Test de systèmes ubiquitaires avec prise en compte explicite de la mobilité

Pierre André

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)

Présentée et soutenue le *17/11/2015* **par :**

Pierre ANDRÉ

Test de systèmes ubiquitaires avec prise en compte explicite de la mobilité.

JURY

Ana Rosa Cavalli	Professeur	Rapporteur
François Taïani	Professeur des universités	Rapporteur
Olivier Alphand	Maître de conférence	Examinateur
Thierry Gayraud	Professeur des universités	Examinateur
Hélène Waeselynck	Directeur de recherche	Directeur de thèse
Nicolas Rivière	Maître de conférence	Directeur de thèse

École doctorale et spécialité :
EDSYS : Informatique 4200018
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Karama Kanoun, *Nicolas* Rivière et *Hélène* Waeselynck
Rapporteurs :
Ana Cavalli et *François* Taïani

Remerciements

Le travail présenté dans ce mémoire a été réalisé au sein du Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique (LAAS-CNRS). Je remercie M. Jean-Arlat de m'avoir accueilli dans son unité de recherche.

Au sein du LAAS-CNRS, je tiens à remercier Mme Karama Kanoun et M. Mohammed Kaâniche, responsables successifs de l'équipe TSF *Tolérance aux fautes et Sûreté de Fonctionnement* dans laquelle j'ai réalisé mes travaux de thèse.

Je tiens à exprimer ma profonde reconnaissance à mes encadrants Nicolas Rivière et Hélène Waeselynck, pour leurs conseils, le savoir scientifique et technique qu'ils m'ont enseigné, leur disponibilité, et leur patience. Ils ont su durant ces quelques années me pousser à développer mes idées et à les mettre en œuvre.

Je remercie également Mme Ana Rosa Cavalli et M. François Taïani pour avoir accepté de rapporter sur cette thèse. Merci également à M. Thierry Gayraud qui a présidé le jury de thèse. Enfin, je remercie l'ensemble des membres du jury, pour leur nombreuses questions pendant la soutenance, à savoir Mme Ana Rosa Cavalli, M. François Taïani, M. Thierry Gayraud et M. Olivier Alphand. Je leur suis très reconnaissant de l'intérêt qu'ils ont porté à mes travaux.

Je tiens aussi à remercier l'ensemble des personnes ayant travaillé sur la mise en place de notre méthode de test de systèmes mobiles, que ce soit lors de leurs thèses, de stages ou en tant qu'ingénieur d'études à savoir Zoltan Micskei, Aron Hamvas, Minh Duc Nguyen, Irina Nitu, Roxana-Alexandra Teodosiu, Mariem Touihri, Alexandre Leclercq et Jean-Marc Larré.

Durant ces quelques années au LAAS, j'ai aussi eu l'occasion d'enseigner. Cette expérience a été très enrichissante et l'accueil chaleureux qui m'a été réservé dans les deux départements où j'ai enseigné m'a beaucoup touché. Je tiens à exprimer mes remerciements à l'ensemble des personnels des départements GEII et EEA de l'Université Paul Sabatier.

Merci à l'ensemble des doctorants et post-doctorants TSF qui ont été là pour moi au cours de ma thèse, et avec qui j'ai partagé de très bons moments, Alexandru, Amina, Amira, Anthony, Benoît, Camille, Carla, Fanny, Fernand, Guillaume, Guthemberg, Hélène, Ivan, Jimmy, Joris, Julien, Kalou, Kossi, Lola, Ludovic, Mathilde, Matthieu, Maxime, Miguel, Miruna, Moussa, Olivier, Quynh Anh, Rim, Roberto, Rui, Thibaut, Thierry, Ulrich, William, Yann et ceux que j'oublie.

Enfin, je tiens à remercier ma famille, qui a toujours été là pour moi, et qui a su faire preuve de patience pendant ces années de thèse. Je remercie mes parents qui ont fait preuve d'un soutien sans faille depuis toujours, et grâce à qui j'ai pu arriver à ce stade. Pour finir, je tiens à remercier Iohan, Héloïse, Alicia, Maxime et Gabrielle mes neveux et nièces qui m'ont beaucoup apportés ces dernières années.

