, les événements permettent d'identifier de manière abstraite différents types «d'instants». Il est fréquent de différencier les événements corres-

pondant à des dates, et les événements correspondant à des étapes de la logique d'exécution. Le premier type d'événement correspond par exemple aux échéances du système, et représente l'aspect «temps physique» de la dimension temporelle. Le second type d'événement peut représenter par exemple des appels de fonction, ou l'affectation d'une variable, et correspond à l'aspect logique de la dimension temporelle. La définition des étapes logiques de l'exécution dépend donc du modèle de programmation.

Une séparation similaire peut être définie sur l'état du système et sa structure (d'où le nom «dimension structurelle»). À partir du moment où le vérifieur peut posséder sa propre mémoire, ce dernier rajoute une variable à l'état du système. Dans certains cas, il est pratique de créer des variables d'états qui seront mises à jour à la volée et directement utilisées pour décider de la présence d'une erreur. Ces variables permettent de donner corps à une information inexistante ou difficilement manipulable dans l'état du système mais dont la connaissance permet une décision rapide sur la présence ou l'absence d'erreur dans l'état du système. Par exemple, pour une fonction récursive, si l'on maintient un compteur représentant la profondeur de la pile d'appels, il est possible de détecter un défaut de programmation pouvant causer un dépassement de pile à cause d'une fonction récursive mal programmée. Une telle variable est dit «abstraite» puisqu'elle ne faisait pas partie des variables d'état de l'application à l'origine. L'état de l'application est habituellement décomposé entre les variables et structures concrètes de l'application, celles qui sont observables par simple lecture de la mémoire, et les variables et structures abstraites qui doivent être déduites de l'état et/ou du comportement de l'application.

De manière historique, les premiers vérifieurs en ligne se sont attelés à la vérification de la combinaison entre la dimension temporelle «logique», et la structure «concrète» de l'état du système. Ceci a donné lieu à divers langages d'assertions exprimant des contraintes sur la valeur des variables du programme à différents points du code. Très vite la dimension temporelle physique a été intégrée. Les avancées les plus récentes concernent la prise en compte des quatre aspects du comportement, et la définition de contraintes liant deux à deux ces aspects du comportement de l'application : le temps physique et une variable abstraite, la progression de l'exécution et le temps, une variable abstraite et les variables concrètes ... La classification des objets d'intérêt dans le comportement de l'application proposée ici est cohérente avec les quatre classes de propriétés à vérifier proposée dans [HG05] et permet du coup de mieux comprendre l'intérêt des quatre classes de propriétés à vérifier définies dans leur papier.

1.2.1.2 De la vérification vers la détection

Un problème de vérification en ligne revient à déterminer en ligne si le comportement courant d'une application est conforme à une spécification comportementale. La vérification peut déboucher sur deux résultats : la confirmation de la présence du com-

portement à détecter dans l'exécution en cours, ou la confirmation de son absence. La détection d'erreur peut être définie par défaut ou présence d'un comportement à l'exécution. Le symptôme d'une erreur constitue le comportement (état, ou séquence d'état) permettant d'identifier la présence, ou supposer la présence d'une erreur dans l'état du système. Une spécification comportementale correspond à un ensemble de contraintes permettant d'identifier un ensemble d'états ou de séquences d'états.

Définition directe les spécifications comportementales peuvent servir à identifier les comportements redoutés. La présence d'un dénominateur nul dans une division est un exemple de définition directe d'une erreur. L'identification directe de l'erreur suppose que la nature de l'erreur est connue ou définie de manière arbitraire à l'avance. Il est alors possible de proposer un rétablissement de l'état du système spécifiquement adapté aux erreurs ainsi détectées, par recouvrement avant par exemple.

Définition indirecte Les spécifications comportementales peuvent servir à décrire les comportements corrects de l'application. Le signalement d'une erreur passe dans ce cas par l'absence de comportements corrects correspondant à l'exécution courante. Le comportement correct d'une application peut être partiellement caractérisé par le respect d'une échéance pour une des ses tâches internes. Le dépassement de l'échéance sans nécessairement entraîner la défaillance du système est la manifestation d'une erreur. Les invariants de boucle sont un autre exemple de définition indirecte de la manifestation d'une erreur. La nature exacte de l'erreur n'est pas nécessairement identifiable. Cependant, ce type de détecteur permet d'identifier des erreurs qui n'ont pas été «prévues» mais dont la manifestation peut être observée de manière indirecte.

En fin de compte, il existe une partition en trois de l'ensemble des exécutions du système (en occultant les processus de rétablissement de l'état du système) : les exécution déclarée correcte, les exécution déclarées incorrecte et le reste comme indiqué sur la figure 1.1. Une définition indirecte des symptômes d'erreur fait l'approximation suivante : *toute trace qui n'est pas déclarée correcte est par défaut erronée et redoutée*.

1.2.1.3 Éléments d'un problème de vérification en ligne

La maîtrise de la vérification en ligne est l'occasion de concevoir des tests de vraisemblance qui tirent parti au maximum de la connaissance de la sémantique et des contraintes d'implémentation de l'application. Pour pouvoir faciliter la conception de ce type de détecteur, il est nécessaire de permettre la définition des symptômes déclenchant le signalement d'une erreur. Les systèmes permettant ce type d'approche ont été recensés récemment dans [DGR04, CM05]. Ces papiers proviennent de communautés différentes mais s'accordent sur la nature et la structure de ces détecteurs. Ces vérifieurs exhibent de

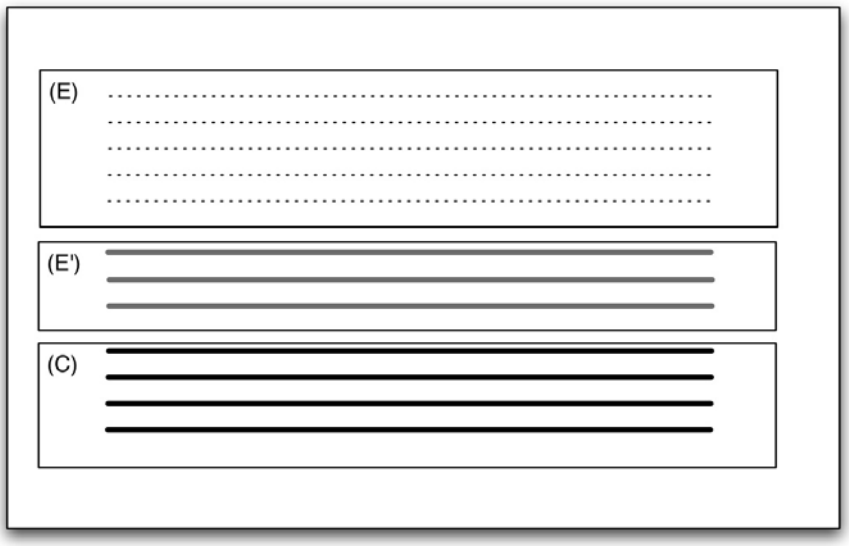

FIG. 1.1: Partition de l'ensemble des exécutions d'une application.
L'ensemble des exécutions peut se décomposer en trois ensembles : a) les exécutions explicitement identifiées comme correctes (C), b) les exécutions dans lesquelles une erreur est explicitement décrite par un symptômes d'erreur (E), et c) un ensemble d'exécutions pour lesquelles la présence d'une erreur n'est pas clairement définie (E').

manière récurrente une organisation, au moins conceptuelle, en trois activités qui se transforment fréquemment en trois composants séparés dans l'implémentation des vérifieurs (que nous appellerons blocs par la suite). Ces trois composants permettent de distinguer les rôles d'observation, d'analyse et de réaction du vérifieur en ligne. Ce type d'architecture permet de considérer que la réalisation de ces systèmes est relativement modulaire. Cette structure est désormais reconnue et adoptée par la communauté même si pour des raisons d'efficacité elle n'est pas physiquement implémentée . Chaque élément doit être configuré à partir d'une description de la nature des observations à réaliser, des comportements à détecter, et des actions à entreprendre en cas d'identification de l'un d'entre eux.

Pour résumer, un vérifieur en ligne est un programme qui confronte une abstraction de l'activité du système, capturée pas-à-pas par l'observateur, et la définition univoque d'un ensemble de comportements dont on souhaite détecter la présence. La vérification en ligne permet de confirmer la présence ou l'absence des comportement à détecter dans

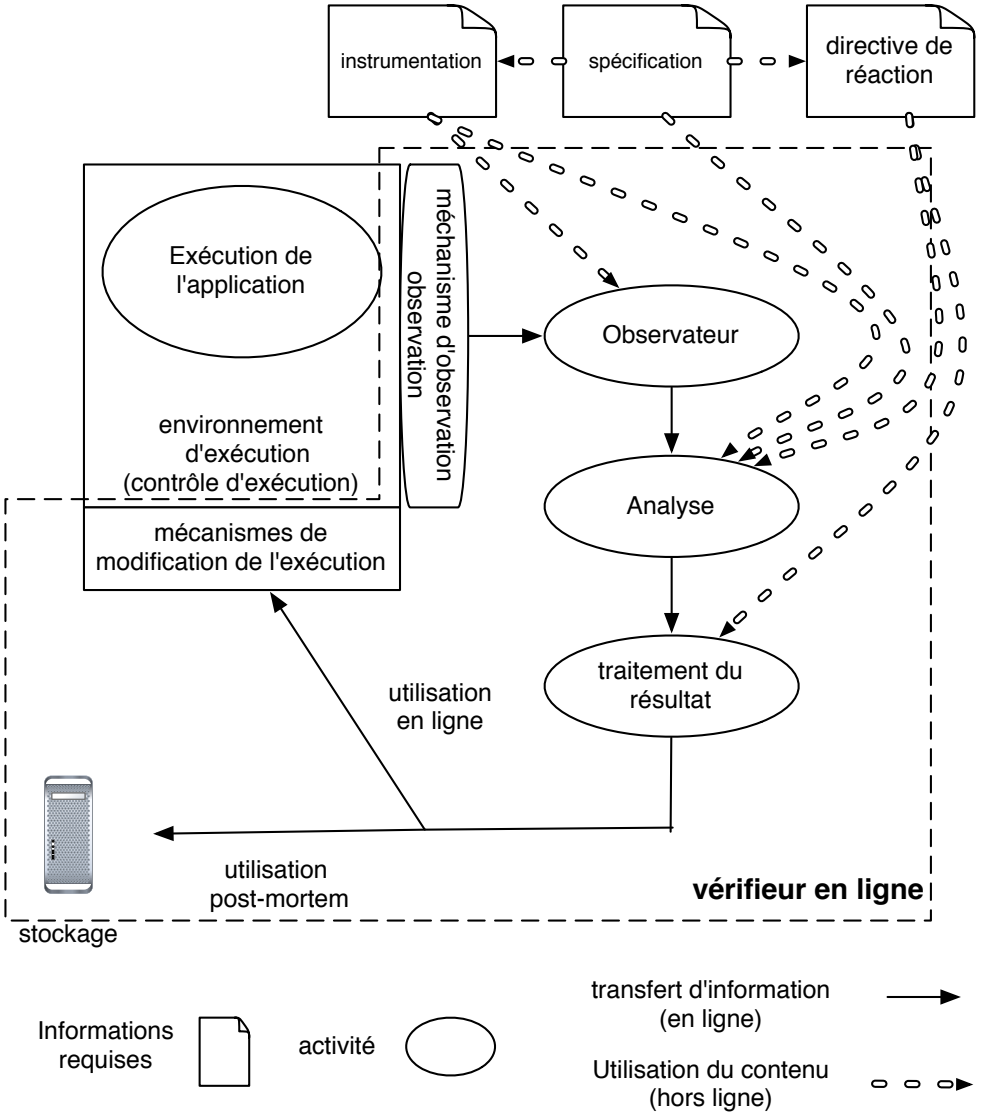

FIG. 1.2: Architecture abstraite d'un vérifieur en ligne

l'exécution observée. La détection d'erreur correspond à l'une de ces deux situations. Si les comportements à «vérifier» représentent des comportements redoutés, alors leur observation permet de justifier l'émission d'un signal d'erreur (cas de la détection des exécutions explicitement erronées (E)). En revanche, si les comportements à «vérifier» correspondent à la description des comportements normaux, alors vérifier que le compor-

tement courant de l'application n'est pas un comportement normal permet de motiver le signalement d'une erreur. Ce cas correspond à l'identification de l'ensemble des exécutions erronées «par défaut» (E'), mais aussi des exécutions explicitement erronées (E), cf figure 1.1. Notez que la vérification en ligne est un moyen et non un but en soi. Il est parfois pratique d'avoir ces deux visions de la détection d'erreur par vérification en ligne. Une définition exacte du comportement redouté permet d'établir une classification des erreurs et de proposer des mécanismes de recouvrement spécialisés pour un type particulier d'erreur. En pratique, le modèle abstrait d'une exécution possède nécessairement une sémantique par «pas». La définition des pas du modèle de l'exécution est fortement reliée à la définition des changements d'état élémentaires pouvant s'opérer sur le système. La représentation la plus simple est la séquence d'états et/ou de changements d'état de l'application. Dans ce contexte, une unité d'information peut être la détection par l'observateur d'un nouveau pas dans l'exécution de l'application. En pratique, un pas dans l'exécution est un événement (une date, une étape précise dans le code de l'application ou de l'exécution du système) permettant d'identifier la progression de l'exécution sur l'axe temporel. Le rôle de l'observateur est de surveiller l'exécution de l'application et de signaler les pas d'exécution significatifs pour la vérification en ligne dont le bloc d'analyse est en charge. Le modèle abstrait de l'exécution de l'application n'est que très rarement réel-

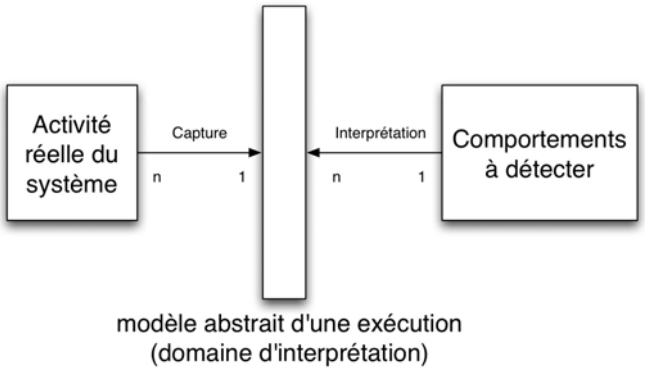

FIG. 1.3: Le principe d'interprétation comme médiateur à l'exécution

lement construit et manipulé à l'exécution. Sa définition permet essentiellement d'établir le lien en terme de sémantique entre l'activité réelle, et la spécification des comportements à détecter. La figure 1.3 représente le lien logique liant ces éléments. En comprenant la nature des «domaines d'interprétation» des formalismes comportementaux, il est possible de décrire leur logique d'exécution mais aussi de les comparer les uns, les autres.

1.2.2 Traces et signaux : les modèles d'exécution

Dans l'ensemble des systèmes considérés dans [DGR04], la dynamique de l'application se décompose en états discrets (constants), et/ou en événements. Les séquences temporisées sont des modèles classiques permettant de représenter une exécution, ses étapes et sa dynamique. Les séquences d'événements temporisés sont parfois appelées traces temporisées. Les séquences d'états temporisées sont appelées signaux. Ces deux modèles seront introduits, ainsi que les passerelles qui ont été jetées entre ces modèles, via la définition de processus de traduction de traces en signaux et inversement.

1.2.2.1 Les signaux

De manière classique, les signaux permettent une structuration forte de l'état du système. Le type du signal désigne la nature des valeurs que ce dernier peut prendre au cours du temps. En utilisant des types construits complexes, il devient possible de donner une structure à l'état. De manière classique, le signal est décomposé en un vecteur de variables élémentaires, chacune représentée par un signal.

À partir du moment où l'on considère un temps dense ayant la même structure que les nombres réels par exemple, un intervalle peut avoir 4 formes différentes : $[a,b]$, $]a,b]$, $]a,b[$, $[a,b[$. On suppose souvent que les intervalles sur lesquels l'état du système est constant ont tous la même forme. Le choix le plus fréquent consiste à supposer que la discontinuité lors du passage d'un état à l'autre favorise le nouvel état. Ainsi, un état sera constant sur un intervalle $[a,b[$. De manière usuelle, le symbole '.' est utilisé pour désigner l'opération de concaténation de séquences.

Définition 1 (Signal) *Un signal défini sur D est une séquence de termes "(σ^d)", où σ représente une «valeur du signal» prise dans D et d la durée de cet état.*

La figure 1.4 représente le signal $v^8.x^{7,4}.y^{14,6}.(z,\infty)$. Le symbole ∞ indique que le système reste définitivement dans l'état z. On notera que dans cette représentation un état peut être coupé en deux tel qu'indiqué sur la figure.

1.2.2.2 Les signaux avec des événements

De manière relativement naturelle, les événements peuvent être intégrés à ce modèle en les intercalant entre chaque état. En supposant que l'événement e_i indique la transition entre l'état σ_i et $sigma_{i+1}$, alors le signal $\sigma_1^{d_1}.\sigma_2^{d_2}.\cdots.\sigma_n^{d_n}$ deviendrait $\sigma_1^{d_1}.e_1.\sigma_2^{d_2}.e_2.\cdots.e_{n-1}.\sigma_n^{d_n}$. L'ensemble des événements est souvent désigné par le symbole Σ. Rappelons que le sens d'un événement était de permettre d'identifier un changement d'état de l'application.

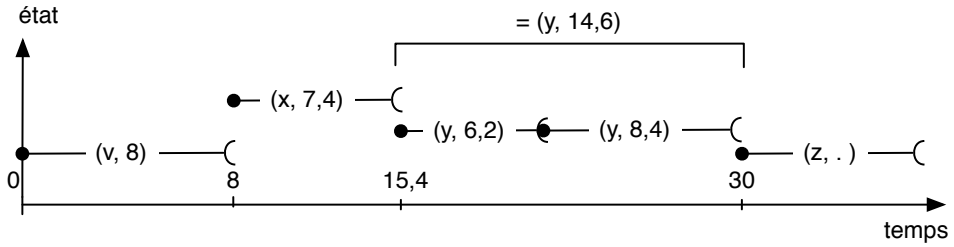

FIG. 1.4: Représentation d'un signal et décomposition d'un état
Le signal représenté ci-dessus correspond à la trace $v^{8}.x^{7,4}.y^{14,6}.(z,\infty)$. Notez le fait que l'état $y^{14,6}$ puisse se décomposer en $y^{6,2}$ suivi de $y^{8,4}$

1.2.2.3 Les traces d'événements temporisés

Dans ce cas, seule la durée des états importe entre deux événements, leur valeur n'a pas d'intérêt ou n'est pas connue. Une syntaxe plus appropriée revient à ne conserver que la durée dans les termes σ^{d} d'une trace. Si l'on considère une trace complète $\sigma_1^3.b.\sigma_2^{11}.a$ on obtient la trace $3.b.11.a$.

Historiquement ce modèle était remplacé par un modèle associant à chaque événement sa date absolue. D'un point de vue syntaxique, la seconde notation était moins pratique puisque la concaténation devenait une opération assez complexe. En revanche ce modèle était très adapté pour calculer la superposition de plusieurs activités. Ces deux modèles de traces sont décrits et discutés dans [ACM02].

1.2.3 Catégories de vérifieurs en-ligne

Notre décomposition des contributions du domaine est inspiré du travail réalisé dans [DGR04]. Cependant, au lieu de tenter de capturer d'un coup l'ensemble des caractéristiques d'un vérifieur en ligne, nous allons procéder de manière progressive, en insistant sur les concepts et en particulier sur la nature de la vérification réalisée. Nous allons distinguer trois familles de «syntaxes» utilisées pour décrire tout ou partie des spécifications comportementales servant de support à la définition de la tâche de vérification. Les trois catégories considérées sont respectivement les langages dédiés, les modèles exécutables, et les logiques temporelles. Il faut bien comprendre qu'ici, seule la spécification comportementale est concernée. La question de la capture et de la traduction de l'activité du système pour permettre sa vérification est traitée dans la section suivante.

1.2.3.1 Langages dédiés pour la vérification en-ligne

La notion de langage dédié permet de désigner un système de description adapté à la «programmation» d'un type précis de système ou à sa spécification. Ces langages dédiés peuvent être des raccourcis pour décrire une spécification selon les deux types de formalismes que nous verrons par la suite : les logiques ou les modèles exécutables. Ce phénomène de description est fréquent, par exemple le langage MEDL de Monitoring and Checking [KLS+02]. Seuls les premiers systèmes de vérification en ligne utilisaient des mots clés définissant de manière exacte la tâche de vérification à réaliser. [Ros95] propose une méthode pour coordonner efficacement l'utilisation de mots clés à insérer dans le code d'une application. Ces mots clés permettent de définir des tâches élémentaires de vérification à exécuter à chaque fois que la ligne en question est atteinte. Dans ce cas précis, l'observation, l'analyse et la réaction sont fusionnées en un seul élément dans lequel il est facile de relier ce qui est observé, la définition d'une erreur et la réaction associée.

L'avantage de cette approche repose sur une prise en main rapide du langage permettant de définir les problèmes de vérification de manière simple. Le défaut principal de ce type d'approche vient de la difficulté à traiter des symptômes d'erreur qui ne s'expriment pas simplement sous la forme d'invariants ou de comparaisons simples entre deux états successifs. Très vite, ces langages se sont adaptés pour intégrer des moyens de description permettant de décrire des comportements et non plus simplement des états isolés.

1.2.3.2 Vérification en-ligne de modèles exécutables

Un modèle exécutable est un formalisme qui définit explicitement une notion d'états pour le modèle, et des règles permettant de comprendre comment ce dernier peut ou doit évoluer. La vérification en ligne par un modèle exécutable possède différent niveaux de raffinement. Le modèle «observer-worker» proposé dans [DJC94], peut être vu comme un exemple emblématique de ce type de vérifieur. Le principe est le suivant : le modèle de l'application attend que la couche d'observation capture une variation de l'état de l'application. Cette variation est transformée en une variation de l'état du modèle exécutable. Une fois l'état du modèle mis à jour, un test est réalisé pour déterminer si cet état est valide ou non. De manière usuelle, la réalisation de ce test est améliorée en interdisant certaines transitions dans le modèle exécutable.

Plusieurs systèmes de vérification en ligne reposent sur la simple animation «réactive» d'un modèle exécutable en parallèle de l'exécution de l'application : [DJC94, GH01, HG05]. De manière classique ces modèles permettent essentiellement de définir des contraintes de sûreté en interdisant certains états du système.

Les mécanismes de détection d'erreur proposés dans [ML97, BLS06, BBKT05] se proposent de traiter des modèles intégrant des contraintes temporelles fortes qui se changent en échéances dont le dépassement constitue un comportement redouté. La notion de dé-

tection au plus tôt est introduite dans [ML97] comme la capacité à déclencher le signal d'erreur dès que possible. La définition de ce «dès que possible» a été proposé relativement au modèle manipulé, les chroniques [Gha96]. Ce formalisme est souvent utilisé pour décrire des phénomènes où l'ordre des événements importe moins que leurs distances temporelles relatives. Des modèles similaires existent pour représenter des exécutions «concurrentes» et reposent sur des extensions de réseaux de Petri temporisés. La construction de vérifieur à partir de tels systèmes n'a pas été explicitement traitée à notre connaissance. En revanche, la question de la reconstruction de la séquence d'état parcourue à partie d'une trace partiellement observable a attiré beaucoup d'attention dans la communauté du diagnostique, [BCTD98, Bou03]. Le problème principal des modèles repose sur la complexité de leur espace d'état. L'avantage de ce type de représentation réside dans l'opportunité d'étudier les séquences d'états amenant le système dans un état erroné, ce que l'on pourrait considérer comme les «chemins redoutés».

1.2.3.3 Vérification en-ligne de formules logiques temporelles

La vérification en ligne de formule de logique temporelle consiste à tenter de répondre à l'une des deux questions suivantes pour une formule ϕ.

- L'exécution courante permet-t-elle d'affirmer que ϕ est vrai ?
- L'exécution courante permet-t-elle d'affirmer que ϕ est fausse ?

Si l'état est structuré, la vérification en parallèle de plusieurs formules peut permettre de déterminer quelle fraction de l'état est corrompue par l'erreur lorsque cette dernière est détectée. Les systèmes concernés par ce type de définition d'objectifs de vérification sont : JavaPath Explorer [HR01], DBRover et Temporal Rover [Dru00], MaCS (Monitoring and Checking and Steering) [KLS+02]

La définition d'une formule de logique temporelle repose sur la combinaison d'expressions logiques «instantanée» et d'opérateurs permettant de définir à quels instants ces propriétés instantanées s'appliquent. Les logiques modales permettent de considérer qu'une variable peut avoir plusieurs valeurs possibles en fonction d'un contexte, appelé «univers». Plusieurs univers permettent de décrire une différence de point de vu ou une évolution de la valeurs d'une ou plusieurs variable. Les univers d'une logique temporelle représentent les différents instants de l'exécution. Les opérateurs modaux permettent de définir à quel univers, et donc à quel instant, s'applique une formule. Les valeurs des différentes variables contenues dans l'état de l'application et leur variation au cours du temps forment un signal, tel que définit précédemment. Il existe deux variantes de logiques modales : les logiques linéaires, [Pnu77], et les logiques arborescentes, [EH85]. En pratique, un problème de vérification en ligne revient à s'assurer que le signal représentant la séquence d'états du système satisfait une formule logique. Ceci fait que seules les logiques temporelles linéaires sont considérées pour définir des objectifs de vérification en ligne, à

notre connaissance.

La table 1.1 récapitule les opérateurs modaux les plus connus de la logique temporelle linéaire.

TAB. 1.1: Opérateurs modaux pour une logique temporelle linéaire – la fraction «futur» La table ci-dessous présente un ensemble d'opérateurs modaux permettant de définir des obligations sur le futur d'un état.

Always ϕ	tout état futur vérifie ϕ (invariant)
Eventually ϕ	il existe au moins un état futur satisfaisant ϕ (invariant)
Next ϕ	l'état suivant satisfait ϕ
ϕ *Until* ψ	l'état courant et ses successeurs vérifient ϕ jusqu'à ce que ψ le soit

Ces opérateurs permettent essentiellement de décrire l'enchaînement des caractéristiques logiques de l'exécution (invariant, pre et post conditions ...) La dimension temporelle «physique» doit être rajoutée. Les deux moyens les plus utilisés reposent sur l'annotation des opérateurs modaux par des intervalles, et la mémorisation de dates ou de variables. Une étude précise de ces différentes logiques peut être trouvée dans [BMN00, Hen98, AH91].

Ces formules sont très pratiques puisque leur sens est souvent défini par rapport à «l'instant courant». La définition des comportements à vérifier se fait donc par rapport à un événement qui doit signaler à partir de quand ces propriétés doivent être vérifiées.

Il est possible de distinguer deux méthodes d'évaluation pour réaliser la vérification de ces formules : les méthodes dites de réécriture, et la conversion de la formule en système état/transition.

Réécriture ou méthode des Tableaux Plusieurs algorithmes ont été proposés pour la vérification à la volée de formules de telles logiques sans forcément être réellement implémentés, [RFA00, KPA03, TR04, MN04, PM05]. La formule est vue comme un ensemble de contraintes à satisfaire. La vérification repose sur la décomposition de la formule entre les contraintes passées et présentes d'un coté, et celles concernant le futur de l'autre. Cet ensemble de contraintes est mise à jour, «réécrit», au fur et à mesure de l'exécution de l'application. La vérification se poursuit tant que la valeur de vérité de la formule n'est pas déterminée.

Par exemple, $Always(\ V \neq 0 \Rightarrow Eventually_{[0,10]} V = 0)$ signifie que V ne peut pas être différent de zéro plus de dix unités de temps d'affilées. Tant que V vaut 0 aucune contrainte

n'est définie pour l'état présent. La contrainte future reste identique : $Always\ V > 0 \Rightarrow Eventually_{[0,10]} V = 0$. En revanche, si V est strictement positif la formule à vérifier devient $Always\ (V > 0 \Rightarrow Eventually_{[0,10]} V = 0) \wedge (V > 0 \Rightarrow Eventually_{[0,10]} V = 0)$. Cela veut dire que si V ne devient pas nul dans moins de 10 unités de temps, les contraintes passées et présentes ne pourront pas être satisfaites : la formule sera prouvée fausse. L'évaluation est «glissante» : lorsque la formule concernant le futur de l'exécution courante devient trivialement une tautologie, ou une contradiction, la valeur de vérité de la formule à vérifier est déterminée.

Tout le problème consiste à déterminer cet instant tant qu'il reste des formules modales dans l'ensemble des obligations futures. Bien que la méthode pour décider si une formule de logique temporelle est une tautologie ou une contradiction est un problème déjà résolu pour des traces finies, sa mise en œuvre pratique est peu réaliste pour une utilisation en ligne. Les auteurs de [KPA03] pointent du doigt cette difficulté sans proposer de solution autre que de réaliser cette analyse à la volée.

Transformation en modèle exécutable La formule logique est transformée en un système à transitions, c'est à dire une machine à états finie ou l'une de ses extensions. Cette démarche a été suivie à divers degrés dans [Dru00, BLS06, DR05, MLWK02, Dru06]. Le vérifieur capture les changements d'état du signal à vérifier et les «exécute» dans le système à transitions censé traduire les différents états de l'évaluation de la formule au cours du temps. Les états permettant de conclure sur la satisfaction de la formule sont étiquetés de telle sorte que dès qu'ils sont atteints un événement spécial est émis pour signaler l'erreur.

Les avantages de cette approche sont l'optimisation du système à transition en fonction des divers critères de coûts : occupation mémoire, temps de vérification, [ML97]. La difficulté de cette approche réside souvent dans la traduction de la formule dans le système à transitions dont la taille devient parfois prohibitive. En effet, en enchaînant les modèles intermédiaires générés automatiquement, il est fréquent que la taille de ces modèles explose et ne puisse plus permettre leur utilisation.

Nous avons vu jusqu'ici les différents moyens disponibles pour décrire un problème de vérification en-ligne, et la nature des problèmes ainsi définis. Nous allons voir maintenant comment l'information est capturée à l'exécution, ainsi que les mécanismes permettant d'obtenir un modèle de signal à partir de traces d'événements.

1.3 Observation et analyse

La couche d'observation permet de traduire une portion de l'activité du système en des éléments primitifs d'une trace (événements) ou d'un signal (état) représentant l'abs-

traction de l'exécution de l'application. Du point de vue de l'observation l'élément primitif est l'événement puisqu'il fait soit partie de l'abstraction de l'exécution, soit permet d'identifier une variation d'une partie de l'état du système lors de la capture d'un signal, cf figure 1.5.

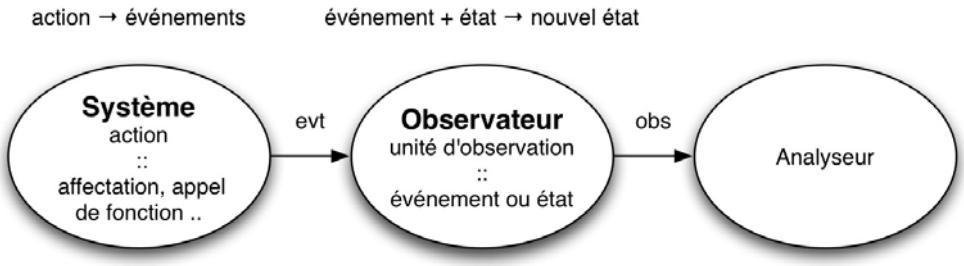

FIG. 1.5: De l'activité du système aux traces d'événements et aux signaux

À chaque événement capturé, la couche d'observation peut directement associer une information caractéristique d'une fraction de l'état du système lors de l'occurrence de l'événement.

1.3.1 De l'exécution d'un code vers les événements

Les événements capturés ne sont pas forcément tous internes à l'exécution de l'application mais peuvent aussi correspondre à des étapes de l'exécution des couches inférieures de l'architecture du système informatique : préemption, blocage sur primitive de synchronisation, appel d'une fonction de l'intergiciel. Des langages existent pour définir ces événements à partir d'éléments concrets de l'exécution de l'application.

Exemple 1: *Langage d'instrumentation PEDL,[KLS+02]*

Un langage, Primitive Event Description Langage (PEDL), a été proposé dans Java MaCS pour capturer des événements dans l'exécution d'un programme réalisé en Java. Par exemple, on peut capturer les appels et retours de méthodes à travers les expressions `StartM(<nom de la méthode>)` et `EndM(<nom de la méthode>)`. L'événement généré lors de l'appel d'une méthode est un triplet qui associe une date, un nom (i.e. `start<nom de la méthode>`), et les valeurs des paramètres. Un autre exemple caractéristique est la possibilité de capturer les accès à un champ, dans une instance d'objet, dans une classe.

Des langages dédiés permettent de décrire ces événements, [KLS+02] et sont associés à une méthode de capture. De manière usuelle, les événements sont extraits de l'exécution

de l'application. La programmation par aspects [FH02, MK03, OGRG07], voire même des techniques de reflexivité permettent de capturer ces événements de manière relativement efficace. Très rapidement deux autres catégories d'événements ont été traitées : les changements d'état du support d'exécution, et les changement d'état des variables abstraites artificiellement mises à jour par le vérifieur.

Capture avancée des événements internes La capture des événements «internes» à l'application passe essentiellement par une instrumentation de son code, ou par l'utilisation de fonctionnalités réflexives du support d'exécution. Pour permettre d'identifier des modifications de l'état du système à un niveau très bas, le code objet des programmes est parfois manipulé pour analyser les opérations arithmétiques qui modifient l'état d'une variable (exemple `i++` en C), [TJ98, Jef99, GRMD01]. L'utilisation de machines virtuelles permet d'avoir une instrumentation facilement déployable sur un grand nombre de plateformes avec une grande finesse d'observation. Sous Java, les vérifieurs JavaPath Explorer [HR01], et MaCS, [KVBA+99], reposent sur ce type de mécanismes d'instrumentation.