À tous un grand merci.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction : contexte et problématique

Sommaire

Ce manuscrit de thèse rapporte les travaux menés au *Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS)* de Toulouse et plus particulièrement au sein de l'équipe *Tolérance aux fautes et Sûreté de Fonctionnement Informatique (TSF)*. Cette thèse se focalise sur le test de systèmes informatiques mobiles, plus précisément sur l'utilisation de scénarios pour décrire et tester le comportent de ce type de systèmes.

1.1 Contexte

Les réseaux mobiles ad-hoc posent de nouveaux challenges pour les activités de développement, de vérification et de validation. En plus des questions posées par les systèmes distribués fixes, les systèmes distribués mobiles introduisent des questions supplémentaires liées par exemple à la dynamicité de la structure du système ainsi qu'à la sensibilité au contexte. Les dispositifs mobiles sont constamment en mouvement, les connexions et déconnexions des liens de communication entre eux sont fréquentes. Dans ces conditions, il n'est pas rare que l'envoi d'un message se traduise par un échec. L'état d'une application ne dépend plus seulement des messages reçus des autres, il prend aussi en compte son contexte, par exemple sa position actuelle fournie par un GPS ou toute autre information sur son environnement. Une approche de test pour ces systèmes devra prendre en compte toutes ces spécificités.

Le test passif est une approche classique pour tester les systèmes distribués, c'est-à-dire que le testeur peut vérifier le comportement du système sous test par l'analyse de traces d'exécutions, mais sans pouvoir interagir avec lui. Dans ce-cas là, plutôt que de définir des cas de test avec des entrées précises et les sorties attendues, la trace des messages est utilisée pour vérifier des propriétés. Ces propriétés expriment des exigences de test. L'objectif du test est alors d'exécuter le système

sous test dans différentes situations jusqu'à ce que l'ensemble des objectifs de test soit couvert et que toutes les exigences soient satisfaites.

1.2 Problématique

La qualité d'un logiciel est un enjeu important pour les entreprises qui les conçoivent. Généralement, pour les applications non critiques, la qualité du logiciel est assurée par une étape de test mais dans le cadre d'applications ubiquitaires les techniques de vérifications classiques sont-elles adaptées ?

Ce travail de thèse s'inscrit dans le développement d'une méthode de test pour des applications mobiles évoluant dans un environnement réseau sans fil peu structuré. Un premier travail a été initié dans l'équipe TSF et a abouti à la soutenance d'une thèse [Nguyen 2009]. Cette première thèse a permis, de définir une approche basée sur des descriptions de scénarios de test. Cette approche étend les langages de scénarios classiques afin de couvrir les spécificités des systèmes ubiquitaires. Ces extensions sont, par exemple, la représentation graphique des relations spatiales entre les nœuds mais aussi la gestion des événements de diffusion dans le voisinage couramment appelé *broadcast*.

1.3 Contributions

La méthode de test définie durant la précédente thèse était limitée aux spécifications du langage et à la partie recherche de relations spatiales entre les nœuds qui avait été mise en œuvre dans un logiciel appelé *GraphSeq*. Pour répondre à notre problématique, nous avons dans un premier temps réalisé une implémentation du langage de description de scénarios dans un environnement *UML* à l'aide de *Profils UML*, ce langage est nommé *TERMOS* pour *TEst Requirement language for MObile Settings*.

Une deuxième étape consiste à transformer les scénarios écrits en suivant la syntaxe du langage *TERMOS* en deux éléments : une séquence de graphes utilisable par *GraphSeq* d'une part et un automate facilement vérifiable sur une trace d'exécution d'autre part. Préalablement à cette étape de transformation, nous avons intégré un ensemble de vérifications réalisées sur chaque scénario permettant de garantir la bonne forme de celui-ci.

Dans un même temps, afin de pouvoir représenter des scénarios toujours plus complexes, nous avons étendu la spécification du langage *TERMOS* afin qu'il soit capable de gérer les prédicats. Ces prédicats nous permettent d'analyser finement le contenu des messages échangés entre des nœuds participants à un scénario.

Ces scénarios sont ensuite vérifiés sur des traces d'exécutions. Cette vérification se déroule en trois étapes. La première utilise l'outil *GraphSeq* dont l'objectif est d'identifier des motifs de connexions entre les nœuds de la traces correspondant au scénario à tester. Ensuite, à partir de chaque résultat fourni par *GraphSeq*, l'automate du scénario est exécuté et conduit à un verdict. Finalement, l'ensemble

des verdicts de la vérification d'un scénario sur une trace est présenté à l'utilisateur pour permettre une analyse des résultats.

Lors de la phase de test, nous avons été contraints de reprendre une partie de l'application *GraphSeq* pour lui permettre de traiter des fichiers beaucoup plus importants tout en réduisant son empreinte mémoire et l'utilisation de ressources.

Cette étape nous a conduits à construire une plate-forme de simulation de systèmes mobiles nous permettant de collecter des traces d'exécution afin de tester notre méthode. Cette plate-forme a nécessité l'instrumentation d'une étude de cas, l'utilisation d'un simulateur de mobilité, la création d'un gestionnaire de contexte ainsi que l'intégration de ces outils.

1.4 Présentation du plan

Cette thèse se découpe huit chapitres.

Le chapitre 1 présente le contexte et les problématiques associés à ces travaux. Il présente succinctement les contributions permettant d'adresser ces problèmes.

Le chapitre 2 présente l'état de l'art, le contexte scientifique global des différents domaines auxquels cette thèse se rattache.

Le chapitre 3 présente les spécifications du langage de scénario *TERMOS*.

Le chapitre 4 présente l'organisation des différentes étapes de la vérification de propriétés d'un système mobile à l'aide de scénarios.

Le chapitre 5 présente les étapes permettant la spécification d'un scénario ainsi que la transformation de celui-ci en une séquence de graphe et un automate.

Le chapitre 6 présente les étapes nécessaires à l'analyse d'un scénario sur une trace d'exécution.

Le chapitre 7 présente la plate-forme d'expérimentation mise en œuvre dans le cadre de l'étude d'un protocole d'appartenance de groupe ainsi que l'analyse de ce protocole à l'aide du langage *TERMOS*.

Enfin, le chapitre 8 conclut ce mémoire et présente les perspectives issues de ces travaux.

Chapitre 2

État de l'art

Sommaire

Ce premier chapitre introduit le test de systèmes informatiques mobiles. Pour cela nous définirons dans un premier temps les systèmes mobiles et ubiquitaires (section 2.1). Dans un second temps nous aborderons les techniques de vérification et de validation, principalement le test (section 2.2). Finalement nous expliquerons pourquoi les systèmes mobiles nécessitent des nouvelles techniques de test (section 2.3).

2.1 Systèmes mobiles et ubiquitaires

De plus en plus de systèmes sont mobiles et intègrent des fonctionnalités de communication sans fil. En plus de la mobilité, ces équipements disposent d'un grand nombre de capteurs pour s'adapter à l'environnement dans lequel ils évoluent. Conformément à la classification introduite par [Lyytinen 2002] et détaillée dans la sous-section 2.1.1, les systèmes peuvent être répartis en quatre catégories en fonction du niveau d'instrumentation de l'environnement dans lequel ils évoluent et de leur niveau de mobilité comme illustré Figure 2.1. Par rapport aux systèmes informatiques traditionnels auxquels nous sommes habitués, l'évolution