Capture d'événements externes à l'exécution de l'application La capture d'information sur le contexte d'exécution de l'application repose essentiellement sur une instrumentation du code du système d'exploitation, soit de manière native tel que c'est le cas dans les système OSEK via la notion de «hooks», [Ose05], ou par modification du système, tel que cela a été réalisé pour Chorus dans [Rod02]. Il est aussi possible de capturer des événements du support d'exécution à plus haut niveau en instrumentant les primitives de synchronisation, ainsi que certaines fonctions d'ordonnancement, [GH03]. L'instrumentation du noyau pose des problèmes de performance et faisabilité (disponibilité du code, licences ...) mais reste l'une des solutions les plus sûres, [Rod02].

Capture des variations des variables abstraites Le dernier niveau d'instrumentation est en fait extérieur à l'application et repose sur l'observation de la variation de variables créées de toutes pièces par le vérifieur. L'outil MaCS prend en compte ce type de variables à travers le langage MEDL. Un mécanisme interne permet de mettre à jour à la volée les variables abstraites pour permettre la vérification des comportements faisant référence à cette variable.

La malédiction des spécifications exhaustives d'implémentations De nombreuses contributions résistent difficilement du point de vue de leur auteur à la vérification de modèles trop près de l'implémentation de l'application. Ces termes sont utilisés lorsque de nombreuses variables de l'implémentation doivent être surveillées en permanence. Dans

ce cas précis, une fraction non négligeable de l'état du système est dupliquée dans le vérifieur (l'oracle). Il est normal que ce phénomène pose problème puisque l'oracle tend à devenir une réplique de l'implémentation du système, en moins efficace. De manière naturelle, les systèmes visant des spécifications particulièrement précises préconisent l'exécution de la vérification sur un support matériel indépendant.

1.3.2 Etat du bloc d'analyse et synchronisation avec l'application

Le bloc d'analyse confronte la mise à jour de l'état courant de l'application aux différentes tâches de vérification à exécuter. Pour chaque tâche de vérification, le bloc d'analyse maintient l'état du processus de vérification. Dans la majorité des systèmes,[2] cet état peut être résumé en trois cas de figure : présent, absent, en cours d'identification. C'est la valeur de cet état du processus de vérification qui permet de décider quelle erreur doit être signalée, et plus généralement des actions à entreprendre à partir de cet état. En effet, les systèmes récents de vérification en ligne intègrent un module de «réaction» tout à fait capable de déclencher ou prendre en charge le rétablissement du système dans un état correct.

L'étude réalisée dans [KS00] souligne l'importance de l'impact de la latence de détection d'une erreur sur l'efficacité des procédures de rétablissement du système dans le cadre des applications temps réel. Le coût de l'analyse en terme de temps d'exécution peut être prohibitif. Lorsqu'une spécification comportementale tente de contrôler une trop grande fraction des variables de l'implémentation, alors elle devient aussi complexe, voire même plus complexe, à vérifier que la simple exécution d'une réplique de l'application.

La vérification de ce type de spécification de manière synchrone à l'exécution de l'application devient de plus en plus difficile. On peut distinguer trois modes d'exécution du bloc d'analyse : synchrone, quasi-synchrone, et asynchrone. D'un point de vue temps réel, le cas asynchrone est inacceptable puisque cela veut dire que l'activité de l'application est peut être terminée lorsque le verdict de la vérification est présenté. Le cas asynchrone est ce que certains appellent une vérification «post-mortem».

Dans le cas synchrone, le système dont l'état est soumis à une vérification ne peut évoluer tant que la vérification n'est pas terminée. Ainsi, lorsque la vérification prend fin, le système est toujours dans l'état ayant permis de détecter l'erreur.

Dans le cas quasi-synchrone, le temps entre l'observation d'un changement d'état et la mise à jour du processus de vérification est borné. Ceci veut dire qu'au moment du signalement d'une erreur, l'état à partir duquel le vérifieur a pu prendre la décision de générer un signal d'erreur appartient au passé proche de l'état courant de l'application (l'application poursuit son exécution).

[2] À l'exception des versions reposant sur des analyses probabilistes complexes [MLWK02]

Modes de blocage pour le modèle synchrone On peut distinguer deux modes de blocage de l'activité de l'application.

- Un blocage qui maintient simplement l'ordre des événements capturés (méthode la plus utilisée, [GH03, HG05, HR02a, HR01, KLS+02]).
- Un blocage qui assure une pause complète de l'application (suspension des threads liés à l'application voire du système complet, [Rod02]).

Une synchronisation forte de l'observateur et de l'analyseur permet d'intercepter les erreurs. Dans le cas d'une synchronisation «quasi-synchrone», la latence de détection n'est pas réellement prise en compte pour décider quelles actions doivent être exécutées par le vérifieur. On peut remarquer que les travaux réalisés dans [EKS06], tentent de traiter cette problématique à travers la théorie de la synthèse de «contrôleur à délais bornés».

1.4 Position de l'étude

1.4.1 Attributs orientés «détection» d'un vérifieur en-ligne

Nous avons présenté jusqu'ici les mécanismes disponibles pour implémenter chaque bloc d'un système de vérification en ligne. La taxonomie établie dans [DGR04] vise à déterminer les conditions d'utilisation d'un très large éventail de vérifieurs. Nous souhaitons compléter cette analyse par la définition de trois exigences, ou attributs, permettant d'évaluer l'adéquation entre un problème de détection et un système de vérification en ligne. Ces trois qualités permettent d'évaluer les trois facettes du service que doit fournir un vérifieur en ligne.

Le pouvoir d'interception
: Le pouvoir d'interception du vérifieur définit sa capacité à contrôler l'exécution de l'application lorsqu'une erreur est signalée par rapport à un état présent ou passé de l'application.

Le pouvoir d'isolation
: Dans le cas d'un vérifieur pouvant supporter l'analyse d'un état structuré de l'application, le pouvoir d'isolation représente la précision avec laquelle le détecteur sépare la fraction de l'état corrompue du reste de l'état de l'application.

La prédictibilité
: Ce dernier attribut caractérise le comportement du vérifieur entre le début du processus de vérification et la production d'un verdict. Cette notion peut couvrir sa latence de détection, la qualité du processus de décision, l'impact de l'exécution du vérifieur sur le système. Ce dernier point couvre entre autre l'estimation du coût de la vérification, mais aussi la question de l'intrusivité du détecteur[3]

[3] L'intrusivité du détecteur regroupe l'ensemble des transformations comportementales d'une exécution

Un symptôme décrit la quantité d'information permettant de décider de la présence d'une erreur. Le point le plus critique concerne de notre point de vue la prédictibilité du service de détection et son impact sur l'application et son environnement. Cette caractéristique pouvant influer sur l'interception et l'isolation de l'erreur dans le comportement de l'application, il est de notre point de vue nécessaire de la traiter en premier.

De l'optimisation de la latence, vers la notion de symptôme optimal

Bien que l'état du processus de vérification finisse toujours par converger pour toutes les propriétés de sûreté, dans le cadre de la détection d'erreur pour des applications temps-réel, il est crucial de comprendre quand l'on est en droit d'attendre une réponse du vérifieur, et comment l'erreur signalée doit être interprétée.

La latence de détection est la somme de deux influences. D'une part, l'efficacité du processus de décision détermine le temps de traitement des données, ainsi que l'efficacité avec laquelle chaque donnée est prise en compte. Ceci revient à tenter de déterminer la «quantité» minimale d'information nécessaire pour signaler une erreur. La seconde influence provient de la synchronisation entre l'application, le bloc d'observation, et le bloc d'analyse de la vérification en ligne.

Perturbations introduites par le vérifieur

L'impact de la vérification en termes de temps processeur nécessaire pour la vérification doit pouvoir être borné soit en fonction d'une durée, soit en fonction de l'intensité de l'activité[4]. Dans la plupart des algorithmes, le premier aspect est souvent déduit du second en faisant l'hypothèse d'un nombre maximal d'événements dans un intervalle donné.

L'analyse de cette perturbation a été étudiée dans les travaux de thèse réalisés dans [Rod02]. Dans ces travaux, l'impact de l'exécution d'un vérifieur en ligne sur la politique d'ordonnancement des différentes tâches du système est entre autres analysée. L'approche expérimentale retenue dans ces travaux, l'injection de fautes par campagnes de sabotage, permet de comprendre l'efficacité de la détection fournie par ce type de systèmes en parallèle de l'impact réel de ce type de service sur l'exécution du système. La rigueur de la méthode d'expérimentation fournit de bonnes indications sur le potentiel de telles méthodes.

Nous avons souhaité dans ce travail de thèse rassembler des éléments théoriques complémentaires permettant de mieux comprendre les enjeux de l'utilisation de tels systèmes pour la vérification de comportements contraints du point de vue temporel .

---

valide de l'application en présence du détecteur. Le but est d'avoir une intrusivité nulle dans le cas idéal.

[4] ici il est possible ce terme par fréquence des événements

1.4.2 Synthèse de la démarche suivie

Nous avons vu dans ce chapitre la nature des exigences associées à la conception d'une application temps-réel.

Les motivations

La mise en place de la tolérance aux fautes peut prendre dans ce contexte diverses formes. L'utilisation de techniques de réplication de l'application pour assurer la tolérance aux fautes reste la référence dans le domaine pour obtenir un degré maîtrisé de fiabilité. Cependant, cette approche nécessite une connaissance très précise des caractéristiques du support d'exécution, et de l'environnement. L'un des freins reconnus à la mise en place de ces méthodes vient des mauvaises performances de tels systèmes lorsque qu'une des répliques a un comportement très éloigné de celui qui été prévu à l'origine (souvent désigné comme un comportement fortement non-déterministe par abus de langage). Plus le support d'exécution est sujet à des variations de la qualité des services qu'il délivre, plus il est difficile de maîtriser le comportement des répliques. Un fossé est apparu entre une majorité d'applications temps-réel développées sur des supports d'exécution ne permettant pas de mettre en place ces architectures, et l'ensemble des applications temps-réel ultra-critiques.

Un moyen d'améliorer le fonctionnement du logiciel pour ces deux classes de systèmes consiste à mettre en place des détecteur efficaces assurant un confinement efficace des erreurs. Ces applications auto-contrôlées permettraient du coup de contenir la propagation des erreurs en s'assurant que le comportement de l'application reste toujours dans certaines limites raisonnables.

Le point de départ

La réalisation d'une application auto-testable définit en réalité un problème de vérification à l'exécution. Deux approches permettent d'implémenter un composant logiciel auto-testable : l'utilisation d'implémentations diversifiées du logiciel et la compensation d'erreur, ou la définition d'oracles permettant de détecter les erreurs à travers des tests de vraisemblance. Il semble que la tendance générale ne soit plus à l'insertion ad-hoc d'assertions exécutables permettant de vérifier essentiellement des invariants relativement simples. Le domaine a évolué vers des systèmes permettant de générer les vérifieurs à partir de descriptions formelles des comportements à détecter.

La problématique : la détection au plus tôt

À partir du moment où la vérification en ligne devient un processus complexe avec de nombreuses fonctionnalités, il est raisonnable d'analyser l'efficacité de ces processus de vérification. Ici l'efficacité de la vérification est reliée à la latence de détection des erreurs, son coût mémoire, sa robustesse aux défauts d'observation... De notre point de vue, le premier critère à évaluer concerne la latence de détection. La notion de «détection au plus tôt» a déjà été évoquée périodiquement dans différents modèles sans qu'elle soit formalisée clairement. La définition de ce concept se faisait formalisme de description d'exécution par formalisme de description, [ML97] et de manière parallèle à nos travaux de thèse dans [BLS06]. Une définition univoque pour une large gamme de modèles présenterait l'intérêt de s'assurer que chacun possède la même interprétation de la détection au plus tôt. Nous avons aussi essayé de comprendre pourquoi un mécanisme d'évaluation d'une spécification comportementale n'atteindrait pas cette propriété de manière naturelle.

L'approche suivie et les contributions

L'approche suivie consiste à définir la détection au plus tôt sur le domaine d'interprétation permettant de comparer les formalismes de spécification présentés auparavant : les traces d'événements temporisés. Au travers des études faites pour définir cette notion, nous avons identifié différent types de détecteurs et différents types d'erreurs. Ces définitions ne sont pas constructives au sens où elles ne fournissent pas de solution pour les implémenter. Cependant, ces définitions et leur propriétés permettent de mieux comprendre de notre point de vue quelles propriétés doivent posséder les algorithmes de vérification en-ligne. Ce travail de caractérisation est rassemblé dans le chapitre 2, et correspond à l'identification de la séquence minimale d'observation permettant de signaler une erreur.

Nous allons proposer une architecture et un algorithme de haut niveau pour réaliser la détection au plus tôt d'erreur. Nous utiliserons des automates temporisés pour définir l'ensemble des exécutions correctes et illustrer le concept de «détection au plus tôt». La notion de détection au plus tôt sera traduite sur les états de l'automate, de même que le type de l'erreur associée à chaque état erroné de l'automate.

Dans une troisième phase, nous présenterons un prototype de ce type de détecteur pour des applications temps-réel développées sur l'interface de programmation temps-réel Xenomai. Les caractéristiques temporelles de ce prototype seront analysées en fonction de différent profils de modèles vérifiés. L'objectif de cette thèse est d'aboutir à une compréhension de bout en bout de ce type de système. Nous avons décidé de traiter l'une des caractéristiques clés de leur efficacité : la latence de détection.

Chapitre 2

Critère de Détection au plus tôt

Introduction

Nous avons vu dans le chapitre précédent les différents éléments constitutifs d'un logiciel de vérification à l'exécution ainsi que leurs caractéristiques pratiques. Lorsque le comportement à vérifier s'étale sur un intervalle, la détection possède une dimension temporelle. La latence de détection correspond au temps écoulé entre la date où l'erreur peut être détectée, et celle où elle est effectivement signalée. Cependant, cette durée peut se mesurer selon les deux échelles disponibles sur l'axe «temporel» : l'échelle de temps physique, et l'échelle mesurant la progression de l'application dans son exécution (temps logique).

Ce chapitre présente notre contribution sur le plan de la modélisation du processus de détection. Cette contribution est en deux parties. Premièrement, nous fournissons un cadre complet de description du processus de détection, suivie d'une analyse des différents phénomènes à l'origine de la latence de détection. Deuxièmement, nous définissons une instance particulière de détecteur d'erreurs comportementales permettant le signalement des erreurs au plus tôt. Sa définition comprend une partie descriptive des propriétés et des contraintes de ce type de détecteur. Ces définitions sont réalisées sur des ensembles de traces. Ceci nous permet de les transposer assez facilement à n'importe quel modèle permettant de définir un ensemble de traces. Le but de cette section est de poser des définitions qui ne dépendront pas de la «syntaxe» des modèles manipulés mais de leur sémantique. Nous souhaitons pouvoir modéliser l'impact de chaque bloc du vérifieur sur le déroulement du processus de détection. Ceci passe par l'étude des éléments clés utilisés tout au long du chemin logique menant, de la spécification du problème jusqu'à sa vérification.

2.1 Rappels sur les traces et leurs opérations

Tout d'abord, nous devons rappeler certaines opérations élémentaires sur les traces et les ensembles de traces. Nous avions vu dans le chapitre précédent que les traces temporisées pouvaient servir à représenter la séquence d'événements ou d'états observés lors de l'exécution d'une application temps-réel. Ces traces seront utilisées comme abstraction de l'exécution du système. Nous nous restreignons volontairement aux traces d'événements puisque nous avons déjà vu qu'un problème de vérification de signaux se transforme assez rapidement en un problème de vérification de traces d'événements grâce à un mécanisme semblable à celui décrit dans [DJC94]. Pour nous permettre de poursuivre notre étude de manière efficace nous rappelons certains concepts et éléments de notation utilisés tout au long de ce chapitre.

ensemble d'événements : la notation usuelle pour un ensemble fini d'événements est le

symbole Σ, indicé si nécessaire par un nom précisant la nature des événements.

ensemble des traces temporisées : l'ensemble des traces finies d'événements non temporisées pris dans Σ se note Σ^*. Nous utiliserons $T\Sigma^*$ pour désigner l'ensemble des traces d'événements temporisées finies.

durée d'une trace temporisée : une trace temporisée représente une période continue de l'exécution du système et possède une durée noté $\Delta(u)$ égale à la somme des durées contenues dans u.

concaténation de traces : Pour deux traces temporisées u et v, la trace $u.v$ représente l'enchaînement des comportements décrits dans u et v. La durée de $u.v$ est égale à la somme des durées respectives de u et v.

Opérations ensemblistes : Les opérations usuelles sur une paire d'ensembles sont l'intersection, l'union et le complémentaire. L'intersection et l'union, notées $X \cap Y, X \cup Y$, ne posent aucun problème particulier. L'opération $\overline{X}^Y$ désigne le complémentaire de l'ensemble X dans Y, $Y - Y \cap X$. L'ensemble Y est souvent omis lorsque sa nature est évidente.

Concaténation et répétition non bornée Soit L_1 et L_2 deux ensembles de traces, $L_1.L_2$ désigne l'ensemble des traces $u.v$ telles que $u \in L_1$, et $v \in L_2$. L_1^* désigne l'union pour tout entier k des langages $\underbrace{L_1.\cdots.L_1}_{k\ fois}$. Par extension, si u est une trace $u.L_1 \stackrel{\Delta}{=} \{u\}.L_1$ et $u^* = \{u\}^*$.

Ces éléments de notation vont nous permettre de définir dans l'ordre suivant : l'historique d'exécution, l'état du processus d'observation, et la caractérisation des symptômes d'erreur.

2.2 Historique d'une exécution et processus d'observation

Dans la majeure partie des systèmes décrits dans [DGR04], l'événement daté constitue l'unité d'observation (l'information élémentaire acquise). Cependant, la nature de ce qui est capturé et la façon dont cette information est transmise peuvent différer.

2.2.1 Approche de modélisation suivie

La modélisation du processus d'observation doit permettre de clairement identifier la nature de l'information échangée entre le bloc d'observation et le bloc d'analyse. Nous

supposons que les blocs d'analyse et d'observation ont un comportement purement séquentiel. A priori le bloc d'analyse est un bloc réactif pur. Cela signifie que ce bloc d'analyse ne s'active que lorsqu'il reçoit un stimuli du bloc d'observation. Le bloc d'observation indique chaque fois que nécessaire ses changements d'état. Ce changement d'état est analysé pour déterminer si une erreur peut être signalée. Il est donc crucial de comprendre comment l'observation fonctionne et influence le processus de vérification. Nous allons modéliser conjointement le déroulement d'une exécution en cours et le processus d'observation à travers la mise en parallèle à chaque instant de l'exécution de deux traces. L'une représentera l'exécution de l'application jusqu'à un instant donné, l'autre servira à modéliser ce qui est perçu par le processus d'observation au même instant. La trace représentant l'exécution sera appelée l'historique de l'exécution, tandis que l'autre trace sera considérée comme l'état du processus d'observation.

2.2.2 Historique d'une exécution en cours

L'historique d'une exécution en cours est une trace qui représente l'activité du système tout au long de l'exécution en cours. Il est nécessaire dans un premier temps de définir l'ensemble des événements concernés par la définition de l'historique de l'exécution.

Définition 2 (Support d'une trace) *Le support d'une trace est l'ensemble des événements caractéristiques de l'exécution de l'application*

La définition du support d'une exécution nous permet en réalité de savoir quels événements doivent être capturés. Par exemple, considérons quatre événements, $\{a, b, c, d\}$. En fonction du support de trace choisi, la signification de la trace suivante, $a.3.b.13.a.7.b.40$ – notée u, possède différentes interprétations. Si le support de la trace correspond aux quatre événements, alors c et d ne se sont pas autorisés durant l'exécution de u. Nous allons considérer pour la suite que le support d'un ensemble de traces S sera noté Σ_S, ou Σ tout cours lorsqu'aucune ambiguïté ne sera possible sur la nature S.

Pour une exécution donnée, si la date 0 correspond au début de l'exécution, alors une trace de durée t peut représenter «l'état» d'une exécution en cours à la date t sous la forme d'un historique. Puisqu'un événement e n'a pas de durée, il y a potentiellement deux traces distinctes ayant la même durée. Pour pouvoir associer une unique trace à une date donnée, nous allons définir le concept de pas maximal d'exécution. Tout d'abord voyons d'où vient le problème. Pour tout trace de la forme $u.e$, u et $u.e$ ont toutes les deux la même durée. Ainsi, si l'on considère l'historique $a.3.b.13.a.7.b.40$, deux traces peuvent correspondre à la date 16 : $a.3.b.13$ et $a.3.b.13.a$.

Définition 3 *Pas maximal*
Nous parlerons de pas maximal pour désigner une observation qui correspondrait à une

trace ne pouvant être poursuivie que par une trace de la forme $\epsilon.v$ avec ϵ une durée non nulle.

Il en découle que si un pas maximal de durée D ne se termine pas sur un événement, cette observation indique qu'aucun événement ne s'est produit à la date D.

La trace $a.3.b.13.a$ correspond au pas maximal d'exécution associé à la date 16. A partir de maintenant, l'état de l'exécution à la date t est le pas maximal d'exécution de durée t.

2.2.3 État du processus d'observation

Dans le cas de l'analyse des comportements temps-réel, l'état du bloc d'observation va correspondre à une trace temporisée représentant l'historique de l'exécution tel qu'il est perçu par le bloc d'observation. Il est désormais possible de comparer l'historique réel de l'exécution et ce que le bloc d'observation est capable de communiquer au bloc d'analyse. Nous nous intéressons à ce qui pourrait être transmis et non à ce qui l'est effectivement. C'est pourquoi nous conservons l'historique complet perçu par le bloc d'observation. La comparaison de l'état du processus et de l'historique de l'exécution permet de déterminer les caractéristiques du processus d'observation.

Définition 4 (État du processus d'observation) *Notons Σ l'ensemble des événements capturés par le processus d'observation. L'état du processus d'observation correspond à une trace temporisée, prise dans $T\Sigma^*$, avec Σ inclus dans le support des traces représentant l'activité du système.*

Le processus d'observation est dit partiel lorsque le support des traces utilisées pour définir l'état du processus d'observation est strictement inclus dans le support des traces représentant l'exécution proprement dite. De nombreuses propriétés du processus d'observation peuvent se définir en comparant l'état du processus d'observation à une date t, et la trace représentant l'état de l'exécution à ce même instant. Pour la suite, nous avons simplifié un peu la nature de la couche d'observation en supposant que les supports des deux traces coïncident. Cela signifie que tous les événements caractéristiques de l'exécution sont observables.

2.2.4 Monotonie des observations

Il est possible d'ordonner deux traces pour refléter le fait qu'une trace représente le passé d'une autre. La définition de cette relation d'ordre repose sur la notion de préfixe d'une trace.

Définition 5 (Rappel – Relation préfixe) *Une trace u, est un préfixe d'une trace w si et seulement si il existe une trace v telle que u.v = w. On dit alors que u est inférieure à w.*

La fonction d'observation est monotone (croissante) si deux états successifs du processus d'observation u_1 et u_2 vérifient les deux contraintes suivantes : la durée de l'historique u_2 est strictement supérieure à la durée de u_1, et u_1 est un préfixe de u_2. D'un point de vue plus concret, le processus d'observation est monotone lorsqu'il construit pas à pas un historique de l'exécution du système sans jamais remettre en cause ce qui a été observé dans le passé. Nous verrons par la suite en quoi cette propriété est cruciale pour éviter un détecteur d'erreur qui bégaye ou commet des erreurs.

Dans le cas d'un processus d'observation monotone, l'unité d'observation d'un processus d'observation est une trace v appartenant à un ensemble clairement identifié de traces noté UO tel que le successeur d'un état u_1 du processus d'observation appartient toujours à $u_1.UO$.

Pour illustrer ce phénomène, nous allons décrire cette évolution pour deux processus d'observation ayant des unités d'observation de natures différentes. Dans le premier cas l'unité d'observation est toujours constituée d'une durée et d'un événement : $T.\Sigma$ (sans l'étoile). Dans le second cas, l'unité d'observation correspond à une période de temps fixe, 20 unités de temps. Dans ce cas, l'ensemble UO correspond à l'ensemble des traces de $T\Sigma^*$ de durée exactement égale à 20. L'évolution de l'état de ces deux processus sera explicitée pour une même trace d'exécution. L'exécution considérée est abstraite en une trace contenant seulement trois événements : *receive_req*, *start_processing*, *send_result*.

- L'exécution démarre par l'événement *receive_req* représentant l'arrivée d'une requête sur un serveur. La requête est mise en attente pendant 11 unités de temps.
- Puis le traitement de la requête est initié en émettant l'événement *start_processing*.
- Le traitement de la requête dure 29 unité de temps, et se termine par l'envoi du résultat au client. Cet envoi est symbolisé par l'événement *send_result*

Les dates des événements sont les suivantes : (*recieve_req* : 0), (*start_processing* : 11), (*send_result* : 40). Les tableaux 2.1 résument les séquences d'états des processus d'observation au cours du temps, ainsi que les dates auxquelles un changement d'état intervient.

2.2.5 «Latence» et exactitude du processus d'observation

A chaque état du processus d'observation, il est possible d'associer une date. La latence du processus d'observation peut varier au cours du temps mais représente la différence entre la date courante[1] et la durée de la trace contenue dans l'état du processus

[1] définie comme la durée de la trace représentant l'exécution courante

TAB. 2.1: tableaux récapitulatifs de l'évolution de l'état de processus d'observation

Cas du processus d'observation n°1

date	nouvelle observation	mise à jour de l'état d'observation
0	*receive_req*	*receive_req*
11	11.*start_processing*	*receive_req*.11.*start_processing*
40	29.*send_result*	*receive_req*.11.*start_processing*.29.*send_result*

Cas du processus d'observation n°2

date	nouvelle observation	mise à jour de l'état d'observation
0	*receive_req*	*receive_req*
20	11.*start_processing*.9	*receive_req*.11.*start_processing*.9
40	20.*send_result*	*receive_req*.11.*start_processing*.9.20.*send_result*

d'observation. Cette latence peut être positive ou négative. Une latence négative fait référence au cas où l'état du processus d'observation est une prédiction du comportement futur de l'application. Une latence positive signifie qu'a priori la trace décrite dans l'état du processus d'observation concerne le passé de l'exécution courante.

La définition de l'exactitude du processus d'observation est relativement aisée dans le cas où la latence d'observation est positive ou nulle. L'observation de l'exécution est exacte si l'état du processus d'observation est un préfixe de la trace temporisée modélisant l'exécution effective du système. Cela signifie simplement que l'état du processus d'observation correspond à une trace appartenant au passé de l'exécution courante. Si ces deux traces ne sont pas comparables, cela signifie alors que le processus d'observation est inexact. Dans le cas où la latence de détection est négative, si le processus d'observation est exact, alors la trace temporisée représentant l'exécution du système est un préfixe de l'état du processus d'observation tout au long de l'exécution du système. Si de plus le processus d'observation est monotone alors cela signifie que l'état du processus d'observation coïncide avec l'exécution courante sur le passé et le présent, et que la prédiction du futur de l'exécution du système finit toujours par se vérifier.

Il est très rare d'avoir à la fois une latence négative, un processus d'observation monotone et exact. La perte de la monotonie signifie que la mise à jour de l'état au cours du temps est potentiellement très coûteuse et peut être totalement contradictoire avec un état précédant. Ce type de processus d'observation est à bannir. La définition du processus d'observation est la première étape vers la compréhension du processus complet de détection. L'étape suivante est la procédure de décision permettant de transformer l'état du processus de détection en une décision vis à vis de la présence ou non d'une erreur

dans le comportement de l'application.

2.3 Symptômes d'erreur et apparition de l'erreur

Nous supposerons à partir de maintenant que la spécification comportementale fournie par un utilisateur du système de vérification désigne les comportements valides de tout ou partie de l'application. Cette description des comportements corrects sera notée $\mathcal{S}pec$.

2.3.1 Symptômes d'erreur associés à $\mathcal{S}pec$

La notion de symptôme d'erreur désigne les traces qui lorsqu'elles sont observées déclenchent le signalement d'une erreur. Dans le cas qui nous concerne les symptômes d'erreur sont définis de manière indirecte. En effet, il convient de déduire leur nature à partir de celle de $\mathcal{S}pec$. La spécification détermine un ensemble de traces dites valides. Les traces considérées sont des traces finies. Les traces invalides correspondraient naturellement au complémentaire de $\mathcal{S}pec$ dans l'ensemble des traces possibles. Nous noterons Σ le support de $\mathcal{S}pec$. Ainsi, l'ensemble des traces possibles correspond à $T\Sigma^*$. Cependant, avec une telle définition des traces invalides, bon nombre d'entre elles pourraient être des préfixes de traces valides. Cela signifie qu'une trace invalides pourrait être continuée en une trace correcte sous certaines conditions.

Les traces représentant des exécutions complètes sont souvent représentées par des séquences infinies pour «occuper le futur» et évacuer ce problème de poursuite d'exécution. L'utilisation de traces temps-réel offre une alternative. Soit l'on utilise des traces représentant une infinité d'événements, soit l'on considère des traces représentant une activité finie sur une durée infinie. Ce subterfuge permet de déclarer la fin d'une exécution en accolant à la fin de la trace une durée infinie interdisant tout nouvel événement. Le complémentaire de $\mathcal{S}pec.\infty$ serait alors défini par rapport à $T\Sigma^*.\infty$. Ainsi, toute trace appartenant au complémentaire de la spécification ne pourra être le préfixe d'aucune trace finie.

Un mauvais préfixe pour un ensemble de traces de durée infinie noté L, est une trace finie u telle que $u.T\Sigma^*.\infty$ ait une intersection nulle avec L. Cela signifie que u ne peut plus désormais être «continué» en une trace de L. Un ensemble de traces L est dit un langage de sûreté si son complémentaire peut s'écrire comme l'union des langages $u.T\Sigma^*.\infty$, pour chaque mauvais préfixe de L, noté ici u. De manière pratique toute trace finie, qui n'est pas un préfixe fini d'une trace appartenant à $\mathcal{S}pec.\infty$ est un mauvais préfixe de $\mathcal{S}pec.\infty$ et permet donc d'identifier tout un ensemble de traces invalides.

Définition 6 (Symptômes d'erreur de $\mathcal{S}pec$) *Soit $\mathcal{S}pec$ l'ensemble de traces identifiées comme valides. L'ensemble des symptômes d'erreur pour $\mathcal{S}pec$ correspond à l'ensemble*

des traces n'appartenant pas aux préfixes finis de $\mathcal{S}pec.\infty$

Nous noterons $Pref(L)$ l'ensemble des préfixes finis d'un ensemble de traces L. La procédure de détection repose sur l'identification d'un ensemble de symptômes d'erreur permettant de capturer toutes les traces invalides associées à $\mathcal{S}pec.\infty$.

2.3.2 Date d'apparition de l'erreur

Pour pouvoir définir la latence de détection, il faut être capable de définir une date de référence identifiant l'apparition des erreurs dans la trace représentant l'état du système. Cela revient à déterminer les plus petites séquences d'observations correspondant à des symptômes d'erreur. Ce concept est à rapprocher de la notion de «minimal bad préfix» définie sur les traces non temporisées, [KV01], et en constitue l'extension au cadre des traces temporisées.

Propriété 1 (Monotonie du processus d'identification des symptômes d'erreur)
Pour tout symptôme d'erreur s, alors toutes les continuations finies de s sont des symptômes d'erreur.
Pour tout préfixe de la spécification s', alors tout préfixe de s' est aussi un préfixe de la spécification

La première partie de la propriété est l'une des conséquences directes de la définition d'un symptôme d'erreur. Si u est un symptôme d'erreur alors pour tout w tel que u est un préfixe de w, w peut s'écrire $u.v$ d'où $w.T\Sigma^*.\infty$ est inclus dans $u.T\Sigma^*.\infty$. Or tout élément de $u.T\Sigma^*.\infty$ est une trace incorrecte puisque u est un symptôme d'erreur. il en découle que w est aussi un symptôme d'erreur. La seconde partie de la propriété de monotonie est déduite de la propriété de transitivité intrinsèque à toute relation d'ordre : si u est inférieur à v et v est inférieur à w, alors u est inférieur à w. Ainsi, si s'' est un préfixe de s', et s' est un préfixe d'une trace correcte, alors s'' est aussi un préfixe de cette trace. L'ensemble des passés d'un symptôme d'erreur peut être totalement ordonné. Une fois cet ensemble ordonné, la monotonie nous assure que cet ensemble est décomposé en deux sous-ensembles eux mêmes ordonnés : l'ensemble des passés exempts d'erreur, et l'ensemble des passés qui sont des symptôme d'erreur. Il est donc possible de trouver une unique trace à la frontière de ces deux ensembles, et cette trace constituera le plus petit symptôme d'erreur associé au symptôme d'erreur traité.

Pour chaque symptôme d'erreur s, $Pref(s)$ contient l'ensemble des traces représentant un instant passé de l'exécution de s. $Pref(\mathcal{S}pec.\infty)$ et son complémentaire dans l'ensemble des traces finies forment une partition en deux de l'ensemble des traces finies $T\Sigma^*$. Il en va de même pour tout sous ensemble de $T\Sigma^*$. Ainsi, $Pref(s)$ est séparé en deux sous ensembles disjoints. D'un coté, il y a les passés de s encore susceptibles de mener à une

exécution correcte, noté $Pref^{+}(s)$. De l'autre, on trouve l'ensemble des passés de s qui font partie des symptômes d'erreur, notés $Pref^{-}(s)$.

Propriété 2 (Séparation unique de $Pref^{+}(s)$ et $Pref^{-}(s)$) *Il existe une unique trace de $Pref(s)$ telle que tout préfixe correct de s est inférieur à cette trace, et que tout préfixe incorrect est supérieur à cette trace : la borne inférieure de $Pref^{-}(s)$ selon la relation préfixe.*

Preuve :

A cause de la propriété de monotonie du processus de décision concernant la présence d'une erreur dans une trace, nous pouvons affirmer que $\forall x, y, x \in Pref^{+}(s) \wedge y \in Pref^{-}(s) \Rightarrow x$ est un préfixe de y. En effet, le contraire serait absurde puisque tout préfixe d'un préfixe correct est un préfixe correct, et qu'une trace ne peut appartenir à la fois à l'ensemble des préfixes de traces correctes et à l'ensemble des symptômes d'erreur. À partir de là, nous en déduisons que $Pref^{-}(s)$ succède à $Pref^{+}(s)$. Or, tout sous-ensemble X totalement ordonné d'un ensemble Y, possède au moins une borne inférieure contenue dans Y. Deux cas de figure peuvent se présenter en fonction de la façon dont on considère l'écoulement du temps. Si le temps s'écoule de manière discrète, alors $Pref^{-}(s)$ est un domaine discret[2] et contient sa borne inférieure. Si le temps s'écoule de manière continue, alors la borne inférieure de $Pref^{-}(s)$ peut en toute théorie ne pas faire partie de cet ensemble et donc appartenir à $Pref^{+}(s)$. Ainsi, la borne inférieure de $Pref^{-}(s)$ peut ne pas être stricto sensu un symptôme d'erreur. Cependant, nous souhaitons utiliser cette borne inférieure comme un marqueur permettant d'identifier les erreurs dans tous les cas. Cette approximation se justifie puisque lorsque cette borne inférieure n'est pas un minimum, alors elle possède des successeurs infiniment proches qui eux sont des symptômes d'erreur.

Notons $\prec$ la relation d'ordre définie par la relation préfixe sur les traces. La borne inférieure d'un ensemble de traces totalement ordonnées selon $\prec$ se note $Inf_{\prec}(L)$. Ceci nous permet de simplement définir la plus petite séquence d'observation permettant de «prouver» la présence d'une erreur, ou son infinie proximité.

Définition 7 (Ensemble des plus petits symptômes d'erreur) *Étant donnée une spécification $\mathcal{S}pec$, l'ensemble des plus petit symptômes d'erreur correspond à*

$$Sympt_{min}(\mathcal{S}pec) = \bigcup_{u \in \overline{Pref(\mathcal{S}pec.\infty)}} Inf_{\prec}(Pref^{-}(u))$$

[2] un ensemble est discret si la distance entre deux de ses éléments est minorée. Ici la distance utilisée sur les traces repose sur la mesure de la durée d'une trace.

La surveillance à l'exécution de l'ensemble de ces traces nous assure que la présence d'une erreur sera signalée dès que l'information sera disponible. Nous avons défini jusqu'à présent les trois points clés permettant la caractérisation du processus de détection et de la détection au plus tôt : le processus d'interprétation d'une trace en symptôme d'erreur, la définition des attributs du processus d'observation, et pour finir la définition de l'ensemble des symptômes d'erreur permettant la détection des erreurs dès que l'information est disponible.

2.4 Latence de détection et détection au plus tôt

La latence totale du système de détection créé est due au processus de capture, de propagation, et d'interprétation de l'information à travers l'architecture du moniteur. Nous avons défini dans la section précédente la notion d'erreur comportementale et par la même occasion sa date précise d'occurrence. La définition de cette date a été obtenue grâce au concept de symptôme d'erreur. Nous allons énoncer le principe de détection causale puis nous verrons quelles peuvent être les sources de latence dans ce processus de détection. À l'issue de cette analyse, nous allons définir la détection causale «au plus tôt».

2.4.1 Contribution à la latence de détection

La latence de détection peut être considérée comme l'accumulation de retards introduits à chaque étage de l'architecture du moniteur. Ainsi, il faut considérer le mécanisme d'observation, le processus d'analyse des événements, et les communications entre ces deux couches. Le bloc de traitement n'est pas pris en compte car le signalement de l'erreur correspond à l'événement qui transite justement entre le bloc d'analyse et le bloc de traitement. La figure 2.1 illustre ce phénomène de propagation de l'information. Considérons qu'une erreur comportementale se produit, ce qui signifie que la trace représentant l'exécution courante est un plus petit symptôme d'erreur. À partir de ce moment, le temps séparant l'apparition de l'erreur et son signalement peut être décomposé selon trois phénomènes responsable de la latence de détection

- Les retards introduits par les mécanismes de communication inter-blocs du moniteur (du bloc d'observation vers le bloc d'analyse).
- La latence introduite par le processus d'observation. d'un point de vue pratique cette latence peut être due à un mécanisme qui s'assure que les événements sont bien reçus dans l'ordre et sans omission par le bloc d'observation avant de les restituer sous la forme d'une trace.
- Le temps de retard introduit par le processus d'analyse qui assure le service de détection d'erreur à travers des symptômes d'erreur qui ne sont pas minimaux

Ces trois contributions à la latence sont représentées sur la figure 2.1. L'état du processus d'observation ne permet pas nécessairement de réveiller le processus d'analyse sur n'importe quelle type de trace. Il est alors possible de «laisser passer» certaines traces erronées. Ce phénomène se produit par exemple pour un processus d'observation indiquant seulement les nouveaux événements. Un symptôme d'erreur identifiant une erreur se traduisant par un dépassement d'échéance peut se poursuivre par un silence définitif du système (équivalent à son crash). Plus généralement, la latence de détection se répartie entre le temps de calcul nécessaire à chaque étapes et les temps de pause induits par la façon dont l'information est relayée puis traitée.

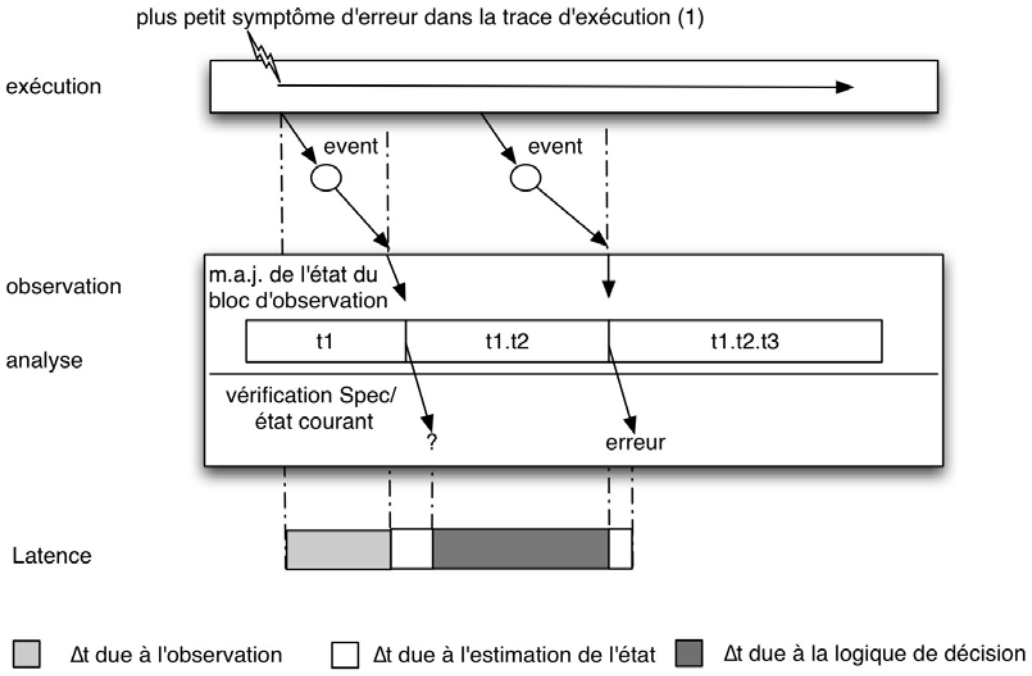

FIG. 2.1: Les différentes sources de latence dans le processus de détection d'erreur
L'apparition de l'erreur dans l'exécution en (1) correspond à la capture possible d'un plus petit symptôme d'erreur.

Nous allons dans la suite définir une classe particulière de détecteurs qui rassemble un type de processus d'observation, et un processus d'analyse utilisant les plus petits symptômes d'erreur.

2.4.2 Détection causale, «au plus tôt»

Pour réduire la latence de détection, il est important d'harmoniser le processus d'observation et la procédure de décision utilisée pour identifier les erreurs. Cependant, nous allons nous restreindre à un certain type de détecteur.

2.4.2.1 Détection Causale

Nous avons déjà évoqué la possibilité de faire en sorte que la latence du bloc d'observation soit «négative» en réalisant des pronostics sur les événements futurs d'une exécution. Nous avons cependant pointé du doigt la fragilité de l'information ainsi produite. Si l'état du bloc d'observation n'est pas monotone, cela veut dire que soit le bloc d'analyse doit prendre en compte cette volatilité de l'information, soit que le signalement de la présence d'une erreur puisse être annulé. C'est pour ces raisons qu'il est préférable que l'évolution de l'état du processus d'observation soit monotone.

Définition 8 (Détection Causale) *La détection est dite causale lorsque le processus d'observation est exact et possède une latence positive.*

Sous ces conditions, la latence du processus d'observation vient s'ajouter directement à la latence totale de détection. Nous proposons d'utiliser les plus petits symptômes d'erreurs pour permettre la minimisation de la latence totale à travers la suppression d'attentes inutiles dues à une utilisation inefficace de l'information collectée.

Ces attentes peuvent provenir d'un processus de collecte et diffusion inadapté des observations. Ceci se produit lorsque les traces représentant l'état du processus d'observation ne coïncident pas bien avec les plus petits symptômes d'erreur.

2.4.2.2 Détection au plus tôt et plus petits symptômes d'erreur

Le critère de détection au plus tôt est défini par rapport à la quantité d'information utilisée par le détecteur. En bref cela signifie que l'on souhaite que le détecteur puisse signaler l'erreur dès que celle-ci est avérée. Nous traduisons ceci par : «le détecteur d'erreur doit signaler une erreur dès qu'il peut affirmer avoir observé un plus petit symptôme d'erreur».

Définition 9 (Détection au plus tôt d'erreur comportementales) *Étant donnée la spécification $\mathcal{S}pec$, le détecteur au plus tôt est défini comme le système qui lève un signal d'erreur dès que il observe une trace appartenant à l'ensemble des plus petits symptômes d'erreur de $\mathcal{S}pec$*

Par définition, le détecteur observe l'exécution en cours. Dans le cas des traces temps-réel, les symptômes d'erreur ne se terminent pas tous par des événements. Il faudra donc créer de toute pièce des événements spéciaux pour réveiller le bloc d'analyse lorsqu'un plus petit symptôme d'erreur correspondant à un dépassement d'échéance sera rencontré. Pour illustrer ce concept de détection "au plus tôt", nous utiliserons un exemple assez proche de ce qui est proposé dans [RRJ92]. Ceci nous permettra de pointer du doigt en quoi il est raisonnable de parler d'erreur et non de défaillance.

2.4.2.3 Détection au plus tôt et spécifications modulaires

On considère un système dont la spécification comportementale est construite comme l'intersection de diverses contraintes comportementales Voici une description informelle des comportements autorisés pour le composant vérifié.

- Le système est un serveur qui reçoit des requêtes et émet des réponses contenant un résultat numérique.
- La réception d'une requête (événement *receive_req*) est toujours suivie de l'envoi de la réponse en mode *one_shot* représentée par l'événement *send_os*. La réponse doit être émise au plus 10 unités de temps après la requête. (contrainte P1)
- Le système possède deux modes d'envoi : le mode *one_shot*, et le mode "flux" *stream*. Périodiquement le système émet un flux de données. Le début de l'émission est représenté par l'événement *send_str*
- L'envoi d'un flux de données (mode *stream*) interdit tout envoi en mode *one_shot* pendant une durée de 4 unités de temps. (contrainte P2)

Un scénario de détection au plus tôt est illustré sur la figure 2.2. Une erreur peut être détectée lorsque l'on reçoit une requête *receive_req*, puis sept unités de temps plus tard l'événement *send_str*. Si ceci a lieu avant l'occurrence de l'événement *send_os*, bien que P1 et P2 puissent encore être satisfaites prisent séparément, la conjonction des deux contraintes ne peut être respectée. Si l'on considère que P1 est une exigence liée à la fonction du service et que P2 est une contrainte liée à son implémentation le conflit entre P1 et P2 est bien une erreur indiquant l'impossibilité de concilier le comportement attendu du système avec ses exigences de service. Si une autre implémentation du service existe, il est alors peut être possible d'ignorer la contrainte P2 et tout de même satisfaire P1, tout ceci à condition d'agir suffisamment vite. Sur la figure, notez la présence de deux tâches à l'extrémité des cônes représentant les continuations de la trace courante. La tâche noire représente les traces satisfaisant P1, alors que la trace tache blanche correspond aux traces satisfaisant P2. Les deux ensembles de traces sont disjoints. Il en découle que la trace courante ne possède pas de continuation satisfaisant à la fois P1 et P2.

Une méthode classique d'évaluation des formules logiques repose sur le principe de diviser pour régner sur une conjonction de formules. La trace est évaluée à faux dès que

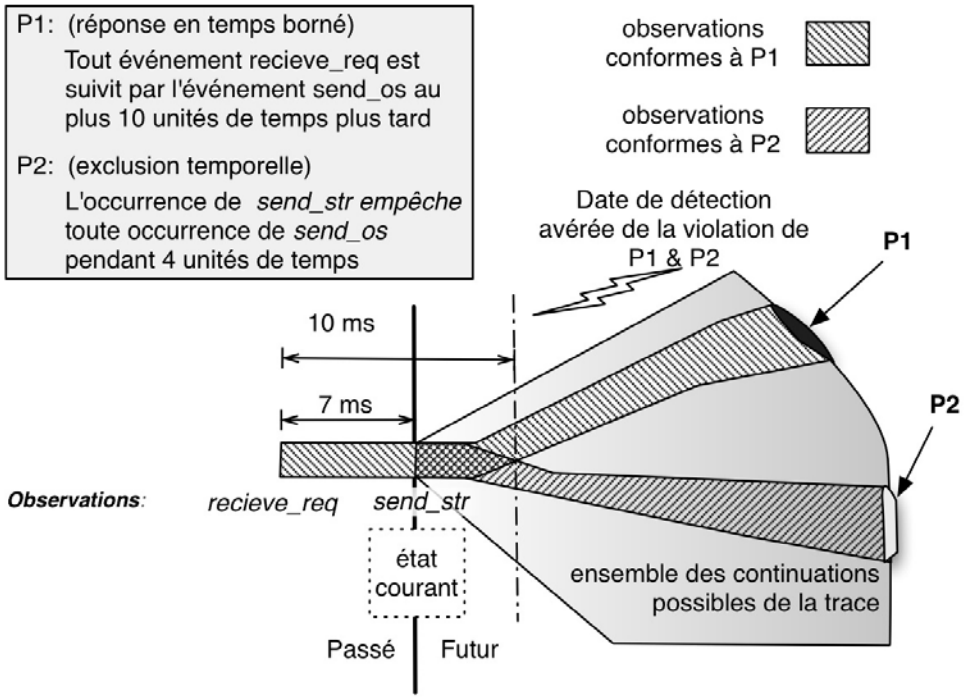

FIG. 2.2: Détection au plus tôt et contre exemple

la trace invalide l'une ou l'autre des formules de la conjonction. En effet, une conjonction est fausse si au moins l'un de ses membres l'est. Si l'on n'y prend pas garde, alors il est relativement facile de ne pas détecter la violation de $P1 \wedge P2$ au plus tôt et d'attendre que l'événement *send_os* soit réellement émis pour finalement se rendre compte que c'était perdu d'avance. Ce type de procédure de décision est un contre exemple de la propriété de détection au plus tôt.

Conclusion

Dans ce chapitre nous avions pour objectif de modéliser les éléments clés du processus de détection à travers les éléments primitifs de la théorie des traces : des ensembles, la relation préfixe, les continuations... La contribution principale de ce chapitre tient au cadre formel mis en place pour décrire ces éléments : le processus d'observation et le processus de détection. Ce cadre de description nous a permis de définir proprement ce que signifie la détection au plus tôt et la relation liant le bloc d'observation et le bloc

d'analyse. Le concept d'erreur comportementale est un peu contre nature puisqu'il associe une erreur à une trace et non un état. Cette vision est particulièrement adaptée dans le cas où le composant est un composant «boite grise» pour lequel seuls des événements sont capturables. Dans ce cas, il est nécessaire de reconstruire un état abstrait à partir de cette trace. Ensuite, il est possible de l'utiliser pour réaliser la vérification en-ligne. La plus petite période d'observation continue nécessaire pour «prouver» l'apparition d'une erreur a été appelé un plus petit symptôme de défaillance. Cette étude pourrait se poursuivre par un questionnement sur des modèles de "traces"[3] plus complexes utilisés dans le cadre des systèmes temps-réels distribués. Sur ce chemin, des modèles d'horloges distribuées quantitatives quasi-synchrones furent proposées dans [VR00, CMV00]. Il serait intéressant de comprendre leur impact sur la topologie de l'espace des exécutions. Cela permettrait en particulier de voir si un concept équivalent à celui de la monotonie du processus de décision et d'observation peut être défini. À partir de ces objets théoriques, le concept de détecteur causal au plus tôt a été proposé. Ce détecteur représente une sorte de référence parfaite pouvant servir d'étalon pour évaluer la latence de détection d'un détecteur concret. Nous verrons comment appliquer en pratique ces concepts sur un formalisme adapté à la modélisation de comportements temps-réel.

[3] ordres partiels

Chapitre 3

Synthèse d’un détecteur au plus tôt

Introduction

Dans le chapitre précédant, le service optimal attendu d'un détecteur a été formellement spécifié dans le cadre d'une détection causale. Cette définition fixe l'objectif à atteindre en termes d'identification des erreurs. La notion de plus petit symptôme d'erreur correspond à l'ensemble des plus petites traces permettant de prouver l'apparition d'une erreur vis-à-vis d'une description des comportements attendus de l'application. Cette définition sur des ensembles de traces est assez générale pour être applicable à de nombreux problème de vérification en ligne, à savoir à de nombreux formalismes.

Pour assurer la détection au plus tôt, il est nécessaire de pouvoir « reconnaître » à la volée les plus petits symptômes d'erreur. La difficulté de cette approche réside essentiellement dans la prise en compte de modèles temporisées et non de simple séquences d'événements.

À partir du langage de trace à identifier, un système à transitions est généré et est animé à l'exécution. L'effort consenti lors de la modélisation du processus d'observation nous permet de comprendre comment les différents symptômes d'erreurs peuvent être capturés. La spécification du comportement d'une application est définie à travers un modèle d'automates temporisés, celui proposé par Alur & Dill, [AD94]. Ainsi, les traces de la spécification sont déjà représentées par un système à transition. Nous avons donc proposé une interprétation de la définition des plus petits symptômes d'erreur directement sur l'automate. Grâce à des modèles dérivés de l'étude des automates temporisés dans le domaine de la vérification hors-ligne, il est possible de transformer un automate temporisé en un modèle permettant d'identifier ses plus petits symptômes d'erreur. C'est ce dernier modèle qui sera utilisé en ligne pour effectuer la vérification du comportement de l'application.

La première section présentera le formalisme des automates temporisés utilisé comme support pour définir un ensemble de traces temporisées. Nous présenterons ensuite la logique de détection utilisée, et en particulier comment le processus de vérification est animé pour assurer la capture de tous les plus petits symptômes d'erreur.

3.1 Automates temporisés et traces temporisées

Nous allons dans cette section détailler le formalisme des automates temporisés évoqué dans le premier chapitre. Le modèle d'automate temporisé, introduit par Alur & Dill, a connu un franc succès grâce aux méthodes d'analyse proposées pour appuyer la vérification formelle hors-ligne de ces objets. Ces bases théoriques ont permis de faire de ce modèle une référence supportées par de nombreux outils de vérification, de test, et de synthèse de contrôleurs, dont la suite complète des outils UPPAAL est un bon exemple

[UPP]. Ces automates temporisés permettent de définir un ensemble de traces temporisées.

3.1.1 Rappel sur les automates finis non temporisés.

Tout d'abord, il est nécessaire de comprendre qu'un automate est avant tout un graphe étiqueté et surchargé d'information permettant de définir une relation décrivant un ensemble de changement d'états possibles. Les changements d'états seront appelés transitions. Dans notre cas, un événement sera associé à chaque changement d'état. Par définition, à partir d'un état de l'automate, l'ensemble des transitions que l'on peut réaliser est statiquement défini. Cela implique que si l'écoulement du temps influe sur les transitions que l'on peut réaliser, alors l'état possède une composante temporelle.

La représentation de l'automate sous forme de graphe n'associe pas nécessairement un état à chaque nœud. Il est suffisant de pouvoir déterminer les transitions exécutables en chaque état à partir de l'information associée aux noeuds et aux arcs.

Une exécution de l'automate dans ce contexte est une séquence de transitions. En ne conservant que les événements associés aux transitions, il est possible d'obtenir une trace d'événements. Dans le cadre temporisé, les transitions sont de deux types : événementielles, et temporelles. Cela permet d'être cohérent avec la définition des traces temporisées. Le graphe permettant de représenter l'ensemble des transitions n'associe pas nécessairement un noeud à chaque état de l'automate. De manière similaire, la formation de cycles dans ce graphe permet de représenter un nombre a priori non borné de scénarios d'exécution distincts. L'important est de bien comprendre cette distinction entre l'ensemble des transitions réellement exécutable sur l'automate et le graphe utilisé pour les représenter. On utilisera l'expression exécuter une trace sur l'automate lorsque l'on franchira une séquence de transitions dont la séquence d'étiquettes correspond à la trace.

Une transition sera noté (état source $\overset{\text{étiquette}}{\rightarrow}$ état de destination). De manière similaire, le changement d'état dû à l'exécution d'une trace u sera noté (état source$\overset{u}{\rightarrow}$ état de destination). Cela signifie qu'il existe une séquence de transitions élémentaires menant de l'état source à l'état de destination dont les étiquettes sont successivement les éléments de la trace u.

La définition des états initiaux et finals de l'automate permet de fixer le point de départ et d'arrivée des exécutions de l'automate. Si l'on se réfère à la figure 3.1, le comportement d'un serveur de calcul y est représenté sommairement. Sur l'exemple l'automate possède un seul état initial, $s0$, et un seul état final $s6$. La figure représente un automate où il existe au moins une exécution de l'automate correspondant à la trace *req.start_proc.end_process*. On notera aussi qu'un seul chemin suffit pour faire que cette trace soit acceptée mais que cela n'empêche pas l'existence de plusieurs chemins

distincts correspondant à la même trace. Dans le cas ou plusieurs chemins existent pour une seule et même trace, on dira que l'automate est **non déterministe**. Les deux flèches en double ligne montrent les deux chemins que l'on aurait pu suivre en exécutant la trace *req.start_proc.end_process*. Lorsqu'il devient impossible d'atteindre un état final à partir de l'état courant, cet état correspond à la détection d'un symptôme d'erreur. On parle alors d'état d'impasse ou de rebut dont l'état $s7$ est un exemple dans notre exemple. En

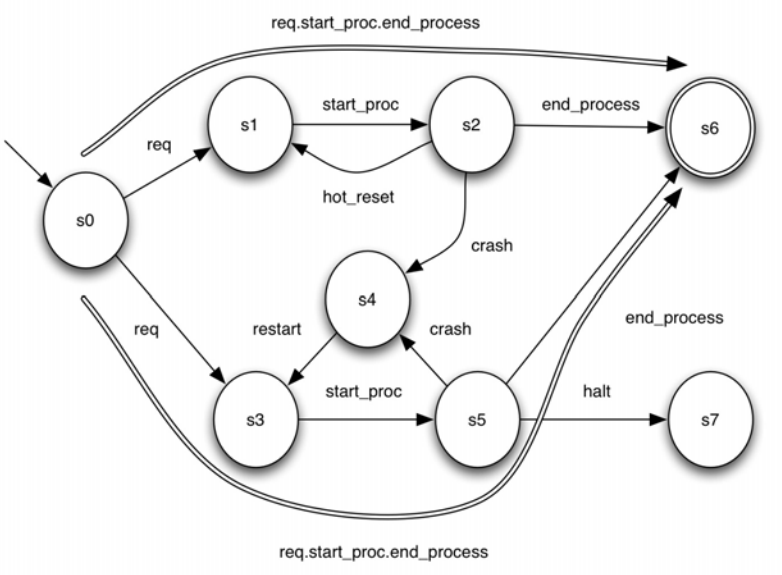

FIG. 3.1: Chemins dans un automate, des états initiaux vers les états finals.

toute théorie si l'on ne s'intéresse qu'au langage de l'automate, il est possible de transformer l'automate fini pour minimiser son nombre d'états, et déterminiser ses exécutions (créer un graphe de transitions tel que chaque trace est associée à un unique chemin dans l'automate). Ceci permet d'optimiser la manipulation de l'automate en ligne.

3.1.2 Les automates temporisés.

L'extension des automates classiques proposée par Alur & Dill permet d'intégrer la dimension temporelle en introduisant :

- Des variables jouant le rôle d'horloges réinitialisables.
- Des contraintes associées aux arcs de l'automate qui rendent ces arcs franchissables sous conditions, en fonction de la valeur des horloges.

Les horloges mesurent l'écoulement du temps de manière continue. Cela signifie que l'on ne peut pas les mettre en pause. La seule opération autorisée sur ces variables est

leur remise à zéro lors du franchissement d'un arc. Cela permet d'associer la mise à zéro de ces horloges à l'occurrence d'un événement et donc de mesurer le temps écoulé depuis le franchissement d'une transition précise. A cause des variables d'horloge l'état de l'automate ne se limite pas à sa position dans le graphe de l'automate.

Il faut pouvoir expliquer comment une trace temporisée s'exécute sur un tel système. L'exemple du serveur est adapté pour intégrer les horloges : le serveur à l'issue de la réception d'une requête, représentée par l'événement *request*, peut engendrer en fonction de l'origine de la requête deux comportements distincts. Le premier scénario doit s'exécuter en au plus 20 unités de temps. Son exécution se déroule en deux étapes : tout d'abord le serveur obtient l'accès à une ressource de calcul prioritaire représenté par l'événement *get_prio* puis délivre le résultat. La ressource prioritaire ne délivre pas de résultat avant 5 unités de temps. Le second scénario d'exécution doit lui s'exécuter en au plus 25 unités de temps. Il se déroule lui aussi en deux étapes.Tout d'abord le serveur obtient une ressource de calcul de seconde zone, et réalise la tâche demandée. La ressource de calcul de seconde zone n'est pas totalement spécifiée et donc ne fournit aucune information sur sa vitesse d'exécution et son temps de blocage.

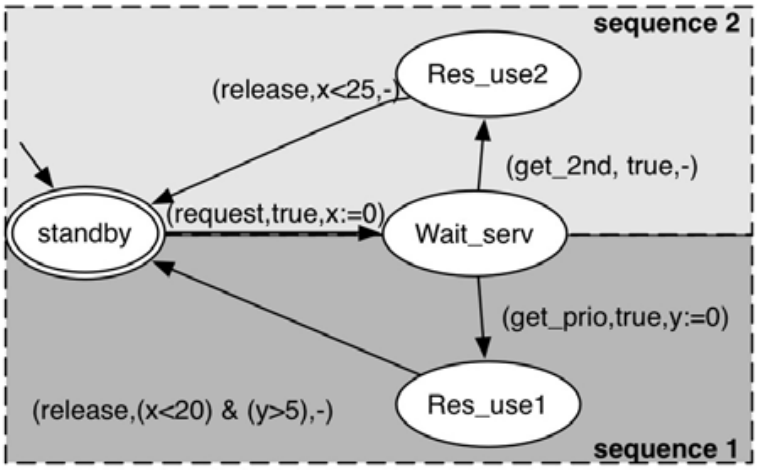

FIG. 3.2: Spécification par automate temporisé d'un serveur de calcul

De manière plus formelle l'automate temporisé contient tous les éléments d'un automate classique, plus les éléments nécessaires à la définition des contraintes temporelles définies à partir des horloges. Nous avons retenu la définition étendue par des contraintes d'horloges sur les lieux de l'automate en supplément des contraintes sur les arcs. Ces contraintes définies sur les lieux de l'automate sont appelés les invariants de lieux.

Définition 10 (Automates temporisés Alur & Dill [AD94, HNSY94]) *Un automate temporisé est un sextuplé* $\mathcal{A} =< Loc, C, \Sigma, Inv, Edge, l_0 >$ *avec :*

- *Loc est l'ensemble des lieux de l'automate.*
- *C est un vecteur de noms de variables représentant des horloges :* $[x_1, \cdot, x_k]$[1]
- Σ *est l'ensemble des noms d'événements utilisés par la suite dans la définition des changements de lieux dans l'automate.*
- *une fonction, Inv, qui associe à chaque lieu une contrainte temporelle. Les contraintes temporelles sont définies comme une conjonction de contraintes élémentaires de la forme du* $x_i \sim c$ *avec* $\sim$ *pris au choix dans* $\{<, \leq\}$*, et c un nombre.*
- *Edge l'ensemble des arcs définissant les conditions de passage entre les lieux de l'automate lors de l'occurrence d'un événement. Ces arcs sont caractérisés par :*
  - *l et l' les lieux d'origine et de destination de l'arc, tous les deux dans Loc.*
  - *g la garde de l'arc. Cette garde est une contrainte numérique sur la valeur des horloges. Cette contrainte est un pré-requis au franchissement de l'arc. Elle se définit comme une conjonction de contraintes élémentaires de la forme* $x_i \sim c$*, ou* $x_i - x_j \sim c$ *avec c une constante,* x_i, x_j *des horloges, et* $\sim$ *un opérateur de comparaison au choix parmi* $\{==, <, \leq, >\geq\}$*.*
  - *l'événement e requis pour franchir l'arc, e est pris dans* Σ
  - *l'ensemble* λ *des horloges à remettre à zéro après franchissement de l'arc.*

  Ainsi, un arc est un quintuplé (l, g, e, λ, l')
- loc_0 *est le lieu initial à partir duquel seront débutées toutes les exécutions de l'automate.*

Le langage d'un automate est constitué de l'ensemble des traces exécutables sur l'automate. Dans sa version améliorée, une condition d'acceptation est utilisée pour introduire des contraintes de vivacité sur les exécutions de l'automate. Les conditions d'acceptation les plus classiques sont des conditions reposant sur la propriété d'accessibilité que nous détaillerons par la suite.

3.1.3 Interprétation des automates temporisés

L'état d'un automate temporisé est représenté sous la forme d'une paire liant un lieu et une valuation de l'ensemble d'horloges. Une valuation d'un ensemble de variables est une fonction qui associe au nom de chaque variable une valeur. Pour simplifier à la fois les notations et le discours, nous supposerons un ordre pré-défini sur les noms de variables. Puisque l'ensemble des horloges utilisées dans l'automate est fini, une valuation peut

[1] Un ensemble aurait tout aussi bien pu être considéré mais en introduisant un vecteur dès maintenant, la définition de l'état de l'automate et de sa sémantique sera grandement simplifiée

être représentée par un vecteur de valeurs réelles positives. Un tel vecteur sera appelé un vecteur d'horloge.

Les traces acceptées par l'automate correspondent à la séquence d'étiquettes d'une exécution de l'automate. Il existe deux types de changements d'état. Nous noterons un vecteur d'horloge γ. L'impact de l'écoulement de d unités de temps sur un vecteur d'horloge sera noté $\gamma+d$ et correspond à l'ajout de d à chaque coordonnées du vecteur. $[\lambda \rightarrow 0]\gamma$ indique la mise à zéro des horloges contenu dans l'ensemble λ pour le vecteur d'horloge γ. $\gamma \models \phi$ signifie que le vecteur d'horloge γ satisfait la condition ϕ, qui peut être une garde ou un invariant.

Les deux types de changements d'état autorisés dans un automate $\mathcal{A}$ défini par $< Loc, C, \Sigma, Inv, E, l_0 >$, sont :

a) les changements d'état sur un événement a qui se traduisent par le passage d'un état (l, γ) à un état (l', γ') si et seulement s'il existe un arc (l, g, a, λ, l') tel que :
- le vecteur d'horloge avant le changement d'état satisfait la garde g de la transition, $\gamma \models g$
- le vecteur d'horloge après le changement d'état satisfait l'invariant de l'. Lors du changement d'état, le vecteur d'horloge est mis à jour en effectuant les mises à zéro d'horloges indiquées sur la transition. Ainsi, il faut s'assurer que l'invariant de l' est bien satisfait par le vecteur mis à jour : $[\lambda \rightarrow 0]\gamma \models Inv(l')$.

Un tel changement d'état sera noté $(l, \gamma) \xrightarrow{a} (l', \gamma')$ et correspond à une « transition événementielle ».

b) les changements d'état dus à l'écoulement d'une durée d qui se traduisent par le passage d'un état (l, γ) à l'état $(l, \gamma + d)$. Ce changement d'état est autorisé si et seulement si le vecteur d'horloge après ajout de cette durée satisfait encore l'invariant du lieu l. Un tel changement d'état sera noté $(l, \gamma) \xrightarrow{d} (l, \gamma')$. Ce type de transition sera appelé dans la suite une transition temporelle.

Considérons la trace $u = d_0(a_1.d_1).\cdot.(a_p.d_p)$ où chaque a_i est un événement, et chaque d_i une durée. L'exécution de cette trace sur l'automate se traduit par l'alternance des deux types de transitions que nous venons de décrire. Une telle trace est acceptée par un automate temporisé A si et seulement si il existe une séquence d'états $(\rho_j)_{0\leq j\leq 2p+1}$, avec chaque $\rho_j = (l_j, \gamma_j)$ tel que :
- l_0 est le lieu initial loc_0, et γ_0 correspond au vecteur nul.
- Pour tout i, entre 1 et p, $t_{2i-1} = (l_{2i-1}, \gamma_{2i-1}) \xrightarrow{a_i} (l_{2i}, \gamma_{2i})$ est une transition événementielle autorisée par l'automate.
- Pour tout i entre 0 et p $t_{2i} = (l_{2i}, \gamma_{2i}) \xrightarrow{d_i} (l_{2i+1}, \gamma_{2i+1})$ est une transition temporelle autorisée par l'automate

La séquences des transitions $(t_j)_{0\leq j\leq 2p}$ représente alors "l'exécution de la trace u" dans l'automate.