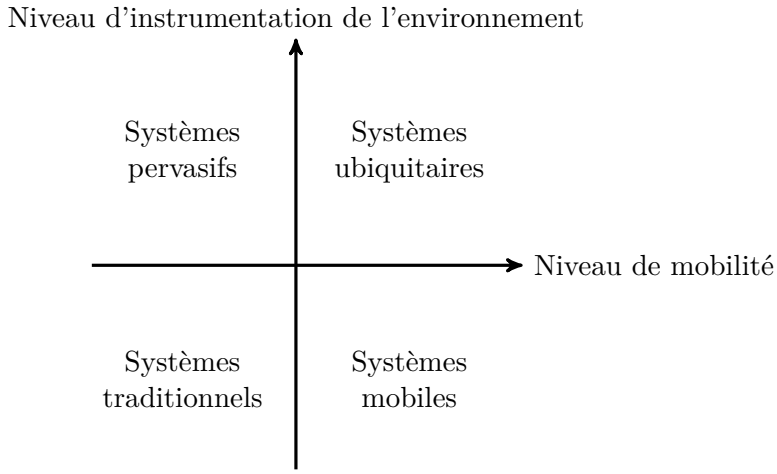


FIGURE 2.1 – Dimensions de l'informatique ubiquitaire

vers l'informatique ubiquitaire pose des défis multiples d'ordre technique, social et organisationnel [Sama 2010, Dadeau 2014].

Cette évolution vers l'informatique ubiquitaire, nous conduit à voir émerger de plus en plus de dispositifs mobiles communicants, équipés de capteurs permettant une instrumentation de l'environnement et capables d'interagir entre eux. Un ensemble de dispositifs mobiles forment un système mobile. Lorsque l'ensemble des dispositifs mobiles est capable de coopérer et de s'adapter à l'environnement dans lequel il évolue, il forme un système ubiquitaire. Bien que nos travaux portent essentiellement sur les systèmes ubiquitaires, ils peuvent s'appliquer en partie aux systèmes pervasifs ainsi qu'aux systèmes mobiles.

2.1.1 Définitions

Systèmes mobiles : L'informatique mobile se concentre sur la capacité d'un système à se déplacer. La mobilité est possible grâce à la réduction du poids et de la taille des équipements mais aussi grâce à la mise en place de réseaux de communication sans fil. Dans l'informatique mobile, une limitation importante est que le comportement de l'application n'est pas évolutif en fonction de ses déplacements. Cela est du à l'impossibilité d'obtenir des informations sur le contexte dans lequel le système prend place. Dans ce cas, la seule solution pour adapter le comportement du système est de laisser l'utilisateur changer la configuration de l'application au fur à mesure de ses déplacements.

Systèmes pervasifs : Une autre dimension de l'informatique ubiquitaire est de rendre l'informatique invisible : c'est l'idée de l'informatique pervasive. Ce concept implique que le système a la capacité d'obtenir des information sur l'environnement dans lequel il est intégré. Ce procédé est réciproque, l'environnement peut et doit aussi devenir "intelligent" avec notamment la capacité de détecter les autres

dispositifs présents. Cette dépendance mutuelle est à la base des interactions "intelligentes" des systèmes avec leur environnement. Ces interactions nécessitent un apprentissage de chaque système sur son environnement, cet apprentissage rend ce type de service limité à cause des efforts nécessaires pour concevoir et maintenir de tels systèmes, les ressources utilisées pour l'apprentissage ne sont pas disponibles pour la fonctionnalité principale du système.

Systèmes ubiquitaires : L'informatique ubiquitaire provient de l'intégration de systèmes mobiles à grande échelle avec des fonctionnalités de l'informatique pervasive. Selon Weiser [Weiser 1993], l'informatique ubiquitaire considère les nuances de la vraie vie comme magnifiques et ne vise seulement qu'à les augmenter. Elle préfigure un monde où l'ensemble des équipements est connecté à l'aide de réseau sans fil partout dans le monde.

2.1.2 Caractéristiques des systèmes ayant un niveau de mobilité élevé

Un système informatique peut être considéré comme mobile à partir du moment où l'application s'exécute sur un dispositif pouvant être déplacé tel qu'un lecteur audio, un appareil photo, un téléphone portable ou même un robot. Les principales différences par rapport aux systèmes traditionnels selon [Satyanarayanan 1996] sont classées en quatre contraintes : les ressources limitées, la sécurité et la vulnérabilité, les performances et la fiabilité et une ressource en énergie finie.