Nous l'avons déjà indiqué précédemment, une trace sera autorisée tout d'abord si elle est exécutable. Nous allons faire un point sur les conditions d'acceptation qu'il est raisonnable de considérer. La condition d'acceptation de Büchi [Alu99] permet entre autre de définir des langages de traces correspondant à des propriétés de vivacité. Cette condition requiert que toute trace valide soit une trace infinie telle que pour tout préfixe de cette trace, un état final est accessible depuis l'état atteint par exécution du préfixe. Il en découle qu'un état final est visité une infinité de fois lors de l'exécution d'une trace infinie valide. La définition faisant le consensus dans la communauté identifie les propriétés de vivacité comme l'ensemble des spécifications logiques définissant un ensemble de traces tel que toute trace finie doit posséder au moins une continuation satisfiant la propriété de vivacité. Il en découle qu'il est ridicule de vouloir prouver la violation d'une propriété de vivacité pure à partir d'une séquence d'observation finie. Cependant, la majorité des exigences que l'on souhaite pouvoir définir sont obtenues par conjonction de propriétés de vivacité et de propriétés de sûreté. Le fait qu'une trace soit exécutable correspond à l'expression de contraintes de sûreté. Le fait qu'après une certaine exécution un certain nombre d'état soient toujours accessibles correspond à une contrainte du type : dans le futur, on finira par atteindre l'un de ces état. Cela correspond typiquement à la classe des propriétés de vivacité. Dans le cadre de la vérification en ligne, on distingue une classe de propriétés qui appartient à l'ensemble des propriétés de sûreté : les propriétés de vivacité bornée.

Une telle propriété permet de définir un langage de traces qui exprime des obligations du type : «quelque chose de bien finira par arriver avant T unités de temps».

Nous avons choisit de considérer les TSA, plus une condition d'acceptation désignant une trace finie comme valide si et seulement si l'état atteint après son exécution fait partit d'un ensemble de lieux dits finals.

Dans l'exemple de la figure 3.2, le lieu *standby* est un lieu initial et final. Un vecteur d'horloge de cet automate est constitué de la valeur de x, puis celle de y. La séquence de transitions correspondant à l'exécution de 3.*request*.12.*get_prio*.6.*release* sur l'automate est la suivante : :

$$(standby, [0,0]) \xrightarrow{3} (standby, [3,3]) \xrightarrow{request} (wait\_serv, [0,3]) \xrightarrow{12} (wait\_serv, [12,15]) \rightarrow$$
$$\rightarrow (wait\_serv, [12,15]) \xrightarrow{get\_prio} (res\_use1, [12,0]) \xrightarrow{6} (res\_use1, [18,6]) \rightarrow$$
$$\rightarrow (res\_use1, [18,6]) \xrightarrow{release} (standby, [18,6])$$

Pour la suite de ce travail nous considérerons que l'automate temporisé est déterministe.

Définition 11 (Automate temporisé Déterministe) *L'automate est déterministe s'il vérifie les conditions suivantes*

- *tous les arcs de l'automate sont effectivement étiquetés. Cela signifie qu'aucune horloge ne peut être réinitialisée en l'absence d'événement, et le lieu correspondant à l'état de l'automate reste lui aussi constant tant qu'aucun événement n'est observé.*
- *Pour tous les lieux de l'automate, deux arcs partant de ce lieu ne peuvent à la fois être étiquetée par le même événement, et avoir des gardes g_1, et g_2 telles qu'il existe un vecteur d'horloge satisfaisant à la fois g_1 et g_2*
- *il existe un unique état initial (L'unicité de l'état initial est assurée par l'unicité du lieu initial et la mise à zéro de toutes les horloges en début d'exécution)*

Nous estimons que vu l'objectif de modélisation, il n'est pas essentiel de considérer le cas de spécifications définies à travers des automates temporisés non-déterministes.

Nous souhaitons utiliser ce modèle pour représenter l'existence de choix exclusifs dans la définition de comportement temps réels. Cette phénomène de choix exclusif est illustré dans sur l'exemple de la figure 3.2. Seul un des deux scénarios peut effectivement être exécuté mais il est tout de même possible de factoriser la définition des contraintes temporelles.

À retenir :

- L'état d'un automate temporisé est une paire liant un lieu, et un vecteur d'horloge.
- La classe des automates temporisés considérée se limite aux automates déterministes.
- L'espace d'état de l'automate est à priori infini. Les transitions dans ce systèmes sont de deux types : événementielles ou temporelles. Le franchissement d'un arcs de l'automate correspond à une transition événementielle. Une transition temporelle correspond à la modification induite par l'écoulement du temps sur le vecteur d'horloge d'un état.
- Les automates temporisés permettent de représenter très naturellement des contraintes de sûreté pour interdire certains comportements. L'ajout de l'obligation d'atteindre certains lieux pour terminer une exécution permet de spécifier des contraintes de vivacité bornée assez simplement.

3.2 Synthèse du détecteur au plus tôt

Le vérifieur que nous souhaitons construire est informé de l'occurrence de chaque événement au moment de sa date d'occurrence. Cela revient à faire l'hypothèse d'un mécanisme d'observation monotone et ayant une latence nulle pour raisonner sur la réalisation du bloc d'analyse. Un automate temporisé est un modèle exécutable. Une méthode relativement éprouvée pour la vérification en ligne consiste à exécuter en parallèle du

système le modèle et à vérifier si les états atteints lors de cette exécution sont valides. La validité de ces états dépend de l'accessibilité des états finals.

3.2.1 Des symptômes vers l'analyse d'accessibilité

Parmi les problèmes classiques de vérification de modèles, il est possible de réaliser deux types d'analyses : vérifier qu'une trace peut être exécutée sur un automate (simulation), et déterminer si un état peut être atteint à partir d'un autre (accessibilité). Dans notre cas, ces deux types d'analyses devront être réalisés puisqu'une trace n'est valide vis à vis de l'automate que si cette dernière peut être exécutée sur l'automate et si elle place ce dernier dans un état final. Il en découle que tout préfixe d'une trace valide est exécutable par l'automate et place ce dernier dans état depuis lequel il est possible d'atteindre au moins un état final.

L'exécution de tout symptôme d'erreur place l'automate dans un état où l'une de ces deux conditions n'est pas remplie. Tout état de l'automate ne vérifiant pas au moins l'une de ces deux conditions est un état erroné. Les arcs de l'automate et les gardes qui leurs sont associées définissent pour tout états ses successeurs. Une transition peut exister et cependant avoir pour destination un état erronés lorsque l'état atteint ne vérifie plus la propriété d'accessibilité. Cependant, tout état atteint depuis un état erroné sera nécessairement erroné. A partir de tout état correct, il existe trois cas de figure correspondant à l'apparition d'une erreur :

- l'observation d'un événement suggère une transition événementielle qui n'existe pas dans le lieu occupé par l'état courant.
- l'observation d'un événement suggère une transition dont la garde n'est pas satisfaite par les valeurs des horloges de l'état courant.
- la mise à jour de l'état à la suite de l'écoulement d'une durée d a pour état de destination un état à partir duquel il est impossible d'atteindre un quelconque état final.

Lors de la vérification en ligne, la possibilité d'exécuter la trace observée se vérifie au fur et à mesure. Pour chaque nouvel événement ou durée à prendre en compte, il suffit de vérifier premièrement que la transition en question existe et que l'état atteint permet d'atteindre un état final. La dernière analyse est particulièrement complexe mais peut être pré-caculée et son résultat mémorisé. Ce pré-calcul permet d'échanger l'usage d'une ressource de calcul pour celui d'une ressource de mémoire. Trouver les états à partir desquels un état final peut être atteint correspond à déterminer l'ensemble des états co-accessibles depuis les états finals.

Une technique d'analyse des questions de co-accessibilité consiste à définir un graphe mémorisant le résultat de cette analyse. On parle alors d'abstraction temporelle de l'espace d'état de l'automate. Il existe de nombreuses méthodes de partitionnement de l'es-

pace d'état dont on trouve un panorama assez large dans [TY01, DT98]. On peut distinguer deux grandes familles de partitionnements de l'espace d'état.

- Les partitionnements optimisés pour traiter les questions liées à un parcours avant du graphe, permettent de déterminer l'ensemble des états accessibles à partir d'un état ou d'un ensemble d'état.
- Les partitionnements optimisés pour traiter les questions liées à un parcours arrière du graphe, permettent essentiellement de déterminer l'ensemble des état à partir desquels il est possible d'atteindre un ensemble pré-défini d'états.

Dans notre cas, le second type de partitionnement semble le plus indiqué pour pouvoir déterminer les état d'erreur.

L'abstraction temporelle forte crée une partition de l'espace d'état de telle sorte que l'on soit capable de dire dans chaque classe quelles sont les transitions effectivement franchissables dans le futur de l'exécution courante.

Définition 12 (Abstraction temporelle forte ([AM04, TY01]) *$\mathcal{P}_0$ est un ensemble de classes d'équivalence pré-défini qui permet d'initialiser le partitionnement de l'espace d'état. Ici $\mathcal{P}_0$ contient a priori autant de classes que de lieux finals. Chaque classe regroupe les états correspondants à un lieu final donné. Puis, les classes d'équivalence de l'abstraction temporelle forte sont définies de manière récursives comme suit :*
C est une classe d'équivalence assurant le principe d'abstraction temporelle forte si :

- *$C \in \mathcal{P}_0$*
- *La classe contient un ensemble d'états tels que pour toute paire d'états (p ; p') de C (à savoir p est équivalent à p'), alors les deux conditions suivantes sont vérifiées :*
  - *Pour toute transition due à un événement depuis p vers un état q, $p \xrightarrow{a} q$, il existe q' équivalent à q tel que $p' \xrightarrow{a} q'$ soit une transition valide dans l'automate.*
  - *Pour toute transition due à l'écoulement du temps depuis p vers un état q, $p \xrightarrow{d} q$, si q n'appartient pas à C mais à une classe C', alors il existe un état q' équivalent à q et une durée d' tels que $p' \xrightarrow{d'} q'$.*

La définition des classes permet de réaliser une partition de l'espace d'état à partir de laquelle il est possible de construire un graphe fini ; le graphe quotient de l'abstraction temporelle. Par abus de langage nous utiliserons «abstraction temporelle» pour désigner directement ce graphe. Nous n'avons pas de contribution concernant les algorithmes de calcul de cette abstraction. Nous utilisons l'outil KRONOS [BDM+98] pour calculer cette abstraction à partir de la description d'un automate temporisé.

Définition 13 (Graphe quotient de l'abstraction forte) *On suppose définie la partition $\mathcal{P}_{\mathcal{A}}$ sur l'espace d'état d'un automate temporisé $\mathcal{A}$ de telle sorte que les états associés à chaque lieu final forment une classe d'équivalence distincte pour chaque lieu. La partition de l'espace d'état est construite à partir de ces classes pré-définies.*

Le graphe quotient issu de $\mathcal{P}_{\mathcal{A}}$ est un graphe orienté et étiqueté dont les arcs représentent des changements d'état impliquant un changement de classe d'équivalence. Les arcs sont étiquetés soit par des événements, soit par le symbole spécial τ modélisant une transition temporelle abstraite dont la durée n'est pas spécifiée :

transition événementielle de classe : *Pour tout événement a, l'arc $C \xrightarrow{a} C'$ est dans le graphe quotient si et seulement s'il existe une paire d'états (s,s') telle que $s \xrightarrow{a} s'$, avec s dans C et s' dans C'.*

transition temporelle de classe : *l'arc $C \xrightarrow{\tau} C'$ est dans le graphe quotient si et seulement si il existe une paire d'états (s,s') et une durée d telle que $s \xrightarrow{d} s'$, avec s dans C et s' dans C'.*

En procédant de la sorte, on obtient le graphe détaillé dans la figure 3.3 pour l'exemple du serveur de calcul. Sur ce graphe, on peut noter que le lieu *wait_serv* est décomposé en trois classes bien distinctes. La première classe d'états contient l'ensemble des états avec un vecteur d'horloge tel que x soit strictement inférieur à 15. Depuis ces états, les deux transitions actives dans le lieu *wait_serv* peuvent être franchies inconditionnellement avec l'assurance que l'état de destination de chacune de ces transitions est un état co-accessible depuis un état final. La seconde classe d'équivalence contient tous les états dont l'horloge x vérifie : $15 \leq x < 25$. Dans ces états, seule la transition portant l'événement *get_2nd* est autorisée. En effet, le franchissement de la transition associée à l'événement *get_prio* nous placerait dans un état vérifiant à coup sur x-y > 15. Du coup la conjonction $x < 20 \wedge y > 5$ devient impossible à satisfaire pour l'état atteint juste après la transition en question et tous ses successeurs. Le franchissement de la transition étiquetée par *get_prio* placerait le système dans un lieu non-final sans possibilité de franchir la moindre transition événementielle. La dernière classe d'états du lieu *wait_serv* correspond à des états dont le vecteur d'horloge possède une valeur supérieure à 25 pour x. Dans ce cas, les transitions sur *release* seraient désactivées. Ceci correspond au dépassement d'une échéance. Le système passe d'une classe à l'autre par simple écoulement du temps. Le fait que la troisième classe ne possède aucun état co-accessible depuis un état final fait qu'un signal d'erreur doit être au moment précis où l'état du système change et appartient à cette classe d'états.

Pour réaliser l'analyse de co-accessibilité de manière efficace, la fonction $Pred_a(C)$ retourne l'ensemble des prédécesseurs des états de C à travers une transition événementielle causée par l'événement a. De la même manière, $Pred_\tau(C)$ représente l'ensemble des prédécesseurs de C à travers une $\tau - transition$. Puisque les classes d'équivalence définissent une partition de l'espace d'état, cela signifie qu'il devient possible de définir une fonction qui retourne pour chaque état la classe à laquelle il appartient, soit Classe(.) cette fonction. La fonction abs_τ transforme une trace temporisée en une trace classique pour laquelle tous les symboles de durée ont été remplacés par le symbole τ Il est possible

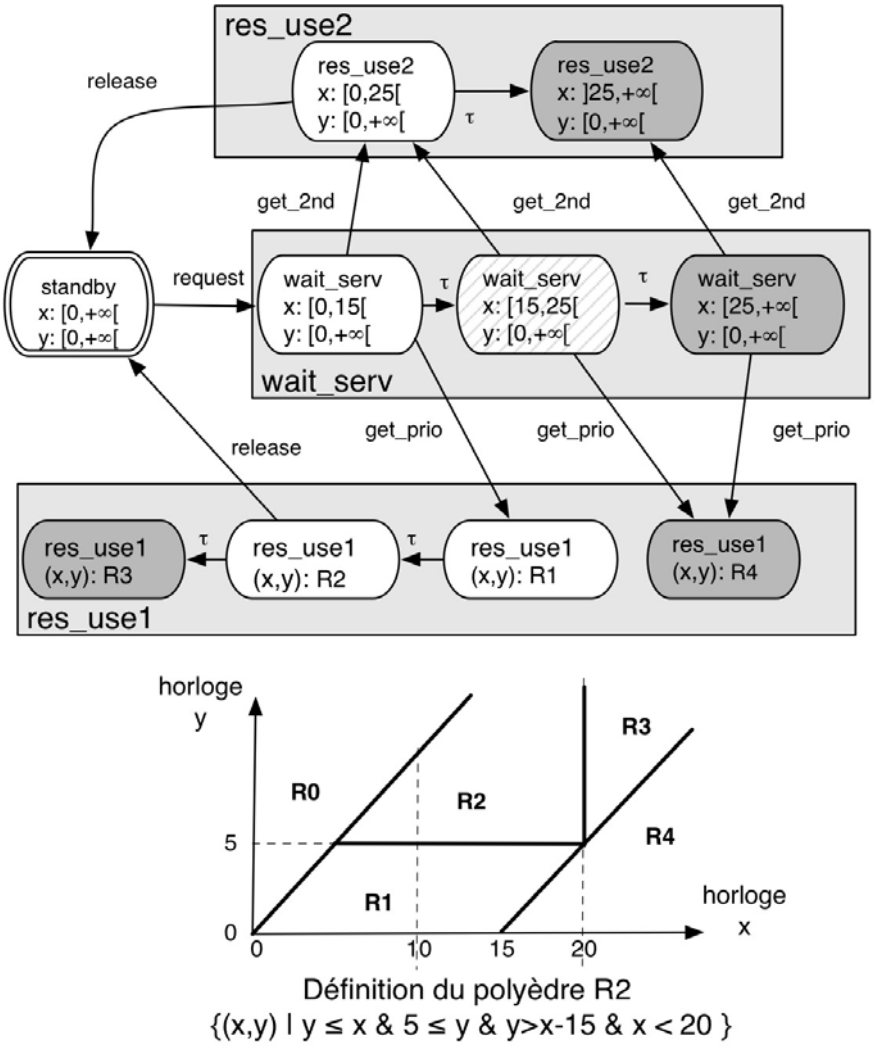

FIG. 3.3: Graphe représentant une abstraction temporelle forte de l'automate du serveur.

d'exécuter une trace sur l'automate mais aussi sur l'abstraction temporelle. Normalement la classe atteinte en simulant l'exécution exacte doit correspondre à celle atteinte par l'exécution de la trace sur l'abstraction temporelle.

Propriété 3 (Equivalence de chemins d'exécution) *Une des propriétés élémentaires du graphe quotient est que s'il existe une trace u menant l'automate de l'état s à l'état s′, alors il existe un parcours du graphe quotient de la classe Classe(s) vers la classe Classe(s′) tel que la séquence d'étiquettes rencontrée lors de ce parcours soit égale à la trace* $abs_\tau(u)$[2].

La réciproque de cette propriété est aussi vraie : S'il existe une trace u dans l'ensemble des traces abstraites $\tau.(\Sigma.\tau)^* + (\Sigma.\tau)^*$ telle que u permet de franchir un chemin dans le graphe quotient menant de la classe C à la classe C', alors pour tout état s de la classe C, il existe une trace temporisée v telle que l'état de destination de v est dans C' et $abs_\tau(v) = u$.

Ce concept d'équivalence est directement lié aux propriétés de l'abstraction, [TY01]. Ainsi, un état final peut être atteint à partir d'un état donné uniquement si un chemin relie les classes d'équivalences correspondantes à ces états dans le graphe quotient. Une analyse classique de co-accessibilité dans un graphe fini peut être appliquée sur l'abstraction temporelle pour déterminer si oui ou non ces dernières sont co-accessibles d'au moins une classe d'états finals. Après cette analyse les nœuds du graphe quotient sont étiquetés par un booléen définissant le résultat de ce test de co-accessibilité. Pour la suite, nous dirons qu'un état est OK si il est co-accessible depuis un quelconque état final. De manière similaire, un état est KO, si il n'est co-accessible depuis aucun état final. Cette convention sera aussi adoptée pour les classes d'équivalence d'états.

D'un point de vue pratique, nous utilisons l'outil KRONOS pour calculer l'abstraction temporelle de l'automate. Les classes d'équivalence d'états sont définies de manière symbolique par la combinaison d'un lieu et la définition d'un polygone convexe de vecteurs d'horloges. Ce type de partitionnement n'est pas forcément optimal mais permet une implémentation efficace de l'algorithme calculant l'abstraction temporelle. Le résultat obtenu est la définition textuelle du graphe quotient. La construction de ce graphe telle qu'elle est réalisée dans KRONOS est relativement efficace puisqu'elle se repose sur un principe de minimisation du nombre de classes d'équivalence. Le graphe quotient présenté dans la figure 3.3 est le résultat obtenu après l'utilisation de KRONOS. Les nœuds grisés représentent les ensembles d'états qui ne sont pas co-accessibles depuis un état final. Il reste maintenant à expliquer précisément comment ce modèle est utilisé en ligne pour réaliser toutes les étapes de vérification.

[2] Vu la définition de la fonction $abs_\tau()$ des transitions stationnaire τ rebouclent sur chaque classe

3.2.2 Etude du comportement en-ligne

La section précédente nous a permis de voir comment l'on pouvait précalculer le test de la violation de la propriété d'accessibilité des états finals. Un plus petit symptôme d'erreur correspond à une trace dont tout préfixe stricte est exécutable et telle soit la trace est non exécutable sur l'automate, soit son état de destination est non co-accessible.

Le détecteur est naturellement activé chaque fois qu'un événement se produit. En effet, tout événement provoque a priori un changement de classe. Ceci correspond potentiellement à l'observation d'un plus petit symptôme d'erreur. Le détecteur peut alors vérifier que l'événement observé correspond à une transition autorisée de l'automate grâce à l'abstraction temporelle. Il reste à s'assurer du réveil du détecteur lorsque l'exécution en cours correspond à un plus petit symptôme d'erreur finissant par une durée. Pour pallier le fait que le processus d'observation ne se réveille qu'à chaque occurrence d'un événement, il est possible de plannifier dans le bloc d'analyse la capture et le signalement d'un tel symptôme en utilisant une projection dans le futur.

Cette approche est relativement similaire à la génération d'un contrôleur permettant de résoudre des objectifs de sûreté, et des objectifs d'accessibilité. Habituellement l'ensemble des événements est coupé en deux : des événements contrôlables d'une part et d'autres non contrôlable d'autre part. Puis, l'objectif de contrôle est défini en spécifiant une liste d'états à atteindre ou éviter en toute circonstances. Le rôle du contrôleur est de forcer l'occurrence des événements contrôlable pour satisfaire l'objectif de contrôle. Un détecteur au plus tôt peut être vu comme un contrôleur assurant la propriété de détection au plus tôt. Cependant, il nous semble que l'objectif de contrôle du détecteur ne peut s'exprimer d'une manière simple. Tout d'abord, il faut déterminer les actions contrôlables. En tout lieu de l'automate, il existe un unique arc contrôlable étiqueté par l'événement signalant l'erreur, noté *ERR*, dont la destination est un lieu puits noté R (pas de transitions sortantes). Le problème de contrôle se pose en des termes assez inhabituels : trouver la stratégie de contrôle spécifiant l'occurrence des événements *ERR* telle que dès qu'aucun état final n'est plus accessible, le lieu *R* doit être atteint au plus tôt. La génération de contrôleur temporellement optimaux a déjà été résolue dans [AM99, BHPR07] mais ne peut directement être appliquée ici. Dans notre cas, le problème provient du fait que les arcs *ERR* sont tout le temps actifs. il faudrait conditionner leur franchissement par la condition « dès qu'aucun état final n'est plus accessible» pour pouvoir utiliser les algorithmes déjà proposés. Il nous semble que les objectifs de contrôle considérées sont assez simples, mais que les conditions sur l'activation des actions de contrôle sont plus complexes que celles considérées dans les résultats connus sur la synthèse de contrôleur. Pour ces raisons nous avons développé notre propre stratégie pour déclencher l'occurrence des signaux d'erreur. Cependant, il est normal que notre solution ait de fortes ressemblances avec les contributions existantes dans ce domaine puisque nous utilisons exactement les

mêmes concepts fondamentaux.

Revenons au problème de la capture des plus petits symptômes d'erreur terminés par une durée et non un événement. Supposons que le processus d'observation correspond à un état correct de l'automate. En bref, si à partir d'un état correct, il est possible par simple écoulement du temps d'atteindre un état erroné, alors il existe une unique durée *ddl* qui ajoutée à la trace représentant l'état «correct» forme un plus petit symptôme d'erreur. Chaque fois que l'état du bloc d'observation est mis à jour, le bloc d'analyse va vérifier la validité de l'événement nouvellement observé. Notons u la trace modélisant l'état du processus d'observation après sa mise à jour. Si l'événement est validé, alors la trace u correspond à un état correct de l'automate. Il est alors possible de vérifier l'existence d'une durée, noté *ddl*, telle que $u.ddl$ est un plus petit symptôme d'erreur. Cette durée n'existe pas nécessairement. Par exemple, lorsque u place le système dans un lieu final, alors le système peut y rester a priori indéfiniment. Lorsque cette durée existe, elle est ajoutée à la date courante pour programmer une alarme utilisée pour signaler l'apparition de l'erreur correspondant à $u.ddl$ dans le cas où effectivement aucun événement ne serait observé entre la date courante, et la date de réveil de l'alarme. Si un événement se produit entre la mise en place de l'alarme et son expiration, cela signifie que l'exécution justifiant le signalement de l'erreur ne se produira pas. Il est impératif de désactiver l'alarme précédemment configurée en attendant le calcul de la nouvelle échéance. Ainsi après chaque événement valide trois choses peuvent se produire dans un futur proche.

1. Un événement valide est à nouveau observé avant expiration de l'alarme. Puisque l'événement est valide, il faut mettre à jour l'alarme en fonction du nouvel état de l'automate.
2. Un événement invalide est observé avant expiration de l'alarme et déclenche un signal d'erreur. L'erreur ayant déjà eu lieu il faut désactiver l'alarme pour éviter tout effet "d'écho" pour le signalement des erreurs.
3. Aucun événement n'a eu lieu depuis la configuration de l'alarme et cette dernière expire. Cela signifie qu'un plus petit symptôme d'erreur finissant par une durée non nulle a été atteint (cf figure 3.4).

Ainsi, le processus d'analyse des événements et l'alarme sont en concurrence. Ils s'exécutent tous les deux en parallèle et collabore pour capturer les deux types de symptômes d'erreur. Le scénario d'exécution représenté dans la figure 3.5 représente le premier cas tout d'abord un événement correct est perçu et place l'automate dans un état où il existe une échéance. Heureusement, un événement valide est observé avant l'expiration de l'alarme précédemment configurée. Lors de l'occurrence d'un événement, il faudra successivement réaliser les étapes suivantes :

1. désactiver l'alarme en cours

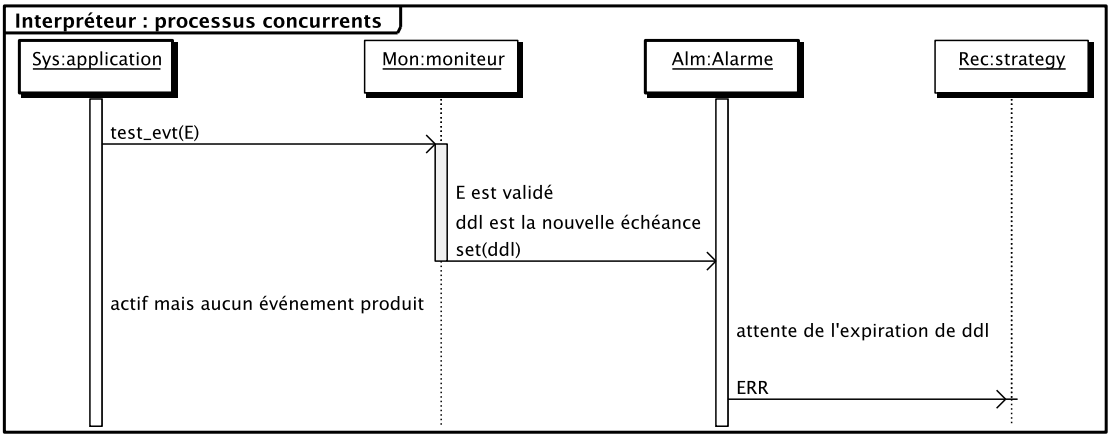

FIG. 3.4: Expiration de l'alarme permettant la détection d'une erreur temporelle

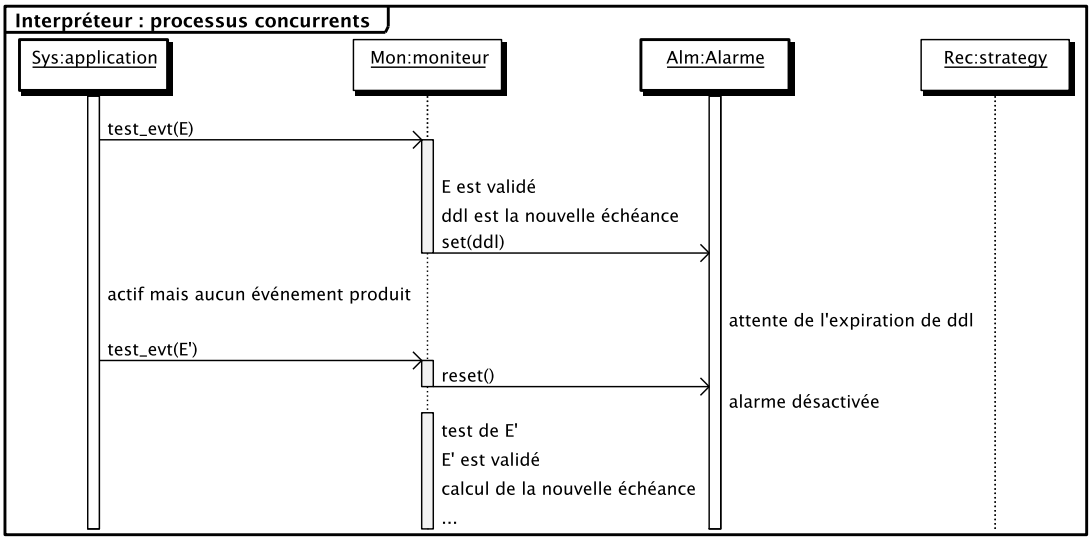

FIG. 3.5: Désactivation de l'alarme à travers l'occurrence d'un événement

2. tester la validité de l'événement reçu. En cas d'échec à ce test, déclencher l'émission du signal ERR et bloquer l'exécution de l'application en attente d'un recouvrement.
3. calculer la nouvelle échéance et régler une alarme en conséquence

Ce principe de vérification en ligne fait l'hypothèse d'un traitement atomique des trois étapes décrites ci-dessus. En fait puisque l'on raisonne sur un état instantané et que l'on configure une alarme, l'hypothèse exacte est celle d'une exécution atomique en temps nul. Pour implémenter ce principe de vérification, il faudra tenir compte du temps de calcul de chaque étape dans la configuration de l'alarme. Les deux sous-sections suivantes détaillent comment sont réalisés le test de validité d'un événement, et le calcul de la durée servant de base pour la configuration de l'alarme.

3.2.3 Test de validité d'un événement.

Dans les sous sections précédentes, nous avons vu dans un premier temps ce que devait réaliser l'interpréteur. Son comportement a été décomposé en différentes étapes.

1. désactiver l'alarme en cours
2. tester la validité de l'événement reçu. En cas d'échec à ce test, déclencher l'émission du signal ERR et bloquer l'exécution de l'application en attente d'un recouvrement.
3. calculer la nouvelle échéance et régler une alarme en conséquence

Les actions 2 et 3 dépendaient d'une analyse des propriétés de co-accessibilité des différents états de l'automate depuis l'ensemble des états finals. Nous avons vu comment pré-calculer ces propriétés . Pour déterminer la validité d'un événement, l'événement doit être soumis à trois tests. On suppose que l'état courant de l'automate au moment de la production de l'événement est un état correct, i.e. de type OK. A savoir, cet état ne correspond ni à l'exécution d'un plus petit symptôme d'erreur, ni à celle d'une trace erronée.

- Premièrement, il faut trouver une transition franchissable par cet événement dans l'automate. Cela veut dire qu'il y a au moins une transition partant du lieu correspondant à l'état courant avec pour étiquette l'événement à tester. Parmi ces transitions, la garde d'au moins l'une d'elle doit être satisfaite par le vecteur d'horloge de l'état courant. Si ces deux conditions sont remplies, le premier test est passé. Ce test correspond aux contraintes de sûreté dans l'automate. Si l'une de ces conditions n'est pas satisfaite, alors la trace qui était jusqu'à présent correcte deviendrait par l'exécution de cet événement, un plus petit symptôme d'erreur.
- Deuxièmement, il faut que la modification de l'état du système soit en accord avec la contrainte de vivacité. Ainsi, le nouvel état doit lui aussi être un état OK. En effet,

si aucun état final ne peut être atteint depuis ce nouvel état, la transition transforme la trace observée jusqu'à présent en une trace erronée qui, elle aussi, est un plus petit symptôme d'erreur.

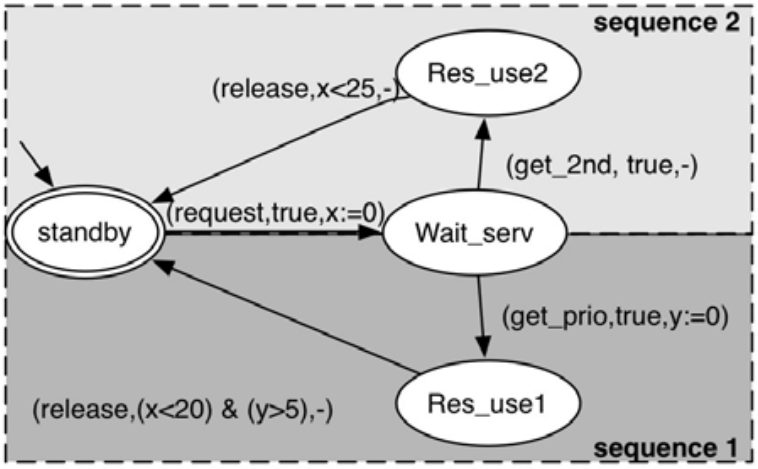

FIG. 3.6: Rappel de l'automate temporisé modélisant un serveur

Trois scénarios d'exécution de l'automate représenté figure 3.6 peuvent illustrer ce processus de vérification d'un événement à partir d'un état correct. La trace 3.*request*.2 place l'automate dans l'état $(Wait\_serv, [2, 5])$ qui est correct puisque l'on peut revenir dans le lieu *standby* en poursuivant l'exécution par *get_2nd*.3.*release*. Cependant, si l'événement *release* est produit par l'automate dans cet état, le premier test ne passe pas et une erreur est déclenchée. En effet aucune transition sur l'événement *release* n'est définie dans le lieu *wait_serv*. Un autre exemple permettant de mettre en défaut le premier test repose sur un conflit entre l'état courant et les gardes sur les transitions. L'application exécute la trace 3.*request*.2.*get_prio*.4 et se place donc dans un état correct, puis émet l'événement *release*. Bien que l'événement *release* soit utilisé dans une transition de res_use1 vers *standby*, la production de l'événement *release* seulement 4 unités de temps après l'événement *get_prio* n'est pas compatible avec la garde, $x < 20 \wedge y > 5$, de la seule transition définie dans l'état courant, $(res\_use1, [x, y] = [6, 4])$. Le second test est mis en défaut par un scénario d'exécution où l'automate est amené dans un état où *get_prio* n'est plus autorisé bien que la transition soit active. La trace 2.*request*.16 place l'automate dans un tel état. Si l'événement *get_prio* venait à être produit, cela placerait l'automate dans le lieu res_use1 avec $x > 15$ et $y = 0$. En franchissant cette transition, la transition sur l'événement *release* est définitivement désactivée dans le lieu res_use1. L'état de destination d'une telle transition est donc non co-accessible depuis un état final, i.e. un état KO.

Ces deux vérifications peuvent être réalisées à travers les raisonnements suivants :

- Si l'état courant autorise une transition sur l'événement soumis au test, alors un arc portant cet événement sort de la classe d'équivalence correspond à l'état courant. Ceci est assuré par définition du graphe quotient.
- Si une transition est autorisée dans l'état courant alors l'état de destination est co-accessible depuis un état final si et seulement si la classe de cet état est co-accessible.

Le premier test devient donc équivalent à retrouver la classe d'équivalence à laquelle l'état courant appartient, puis tester l'existence d'un arc sortant de cette classe et portant l'événement en question. Si un tel arc existe le test est passé et l'on peut procéder à la deuxième vérification. Dans le cas contraire la transition est interdite.

Ainsi, l'état de destination peut effectivement être atteint d'après le premier test. Il faut maintenant vérifier que l'état atteint est bien co-accessible d'un état final de l'automate. En utilisant le principe d'équivalence sur les chemins entre le graphe quotient et l'espace d'état de l'automate, l'appartenance de cet état à une classe d'équivalence étiquetée «OK» permet de s'assurer de cette co-accessibilité. La seconde activité à mener dans le bloc d'analyse concerne la configuration de l'alarme pour le signalement des erreurs dues à un plus petit symptôme s'achevant sur une durée.

3.2.4 Calcul des échéances à partir de l'abstraction temporelle

Pour détecter l'ensemble des erreurs au plus tôt, il faut estimer après chaque événement valide si la trace courante peut générer un plus petit symptôme d'erreur à travers une continuation unitaire associée à l'écoulement du temps, tel que nous l'avons expliqué dans la sous-section 3.2.2.

Cela revient à savoir, s'il est possible de transformer la trace courante en un plus petit symptôme d'erreur par l'ajout d'une durée, notée ddl, à la trace courante. Cette durée ddl se définit comme la durée qui permet depuis l'état courant s de franchir la transition temporelle $s \overset{ddl}{\rightarrow} s'$ telle que :

- l'ensemble des successeurs temporels stricts de s' sont des états erronés :$\forall \epsilon > 0, (s \overset{ddl+\epsilon}{\rightarrow} s") \Rightarrow (s" \in KO)$
- l'ensemble des prédécesseurs temporels stricts de s' sont des états valides :$\forall 0 < \epsilon < ddl, (s \overset{ddl-\epsilon}{\rightarrow} s") \Rightarrow (s" \in OK)$

L'ensemble des successeurs temporels d'un état se répartit naturellement dans un nombre fini de classes d'équivalence. L'ensemble des successeurs temporels d'un état sont strictement ordonnés chronologiquement. Il en est de même pour les classes les contenant : on dira qu'une classe C précède une classe C' si un arc τ (transition temporelle abstraite) permet de passer de la classe C à la classe C'. La relation définie ainsi constitue un ordre

total sur l'ensemble des successeurs temporels d'une classe, à savoir toutes les classes identifiées comme successeurs temporels d'une même classe C doivent pouvoir être ordonnées.

En effet, si $C \xrightarrow{\tau} C'$ et $C \xrightarrow{\tau} C$", alors pour tout s dans C il existe d' et d" deux durées telles que :

- L'écoulement de la durée d' permet de passer de l'état s à l'état s' tel que s' est dans C'.
- L'écoulement de la durée d" permet de passer de l'état s à l'état s" tel que s" est dans C".

Puisque l'écoulement du temps entraîne un changement d'état déterministe, ainsi si C' est différent de C" alors la durée d' est différente de la durée d". Si $d' < d$" alors il existe d_{inter} tel que d" $= d_{inter} + d'$. La transitivité des changements d'état dus à l'écoulement du temps assure que $s' \xrightarrow{d_{inter}} s$" et donc l'existence d'un arc τ entre C' et C". Un raisonnement similaire peut être tenu si d" $< d'$, alors un arc τ symbolisera les transitions temporelles de C" vers C'.

La durée *ddl* est calculée de la manière suivante en utilisant cette propriété d'ordre total :

- Pour tout état s atteint par l'exécution d'une trace u déjà vérifiée, on récupère la classe à laquelle s appartient : $Classe(s)$.
- Si parmi les successeurs temporels de Classe(s), il existe une classe d'états KO alors il existe bien une durée *ddl* telle $u.ddl$ est bien un plus petit symptôme d'erreur (puisqu'un ensemble non vide totalement ordonné possède toujours une borne inférieure).
- Si *ddl* existe, prendre le plus grand successeur temporel de $Classe(s)$ qui contiennent des états OK, noté $MaxOK(s)$.
- La durée *ddl* correspond à la valeur de d telle que si s correspond au lieu l_s et au vecteur d'horloges γ_s, et P est l'ensemble des vecteurs d'horloges des états contenus dans $MaxOK(s)$, alors $ddl = Sup\{d|\gamma_s + d.(1, \cdot, 1) \in P\}$ à savoir la durée permettant d'atteindre le bord de P séparant P d'un ensemble de vecteurs d'horloges correspondant à une classe des états KO.

Le reste du calcul dépend de la nature de l'ensemble P des vecteurs d'horloges de la classe $MaxOK(s)$, et de la façon dont cet ensemble est décrit. L'ensemble des vecteurs d'horloge possède une structure d'espace vectoriel, et le terme de vecteur d'horloge n'avait pas été choisi innocemment puisqu'il est possible de définir les ensembles de vecteurs d'horloge associés aux classes d'équivalence d'états à travers des objets et opérations géométriques.

L'abstraction temporelle ne regroupe que des états qui correspondent au même lieu. Les états regroupés au sein d'une classe d'équivalence ne se distinguent donc que par leur vecteurs d'horloge. Ainsi, lieu par lieu, les classes d'équivalence d'états définissent une

partition de l'ensemble des vecteurs d'horloge. La nature des gardes et des invariants permet de définir de manière symbolique[3] l'ensemble des vecteurs d'horloge d'une classe. Le critère pour séparer deux états et donc deux vecteurs d'horloge dans deux classes différentes repose sur une différence entre les gardes satisfaites par l'un et non par l'autre dans l'état ou ses futurs. Ainsi les séparations observables dans l'espace des vecteurs d'horloges sont essentiellement définies par deux types d'équations : $x = c$ ou $x - y = c$ avec x et y deux horloges distinctes et c une constante. Ainsi, l'ensemble des vecteurs associés à une classe d'équivalence, noté $vect(C)$, est un polyèdre dans l'espace des vecteurs d'horloge. Dans l'exemple du serveur de calcul, la décomposition de l'ensemble des vecteurs d'horloge associés au lieu res_use1 est illustré dans la figure 3.7

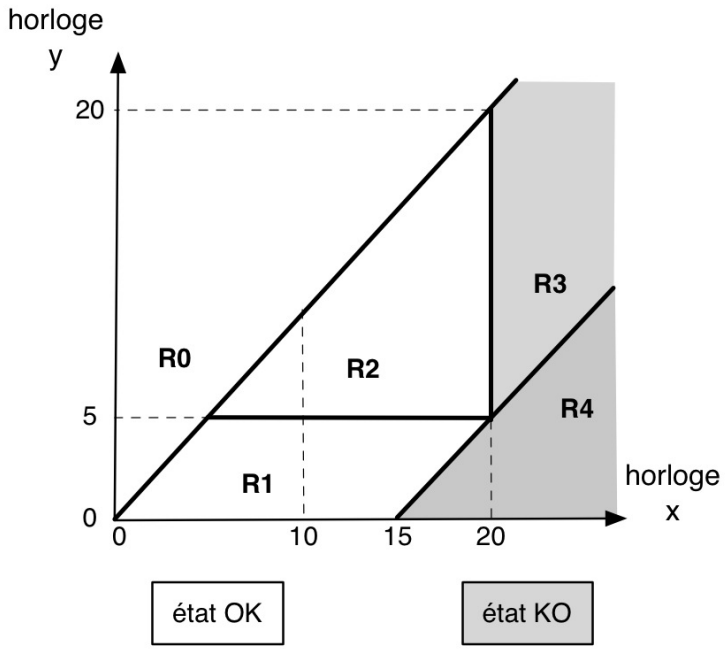

FIG. 3.7: Décomposition de l'espace d'horloge de res_use1 selon les deux horloges x et y.

L'outil KRONOS utilisé pour construire une abstraction temporelle forte de l'automate ne manipule que des polyèdres convexes. Ainsi, même lorsqu'il est possible de regrouper certains vecteurs d'horloge, l'outil conserve des classes d'équivalence bien séparées pour conserver des ensembles de vecteurs d'horloge convexes. Cette restric-

[3] à travers une ou plusieurs inéquations

tion permet une définition symbolique très efficace de ces polyèdres comme l'ensemble des vecteurs d'horloges solution d'un système d'inéquations linéaires du type $x \sim c$, ou $x - y \sim c$ avec $\sim$ pris dans $\{<\leq, >, \geq\}$.

Soit P un polyèdre convexe borné sur au moins l'un des axes[4]. Pour tout point v de ce polyèdre, l'écoulement du temps déclenche une translation selon le vecteur $(1, \cdot, 1)$. On défini le point de sortie de P depuis v par écoulement du temps comme le point $p_s = v + d.(1, \cdot, 1)$ tel que d est la borne supérieure des valeurs λ vérifiant "$v + \lambda.(1, \cdot, 1)$ est dans P". Plus simplement p_s est la projection sur les bords de P de v selon le vecteur $(1. \cdots .1)$. Cette projection est illustrée dans la figure 3.8 Il découle de cette définition que le point de sortie du vecteur d'horloge de MaxOK(s) correspond à l'état dans lequel il faut réveiller le détecteur. Ainsi, si $p_s^+(\gamma_s)$ représente ce point de sorti, alors ddl correspond au facteur d tel que $p_s^+(\gamma_s) = \gamma_s + d(1. \cdots .1)$.

On peut visualiser ce phénomène en dimension deux dans l'exemple du serveur pour le lieu res_use1.

Dans notre cas, les frontières des polyèdres sont perpendiculaires aux axes (x=c), ou contiennent un vecteur parallèle au vecteur $(1, \cdot, 1)$. Ainsi, les points de sortie se situent tous sur des hyperplans définis par une équation du type $(x = c)$. Il est possible de représenter symboliquement l'ensemble des points de sortie d'un polyèdre P par un vecteur collectant pour chaque horloge la constante c (positive) correspondant à l'équation de l'hyperplan définissant la frontière sortante de P selon cet axe. Si P n'est pas borné selon un axe donné, alors le symbole $+\infty$ est pris comme valeur par défaut. Si l'on note $BorderSup(P)$ le vecteur représentant les faces de sortie de P, alors chaque coordonnée du $(BorderSup(MaxOK(s)) - \gamma_s)$ représente le temps pouvant s'écouler selon chaque horloge. La plus petite coordonnée représente la durée servant à définir l'échéance que l'on souhaite capturer.

Pour chaque classe pouvant être retournée par la fonction $MaxOK$, il faut calculer son vecteur de sortie. À l'exécution, le calcul de ddl se déroule en deux temps :

- Récupération de MaxOK(s) avec s l'état courant.
- $ddl = min(BorderSup(MaxOK(s)) - \gamma_s))$

Cette méthode possède un intérêt particulier lorsque l'automate possède un nombre non trivial d'horloges. Le calcul de la date d'expiration de l'alarme après chaque événement validé est donc une opération relativement simple. À partir du moment où l'on a accès à une structure de données stockant pour chaque classe le vecteur BorderSup(MaxOK(s)).

[4] Au moins l'une des horloges est bornée

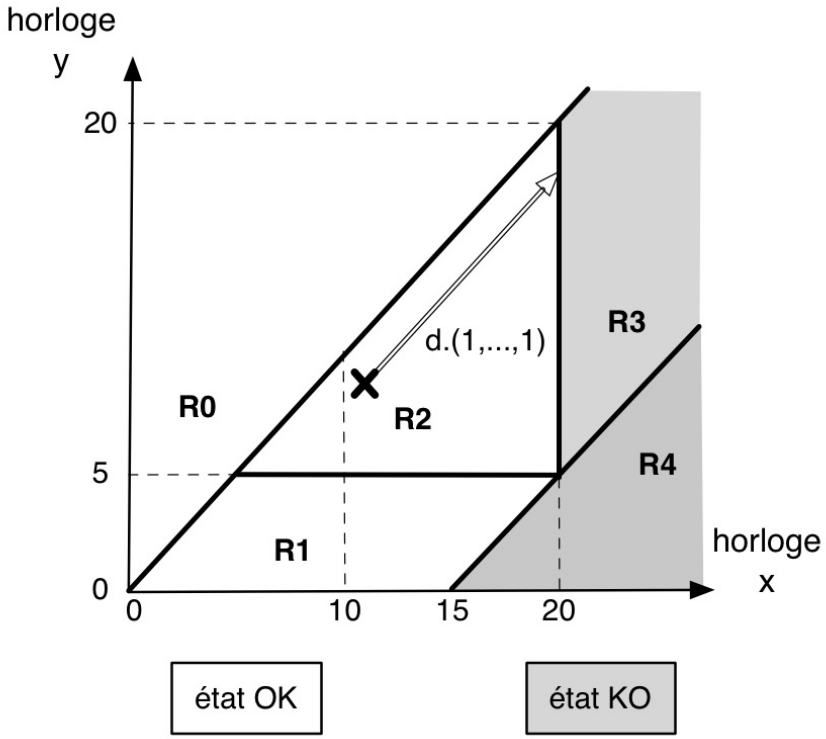

FIG. 3.8: Point de sortie de la polyèdre R2

Conclusion

Nous avons présenté dans ce chapitre une méthode pour créer, à partir d'un automate temporisé, un détecteur assurant, «en théorie», la détection causale au plus tôt des erreurs. À partir d'un automate temporisé déterministe, nous utilisons un modèle abstrait de l'automate pour pourvoir identifier les plus petits symptômes d'erreur. Lors de la création de ce graphe, l'espace d'état de l'automate est découpé en classes d'états disjointes connectés par des transitions simples sans gardes temporelles. Ce graphe permet de déterminer quelles sont les transitions événementielles et temporelles valides.

Nous avons proposé un mécanisme d'analyse des observations pour la capture de tous les plus petits symptômes d'erreur, même ceux ne pouvant être directement capturés par le processus d'observation.

Si l'on résume le processus de synthèse et l'exécution du détecteur, on obtient

Avant Pour générer le détecteur au plus tôt, il faut construire l'abstraction temporelle

forte de l'automate passé comme spécification comportementale. Puis, les classes d'équivalence sont étiquetées soit OK soit KO en fonction du résultat de l'analyse de co-accessibilité d'un état final. Pour chaque classe qui possède un successeur temporel direct qui n'est pas valide, il faut mémoriser l'information permettant de calculer à l'exécution les échéances à surveiller de manière dynamique.

Pendant À l'exécution, le moniteur possède une représentation de l'abstraction temporelle permettant de déterminer de manière instantané la validité des états de l'automate. On suppose que la plate-forme d'exécution offre un système d'alarme permettant de forcer le réveil d'une tâche ou l'exécution d'une fonction à une date précise. Lorsqu'un événement est reçu, le processus d'observation met à jour son état en précisant le temps écoulé depuis le dernier événement, ainsi que l'événement à vérifier. Cette observation déclenche le réveil du bloc d'analyse qui réalise les opérations suivantes :

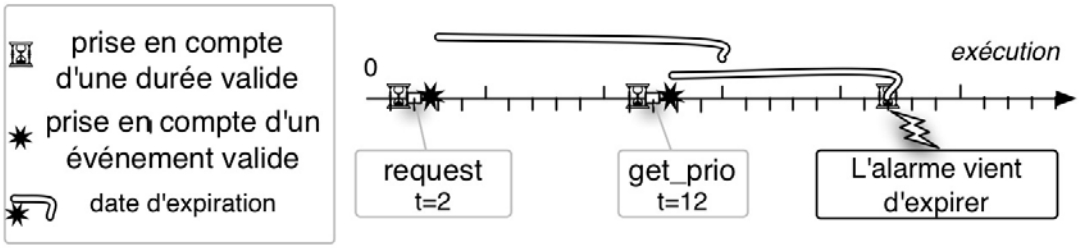

FIG. 3.9: Mise à jour de l'état du moniteur lors du processus de vérification en ligne.

- La mise à jour de la classe d'équivalence à laquelle l'état courant appartient. La connaissance de cette classe permet de vérifier la validité de l'événement.
- La vérification de l'événement se fait en testant si il existe une transition, correspondant à ce dernier, sortant de la classe courante, et que cette dernière place l'automate dans un état valide. Lorsque cette transition est valide, l'état est mis-à-jour en conséquence. La spécification décrite par l'automate génère des échéances dynamiques pouvant changer au cours de l'exécution. L'étape suivante consiste à configurer une alarme pour capturer le dépassement de ces échéances. Il est nécessaire de garder une trace de la valeur réelle du vecteur d'horloges pour permettre le calcul des échéances dynamiques.
- Le calcul de la date d'expiration de l'alarme se fait selon la méthode expliquée en section 3.2.4

La figure 3.9 permet de visualiser les états de l'automate qui sont réellement mémorisés et calculés. On peut aussi noter les différentes échéances calculées, en double trait.

Dans le scénario dépeint sur la figure, une erreur est détectée lors de la violation de la seconde échéance. Ce processus s'exécute en un temps borné puisqu'il ne contient qu'un nombre borné d'opérations de recherche dans des ensembles finis définis statiquement. La logique de vérification que nous proposons permet de notre point de vue d'assurer une détection «au plus tôt» causale assez réaliste pour des symptômes d'erreur déduit d'un automate temporisé.

Chapitre 4

Prototypage et évaluation du détecteur

Introduction

Ce chapitre présente l'étude, la réalisation, et l'évaluation d'un prototype de détecteur au plus tôt. Le détecteur signale des erreurs définies par opposition à la description des comportements acceptables de l'application via un automate temporisé. La logique de détection a déjà été présentée dans le chapitre précédant. Le détecteur sera intégré à des applications déployées dans l'environnement temps réel Xenomai. Cette implémentation a soulevé la question de la stratégie d'intégration du détecteur dans les structures de contrôle fournies par le micro-noyau temps réel utilisé dans Xenomai. L'objectif est de trouver le bon compromis entre les exigences de réactivité définies dans le chapitre précédant, et la perturbation introduite par l'activité du moniteur sur l'ensemble du système (système d'exploitation et application). Nous avons identifié les attributs permettant de déterminer le coût induit par l'exécution du détecteur (en mémoire et en temps processeur). Le prototype présenté ici illustre le compromis nécessaire entre la réactivité du processus de détection, et l'intrusivité de son implémentation. Ce chapitre sera clos par l'évaluation expérimentale du coût du détecteur en fonction de la complexité des automates vérifiés.

4.1 Les enjeux de l'implémentation

Le chapitre précédent s'est attaché à définir le comportement que doit implémenter le détecteur d'un point de vu abstrait. Deux activités concurrentes ont été identifiées pour pouvoir assurer la détection au plus tôt de l'ensemble de tous les symptômes d'erreur. D'une part le processus d'observation assure la capture et la vérification à la volée de chaque événement. D'autre part, après chaque événement validé une alarme est utilisée pour réveiller le détecteur si aucun événement n'est produit entre la configuration et l'expiration de l'alarme. Ces comportements sont définis à haut niveau, en faisant abstraction des contraintes imposées par les couches basses du système (matériel et système d'exploitation).

4.1.1 Analyse du comportement attendu

Lors de l'occurrence d'un événement la vérification est sensée se dérouler dès que le détecteur a pris connaissance de cet événement. Dans le chapitre précédant, la réalisation concrète de cette synchronisation a été temporairement ignorée. Le comportement attendu du détecteur doit au final respecter quatre règles élémentaires.

Réactivité et couverture

- Chaque événement doit être capturé de manière homogène sur l'ensemble de l'application.
- Toute vérification d'un événement doit être terminée dans un délai borné suivant sa capture.

Contrôle de l'application et recouvrement

- L'application ne doit pas pouvoir "progresser" dans son exécution entre la capture d'un événement, et la fin de la vérification de ce dernier.
- Les modalités du déclenchement de la vérification après la capture d'un événement doivent être prédictibles. En particulier l'ordre de traitement, et de production des événements doit être le même tout au long de l'exécution de l'application.

Ces exigences guident la réalisation de l'implémentation. Voici quelques exemples de conflits pouvant exister entre des mécanismes de capture et de traitement des événements, et les exigences décrites ci-dessus.

- *Capture non homogène :* Si l'on définit l'événement *e* comme l'affectation de la variable *x* dans un programme C. Deux interprétations sont possibles pour l'expression "affectation de x" : toute opération modifiant la case mémoire associée à *x*, ou toute opération modifiant le contenu d'une variable désignée par son nom, ici *x*. Si pour des raisons pratiques d'implémentation la couche d'instrumentation réalise en fonction du contexte l'une ou l'autre interprétation, alors l'événement *e* n'est pas couvert de manière homogène.
- *Capture non homogène bis :* Un autre exemple correspond au cas où l'on peut capturer les événements depuis des contextes d'exécution très variés : tâches préemptibles ou non, traitements d'interruptions ... Dans ce cas là, si l'on réalise la vérification dans une tâche, la capture d'un événement dans une tâche non préemptible peut poser un problème de latence de traitement. La prise en compte d'événements depuis des contextes pour lesquels le mode de fonctionnement de l'ordonnanceur varie peut engendrer un blocage du système d'exploitation. En effet, certains appels systèmes sont interdits en fonction du mode d'ordonnancement et du contexte appelant (tâche, traitement d'interruption, privilèges d'exécution (utilisateur/noyau))
- *Comportements erronés en présence de parallélisme :* Si l'implémentation permet aux événements d'être émis depuis plusieurs contextes d'exécution, alors ces événements peuvent être vus comme étant produits par des activités concurrentes. Dès lors si cette concurrence n'est pas prise en compte deux problèmes se posent :
  - Si la capture et la vérification peuvent être désynchronisées, alors plusieurs événements peuvent avoir été signalés mais pas encore vérifiés en présence de pa-

rallélisme.
- Si la vérification d'un événement n'est pas exclusive de toute autre vérification, alors le code de vérification peut être le sujet d'une exécution concurrente. Cela signifie que le même code est exécuté en parallèle dans deux structures de données qui partagent au moins la valeur de l'état courant de l'automate dans l'abstraction temporelle

Dans le premier cas, il y a un risque de modifier l'ordre dans lequel les événements devraient être traités. Cela peut générer des erreurs de service à travers une perception erronée de l'exécution de l'application. En effet, la trace des événements vérifiés ne correspondrait plus à celle qui a été capturée, transformant ainsi un comportement valide en un comportement erroné.
Le deuxième cas n'est simplement pas admissible.et doit être éliminé à la conception. La vérification des événements repose sur une notion d'état qui n'offre a priori aucun support pour un traitement concurrent de la vérification. En effet, si plusieurs exécutions concurrentes du code du détecteur tentent de lire et écrire une variable commune (une variable d'état), alors cette variable pourrait être corrompue. Il y a un risque ici de blocage du détecteur si cette variable correspond par exemple à une adresse mémoire.

Pour pouvoir réaliser une implémentation du détecteur, il est nécessaire de comprendre précisément l'environnement de développement choisi, Xenomai. La satisfaction des exigences énoncées ci-dessus dépend essentiellement de la manière dont on synchronise l'application et le bloc d'analyse du détecteur. Une fois le mode de synchronisation choisi, nous nous concentrerons sur la méthode choisie pour assurer les exigences de contrôle de l'exécution de l'application.

4.1.2 L'environnement Xenomai

Xenomai est le nom d'une interface de programmation destinée à la programmation d'applications temps-réel. Un micro-noyau a été réalisé pour implémenter cette interface. Par abus de langage, nous utiliserons relativement souvent Xenomai pour désigner ce micro-noyau. Le choix de la plate-forme a été guidé par trois caractéristiques de cet environnement de développement :

- Sa généricité : le micro noyau, qui implémente l'interface de programmation Xenomai, repose sur une organisation claire des différents contextes d'exécution disponibles. Cette interface supporte un modèle de tâche préemptible ou non avec priorité fixe. L'ordonnancement peut reposer sur une alarme système avec ou ou sans tics.
- Son extensibilité : de nombreuses interfaces de programmation ont été portées dans l'interface native Xenomai. Ces interfaces de programmation correspondent

à d'autres environnements de développement groupés dans une structure appelée skins. Ces capsules sont disponibles pour des systèmes d'exploitation temps réel, VxWorks, pSOS+, mais aussi pour le standard POSIX 1003.1b-1993 et POSIX 1003.1i-1995. Cela permet le développement d'applications temps-réel à partir de composants applicatifs sur étagère

– Son ouverture : une documentation fournie est disponible sur les appels systèmes implémentés en natif. Une communauté active fait vivre un forum d'assistance.

Comme nous l'avons souligné, le déploiement du détecteur sur une plate-forme d'exécution réelle fait apparaître de nouvelles problématiques. Ces dernières sont essentiellement liées à l'insertion du détecteur dans les structures permettant d'organiser l'exécution des différentes activités à mener de front : la détection, l'application, et toutes autres activités indépendantes de la détection et de l'application hébergées par le système d'exploitation

En schématisant, il faut décider si les traitements d'interruption peuvent où non être intégrés dans l'ensemble des contextes d'exécutions émetteurs d'événements. Pour répondre à cette question, il faut étudier les modes d'interaction entre les tâches, les traitement d'interruption, et le noyau de Xenomai. En particulier, l'aptitude à suspendre un mode d'exécution depuis un contexte donné, et les capacités de prévention de la suspension d'activité seront étudiées pour fixer les caractéristiques de la plate-forme expérimentale. En étudiant précisément ces deux questions, nous pourrons choisir la manière dont le détecteur sera intégré à l'activité de l'application et du système d'exploitation.

4.1.3 Conditionnement des données

Le conditionnement des données consiste à trouver la meilleure organisation d'un ensemble de données vis-à-vis de critères d'efficacité. L'efficacité sera dans notre cas liée au nombre maximal d'opérations exécutées lors d'une vérification. Sans optimiser ce critère, nous souhaitons identifier les caractéristiques des spécifications utilisées pour la détection qui ont le plus d'impact. Dans le chapitre précédent nous avons identifié les opérations réalisées sur l'abstraction temporelle pour permettre la détection des erreurs comportementales. Il existe essentiellement trois phases :

1. La recherche de la classe d'équivalence correspondant à un état s dans l'abstraction temporelle. (Mise à jour de l'index de classe d'équivalence avant la prise en compte de l'événement)
2. Le calcul de l'échéance associée à chaque classe à la volée. La complexité de cette opération dépend de la manière dont est encodée la région temporelle associée à une classe.

3. La recherche d'une transition événementielle dans l'ensemble des transitions autorisées à partir d'une classe d'équivalence.

Ces opérations dépendent fortement de la manière dont les classes d'équivalence sont représentées. Ceci est particulièrement vrai pour le premier point puisque cette opération nécessite un parcours des différents éléments de l'abstraction. L'efficacité de l'exécution de cette première phase dépend donc de la facilité avec laquelle il est possible de consulter le contenu de la structure de données mémorisant l'abstraction temporelle. Les deux autres points sont d'une complexité inférieure. Pour illustrer la nature des opérations réalisées sur l'abstraction, nous allons détailler le déroulement de la première phase.

Analyse détaillé de la première opération

Le problème est le suivant : *trouver la classe d'équivalence contenant l'état de destination s' d'une transition $s \xrightarrow{\Delta} s'$, avec s et Δ connus.*

Notons γ et γ' les vecteurs d'horloges de s et s'. Le vecteur γ' se déduit de γ par simple addition de Δ sur chaque horloge, $\gamma' = \gamma + \Delta.(1, ..., 1)$. Ainsi, deux approches sont possibles pour déterminer la classe de s' : soit l'on trouve la classe de s' dans l'abstraction temporelle en exécutant la transition $s \xrightarrow{\Delta} s'$, soit l'on essaie de trouver directement la classe de s' à partir du vecteur d'horloge de s'. Nous avons opté pour la première solution car cette dernière est plus simple à mettre en place dans le cadre d'un premier prototype.

Rappelons qu'un état de l'automate correspond à un «instant» de l'exécution de ce dernier. Chaque instant est caractérisé par le lieu occupé par l'automate et les valeurs des horloges. Les états à partir desquels il est possible de franchir les mêmes séquences de transitions événementielles sont regroupés dans des classes d'équivalence. Les transitions entre états sont transformées en transitions entre classes. Seul l'impact des transitions temporelles nous intéresse. Considérons que la figure 4.1 représente la partition de l'ensemble des vecteurs d'horloges associés au lieu res_use1. Dans ce cas précis, un vecteur d'horloge vérifiant $y \leq x$ et $y - x > 15$ peut appartenir à trois classes différentes, chacune identifiée par une région temporelle distincte $R1$, $R2$, $R3$. La classe $R1$ n'autorise pas l'événement *release* mais possède toujours une continuation valide ; La classe $R2$ autorise *release* ; La classe $R3$ ne contient que des états erronés.

Toute transition temporelle entraîne un déplacement des vecteurs d'horloges selon la 1ère diagonale de l'espace des vecteurs d'horloges. En réalisant l'abstraction temporelle, la notion de durée associée à une transition disparaît. Il est donc nécessaire de traduire la transition $s \xrightarrow{\Delta} s'$ dans l'abstraction temporelle. Pour cela, il faut comprendre ce que représente une $\tau - transition$. Prenons l'exemple de la figure 4.1, $R1$, $R2$ et $R3$ sont trois régions temporelles liées par des $\tau - transitions$:

$$\{res\_use1\} \times R1 \xrightarrow{\tau} \{res\_use1\} \times R2 \xrightarrow{\tau} \{res\_use1\} \times R3$$

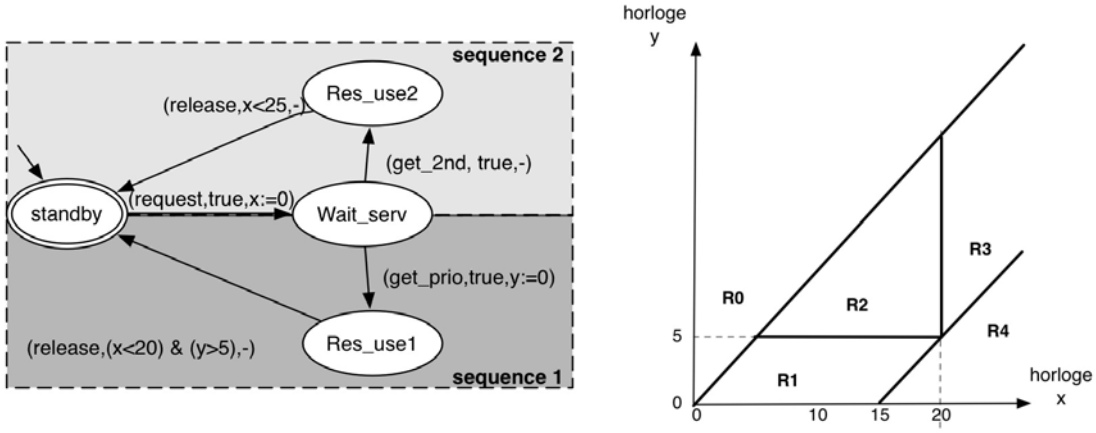

FIG. 4.1: Régions temporelles de res_use1

Sur la gauche, nous avons repris l'automate utilisé tout au long du manuscrit. Sur la droite, nous avons détaillé l'organisation de l'espace des vecteurs d'horloge pour le lieu Res_use1 en quatre régions distinctes. Chaque région correspond à un status différent en terme d'accessibilité et de co-accessibilité.

Cette séquence de $\tau - transitions$ signifie que pour chaque état q de $\{res\_use1\} \times R1$, il existe deux durées caractéristiques d, et d' telles que pour q' le successeur de q par $q \xrightarrow{r} q'$, si :

- $r < d$, **alors** $q' \in \{res\_use1\} \times R1$
- $d < r < d'$, **alors** $q' \in \{res\_use1\} \times R2$
- $d' < r$, **alors** $q' \in \{res\_use1\} \times R3$

Si $q' \in \{res\_use1\} \times R3$, l'exécution de la transition de q à q' dans l'abstraction temporelle passe par un saut de $R1$ à $R2$ puis de $R2$ à $R3$. En pratique, nous procéderons de proche en proche en consommant petit à petit le budget r tel que décrit sur la figure 4.2. Cela revient à sauter de frontière temporelle en frontière temporelle tant que la durée écoulée depuis le dernier événement le permet. Nous avons déjà vu comment calculer le temps nécessaire pour atteindre par écoulement du temps le bord d'une classe d'équivalence, cf 3.2.4 page 77. La section suivante aura pour but de définir avec précision l'algorithme et les structures de données nécessaires pour réaliser cette exécution des transitions abstraites.