Dans le cadre de systèmes mobiles composés de plusieurs dispositifs mobiles communicants entre eux, les communications sont impactées par ces quatre contraintes. Les liens de communication sont très souvent des liens sans fil et la structure du réseau évolue en fonction des déplacements des dispositifs mobiles. Les communications sans fil sont aussi caractérisées par la possibilité de communiquer avec des dispositifs inconnus par exemple dans le cadre de diffusion dans le voisinage appelé *broadcast*. Nous allons détailler ces deux caractéristiques dans les paragraphes suivants.

2.1.2.1 Topologie réseau dynamique

Les systèmes mobiles sont composés d'un ensemble de dispositifs mobiles communicants entre eux appelés nœuds. La topologie du réseau formée par cet ensemble de nœuds est donc dynamique. Elle évolue au fur et à mesure des déplacements [Macker 1999], des liens radios peuvent être interrompus, d'autres peuvent être établis. La topologie du réseau est donc fortement liée aux déplacements des nœuds. En plus de la mobilité physique, les nœuds peuvent aussi êtres activés ou désactivés à la suite d'actions de la part de l'utilisateur par exemple éteindre ou allumer un équipement. Dans le cadre des systèmes ubiquitaires, le système peut aussi réagir de façon autonome par exemple grâce aux informations contextuelles à sa disposition.

2.1.2.2 Communication avec des partenaires inconnus dans le voisinage

Dans un réseau de type ad-hoc, l'envoi de message par diffusion auprès du voisinage est un mode de communication courant [Baumann 1997]. Il est principalement utilisé pour la découverte du voisinage. N'importe quel nœud se trouvant à proximité d'un autre nœud, peut recevoir et répondre à un message diffusé par celui-ci. Lors de l'envoi du message, l'émetteur ne sait pas quels sont les nœuds qui vont pouvoir recevoir et répondre au message.

La communication avec des partenaires dans le voisinage repose sur la découverte de ce dernier. Ces mécanismes de découverte peuvent être une source de surcharge et de collisions dans le réseau. Pour résoudre ces problèmes, il existe des protocoles de diffusion efficaces [Winstanley 2012].

2.1.3 Caractéristiques des systèmes ayant un niveau d'instrumentation de l'environnement élevé

Les applications sensibles au contexte ou *context-aware*, contrairement aux applications traditionnelles, utilisent des informations sur l'environnement dans lequel elles s'exécutent et s'adaptent à lui. Les informations contextuelles peuvent être réparties en deux catégories [Schmidt 2000] : les facteurs humains d'une part tels que les informations personnelles de l'utilisateur et son environnement social, et d'autre part, l'environnement physique tel que la position et les données provenant des capteurs.

2.1.3.1 La sensibilité au contexte

Les systèmes ubiquitaires adaptent leur comportement à leur environnement. Pour cela ils ont besoin d'informations structurées sur celui-ci. Une instrumentation importante permet justement de collecter des informations afin d'avoir une représentation fidèle de l'environnement dans lequel évolue le système [Lieberman 2000]. Afin de développer des systèmes réactifs à leur environnement, il est d'abord nécessaire d'être capable de modéliser celui-ci puis de développer des algorithmes réagissant aux changements d'environnement [Schmidt 1999]. Une grande partie des systèmes dépendant du contexte utilise simultanément plusieurs sources d'informations contextuelles qui sont agrégées entre elles [Cubo 2009]. Cette agrégation permet une représentation plus fidèle de l'environnement dans lequel évolue le système mais nécessite qu'il soit correctement modélisé. Dans une étude [Bettini 2010], l'auteur introduit les différentes techniques de modélisation du contexte existantes. Parmi celles-ci, intéressons nous dans un premier temps à la modélisation du contexte spatial.

La mobilité, ou le contexte spatial L'espace est une information contextuelle importante dans beaucoup d'applications mobiles qui l'utilisent pour adapter leurs comportements à leur localisation. La représentation structurée des informations

concernant l'espace se fait traditionnellement selon un des deux systèmes de coordonnées suivant :

— *coordonnées géométriques :* représentent la position d'un point dans un repère comme par exemple les coordonnées GPS. Ce type de coordonnées facilite les calculs de distances entre nœuds par exemple.