Cette section avait pour but de mettre en évidence les trois problématiques liées à l'implémentation du détecteur à partir de la description de sa logique. Dans la section suivante nous discuterons les moyens mis en œuvre pour répondre au mieux à ces trois problématiques. La description du prototype servira de support, dans la dernière section, à une étude de la criticité des différents composants du détecteur et des mesures qui peuvent

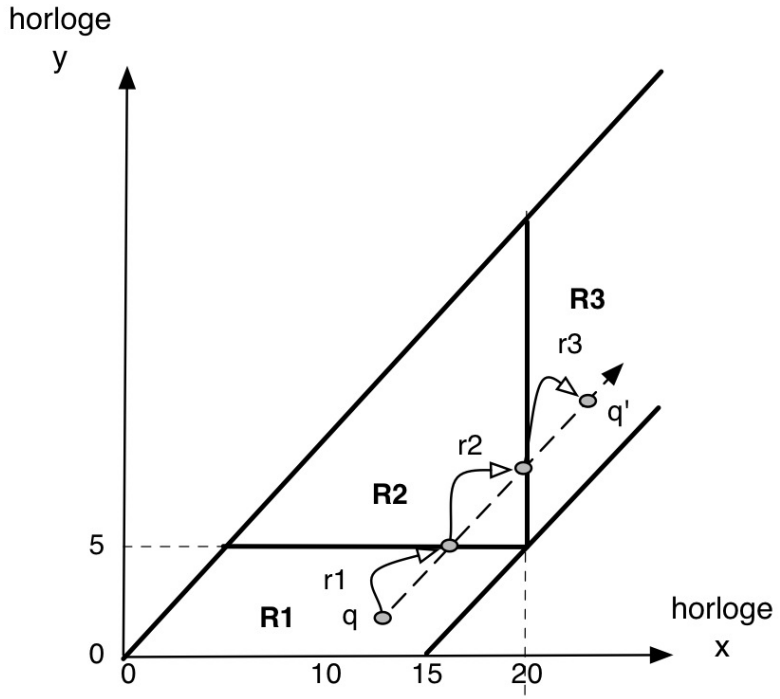

FIG. 4.2: Exécution des transitions abstraites par «sauts de puces», avec $r = r1 + r2 + r3$
Pour prendre en compte l'écoulement de la durée r à partir de l'état q (menant à l'état q′), il faut traduire l'écoulement du temps dans le graphe de l'abstraction temporelle. Franchir la τ-transition $(Res\_use1, R1) \xrightarrow{\tau} (Res\_use2, R2)$ *revient à atteindre le bord de* $R1$ *depuis q en laissant* $r1$ *unités de temps s'écouler. Tant que r le permet, il faut continuer à travers les régions temporelles et franchir les τ-transitions de l'abstraction temporelle. Lorsque r ne permet plus d'atteindre le bord d'une classe d'équivalence, cela signifie que la classe d'équivalence de q′ vient d'être trouvée.*

être prises pour en assurer la correction dans une section supplémentaire.

4.2 Réalisation et intégration du prototype

Trois grandes questions ont été soulevées dans le section précédente. Ces trois problématiques sont directement reliées au raffinement du modèle d'exécution de l'application : le passage de la simple trace temporisée à un contexte réaliste où le détecteur doit intégrer son exécution dans celle de l'application.

1. Quel est pour chaque contexte d'exécution le niveau de contrôle et protection fourni vis-à-vis de l'activité du reste du système ? *Nous identifierons les causes de suspension de chaque contexte d'exécution. Cette information servira à déterminer les contextes d'exécution à partir desquels il est raisonnable de capturer des événements*
2. Comment intégrer la logique de détection dans l'activité de l'application ? *Ce choix doit être un compromis entre la réactivité du mécanisme de déclenchement de la vérification, la sensibilité de l'exécution aux suspensions d'activité, et les fonctions de contrôle d'exécution disponibles dans le ou les contextes choisis.*
3. Comment implémenter les opérations de consultation du contenu de l'abstraction temporelle qui sont pour le moment décrites à très haut niveau ? *Des algorithmes précis doivent être proposés. Cela entraîne l'identification de l'information utile pour la détection au plus tôt dans le calcul complet de l'abstraction temporelle sous KRONOS.*

Nous allons devoir prendre en compte ces éléments pour comprendre où et comment exécuter le code de vérification. Les deux premières questions dépendent toutes les deux d'où et comment sera exécuté le code de vérification. Nous étudierons ce point dans la sous-section suivante. Une sous-section supplémentaire contiendra la description complète du prototype. Cela comprend en particulier les divers algorithmes utilisés pour implémenter le détecteur, ainsi que la description des structures de données utilisées dans ces derniers.

4.2.1 Analyse des structures de contrôle de Xenomai

De notre point de vue, un contexte d'exécution est caractérisé par trois éléments : son mode d'activation, ses modes de protection vis-à-vis de la suspension de son activité, et la mémoire à laquelle il a accès.

Les événements sont potentiellement générés depuis plusieurs contextes d'exécution distincts : différentes tâches, ou différentes routines de traitement d'interruption. L'objectif de cette sous section est double :

- identifier un contexte réaliste pour l'exécution du code de vérification,
- déterminer d'où sont extraits les événements.

Nous allons procéder en trois étapes,

- Faire l'inventaire de l'ensemble des solutions qui s'offrent à nous pour intégrer le détecteur à l'application.
- Proposer un compromis correct entre les contextes considérés comme source d'événements et les contextes dans lesquels ils seront vérifiés.
- Analyser le mode de synchronisation mis en place par la répartition de la logique de vérification et de recouvrement dans les différentes structures de Xenomai.

Xenomai, sa structure, et ses services Xenomai propose un service d'alarme efficace pour déclencher le réveil d'une tâche de telle sorte qu'on puisse capturer les erreurs associées à un dépassement d'échéance. Dans un premier temps, nous allons lister les différents contextes d'exécution disponibles sous Xenomai. Cette partie de l'étude est en grande partie dépendante de l'application visée. En effet, en fonction du matériel utilisé certains mécanismes proposés par Xenomai ne sont pas activés. Cela veut dire que Xenomai possède un comportement qui varie tant au niveau des performance qu'au niveau fonctionnel suivant le matériel disponible. Ainsi, cette partie est spécifique à la plateforme d'expérimentation qui a été utilisée au cours de nos travaux : un pentium III 1Ghz, avec les systèmes d'économie d'énergie désactivés.

Un système de protection mémoire permet de séparer la mémoire associée au noyau de celle de l'application. L'exécution en mode noyau est a priori intéressante si l'on réalise un nombre important d'appels systèmes. Le deuxième critère de séparation concerne le mode d'activation : ce critère nous permet de distinguer les interruptions, des tâches, des fonctions, et des signaux. Ce dernier cas ne sera pas traité car les signaux sous Xenomai ont une sémantique très spécifique les rendant inadaptés au problème à traiter. Ainsi, trois grand types de contextes peuvent être considérés avec chacun leur particularités. D'un point de vue concret le code de vérification est exécuté :

- soit au sein de la structure émettrice de l'événement à travers un appel de fonction,
- soit dans un contexte indépendant, au choix entre une tâche et un traitement d'interruption.

L'exécution du code en mode noyau est de notre point de vue un problème relativement orthogonal. Pour chaque cas, les méthodes de déclenchement, et les méthodes de protection vis-à-vis de la suspension d'activité sont décrites dans le tableau 4.2.1.

Compromis entre réactivité, contrôle et intrusitivité Xenomai permet de définir facilement des traitements d'interruption au niveau utilisateur. Il y aura donc a priori un code avec une forte sémantique comportementale dans ces routines. Cependant, le respect des exigences d'exécution exclusives du code de vérification sont assez dures à assurer si l'on

Contexte	Mode de déclenchement	Mode de protection
Traitement d'interruption	– déclenchement matériel : équivalent à l'exécution d'une fonction en mode noyau, déclenchée indépendamment du code exécuté précédemment. L'espace mémoire accessible de manière prédictible pour un déclenchement de ce type se limite à la mémoire du noyau. – déclenchement logiciel : équivalent à l'exécution d'un appel système dans son implémentation la plus rudimentaire. La routine a accès aux données de la tâche source de l'interruption (softirq)	Masquage des autres interruptions, masquage de l'interruption source de la vérification, exécution en mode privilégié à priori.
Tâche	– activation logicielle implicite : Le code de vérification est exécuté dans une tâche suspendue entre chaque événement. L'occurrence d'un événement déclenche le passage de la tâche du mode «suspendue» à «prête». L'exécution effective de la vérification est à la discrétion de la politique d'ordonnancement. – activation logicielle explicite : La vérification est déclenchée par un appel directe à `rt_task_resume()`. Ceci déclenche explicitement le changement de contexte vers la tâche dans laquelle la vérification aura réellement lieu.	La priorité de la tâche de vérification permet de s'assurer qu'aucune tâche moins prioritaire ne puisse s'exécuter avant que la tâche de vérification ne repasse en mode «suspendue». Pour éviter la suspension de la tâche de vérification par une quelconque tâche, il est nécessaire de changer le mode d'exécution de la tâche de vérification. Il faut la rendre non préemptible. Il n'y a pas de moyen évident de retarder les traitements d'interruption sans les désactiver et perdre les interruptions correspondantes.
Fonction en ligne	activation synchrone par appel de la fonction	la fonction devra utiliser les modes de protection du contexte à partir duquel elle est invoquée (tels que détaillés ci-dessus).

s'autorise à vérifier des événements correspondant à du code exécuté au sein de routines de traitement d'interruptions. Le meilleur moyen d'éviter cette situation est d'interdire la capture d'événement au sein des traitements d'interruption et d'exécuter la vérification événementielle dans une fonction appelée depuis le contexte où l'événement est émis. Nous avons choisi de restreindre la capture à des événements correspondant à une action exécutée au sein des tâches Xenomai constituant l'application. Cette restriction nous amène à considérer une instrumentation relativement classique de l'application par simple annotation du code.

Etat global et synchronisation Le modèle de trace suppose une précision infinie de la mesure de la date d'un événement. L'horloge temps réel de Xenomai fournit un service de datation dont la précision maximale est de l'ordre de la nano-seconde. En pratique, nous avons constaté que la simple mesure du temps coûtait 50 cycles processeur environ. Il en découle une précision utilisable de l'ordre de la centaine de nano-secondes. Les événements ne peuvent être traités en parallèle. En effet, le modèle de trace induit une vision centralisée et linéaire d'une exécution. Cela signifie que chaque événement est complètement vérifié avant de laisser l'exécution se poursuivre. Comme chaque événement peut redéfinir l'échéance la plus urgente à satisfaire, le détecteur se doit donc de traiter chaque événement au plus vite et de manière atomique. Cela évite de créer des incohérences entre la contrainte temporelle implémentée par l'alarme, la trace réellement capturée et le processus de vérification des événements.

La solution choisie repose sur une fonction dont la première action est de modifier le mode d'exécution de la tâche émettrice pour la rendre non préemptible. Le mode d'exécution est restauré à la fin de la vérification. Cette solution doit pouvoir répondre aux trois questions suivantes :

Q1 : l'exécution de l'application peut-elle se poursuivre en dépit de la capture de l'événement ?

R1 : Si le progrès de l'application est mesuré via l'exécution du code inclus dans les tâches, l'utilisation du mode non préemptible assure l'absence de progression de l'exécution de l'application.

Q2 : Y a-t-il un risque de verrouillage du système ?

R2 : L'exécution de la vérification de l'événement sera toujours réalisée en un nombre fini d'opérations élémentaires. Le parcours des structures de données peut être testé hors ligne. Ceci permet d'exclure relativement facilement les causes de verrouillage internes au moniteur.

Q3 : Peut-on borner de bout en bout l'exécution du code de vérification ?

R3 Oui mais ... Le problème de l'évaluation du temps d'exécution de bout en bout du détecteur est de même nature que celui de l'évaluation du temps d'exécution de bout

en bout d'une tâche non préemptible. Pour borner ce temps, il est nécessaire de borner la fréquence des interruptions et le temps de blocage induit par leur traitement. Ainsi, cette borne est obtenue à travers une analyse des conditions opérationnelles.

4.2.2 Analyse algorithmique et description du prototype

Nous avons identifié les moyens disponibles pour implémenter le détecteur. Il reste à détailler les algorithmes concrètement utilisés et leur entrelacement avec les appels systèmes destinés à mettre en place le bon contexte d'exécution pour le code du détecteur. L'implémentation de la fonction de vérification via une fonction spéciale permet de déclencher la vérification. L'identifiant de l'événement est passé en paramètre à cette fonction qui se charge de dater cet événement. Chaque fois qu'un événement doit être capturé dans le code de l'application un appel à la fonction de vérification est inséré avec l'identifiant de l'événement que l'on souhaite générer.

La description du prototype se fera de la manière suivante. Dans un premier temps, la structure de données utilisée pour mémoriser l'abstraction temporelle sera détaillée. Puis chaque algorithme impliqué dans la vérification sera décrit pas à pas. Le processus de synthèse du détecteur permettra de donner le détail de l'assemblage complet. Cette section sera close par la description de l'entrelacement des différents algorithmes avec les appels systèmes permettant d'intégrer l'exécution du détecteur à celle de l'application.

4.2.2.1 La structures de données

Habituellement, les régions de vecteurs d'horloges de l'abstraction temporelle sont mémorisées sous la forme diagrammes de décision booléens (BDD en anglais) [Wan04], ou sous la forme de matrice appelées DBM [Dil90].

Ces structures de données sont adaptées uniquement pour créer l'abstraction et non pour l'utiliser en ligne pour tracer l'exécution du système afin de détecter une erreur. Nous avons déjà vu à la section 3.2.4 que l'information utile dans la définition des classes d'équivalence se limite, dans notre cas, à la définition du «bord supérieur» de la classe d'équivalence. Cette notion de bord supérieur d'une classe d'équivalence C correspond à l'ensemble des vecteurs d'horloge à la fois :

- infiniment proche d'un prédécesseur par transition temporelle appartenant à C.
- infiniement proche d'un successeur par transition temporelle appartenant au successeur temporel abstrait de C, à savoir la classe C' telle que $C \xrightarrow{\tau} C'$, et qu'il n'existe aucune classe C" telle que

$$C \xrightarrow{\tau} C" \xrightarrow{\tau} C'$$

Sur la figure 4.1, le bord supérieur de R1 correspond au vecteur d'horloge satisfaisant : $y \leq x$, $15 < y - x$ et $y = 5$. Nous verrons qu'il est suffisant, pour réaliser la vérification en

ligne, de mémoriser pour chaque classe :

- l'ensemble des transitions événementielles autorisées dans la classe en question. Chaque transition lie trois informations : i) l'identifiant de l'événement, ii) l'identifiant de la classe de destination, iii) la définition des horloges à remettre à zéro lors du franchissement de la transition, tel que décrit dans l'automate
- [*si la classe possède un successeur temporel (une τ-transition) :*]
  - la définition de son "bord supérieur" et l'index du successeur temporel.
  - la définition du vecteur d'échéance tel qu'il est défini dans le chapitre précédent 3.2.4.

La figure 4.3 donne une vue d'ensemble de la nature des données embarquées à l'exécution. La numérotation des classes permet d'utiliser leur index pour encoder les transitions. Une table permet de faire correspondre l'index d'une classe à la structure de données contenant ses caractéristiques. Sur la figure sont représentées la table des classes, et le format de la structure associée à chaque classe. Les flèches en gras indiquent comment les différents champs sont remplis pour le cas de la classe numérotée «1».

Ainsi, la structure associée à une classe d'équivalence contient :

- L'ensemble des transitions événementielles «surchargées» sous la forme d'une liste de structures simples associant l'index d'un événement, celui de la classe de destination, et un ensemble d'index d'horloges à remettre à zéro.
- La définition du vecteur d'échéance qui contient les valeurs maximales acceptables pour chaque horloge pour la classe courante et tous ses successeurs temporels valides. Ce vecteur permet de configurer l'alarme dès que nécessaire sans avoir à parcourir l'ensemble des successeurs temporels de la classe d'équivalence courante. Si aucune échéance n'est définie sur une horloge x dans la classe courante, alors la coordonnée -1 est utilisée comme valeur par défaut.
- l'index du successeur temporel de la classe courante. Si ce successeur n'existe pas, alors la valeur -1 est mise par défaut.
- (si la classe a un successeur temporel) le vecteur définissant le bord supérieur de la région temporelle de la classe d'équivalence.

Nous allons voir maintenant comment les trois opérations clés de la vérification ont été implémentées. On peut voir sur la figure l'organisation des différentes informations associées à une classe. Le processus permettant la synthèse automatique de cette structure ne présente aucune difficulté particulière et ne sera brièvement décrit qu'en fin de section.

4.2.2.2 Principe de simulation des τ-transitions.

Ces structures décrivent explicitement l'ensemble des bordures de chaque région temporelle. Nous avons déjà discuté la notion de frontière supérieure dans le chapitre précédent. Nous pouvons utiliser un raisonnement similaire à celui utilisé pour le calcul des

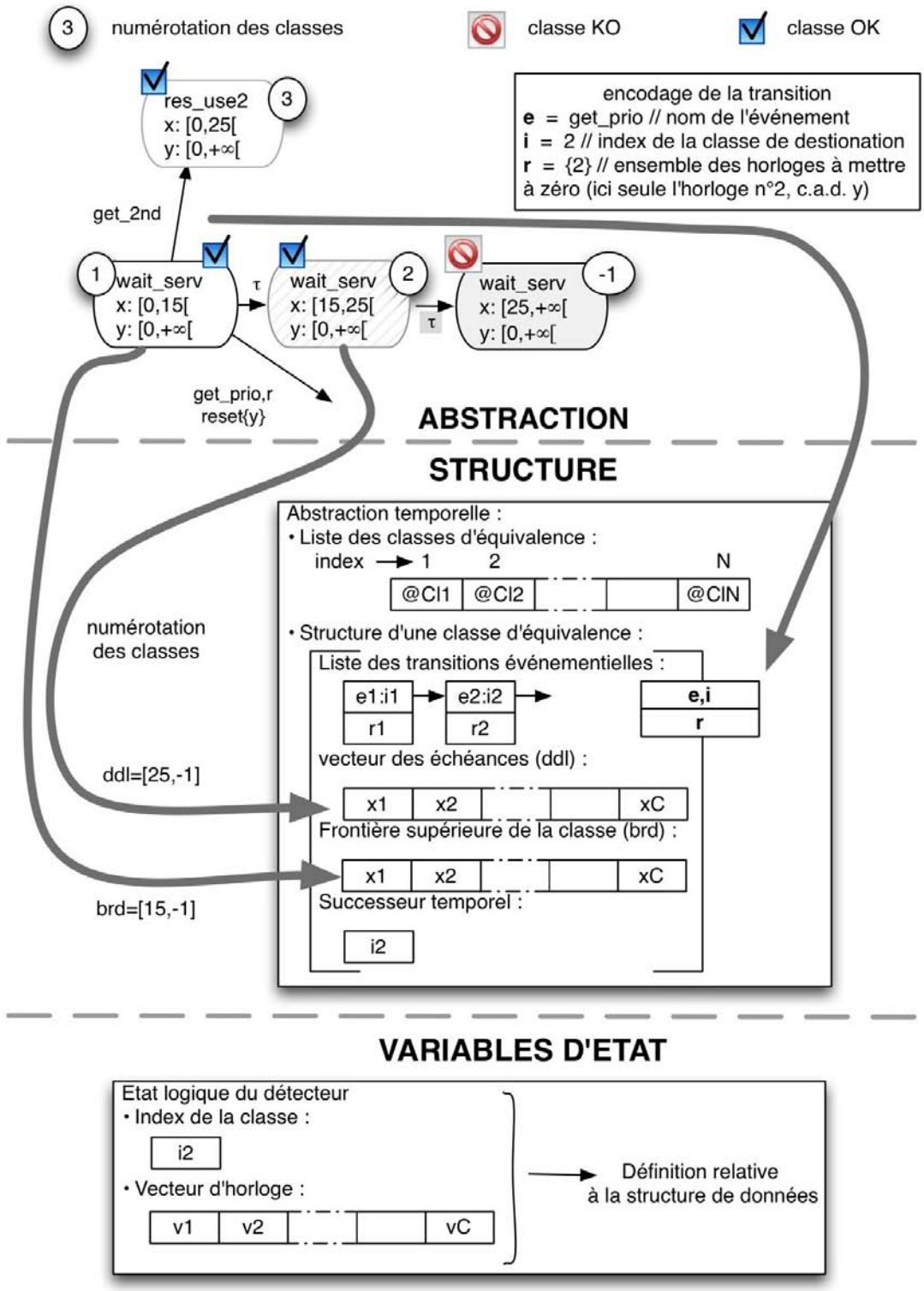

FIG. 4.3: Stockage de l'abstraction temporelle et variables d'état du détecteur

échéances contextuelles, pour simuler les τ-transitions lors de la mise à jour de l'état. Pour poursuivre la description de cette étape de manière efficace nous avons besoins de quelques éléments de notation :

- Les variables utilisées dans l'algorithme, Δ et γ, et S, représentent la durée à simuler, le vecteur d'horloge et l'index de classe de «l'état courant». Ces trois variables sont respectivement initialisées avec : i) la durée écoulée depuis le dernier événement, ii) γ_s et S_s, le vecteur d'horloge et la classe de l'état s atteint après validation de l'événement précédent.
- La fonction $succ_\tau()$ retourne pour une classe donnée son successeur temporel direct s'il existe.
- Dans le cas où la classe S possède un successeur temporel, alors cela signifie que la classe est «bornée» temporellement. La fonction $Dist(\gamma, S)$ retourne le temps nécessaire pour amener γ sur le bord supérieur de S en laissant le temps s'écouler. Ainsi, $Dist(\gamma, S)$ retourne le temps d tel que pour tout $\epsilon, \epsilon > 0$ suffisamment petit, alors $\gamma + d + \epsilon$ est dans $succ_\tau(S)$, et $\gamma + d - \epsilon$ est dans S. En laissant $Dist(\gamma, S)$ unités de temps s'écouler depuis γ, alors le système se place en position de franchir la τ-transition séparant S et $succ_\tau(S)$. Si la classe n'est pas bornée, alors la fonction retourne une valeur par défaut, -1.
- La fonction AdditionVect(v,d) ajoute la constante d à chacune des coordonnées du vecteur v

L'algorithme concret de mise à jour peut enfin être détaillé, cf l'encart 4.2.2.2, page 97. Cet algorithme implémente la première opération, à savoir la mise à jour de l'état avant vérification. D'un point de vue pragmatique, la méthode choisie pour déterminer la classe de s' est relativement facile à mettre en oeuvre puisqu'elle réutilise la méthode qui permet de déterminer les échéances à la volée : le calcul de distance entre un vecteur d'horloge et les «bords» de la région temporelle d'une classe.

4.2.2.3 Recherche d'une transition événementielle

La seconde phase de l'exécution du vérifieur correspond à la vérification proprement dite. L'implémentation de cette phase dépend de la manière dont les transitions événementielles sont mémorisées. L'événement est autorisé si et seulement si la classe atteinte après mise à jour de l'état du détecteur possède une transition événementielle sortante marquée par l'événement à vérifier. Il existe de nombreuses manières d'encoder des ensembles d'objets et d'en faciliter le parcours. Nous avons opté pour une structure simple de liste pour mémoriser l'ensemble des transitions événementielles. L'objectif est d'avoir un ensemble de paires liant un événement et une classe de destination mémorisées dans une liste chaînée. Il suffit de parcourir cette liste en comparant l'événement observé à l'événement associé à chaque transition de la classe courante pour savoir s'il est autorisé.

Algorithme 1 Simulation de la transition $s \xrightarrow{\Delta} s'$ depuis s

```
1  Algorithme Update(Δ, γ_s, S_s)
2  { L'état s correspond au lieu l_s et au vecteur γ_s. S_s est la classe d'équivalence
   associé à s. S et γ définissent conjointement un état. S détermine un lieu et γ le
   vecteur d'horloge correspondant. }
3  Variable
4      TempsRestant : durée
5      S : entier   { index de classe }
6      γ : entier[NC]   { vecteur d'horloge courant }
7      d : durée
8  Début
9      TempsRestant ← Δ
10     S ← S_s
11     γ ← γ_s
12     d ← Dist(γ, S)
13     TantQue (TempsRestant > d) ET (d > 0) Faire
14         { Une valeur négative est utilisée pour indiquer l'absence de borne
       temporelle. }
15         TempsRestant ← TempsRestant − Dist(γ, S)
16         S ← succ_τ(S)
17         AdditionVect(γ, d)
18         d ← Dist(γ, S)
19     FinTantQue
20     AdditionVect(γ, TempsRestant)
21 Fin
22 { L'algorithme termine lorsque le «temps restant» n'est pas suffisant pour
   franchir une τ-transition supplémentaire. }
```

Lorsque aucune transition n'est étiqueté par l'événement à vérifier, cela veut dire que cet événement est interdit dans la classe courante.

La représentation de l'ensemble des transitions sous forme de liste chaînée est particulièrement efficace quand le nombre de transitions actives dans une classe est très faible par rapport au nombre d'événements considérés dans l'ensemble du modèle. Notez que la liste des horloges à remettre à zéro, lorsque la transition est franchie, fait partie du descripteur de la transition. Le comportement attendu du détecteur est de vérifier un événement, et, si ce dernier est valide, de franchir la transition associée. Le franchissement de cette transition entraîne la mise à zéro des alarmes tel qu'indiqué dans le descripteur de la transition.

4.2.2.4 Configuration de l'alarme

Nous avons déjà vu comment calculer la durée nécessaire pour atteindre la frontière supérieure d'une classe d'équivalence à partir d'un vecteur d'horloge de la classe (à travers les $\tau - transitions$).

Chaque classe d'équivalence est dotée d'un vecteur spécial appelé vecteur d'échéance, noté *ddl*. Ce vecteur mémorise la frontière supérieure du dernier successeur temporel valide de la classe. Dans le cas où l'événement est validé, la dernière opération du processus de vérification consiste à calculer la durée de l'échéance associée à la classe d'équivalence courante. Chaque événement déclenche une mise à jour de l'échéance implémentée par l'alarme du détecteur. Il faut recalculer la «nouvelle échéance» à partir de l'état atteint juste après avoir pris en compte l'événement, noté s". L'algorithme suivi est décrit dans l'algorithme 2. Deux points clés méritent d'être détaillés par rapport à cet algorithme :

1. Le bord supérieur de la classe d'une classe n'est pas nécessairement borné sur chaque horloge. La valeur -1 est utilisée dans le vecteur *ddl*, pour signifier que l'horloge n'est pas bornée et peut prendre des valeurs «infiniment» grandes. Ainsi, chaque fois qu'une coordonnée de *ddl* vaut -1, cela signifie que cette horloge ne doit être considérée pour le calcul de l'échéance. Cette remarque se traduit dans l'algorithme, page 100, par le test de la ligne 12. Ce test est généralisé ligne 9, pour s'assurer qu'une échéance existe dans l'état courant. En effet, si *ddl* est le vecteur $(-1,\cdots,-1)$, cela signifie que le lieu atteint est un état sûr (à savoir un état dans lequel on peut rester indéfiniment sans que cela porte à conséquence).
2. La configuration d'une alarme se fait de manière relative. Ainsi, Il faut prendre garde que l'état à partir duquel on calcule la date de l'occurrence correspond à une date relativement éloignée de la date de configuration de l'alarme. Ainsi, la date de réveil de l'alarme doit être corrigé pour correspondre au mieux à l'instant de réveil souhaité. Le diagramme de la figure 4.4 donne le détail du processus de vérification. La date de l'événement considérée tout au long de la vérification correspond à t_0,

la configuration de l'alarme se fait à t_1. A l'issue de l'opération 3, la date de réveil relatif est $\Delta 2$. Cette date relative doit être diminuée de $D = t_1 - t_0$.

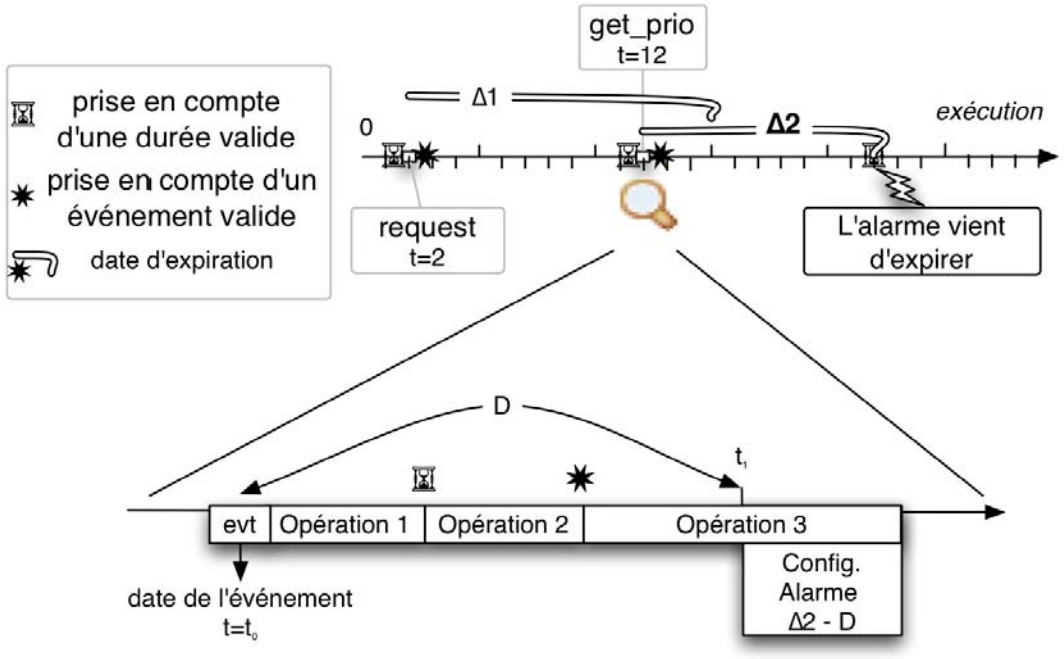

FIG. 4.4: Les phases de la vérification et la correction de l'échéance
Cette figure synthétise le déroulement de la vérification en ligne de l'automate temporisé. Chaque phase correspond à une tâche bien précise identifiée par un petit symbole. Lors de la dernière phase, une échéance est définie sur le prochain événement. Cette échéance est corrigée pour prendre en considération le coût temporel des opérations exécutées durant cette phase d'activité.

Une alarme sous Xenomai, permet de déclencher le réveil d'une tâche en lui affectant une priorité au dessus des priorités accessibles lors de la configuration des tâches. Ainsi, une tâche réveillée par ce mécanisme suspendra toute autre tâche d'après la spécification de l'appel système `rt_alarm_wait()`, [Xen]. Seules les interruptions peuvent retarder l'exécution de la tâche ainsi réveillée. Pour utiliser ce service, il faut avoir un descripteur d'alarme et une tâche, cette dernière restera en attente de l'expiration de l'alarme. Cette tâche est en charge de mettre en place le recouvrement de l'erreur détectée par l'expiration de l'alarme. Ainsi le code associé à cette tâche est constitué d'une boucle infinie contenant :

Algorithme 2 Calcul de la durée séparant l'instant courant de la nouvelle échéance

```
1  Algorithme Echeance(S'', γ_s'' : classe, vecteur)
2  { Cette fonction met directement à jour l'échéance existant sur l'occurrence du
   prochain événement. Cette mise à jour repose sur le calcul d'une date relative Δ
   définie depuis l'état courant (l'état courant après validation de l'événement
   correspond à la date 0) }
3  { s'' est l'état de l'automate juste après avoir franchi la transition correspondant
   à l'événement qui vient d'être validé. γ_s'' est le vecteur d'horloge de s'', et S'' est
   la classe de s'' }
4  { Chaque classe est une structure qui possède un champ ddl qui est un vecteur
   de coordonnées représentant les valeurs maximales acceptables pour chaque
   horloge pour S'' et ses successeurs temporels valides }
5  Constante
6  | DernierEvt = der_evt  { date de l'événement venant d'être validé }
7  Début
8  | Δ ← max(S''.ddl)  { Extraction de la borne supérieure de Δ, cf 1) }
9  | Si Δ > 0 Alors  { existence d'une échéance, cf 1) }
10 | | Pour i variantDe 1 à N Faire  { Pour chaque horloge de γ_s'' }
11 | | | { Calcule la durée disponible avant l'échéance induite par la }
12 | | | { i^me horloge, x_i }
13 | | | Si ddl[i] ≠ −1 Alors  { existence d'une échéance selon x_i, cf 1) }
14 | | | | Δ ← min(Δ, ddl[i] − γ_s''[i])  { Mise à jour de Δ }
15 | | | FinSi
16 | | FinPour
17 | | Δ← Δ − (high_res_clock() − DernierEvt)  { Correction de Δ, cf 2) }
18 | | rt_alarm_start(Δ)  { Configuration de l'alarme }
19 | Sinon
20 | | rt_alarm_stop()  { Désactivation de l'alarme en cours }
21 | FinSi
22 Fin
```

– L'instruction suspendant la tâche jusqu'à la prochaine expiration de l'alarme du détecteur.
– Le code correspondant à la mise en place du recouvrement d'une erreur de type «dépassement d'échéance».

Le descripteur de l'alarme est nécessaire pour pouvoir reconfigurer à la volée l'échéance de cette dernière dans l'opération 3. L'implémentation du recouvrement n'est pas nécessairement concentrée dans la tâche réveillée par l'alarme. Nous conseillons de n'y placer que les instructions permettant de déclencher le recouvrement. Ainsi, ce dernier pourrait lui-même être soumis à la vérification en ligne réalisée par le détecteur. Notez que le temps nécessaire pour configurer et activer l'alarme doit être suffisamment bref pour pouvoir être négligé. Dans le cas contraire, il serait nécessaire de pouvoir prédire ces constantes temporelles pour les inclure dans la correction de la durée Δ.

4.2.2.5 Synthèse et configuration du détecteur

Avant l'exécution, le modèle doit être adapté à la plate-forme. Cela consiste à accorder le modèle et la plate-forme principalement en appliquant un facteur d'échelle sur les grandeurs temporelles présentes dans l'abstraction temporelle. Ce facteur d'échelle permet de donner une unité aux grandeurs définies dans les gardes de l'automate.