— *coordonnées symboliques :* représentent la position par un identifiant comme un numéro de pièce ou de bâtiment. Ce type de coordonnées peut être hiérarchisée et peut véhiculer des informations complémentaires à la position.

Un exemple simple permettant de comparer les deux méthodes peut être un trajet entre deux points en coordonnées géométriques et entre deux pièces d'un bâtiment en coordonnées symboliques. Dans le premier cas, le trajet est un tracé sans valeur ajoutée. Par contre dans le second, on a un enchaînement de pièces et de couloirs à traverser.

Les informations sur l'espace permettent aux logiciels de raisonner par rapport à leur position courante. Les principales informations qui sont extraites à partir du contexte spatial sont :

— Position : Retrouver la position d'un objet

— Portée : Retrouver les objets qui sont à l'intérieur d'une zone

— Voisin le plus proche : Retrouver l'objet le plus proche

Mais le contexte ce n'est pas seulement la localisation d'un nœud, il s'agit de toute information concernant l'environnement du système pouvant être mesurée par un capteur [Bellavista 2012] comme par exemple une température, un niveau de batterie, la présence de réseaux sans fil à proximité mais aussi toute information sur le contexte social des utilisateurs.

2.2 Vérification et Validation

Comme nous venons de le voir, un système ubiquitaire implique une forte dynamicité du système, des connexions et déconnexions fréquentes et l'utilisation d'informations sur le contexte. Pour vérifier le comportement de ce type de systèmes, nous devons adopter des techniques de vérification adaptées. Pour cela, nous allons dans un premier temps introduire les différentes techniques de vérification ainsi que leurs limites dans le cadre de la vérification de systèmes ubiquitaires.

2.2.1 Techniques de vérification

Dans l'objectif de découvrir les fautes de conception, la vérification consiste à déterminer si le système satisfait des propriétés, appelées *conditions de vérification* [Harris Cheheyl 1981]. Pour réduire le nombre de fautes dans un programme informatique, il existe plusieurs techniques de recherches de celles-ci. Une classification des méthodes de recherches de fautes a été réalisée par [Avizienis 2004]. Comme le montre la Figure 2.2, elles peuvent être classées selon qu'elles impliquent ou non

l'activation du système [Laprie 1996]. La vérification d'un système sans activation réelle est la **vérification statique**. Vérifier un système en l'activant constitue la **vérification dynamique** : les entrées fournies au système peuvent être symboliques dans le cas de l'exécution symbolique, ou valuées dans le cas du test de vérification, habituellement appelé simplement test.

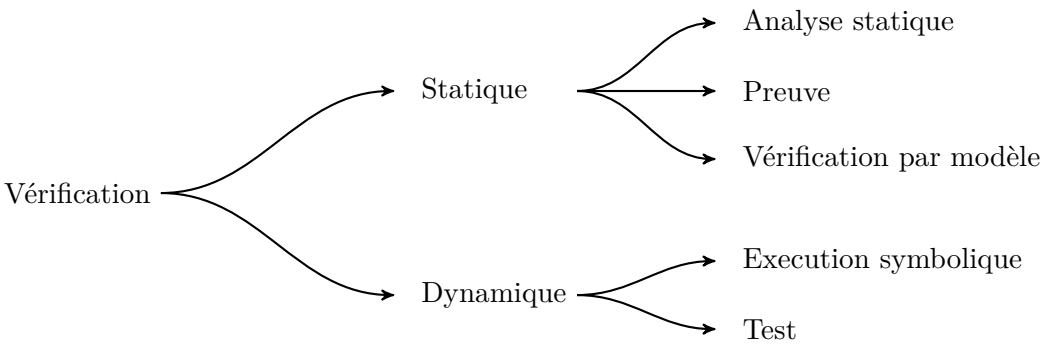


FIGURE 2.2 – Classification des techniques de vérification

Des méthodes formelles de vérification de système pervasif existent telles que celle développée par [Boytsov 2013] qui peut être utilisées pour vérifier des systèmes ubiquitaires. En effet, les auteurs représentent le contexte comme un espace multi-dimensionnel, la localisation et la mobilité pouvant parfaitement être intégrées dans cet espace.