Pré-traitement

Nous avons défini un format de fichier qui reprend dans les grandes lignes la syntaxe utilisée dans KRONOS pour spécifier des automates temporisés. Une description détaillée de cette dernière peut être trouvée dans le manuel de KRONOS, [BDM+98]. La syntaxe des fichiers d'entrée de KRONOS permet de définir des automates temporisés de sûreté, à savoir des automates pour lesquels aucun lieu final n'est défini. De plus, un automate temporisé décrit des contraintes temporelles sans préciser l'unité physique utilisée pour mesurer le temps (secondes, millisecondes, ...). Pour faire bref nous proposons de concentrer dans un fichier trois informations clés pour définir une spécification transformable en détecteur au plus tôt : l'automate, ses états finaux, et l'unité de temps physique considérée. Ces informations sont concentrées dans des fichiers portant l'extension «.rts», un l'exemple est fournit avec le code du prototype. La table 4.1 représente des extraits d'un tel fichier de spécification. Les trois premières lignes de la table correspondent à la syntaxe empruntée à l'outil KRONOS. Ce contenu est directement soumis à l'outil pour calculer l'abstraction temporelle de l'automate ainsi défini.

L'information produite par l'outil KRONOS est filtrée à sa sortie car elle contient toutes les classes d'équivalence accessibles depuis l'état initial de l'automate. Un graphe d'accessibilité est construit pour filtrer les classes depuis lesquelles il est impossible d'atteindre un lieu de la liste des lieux finaux. Nous utilisons ici les algorithmes classiques de

TAB. 4.1: Extraits d'un exemple de fichier d'entrée pour la génération du détecteur.

Extrait	Rôle

`#states 4`
`#trans 5`
`#clocks 2`
`X0 X1`
	En-tête définissant le nombre de «lieux», de transitions, et d'horloges utilisées dans la spécification. Les noms des variables d'horloge ont nécessairement le format `X<nombre>`.

`state: 0`
`invar: TRUE`
`trans:`
`....`
	Définition d'un «lieu» avec son index, son invariant et la liste de ses transitions (ici réduite à une seule transition détaillé dans la suite du tableau)

`TRUE =>`
`  REQUEST; RESET{ X0 }; goto 1`
`X0 < 20 =>`
`  RELEASE; RESET{ }; goto 0`
	Définition de deux transitions. La première est une transition sans garde correspondant à l'événement REQUEST. L'état de destination est le lieu 1, et le franchissement de cette transition entraîne la mise à zéro de l'horloge X0. La seconde transition diffère essentiellement de la première par la définition d'une garde : $x_0 < 20$.

`FINAL STATES : 0;` `TIME UNIT 1000`	Définition des états finaux et de l'unité temporelle exprimée en multiple de nano-secondes, à savoir ici la 1 micro-seconde.

contrôle d'accessibilité dans un graphe orienté fini. Après avoir élagué l'abstraction temporelle de la sorte, chaque classe est encodée dans la structure définie dans cette section.

Assemblage et vue d'ensemble du détecteur

Le code est organisé en trois unités pour assurer une certaine lisibilité du processus d'assemblage. Chaque bloc possède un rôle bien défini :

- Le moteur d'interprétation de l'abstraction temporelle est concentré dans un fichier sous la forme d'un ensemble de fonctions (hrtmonit.c/h – code constant)
- Une tâche de recouvrement dont le code est défini dans un fichier indépendant des fichier sources contenant le moteur de détection (rec.c/h – code défini par l'utilisateur)
- L'abstraction temporelle encodée tel que décrit dans cette section. (model.c / model.h – généré automatiquement à partir du fichier de modélisation)

Il convient de noter que d'un point de vue pratique le code constituant le moteur de détection, à savoir implémentant les trois phases du processus de vérification, ne fait pas plus de 300 lignes de code.

Les événements capturés par notre prototype se limitent à des points de code exécutés dans le contexte d'une tâche de l'application. Grâce à cette limitation, il suffit de déclarer une variable globale au processus pour pouvoir avoir accès en permanence à l'encodage de l'abstraction temporelle. L'ensemble des classes d'équivalence est énuméré puis mémorisé dans un simple tableau. Les variables d'état du moniteur sont au nombre de trois : *CC*, *vech*, et *DernierEvt*. *CC* et *vech* représentant l'index de classe et le vecteur d'horloge du dernier état estimé. Au début de l'exécution du moniteur, ces deux variables sont initialisées avec 0 et $(0, \cdot, 0)$. La date du dernier événement est toujours mémorisée dans *DernierEvt*, et est initialisée à 0. Le moteur de détection est décrit dans son intégralité par l'algorithme défini dans l'algorithme 3.

Si l'on devait résumer la réalisation du prototype en cinq points cela donnerait ceci :

- Les trois étapes logiques de la vérification d'un événement sont intégrées à une fonction non préemptible appelée à chaque événement à vérifier.
- La vérification est uniquement perturbée par l'exécution des routines de traitement d'interruption
- La capture et le traitement des événements se déroule dans le même ordre et de manière atomique pour l'ensemble des tâches temps réel de Xenomai. Cela implique qu'aucune tâche ne progresse dans son exécution pendant la vérification d'un événement.
- La configuration de l'alarme pour la capture des erreurs par dépassement d'échéance se fait pour chaque nouvel événement.
- La durée du processus de vérification est integrée dans le calcul de la durée séparant l'événement vérifié de la prochaine échéance.

Algorithme 3 Moteur de détection, une vue d'ensemble

```
1  { Description de la fonction invoquée lors de l'occurrence d'un événement }
2  Algorithme Verif_evt(evt, t)
   (evt, t) :: index et date de l'événement à vérifier
3  Variable
4  |  Tr : transition
5  Début
6  |  { Blocage de l'ordonnanceur }
7  |  rt_task_set_mode(0,T_LOCK,NULL)
8  |  { Mise à jour de l'état du système avant vérification de evt }
9  |  Update(t − DernierEvt, vech, CC)   { cf Algorithme 1 }
10 |  DernierEvt ← t
11 |  { La fonction FoundEvent(evt, CC) retourne la transition associée à evt
   dans la classe CC, la constante NULL sinon. }
12 |  Tr ← FoundEvent(evt, CC)
13 |  Si ( Tr = NULL) Alors
14 |  |  SIGNAL_ERROR et sortir   { signaler l'erreur détectée ! }
15 |  Sinon
16 |  |  { L'événement est valide, l'état doit être mis à jour }
17 |  |  CC ← Tr.destination
18 |  |  { Mise à zéro des horloges de vech tel que spécifié dans Tr }
19 |  |  reset(Tr.reset, vech)
20 |  |  { Gestion des erreurs temporelles futures }
21 |  |  Echeance(CC, vech)   { cf Algorithme 2 }
22 |  |  { Déblocage de l'ordonnanceur }
23 |  |  rt_task_set_mode(T_LOCK,0,NULL)
24 |  FinSi
25 Fin
```

Nous allons dans la section suivante évaluer de manière expérimentale le coût de l'intégration du détecteur dans une application.

4.3 Évaluation du coût du détecteur

D'un point de vue pratique, l'intégration du détecteur nécessite de la mémoire, du temps processeur, mais aussi des ressources système. Cette section a pour but d'évaluer les besoins du détecteur sur ces trois aspects. En opération, le détecteur consomme de la mémoire, du temps processeur, et des ressources «système» globales (ordonnanceur / alarmes). Les deux premiers points nécessitent une étude approfondie. La mémoire, ainsi que le temps processeur consommés dépendent de la complexité du modèle servant à la détection. Les services système utilisés par le détecteur sont :

- deux tâches temps réel. Ces deux tâches permettent la prise en charge du signalement d'un dépassement d'échéance, ainsi que les activités de recouvrement (ici l'arrêt de l'application).
- une alarme. Cette alarme est reconfigurée à chaque fois que l'échéance temporelle change suite à un changement d'état de l'automate.

TAB. 4.2: Quelques paramètres caractéristiques de l'abstraction temporelle d'un automate

N	(défini pour tout l'automate) nombre d'horloges utilisées dans l'automate temporisé.
BR_C	(défini pour une classe de l'automate) nombre de transitions sortantes pour une classe donnée, ici C. Ce nombre est appelé le degré de branchement de la classe d'équivalence
L_C	(défini pour une classe d'équivalence de l'automate) longueur du chemin maximal contenant la classe C et constitué uniquement de τ-transitions
BR_{max}	(défini pour l'automate) maximum des degrés de branchement de l'ensemble des classes d'équivalence de l'automate
L_{max}	(défini pour l'automate) maximum des valeurs L_C pour l'ensemble des classes d'équivalence de l'automate.
C_{max}	(défini pour l'automate) nombre de classes d'équivalence dans l'abstraction temporelle de l'automate.

Xenomai permet la création dynamique des tâches à l'exécution. L'ajout de deux tâches à une application au moment de son démarrage ne pose a priori aucun problème. Un effet de bord de la création de ces deux tâches pourrait être la réservation de deux niveaux de

priorité pour contrôler le comportement temporel de ces deux tâches. Xenomai compte 100 niveaux de priorité. Nous avons donc jugé, la encore, le coût raisonnable.

Xenomai implémente un service classique d'alarme logicielle temps réel. L'utilisation d'une alarme supplémentaire ne doit pas poser de problème. Ainsi, le détecteur peut être caractérisé en terme de coût de déploiement essentiellement par l'estimation de son empreinte mémoire et du temps processeur qu'il consomme. Nous allons tout d'abord introduire les notations utilisées pour désigner les paramètres clés de la complexité de l'abstraction temporelle.

4.3.1 Occupation de la mémoire

L'empreinte en mémoire du détecteur se répartit classiquement entre une plage réservée pour le code, une plage réservée pour les données, et la quantité de mémoire potentiellement consommée sur la pile du processus hébergeant l'exécution du détecteur. Dans notre cas le code du détecteur est constitué de deux parties. La première partie est constituée des fichiers, dits constant, dont la taille ne varie pas : le code du moteur de vérification et celui de manipulation de l'abstraction temporelle en ligne. La seconde partie correspond à la définition de l'abstraction temporelle[1] et des tâches de recouvrement. Cet aspect de notre prototype est clairement sous-optimal. Il est possible de concevoir l'abstraction temporelle d'un seul tenant et de la mémoriser sous la forme du plage de donnée à charger au moment du démarrage. En pratique, le chargement de ces données peut se faire avec un code relativement simple de taille constante. Au final, le code du moteur de vérification et celui utilisé pour le chargement de l'abstraction temporelle pourraient être de taille constante indépendante de la complexité du modèle à charger en mémoire. Ainsi, le facteur clés du coût mémoire du détecteur concerne la quantité de mémoire occupée par la structure de l'abstraction temporelle.

En ce qui concerne la zone de données, il faut comptabiliser la mémoire occupée par l'abstraction ainsi que celle occupée par les variables d'état du détecteur. La taille de l'état du détecteur est constante et relativement faible. Nous utilisons un entier pour désigner la classe courante, un entier long pour mémoriser la date du dernier événement, et un vecteur d'horloge au total l'état du détecteur est de l'ordre de $(4 + (N + 1) * 8)$ octets.

En ce qui concerne l'espace occupé par le modèle, cet espace peut être calculé statiquement. Il suffit de parcourir la liste des classes et de déterminer pour chacune d'elles la place qu'elle occupe. La structure mémorisant l'information associée à une classe d'équivalence C contient :

1. La liste des transitions événementielles sortant de C sous la forme d'une liste chaînée de structures. La liste contient par définition BR_C éléments constitués d'une

[1] L'abstraction est construite en mémoire sous la forme d'une structure chaînée

structure contenant :
– l'index d'événement : 4 octets
– l'index de classe d'équivalence : 4 octets
– le vecteur des resets (un vecteur de N booléens représentés ici par le type `int`) : $4N$ octets
– un pointeur vers la transition suivante : 4 octets.
sous-total $= BR_C * (4N + 12)$ octets

2. un entier indiquant l'index du successeur temporel de la classe courante, s'il existe : 4 octets.
3. un vecteur d'entiers pour stocker les bornes de la région temporelle associée à la classe : $4N$ octets
4. un vecteur d'entiers (4 octets) pour mémoriser les bornes temporelles servant à définir le vecteur d'échéance de la classe : $4N$ octets
5. $* \; total = BR * (4N + 12) + 8N + 4$ octets

En bref, nous obtenons une structure dont la taille est bornée par $O(BR_{max} * N * C_{max})$ dans le pire des cas[2].

On peut distinguer grossièrement deux phénomènes menant à des abstractions temporelles complexes. Le premier cas est relativement évident et correspond au cas où l'automate temporisé est lui-même très complexe. Dans ce cas, son abstraction est au moins aussi complexe. Il est fréquent de définir une spécification comportementale à travers un "réseau" d'automates communicants. La composition de nombreux modèles élémentaires de faible complexité a tendance à engendrer un grand nombre de lieux et de transitions. Le protocole CSMA/CD a déjà été décrit en suivant ce procédé. Chaque émetteur possède un modèle. Le support de communication est lui aussi modélisé. La spécification s'obtient en composant tous ces éléments. Avec 5 émetteurs, le graphe contient environ 4000 lieux et 190000 transitions. Cela signifie que par lieu, il y a en moyenne un peu plus de 47 transitions possibles. Les cinqs horloges entraînent une explosion combinatoire de la complexité du processus de calcul de l'abstraction temporelle.

La deuxième source de complexité vis à vis de l'abstraction temporelle est l'existence de cycles au sein desquels au moins une horloge n'est pas remise à zéro. Ces cycles servent essentiellement à décrire le fait que l'on puisse réexécuter en boucle une séquence d'événements de telle sorte que l'exécution totale de la boucle est contrainte temporellement.

Si l'on considère les automates comme un moyen de spécifier des contraintes comportementales et non de décrire des implémentations. Il est raisonnable de considérer le second cas comme étant le problème numéro 1 auquel nous seront confrontés lorsqu'il

[2]Les constantes BR_{max}, N et C_{max} sont définies en début de section cf page 105

sera nécessaire de traiter des spécifications complexes écrites «à la main». Ceci ne signifie pas que nous réfutons le premier cas de figure. Nous souhaitons simplement souligner le fait que dans le second cas, la partition de l'ensemble des vecteurs d'horloge pour chaque lieu sera relativement complexe.

Dans le cas du modèle global de CSMA/CD avec deux émetteurs, nous obtenons un peu plus de 13000 classes, avec un degré de branchement moyen inférieur à 10, des séquences de τ-transition de taille inférieure à 18, et trois horloges. Cela donne une occupation mémoire de l'ordre du méga-octets pour ce modèle relativement complexe.

4.3.2 Temps d'exécution

Nous nous sommes surtout intéressés à analyser le coût du moniteur en terme de temps d'exécution. Le coût de l'exécution peut être considéré selon deux aspects en fonction de l'utilisation que l'on souhaite faire de son analyse. La complexité de l'algorithme de vérification utilisé pour chaque événement nous permettrait de prédire la vitesse d'accroissement du temps nécessaire pour vérifier des événements dans des modèles de compléxité croissante. Ceci permet d'obtenir une idée de la complexité des descriptions comportementales auxquelles ce type de détecteur peut être appliqué. D'un autre coté, le prototype intègre de nombreuses opérations liées au contrôle de l'exécution du moniteur et de l'application qu'il surveille. Ces opérations sont toujours exécutées que la spécification soit complexe ou non. Il est donc intéressant de comprendre quel est le coût de ces actions de contrôle d'exécution vis-à-vis du coût des calculs de vérification.

Nous proposons de mesurer séparément le coût des appels systèmes, et les portions du code correspondant à la logique de vérification. Nous avons tenté de limiter au maximum l'influence des interruptions sur cette campagne de mesures. Cependant, ces interruptions peuvent déclencher un accroissement brusque du temps d'exécution de bout en bout du moniteur. Elles font partie du bruit de fond du système d'exploitation. Nous avons donc essayé de limiter ce bruit de fond dans l'expérimentation en ne créant pas d'interruption dans l'application témoin utilisée au cours de l'expérience.

4.3.2.1 Estimation de la complexité de la vérification

Dans un premier temps, il faut déterminer les opérations réalisées lors de la vérification dont le coût change en fonction de la complexité du modèle.

La décomposition du processus de vérification peut être raffinée pour séparer les actions directement reliées au contrôle de l'exécution de l'application et les opérations spécifiques à la vérification de l'automate, cf figure 4.5. Dans l'implémentation courante du moniteur, l'état est mis à jour pas à pas en franchissant les τ-transitions représentant l'écoulement du temps dans l'abstraction temporelle. Cela correspond à parcourir

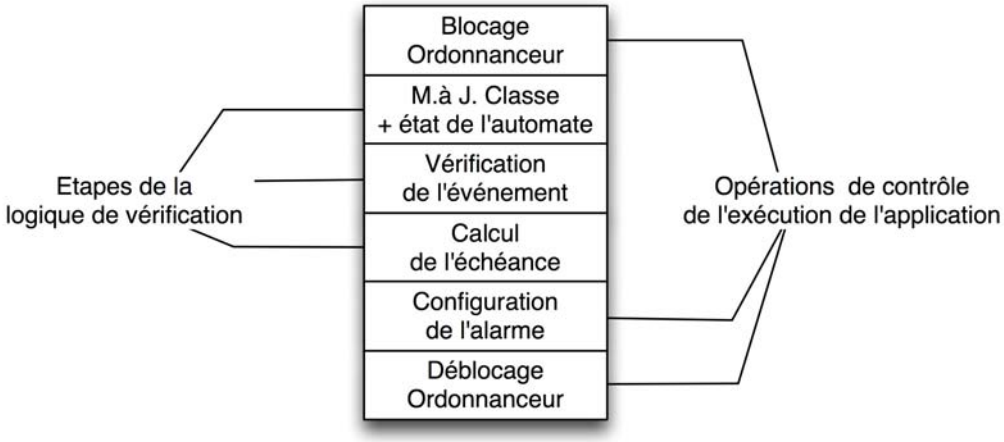

FIG. 4.5: Séparation des étapes de vérification

les transitions temporelles dans l'abstraction temporelle jusqu'à trouver la classe classe d'équivalence contenant le vecteur d'horloge mis-à-jour, cf page 97. Si l'on réalise cette recherche en parcourant le graphe, la recherche nécessite au plus L_{max} tests, rappelons que L_{max} est la longueur maximale d'une séquence de τ-transitions dans l'abstraction temporelle. Le test pour déterminer si l'on doit franchir une τ-transition correspond dans le pire cas à effectuer N comparaisons élémentaires entre le vecteur d'horloge de l'état courant et celui définissant la frontière de la région temporelle. Ainsi, le temps nécessaire pour mettre à jour la classe d'équivalence occupée est, dans le pire des cas, proportionnel à $L_{max} * N$.

La seconde opération consiste à tester dans la classe occupée après mise à jour, si l'événement correspond à une transition valide depuis cette classe. Dans ce cas les transitions valides sont mémorisées dans une liste par index d'événement croissant. Le pire cas, en terme de temps d'exécution, correspond au cas où l'on a un événement interdit. En effet, il faudra parcourir l'ensemble de la liste des transitions valides avant de pouvoir conclure sur l'absence d'une transition événementielle sortante. Pour chaque transition, il suffit de tester si l'index de l'événement porté par la transition testée correspond à l'index de l'événement en cours de vérification. La complexité de cette opération est proportionnelle dans le pire des cas au nombre de transitions événementielles permettant de sortir d'une classe d'équivalence. Ce nombre est majoré dans tous les cas par BR_{max}

La troisième opération dépend de la complexité du modèle, mais dans une moindre mesure comparativement aux deux opérations précédentes. Lors de la mise en place l'alarme permettant d'intercepter les erreurs dues à des dépassements d'échéances, une soustraction vectorielle est réalisée entre deux vecteurs d'horloge de taille N.

Pour résumer, le temps nécessaire pour exécuter par le vérifieur dépend des trois paramètres N, BR_{max} et L_{max}. Cependant, la dépendance entre le temps d'exécution et chaque paramètre est linéaire. Cette propriété est l'un des bénéfices attendus du calcul de l'abstraction temporelle et de son intégration au détecteur. Après avoir estimé l'accroissement de la complexité de la vérification d'un événement, il faut déterminer le coût réel de la vérification pour un nombre raisonnable de modèles.

4.3.2.2 Processus expérimental

La démarche expérimentale est relativement simple. Nous souhaitons évaluer le coût temporel du détecteur en l'absence d'erreur. Cela revient à mesurer son temps d'exécution pour chaque événement vérifié au cours d'une exécution.

Objectif détaillé de l'expérimentation :
Nous souhaitons mesurer le temps d'exécution du détecteur en fonction de la complexité du modèle. Nous avons vu que la complexité de la vérification d'un événement varie en fonction du nombre d'horloges, du nombre de τ-transitions à franchir pour mettre à jour l'état du détecteur, et du nombre de transitions autorisées dans la classe où l'événement doit être vérifié. Nous souhaitons évaluer le temps d'exécution du détecteur lorsque ces trois aspects de la complexité de la vérification d'un événement sont totalement maîtrisés. Pour cela il faut mesurer de manière répétée le temps d'exécution du détecteur pour des exécution comparable du détecteur. Un contexte de vérification correspond à l'état du détecteur avant la vérification d'un événement, au temps écoulé depuis, et à la fraction de l'abstraction temporelle utilisée durant la vérification. La connaissance de ces trois aspects permet de prédire de manière macroscopique la façon dont les algorithmes de vérification s'exécutent : le nombre d'itération dans les boucles, les branches des conditions exécutées... À partir de là, il est possible de définir des contextes de vérification engendrant des exécutions comparables des algorithmes de vérification correspondant à des scénarios «dévaforables» d'exécution.

Contraintes expérimentales :
Pour évaluer ce temps d'exécution, nous mesurons le temps de bout en bout entre le début et la fin de la vérification d'un événement. Ce temps nous paraît sous certaines conditions être un bon estimateur du coût du moniteur. L'intervalle de temps entre le début et la fin de l'exécution de la vérification d'un événement se décompose en trois types d'intervalles. Chaque type d'intervalle caractérise la nature des actions exécutées sur le processeur.

Tout d'abord, il est possible que le code du vérifieur soit interrompu par une autre activité. Ainsi, une partie du temps séparant le début et la fin de la vérification est po-

tentiellement consacré à l'exécution de traitements d'interruptions et/ou de tâches. La désactivation de l'ordonnanceur consiste à bloquer l'exécution de la fonction permettant de déclencher la réattribution du processeur. Dès lors seuls les traitements d'interruption peuvent interrompre l'exécution du vérifieur en ligne.

Le temps passé à exécuter le code de vérification est ensuite décomposé en deux types d'intervalles : le temps passé à exécuter des appels système de contrôle de l'exécution de l'application et de son environnement, et le temps passé à analyser et mettre à jour l'état du détecteur. La mesure des différentes durées d'exécution se fera à travers des sondes logicielles fournies par le système d'exploitation. Le point délicat lorsque que l'on utilise des sondes logicielles tient dans la précision des dates retournées par les fonctions de mesure du temps. Vis-à-vis des primitives de mesure de l'écoulement du temps, il est nécessaire de s'assurer :

- que le temps d'exécution de la fonction de mesure du temps ne varie pas trop en fonction du contexte d'appel.
- que la quantité retournée ait toujours le même sens vis-à-vis de l'écoulement du temps.

Nous avons sélectionné la primitive `rt_timer_tsc`. Cette fonction lit simplement le compteur de cycle du processeur. Ce compteur est mémorisé sur un champ de taille suffisante pour faire l'hypothèse que sa valeur soit strictement croissante (pas de dépassement de format). De plus, puisque les mécanismes d'économie d'énergie ont été désactivés sur la plate-forme d'expérimentation (un pentium III 1Ghz) le cycle possède une durée constante. Cette opération requiert 50 cycles processeur ce qui est négligeable par rapport à la durée d'un appel système classique de Xenomai depuis l'espace utilisateur, à savoir au moins 1500 cycles. Nous n'avons pas constaté de variation particulière de ce temps de traitement. Cependant pour minimiser la perturbation des résultats expérimentaux nous avons réalisé un grand nombre de mesures du même phénomène pour pourvoir lisser l'impact de la plate-forme d'exécution.

Mise en œuvre :

Nous souhaitons pouvoir mesurer de manière répétée le temps d'exécution correspondant à des exécutions comparables des algorithmes de vérification. Nous allons pour cela positionner avant chaque mesure le détecteur dans des états équivalents engendrant le même parcours de l'algorithmes de vérification. Nous traiterons les actions de contrôle comme des boites noires. Le déterminisme algorithmique n'assure pas nécessairement le déterminisme temporel du détecteur. Ceci est d'autant plus vrai que des appels système sont exécutés au sein du code de vérification. Ceci justifie le fait de répéter un grand nombre de fois le processus de vérification.

L'application génère un flux d'événements qui périodiquement replacent le détecteur dans un contexte de vérification équivalent :

- La classe d'équivalence avant le début de la vérification est la même.
- Le temps écoulé depuis le dernier événement, le vecteur d'horloge et l'abstraction temporelle sont conçus de telle sorte qu'après mise à jour de l'état courant la classe dans laquelle l'événement est vérifié est la même.

Ces deux contraintes permettent de s'assurer que, pour chaque expérience, la mise à jour de la classe d'équivalence de l'état courant correspond au même parcourt de l'abstraction temporelle . Ceci a pour conséquence de générer des exécutions équivalentes de la fonction Update() définie dans l'Algorithme 1. Nous procédons à un nombre raisonnable de mesures du temps d'exécution pour un même contexte de vérification pour limiter la sensibilité du processus de mesure aux défauts du support d'exécution et des mécanismes de mesure (3000 mesure pour chaque expérience). Nous générons artificiellement un automate dont seule une portion de l'abstraction temporelle sera parcourue à l'exécution. Nous allons utiliser un comportement observable suffisamment prédictible pour s'assurer qu'à chaque mesure du temps de vérification, le détecteur se retrouve dans des conditions de vérification «équivalentes».

L'application témoin est une tâche périodique avec une phase d'initialisation terminée par l'émission de l'événement e_0 mettant à zéro toutes les horloges de l'automate. La transition sur e_0 amène le détecteur dans un une classe d'équivalence C_1 contenue dans un chemin cyclique de transitions de l'abstraction temporelle. Un tel cycle donne l'occasion de replacer le système dans un même contexte de vérification pré-défini.

Le cycle de classe d'équivalences considéré est constitué de la plus petite séquence de classes d'équivalence permettant de replacer de manière répétée le détecteur dans un contexte de vérification déclenchant le «pire scénario d'exécution» pour une abstraction temporelle fixée dont les paramètre sont L_{max}, BR_{max}, et N.

Ce cycle est représenté de manière informelle dans la figure 4.6. Plus précisément, le cycle est constitué d'une séquence de $L + 1$ classes uniquement connectées par des τ-transitions. Ces classes correspondent à des intervalles croissants de valeurs de la première horloge utilisée dans l'abstraction, ici notée x_1. Appelons C_1 la première classe de ce cycle. Cette classe est la classe de destination de la transition déclenchée par l'événement e_0 depuis la classe C_0. La dernière classe de cette séquence de τ-transitions est notée C_{L+1}, et correspond à des valeurs de x_1 comprises entre T et $T + \Delta$. Les paramètres T et Δ pourront être ajustés pour nous permettre de créer une application générant au bout d'un certain temps uniquement des événement dans la classe C_{L+1}. Cette classe d'équivalence possède BR transitions événementielles sortantes chacune pointant vers la classe C_1. Cette abstraction temporelle permet de replacer le détecteur de manière périodique dans le contexte de vérification suivant :

État du détecteur : la classe d'équivalence avant le début de la vérification correspond à la classe C_1. L'événement à vérifier doit correspondre à un état de l'automate dans la classe C_{L+1}

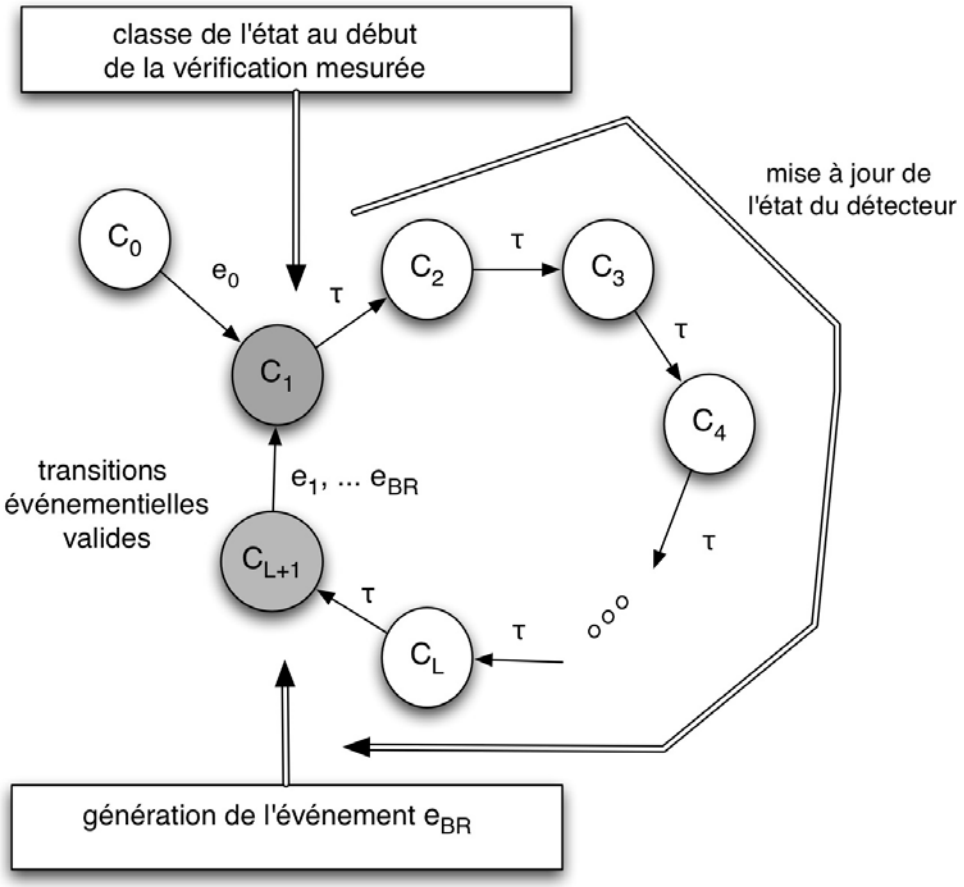

FIG. 4.6: Abstraction temporelle pour la répétition des conditions expérimentales

Phase 1 : Dans ce contexte de vérification la mise à jour de la classe d'équivalence impose le franchissement de L τ-transitions.

Phase 2 : *BR* événements sont autorisés dans la classe C_{L+1}. Nous faisons en sorte que l'événement émis, noté ici e_{BR}, corresponde à la dernière transition de la liste des transitions autorisées. En procédant de la sorte, nous nous assurons que la vérification de l'événement correspond au pire cas : le parcours complet de la liste des transitions valides de la classe courante.

Phase 3 : Le franchissement de cette transition déclenche la remise à zéro de l'horloge x_1. Une fois cette transition validée, la classe de destination est toujours C_1, ceci nous assure que la nature de l'échéance calculée en C_1 est toujours la même.

Pour s'assurer que le détecteur se retrouve toujours dans ce même contexte d'exécution, il suffit d'utiliser une alarme périodique qui déclenche le réveil d'une tâche de priorité maximale dont la première action correspond au signalement de l'événement e_{BR}. T et Δ doivent être judicieusement choisis. En effet, il faut pouvoir s'assurer que l'on est capable de générer l'événement e_{BR} de manière répétée avec une durée séparant deux instances comprise dans l'intervalle T et $T + \Delta$. L'application témoin utilisée pour mener les expériences est relativement triviale et se contente d'émettre périodiquement un événement choisi, e_{BR}. Cette expérience est paramètrable en fonction de L, N et BR et permet de mesurer le temps d'exécution du détecteur pour des scénarios d'exécution comparables où l'on fait varier les attributs caractéristiques de la complexité de l'abstraction temporelle.

Résultats expérimentaux :

Les expériences réalisée correspondent à des paramètres entiers *BR*, *L* et *N* pris respectivement respectivement dans les intervalles $[1, 20]$, $[1, 20]$, $[1, 9]$. Nous avons tronçonné le processus de vérification en 6 phases comme indiqué figure 4.5. La durée de chaque phase est étiquetée «contrôle d'exécution» ou «vérification». Pour chaque expérience une distribution empirique des temps mesurés peut être construite à partir des 3000 mesures collectées. Si l'on considère le temps total d'exécution de la tâche de vérification pour le triplet $BR = 20, L = 20, N = 9$, on obtient l'histogramme des fréquence de la figure 4.8

On remarque que les percentiles[3] permettent de mieux cerner la répartition des valeurs :

pourcentages	90%	95%	98%	99%	99, 5%
valeur (en μs)	24.1	24.3	25.8	30.5	44.1

[3]Rappel : le XX-percentile correspond à la valeur v d'un ensemble de N données telle que XX% des données sont inférieures à v.

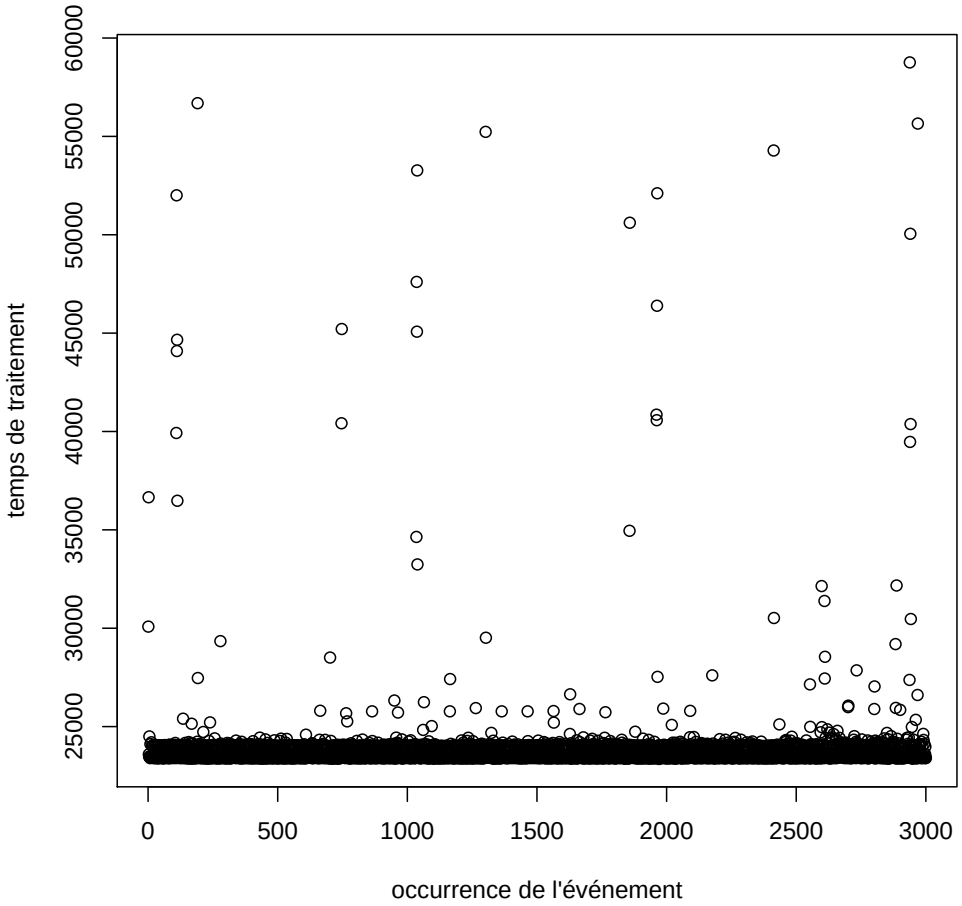

FIG. 4.7: Perturbation homogène des mesures

Ce graphique représente la séquence des 3000 valeurs mesurées au cours de l'expérience. La répartition des valeurs extrêmes est relativement homogène au cours de l'expérience.

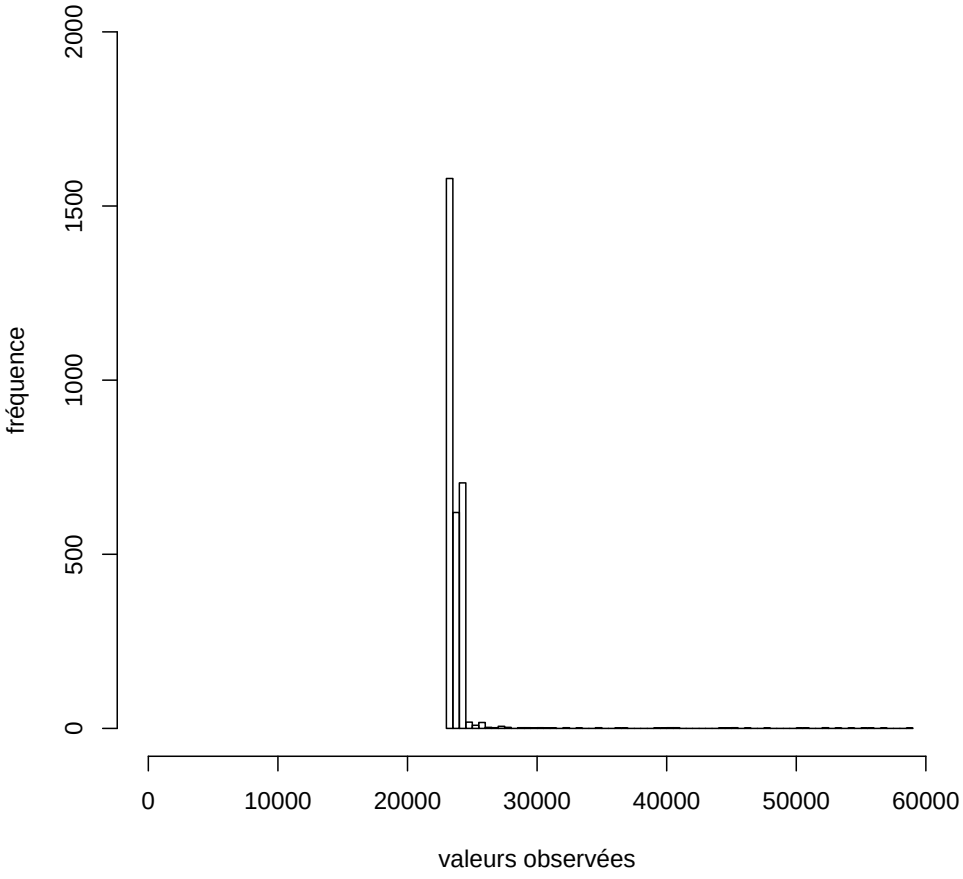

FIG. 4.8: Fréquences des temps d'exécution mesurés dans l'expérience $BR = 20, L = 20, N = 9$

Ce type de répartition se retrouve sur chaque expérience. Nous avons fait l'hypothèse que les cas «isolés» où le temps mesuré est particulièrement grand (deux à trois fois plus grand) correspondent au bruit ambiant du système d'exploitation. Nous avons vérifié cette hypothèse sur une application de calcul non équipée par le moniteur et devant avoir un comportement déterministe, et non préemptible. Le même type de problème a été constaté avec une magnitude moindre.

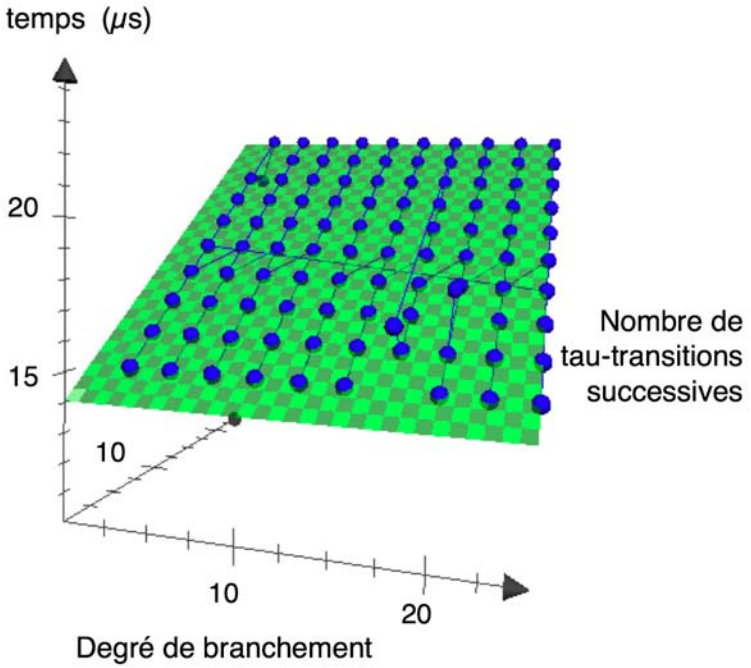

FIG. 4.9: 98^{ime} percentile du temps d'exécution du détecteur en fonction de *BR* et *L* (avec N=9)

Pour représenter l'évolution des temps mesurés, nous avons décidé de fixer l'un des trois paramètres à sa valeur maximale pour tracer l'évolution du temps d'exécution en fonction des deux autres paramètres. Seulement deux paires parmi les trois possibles ont été considérées : BR et L, puis L et N. Pour chaque paire de valeurs, nous avons tracé la surface représentant le 98-percentile du temps total d'exécution du détecteur.

Nous avons pu nous rendre compte qu'à valeur de N fixée, le temps d'exécution croit plus fortement à cause d'un accroissement de L_{max}, qu'en raison d'un accroissement de

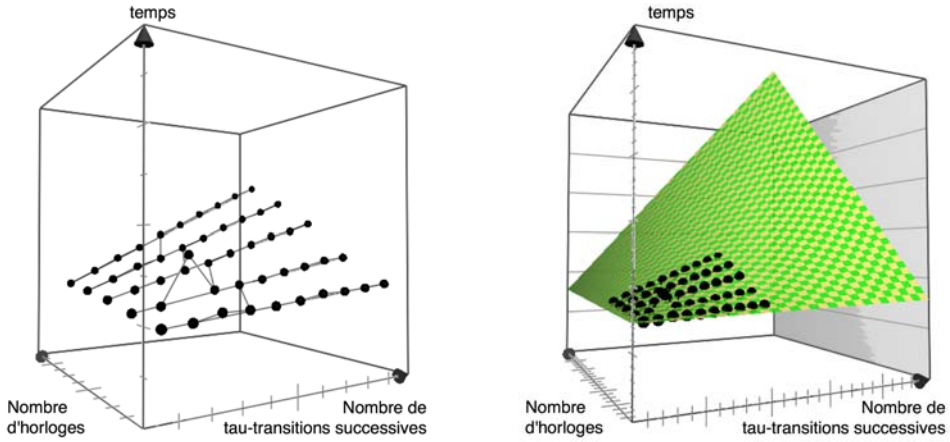

FIG. 4.10: 98^{ime} percentile du temps d'exécution du détecteur en fonction L et N
A gauche, nous avons affiché le nuage de points effectivement mesurés. A droite, nous avons superposé à ce nuage de points la surface extrapolé dont l'équation cartésienne est de la forme $z = a.x * y + b.x + c.y + d$

BR, cf figure 4.9. La configuration la plus négative est un accroissement de N et L qui donnerait l'impression d'un accroissement quadratique du temps d'exécution, cf figure 4.10. Ce dernier cas n'est pas anodin puisqu'il semble assez logique qu'avec un nombre important d'horloges, la complexité des contraintes temporelles soit assez élevée. Si les contraintes temporelles sont complexes la segmentation de l'ensemble des vecteur est potentiellement importante. Ceci se traduirait par une borne L_{max} élevée pour l'abstraction temporelle.

Le dernier point que nous souhaitons souligner concerne le poids relatif du temps passé à exécuter des appels systèmes permettant le contrôle d'exécution, et celui passé à exécuter la logique de vérification. Le temps passé à «contrôler» l'exécution de l'application n'est en aucun cas censé varier en fonction de la complexité du modèle. Nous avons vérifié cette hypothèse avec un bémol puisque ce que nous avons appelé le bruit ambiant semble fausser toujours de 1% à 2% des données. Nous avons constaté que, dans toutes les expériences considérées , le temps passé dans les phases de contrôle de l'exécution et celui passé dans les étapes de vérification proprement dites sont du même ordre de grandeur (à savoir il existe un facteur multiplicatif inférieur à 10 les liant) et ce sur 100% des données.

En analysant le temps passé dans les différentes étapes, il en ressort que l'étape la plus coûteuse est celle de mise à jour de la classe d'équivalence courante avant prise en compte de l'événement. Il serait donc intéressant d'envisager l'encodage des séquences de classes

liées par τ-transitions sous forme d'arbre au lieu d'une liste. De plus, une implémentation de la fonction de vérification dans l'espace noyau permettrait de gagner 1,5 microseconde sur chaque appel système, c'est à dire 4,5 micro secondes au total. Ceci représente entre 10 et 20% du temps d'exécution du détecteur.

Conclusion

A la fin du chapitre 3, nous étions en possession d'un algorithme de haut niveau permettant de mettre en place un service de détection au plus tôt pour une spécification définie via un automate temporisé. Cet algorithme était «désincarné» puisqu'il supposait une exécution indépendante de l'algorithme de vérification et de l'application. En pratique le code de vérification doit se prémunir des différentes suspensions d'activités déclenchées par l'ordonnanceur. Nous avons proposé dans ce chapitre une implémentation de cet algorithme sur la plate-forme de développement et d'exécution d'applications temps réel Xenomai.

Le prototype proposé permet d'intercepter les événements non autorisés par la spécification et de déclencher le processus de recouvrement. Cela se fait par exemple, en bloquant la progression de l'exécution de l'application et en déclenchant le réveil d'une tâche de recouvrement. Le recouvrement des erreurs dues à un dépassement d'échéance se fait lui aussi avec a priori une latence de l'ordre d'un changement de contexte, au mieux.

La description du prototype couvre les aspects liés à l'architecture logicielle assurant un bon contrôle de l'exécution de l'application surveillée, et les algorithmes et structures de données concrètement utilisés pour réaliser la vérification. La conception du détecteur à partir d'un modèle nous a permis de bien séparer le code qui peut être réutilisé de celui qui doit être régénéré pour chaque application. Cette séparation nous a permis de mettre en place un processus automatique pour générer le code complet du détecteur, et l'intégrer à une application donnée. L'outil est distribué librement[4] bien qu'il ne soit qu'à l'état de prototype.

Ce chapitre de description du prototype est clos par une étude du coût du détecteur en terme de code ajouté à l'application, d'empreinte mémoire, et de temps processeur consommé. Le détail des algorithmes et structures utilisés dans les différentes étapes du processus de détection, nous a permis de déterminer l'accroissement de l'empreinte mémoire et temporelle du détecteur. En effet, ces deux aspects du coût d'intégration du détecteur varient en fonction de paramètres représentatifs de la complexité de la spécification vérifiée. Les résultats de cette étude ont pu être complétés par des résultats expérimentaux.

[4] http://www.laas.fr/~trobert/RTRV/

Conclusion générale

Le développement d'applications temps réel critiques est un domaine pour lequel il existe déjà toute une panoplie de méthodes permettant la réalisation de systèmes sûrs de fonctionnement. Il est possible de distinguer les propositions d'architectures et de méthodes de développement visant à assurer la tolérance aux fautes pour l'ensemble du système, des propositions de mécanismes isolés permettant simplement d'améliorer localement le comportement d'un logiciel en présence de fautes. La notion de «wrapper» ou de vérifieur en ligne est séduisante en raison de la facilité avec laquelle ce type de mécanisme s'intègre à un logiciel déjà existant et à sa plate-forme d'exécution. Plus récemment, la possibilité de vérifier des comportements plus complexes à l'exécution a été identifiée comme une arme à double tranchant. Ces mécanismes permettent en théorie de vérifier des comportements formellement décrits relativement complexes. Cependant, en pratique nous avons remarqué plusieurs problèmes. Tout d'abord, la vérification de modèles ou propriétés complexes en ligne a d'abord été traité sans nécessairement analyser les caractéristiques du processus de détection (latence, prédictibilité, robustesse) ainsi créé.

Nous avons étudié au cours des travaux présentés dans ce manuscrit les phénomènes pouvant augmenter la latence de détection de ces détecteurs. Nous avons pris en compte la latence de détection tout au long de la chaîne logique menant d'une définition formelle de la tâche de vérification en ligne et des ses propriétés, à son implémentation pour un formalisme relativement connu, les automates temporisés. Lorsque les erreurs et leur symptômes ne sont pas explicitement définis, alors le processus de dérivation des symptômes d'erreur peut ne pas être optimal du point de vue de la latence de détection des erreurs. La notion de détection au plus tôt caractérise un détecteur qui utilise les symptômes les plus courts pour détecter l'apparition d'une erreur. Cette étude a pris pour fil conducteur la notion de détection au plus tôt et l'a développée tout au long du processus de raffinement menant de la spécification à l'implémentation du vérifieur. Idéalement, un détecteur causal «au plus tôt» est le détecteur d'erreur qui minimise la latence de détection des erreurs pour un bloc d'observation et une définition des exécutions correctes fixés.

En pratique, il est évident que la latence de détection est en partie causée par les calculs et vérifications exécutées pour assurer la détection. Cependant, de nombreuses tâches de vérification ne peuvent être résolues uniquement à partir d'une information instantanée. Dans ce cas, la latence peut aussi être causée par une mise en pause du processus de détection en attendant une nouvelle observation. Si cette observation était inutile, le signalement de l'erreur est donc retardé pour rien. Nous avons proposé un cadre de description pour le processus d'observation et pour le bloc d'analyse constituant le vérifieur en ligne. C'est à partir de là que nous avons commencé à définir les caractéristiques de la détection causale d'erreurs «au plus tôt». En particulier, nous avons souligné la différence entre une prédiction et une détection causale à travers le principe de monotonie (non révocation d'une décision passée) de l'observation et de la détection. Cette propriété de monotonie

du processus d'observation nous assure que le processus de détection n'est pas instable. En parallèle, nous utilisons une caractérisation formelle de la date d'apparition d'une erreur. Cette caractérisation n'est pas triviale lorsque les symptômes d'erreur doivent être déduits d'une description des comportements corrects de l'application. À partir de là, un modèle de détecteur est proposé : le détecteur causal d'erreurs «au plus tôt»

La suite logique de cette définition était de proposer un prototype de ce type de détecteur. Dans un premier temps, nous avons selectionné un formalisme de spécification de comportements temps réels, les automates temporisés. Puis, nous avons décrit comment déduire, d'une spécification proposée sous la forme d'un automate temporisé, les plus petits symptômes d'erreur – les traces identifiant la plus petite séquence d'observation prouvant qu'une trace est incorrecte. La difficulté est de comprendre comment capturer ces traces de manière précise à l'exécution. C'est ici que la compréhension du mode de fonctionnement du bloc d'observation nous a permis de proposer une méthode pour capturer correctement l'ensemble des plus petits symptômes d'erreur associés à un automate temporisé. Dans une troisième phase, nous avons implémenté ce détecteur pour une architecture relativement générique tant au niveau du logiciel que du matériel. Le prototype s'intègre facilement à des applications développées sous l'interface de programmation Xenomai 2.3. Cette interface de programmation propose une série de services temps réel génériques bien différents de ceux habituellement recommandés ou utilisés lors de l'implémentation de systèmes ultra-critiques temps réel. La volonté était de fournir un mécanisme permettant de vérifier des contraintes temporelles de bout en bout dans un contexte où le temps réel est implémenté sur un support d'exécution faible (offrant assez peu de garanties mais suffisamment pour correctement exécuter le détecteur). Une série d'expériences a été conduite pour estimer le coût temporel de l'exécution du moniteur et surtout ses variations en fonction de la complexité des modèles vérifiés.

À l'issue de ces travaux, plusieurs constatations doivent être faites. Nous discuterons des pistes de recherche identifiées et laissées en suspens dans un second temps. Tout d'abord, la vérification en ligne peut devenir assez vite aussi complexe qu'une exécution dupliquée du service que l'on surveille. Ceci est d'autant plus vrai lorsque l'on manipule un modèle exécutable pour réaliser la vérification. Plus le modèle est précis plus la distance entre une implémentation et le modèle se raccourcit. Ensuite, il faut conserver en tête que le vérifieur en ligne est un compromis entre un coût de vérification et un pouvoir de séparation entre les états corrects et erronés. Dans le cas où les symptômes doivent être déduits d'une description des comportement corrects, le système ainsi créé peut s'avérer relativement inefficace. Dans de nombreux cas, il est suffisant de borner la latence de détection des erreurs. Cependant, certains processus de rétablissement de l'état du système voient leur efficacité largement diminuée lorsque cette latence de détection est élevée. Finalement, le processus de détection au plus tôt possède un réel intérêt lorsqu'il est combiné à des mécanismes de rétablissement de l'état du système tirant un réel

profit de la détection au plus tôt. En revanche, si le signal d'erreur est simplement utilisé à des fins de diagnostique post-mortem, le coût de mise en place d'un détecteur au plus tôt est prohibitif. Un exemple de combinaison efficace d'un détecteur au plus tôt et d'un mécanisme de rétablissement correspond au déclenchement de services dégradés alternatifs dès que l'erreur est détectée. Plus l'erreur est détectée tôt, moins il y a de gaspillage du temps processeur. Si le service doit finir par être délivré au bout d'une période fixe, alors la détection au plus tôt augmente a priori les chances du système de terminer son exécution correctement.

Nous avons identifié trois grand axes correspondant à des pistes de recherche dans la continuité du travail réalisé. Tout d'abord, il devient clair que la vérification en ligne d'un composant logiciel a un coût. Il serait intéressant de trouver des compromis pour un vérifieur en ligne entre la complexité de ses traitements, et sa latence de détection. Ce type d'analyse serait à rapprocher de la notion de vérification approchée où le modèle à vérifier est simplifié avant d'être transformé pour permettre la vérification en ligne. Dans un second temps, il serait intéressant de comprendre s'il est possible d'adapter la notion de détection au plus tôt pour des applications distribuées. Cela comprend l'adaptation de la notion de processus d'observation et de bloc d'analyse à une architecture distribuée. Le problème principal viendra de la localité des observations réalisées par le système et des délais de communication entre sites. Une troisième piste revient à tenter de comprendre ce qu'il se passe lorsqu'un bloc d'observation monotone causal est combiné avec un bloc d'analyse réalisant des prédictions non monotones sur l'absence ou la présence d'une erreur dans les états futurs du système. Ce type de comportement permettrait de compenser des latences élevées du bloc d'observation par des prédictions sur le futur d'une exécution.

Bibliographie

[ACCP98] E. Anceaume, G. Cabillic, P. Chevochot, and I. Puaut. HADES : A middleware support for distributed safety-critical real-time applications. In *18th International Conference on Distributed Computing Systems (18th ICDCS'98)*, Amsterdam, The Netherlands, May 1998. IEEE. 18

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2) :172–206, 2002. 26

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994. 5, 56, 59

[AH91] Alur and Henzinger. Logics and models of real time : A survey. In *REX : Real-Time : Theory in Practice, REX Workshop*, 1991. 29

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput*, 1(1) :11–33, 2004. 10, 13, 17

[Alu99] R. Alur. Timed Automata. In *Proc. 11th International Computer Aided Verification Conference*, pages 8–22, 1999. 61

[AM99] Eugene Asarin and Oded Maler. As soon as possible : Time optimal control for timed automata. *Lecture Notes in Computer Science*, 1569 :19–30, 1999. 69

[AM04] Rajeev Alur and P. Madhusudan. Decision problems for timed automata : A survey. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2004. 65

[BBKT05] Saddek Bensalem, Marius Bozga, Moez Krichen, and Stavros Tripakis. Testing Conformance of Real-Time Applications by Automatic Generation of Observers. *Electr. Notes Theor. Comput. Sci*, 113 :23–43, 2005. 27

[BCTD98] Vittorio Brusoni, Luca Console, Paolo Terenziani, and Daniele Theseider Dupré. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence*, 102(1) :39–79, 1998. 28

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos : A model-checking tool for real-time systems. In *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, pages 546–550, London, UK, 1998. Springer-Verlag. 65, 101

[Ber89] G. Berry. Real time programming : special purpose or general purpose languages. Technical Report RR-1065, Inria, Institut National de Recherche en Informatique et en Automatique, 1989. 9

[BHPR07] Thomas Brihaye, Thomas A. Henzinger, Vinayak S. Prabhu, and Jean-François Raskin. Minimum-time reachability in timed games. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 825–837. Springer, 2007. 69

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006. 27, 30, 37

[BMN00] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1) :12–42, 2000. 29

[Bou03] Amine Boufaied. Contribution à la surveillance distribuée des systèmes à évènements discrets complexes. Thèse de l'Université Paul Sabatier, 2003. 28

[CM05] SÈverine Colin and Leonardo Mariani. *Run-Time Verification*, volume 3472 of *Lecture Notes in Computer Science*, chapter 18. Springer, 2005. 21

[CMV00] A. Casimiro, P. Martins, and P. Verssimo. How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–134, 2000. 54

[DGR04] Nelly Delgado, Ann Q. Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Software Eng*, 30(12) :859–872, 2004. 21, 25, 26, 34, 41

[dIA05] Dipartimento di Ingegneria Aerospaziale. Rtai - the realtime application interface for linux from diapm. https://www.rtai.org/, 2005. 18

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Berlin, June 1990. Springer. 93

[DJC94] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. Observer—A concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12) :900–913, December 1994. 27, 40

[DR05] Marcelo D'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In *CAV*, pages 364–378, 2005. 30

[Dru00] Doron Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN*, pages 323–330, 2000. 28, 30

[Dru06] D. Drusinsky. On-line monitoring of metric temporal logic with time-series constraints using alternating finit automata. *J.UCS : Journal of Universal Computer Science*, 12(5) :482–498, 2006. 30

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 1998. 64

[EH85] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30 :1–24, 1985. 28

[EKM+99] Anthony Egan, David Kutz, Dmitry Mikulin, Rami G. Melhem, and Daniel Mossé. Fault-tolerant RT-mach (FT-RT-mach) and an application to real-time train control. *Softw, Pract. Exper*, 29(4) :379–395, 1999. 18

[EKS06] Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of discrete event systems : Control theory approach. *Electr. Notes Theor. Comput. Sci*, 144(4) :21–39, 2006. 34

[FH02] Robert E. Filman and Klaus Havelund. Source-code instrumentation and quantification of events. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings : Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, Technical Report 02-06, pages 45–49. Department of Computer Science, Iowa State University, April 2002. 31

[GH01] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. IEEE Computer Society Press, Providence, 2001. 27

[GH03] Allen Goldberg and Klaus Havelund. Instrumentation of java bytecode for runtime analysis. In *In Proc. Formal Techniques for Java-like Programs, volume 408 of Technical Reports from ETH*, 2003. 32, 34

[Gha96] Malik Ghallab. On chronicles : Representation, on-line recognition and learning. In *KR*, pages 597–606, 1996. 28

[GRMD01] Ann Q. Gates, Steve Roach, Oscar Mondragon, and Nelly Delgado. DynaMICs : Comprehensive support for run-time monitoring. *Electr. Notes Theor. Comput. Sci*, 55(2) :1–17, 2001. 32

[Hen98] T. A. Henzinger. It's about time : Real-time logics reviewed. *Lecture Notes in Computer Science*, 1466 :439–454, 1998. 29

[HG05] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005. 20, 27, 34

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2) :193–244, 1994. 59

[HR01] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. In Klaus Havelund and Grigore Roşu, editors, *RV'01*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 23 July 2001. 28, 32, 34

[HR02a] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, Berlin, 2002. 34

[HR02b] Klaus Havelund and Grigore Roşu, editors. *Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, July 26 2002. Also DIKU technical report 02-14 from University of Copenhagen. 19

[Jef99] Clinton J. Jeffery. The alamo execution monitor architecture. *Electr. Notes Theor. Comput. Sci*, 30(4) :1–18, 1999. 32

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1) :112–126, 2003. 18

[KDK+88] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems : The MARS approach. *IEEE Micro*, 9(1) :25–40, 1988. 18

[KLS+02] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In Klaus Havelund and Grigore Roşu, editors, *Runtime Verification*, volume 70 of *Electronic Notes in Theoretical Computer Science*, pages 96–112. Elsevier Science, July 26 2002. 27, 28, 31, 34

[KPA03] Kåre J. Kristoffersen, Christian Pedersen, and Henrik Reif Andersen. Runtime Verification of Timed LTL using Disjunctive Normalized Equation Systems. *Electr. Notes Theor. Comput. Sci*, 89(2) :1–16, 2003. 29, 30

[KS00] Hagbae Kim and Kang G. Shin. Evaluation of fault tolerance latency from real-time application's perspectives. *IEEE Trans. Comput.*, 49(1) :55–64, 2000. 33

[KV01] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3) :291–314, 2001. 47

[KVBA+99] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally Specified Monitoring Of Temporal Properties. In *Proceedings of 11th Euromicro Conference on Real-Time Systems*, pages 114–22, York, UK, 9–11 June 1999. 32

[Lie95] Jochen Liedtke. On micro-kernel construction. In *In 15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, 1995. 18

[Lyn08] LynuxWorks. Embedded and real-time operating systems series. http://www.lynuxworks.com/, 2008. 18

[Mic] Sun Microsystem. Chorusos open sourced. http://www.experimentalstuff.com/Technologies/ChorusOS/index.html. 18

[MK03] Hidehiko Masuhara and Gregor Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003–Object-Oriented Programming 17th European Conference*, pages 2–28. Springer-Verlag, July 2003. 31

[ML97] A. Mok and G. Liu. Early Detection of Timing Constraint Violation at Runtime. In *The 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 176–186, Washington - Brussels - Tokyo, December 1997. IEEE. 27, 28, 30, 37

[MLWK02] Aloysius K. Mok, Chan-Gun Lee, Honguk Woo, and Prabhudev Konana. The monitoring of timing constraints on time intervals. In *IEEE Real-Time Systems Symposium*, pages 191–200, 2002. 30, 33

[MN04] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *FTRTFTS : Formal Techniques in Real-Time and Fault-Tolerant Systems : International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS*. LNCS, Springer-Verlag, 2004. 29

[OGRG07] Omar Ochoa, Irbis Gallegos, Steve Roach, and Ann Q. Gates. Towards a tool for generating aspects from MEDL and PEDL specifications for runtime

verification. In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 2007. 31

[Ose05] Osek/vdx, operating system specification 2.2.3. Osek group official release, 2005. 32

[PABD+99] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings. GUARDS : A generic upgradable architecture for real-time dependable systems. *IEEE Trans. Parallel and Distrib. Systems*, 10(6) :580–599, 1999. 18

[PM05] Gergely Pintér and István Majzik. Automatic Generation of Executable Assertions for Runtime Checking Temporal Requirements. In Mario Dal Cin, Andrea Bondavalli, and Neeraj Suri, editors, *Proc. The 9th IEEE International Symposium on High Assurance Systems Engineering (HASE 2005)*, pages 111–120, Heidelberg, Germany, October 12–14 2005. 29

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977. 28

[RFA00] Manuel Rodriguez, Jean-Charles Fabre, and Jean Arlat. Formal specification for building robust real-time microkernels. In *Proceedings of the 21st Symposium on Real-Time Systems (RTSS-00)*, pages 119–128, Los Alamitos, CA, November 27–30 2000. IEEE Computer Society. 29

[Rod02] Manuel Rodriguez Moreno. Technologie d'empaquetage pour la sûreté de fonctionnement des systèmes temps réels. Thèse de l'Institut National Polytechnique de Toulouse, 2002. 32, 34, 35

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1) :19–31, 1995. 19, 27

[RRJ92] Sitaram C. V. Raju, Ragunathan Rajkumar, and Farnam Jahanian. Monitoring Timing Constraints in Distributed Real-Time Systems. In *RTSS'92*, pages 57–67, 1992. 52

[SAÅ+04] Lui Sha, Tarek F. Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore P. Baker, Alan Burns, Giorgio C. Buttazzo, Marco Caccamo, John P. Lehoczky, and Aloysius K. Mok. Real time scheduling theory : A historical perspective. *Real-Time Systems*, 28(2-3) :101–155, 2004. 12, 18

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems. *Real-Time Systems*, 2 :247–254, 1990. 11

[TJ98] Kevin Templer and Clinton L. Jeffery. A configurable automatic instrumentation tool for ANSI C. In *ASE*, page 249, 1998. 32

[TR04] Prasanna Thati and Grigore Roşu. Monitoring algorithms for metric temporal logic specifications. In *RV'04*, 2004. 29

[TY01] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1) :25–68, 2001. 64, 65, 68

[UPP] site web de uppaal. http ://www.uppaal.com/. 56

[VR00] Paulo Veríssimo and Michel Raynal. Time in Distributed System Models and Algorithms. *j-LECT-NOTES-COMP-SCI*, 1752 :1–32, 2000. 54

[Wan04] Farn Wang. Efficient verification of timed automata with bdd-like data structures. *International Journal on Software Tools for Technology Transfer*, 6(1) :77–97, 2004. 93

[Win08] WindRiver. Site vxworks. http://www.windriver.com/products/vxworks/, 2008. 18

[Xen] Xenomai homepage. https://www.xenomai.org. 18, 99

Table des figures

Détection d'erreur au plus tôt dans les systèmes temps-réel : une approche basée sur la vérification en ligne

La vérification en ligne de spécifications formelles permet de créer des détecteurs d'erreur dont le pouvoir de détection dépend en grande partie du formalisme vérifié à l'exécution. Plus le formalisme est puissant plus la séparation entre les exécutions correctes et erronées peut être précise. Cependant, l'utilisation des vérifieurs en-ligne dans le but de détecter des erreurs est entravée par deux problèmes récurrents : le coût à l'exécution de ces vérifications, et le flou entourant les propriétés sémantiques exactes des signaux d'erreur ainsi générés. L'objectif de cette thèse est de clarifier les conditions d'utilisation de tels détecteurs dans le cadre d'applications « temps réel » critiques. Dans ce but, nous avons donné l'interprétation formelle de la notion d'erreur comportementale « temps réel ». Nous définissions la propriété de détection « au plus tôt » qui permet de d'identifier la classe des détecteurs qui optimisent la latence de détection. Pour illustrer cette classe de détecteurs, nous proposons un prototype qui vérifie un comportement décrit par un automate temporisé. La propriété de détection au plus tôt est atteinte en raisonnant sur l'abstraction temporelle de l'automate et non sur l'automate lui-même. Nos contributions se déclinent dans trois domaines, la formalisation de la détection au plus tôt, sa traduction pour la synthèse de détecteurs d'erreur à partir d'automate temporisés, puis le déploiement concret de ces détecteurs sur une plate-forme de développement temps réel, Xenomai.

Early error detection for real time applications : an approach using runtime verification

Runtime verification of formal specifications provides the means to generate error detectors with detection capabilities depending mostly on the kind of formalism considered. The stronger the formalism is the easier the speration between correct and erroneous execution is. Nevertheless, two recurring issues have to be considered before using such error detection mechanisms. First, the cost, at run-time, of such error detector has to be assessed. Then, we have to ensure that the execution of such detectors has a well defined semantics. This thesis aims at better understanding the conditions of use of such detectors within critical real-time software application. Given formal behavioural specification, we defined the notion of "behavioural error". Then, we identify the class of early detectors that optimize the detection latency between the occurence of such errors and their signalling. The whole generation process has been implemented for specifications provided as timed automata. The prototype achieves early error detection thanks to a preprocessing of the automaton to generate its temporal abstraction. Our contributions are threefold : formalisation of early detection, algorithms for timed automata run-time verification, and prototyping of such detectors on a real-time kernel, Xenomai.