Une application mobile, qui s'exécute simultanément sur chaque nœud, s'appuie sur des échanges de messages entre des nœuds mobiles. Sa validation est complexe à réaliser par le biais de méthodes de vérification statique car cela nécessite d'avoir une vue globale du système. Par contre, rien ne nous empêche d'utiliser les méthodes de vérification statique localement sur chaque nœud sans vue globale de l'ensemble du système et donc sans prendre en compte les aspects mobilité, topologie du réseau et communication avec les autres nœuds.

2.2.2 Le test

L'activité de test vise à évaluer la qualité d'un logiciel. Pour cela il existe différentes techniques permettant de mettre en évidence d'éventuelles erreurs. Avec le test, on ne peut que vérifier qu'un système se comporte de la manière attendue. Une faute est détectée lorsque le résultat d'une exécution du test ne correspond pas au résultat attendu [Ammann 2008].

Comme nous l'avons vu dans la Figure 2.2, le test est une technique de vérification dynamique, c'est-à-dire qu'elle nécessite l'exécution du système. Le test consiste à stimuler les interfaces du système avec des valeurs particulières contrairement à la vérification statique qui est, elle, focalisée sur la structure du code et ne nécessite pas l'exécution du système.

Contrairement à d'autres méthodes de vérification, le test n'apporte aucune garantie sur l'absence de faute. Il n'est pas possible de tester toutes les valuations possibles de l'ensemble des paramètres d'entrée du système. Le test permet donc de mettre en évidence des fautes mais ne permet en aucun cas de garantir l'absence de fautes dans le système.

Observer les sorties de test et décider si elles satisfont les conditions de vérification est généralement connu sous le nom de problème de l'oracle [Adrion 1982], ou *comment décider de l'exactitude des résultats observés, fournis par le programme en réponse aux entrées de test ?* [Weyuker 1982].

Dans le cadre d'applications mobiles, il peut y avoir des cas où les entrées sont non contrôlables et les sorties non observables. Par exemple, si un dispositif est testé dans un environnement réel, les dispositifs avoisinants peuvent rejoindre et quitter le réseau de manière imprévisible. Les entrées du dispositif cible ne peuvent donc pas être déterminées et les sorties attendues ne peuvent pas être prévues.

La méthode de création de test la plus couramment utilisée reste l'écriture manuelle des cas de tests. Cette tâche s'avère de plus en plus délicate à réaliser avec la complexité des systèmes actuels et demande l'expérience d'un ingénieur de test. Pour cela, la création de test est de plus en plus intégrée au processus de développement d'un logiciel. Par exemple, il existe des outils de génération de tests à partir des spécifications en UML [Offutt 1999]. Ce type de test permet de travailler en faisant abstraction du code source, puisque les propriétés à vérifier sont extraites de la spécification et non pas du code du système sous test, on ne sait pas comment ont été implémentées les spécifications.

Il existe aussi une extension au langage *UML* dédiée au test, il s'agit d'*UML Testing Profile* [Schieferdecker 2003]. Cette extension permet de créer des tests directement grâce à des diagrammes de séquences, cela permet un lien fort entre la spécification et le test. L'*UML Testing Profile* peut très bien être mis en œuvre pour tester des systèmes mobiles. Par exemple, il a été utilisé dans le cadre du test d'un algorithme d'itinérance de dispositifs bluetooth mobiles [Dai 2004].

La Figure 2.3 présente une classification en trois axes des activités de tests selon [Tretmans 2004]. Elle permet de détailler les différents types de tests qui existent. Un premier axe *Support de conception* représente le support à partir duquel sont construits les tests. En effet les tests peuvent être construits en se basant sur le code source de l'application à tester, on parlera de test en *boite blanche* ou à partir des spécifications de l'application, on parlera alors de test en *boite noire*. Le deuxième axe correspond à la taille du système et permet de classer le niveau de détail du système testé. Le troisième axe quant à lui correspond au type de test correspondant à la caractéristique à vérifier du système.

Dans les chapitres suivants de ce manuscrit nous nous intéresserons particulièrement au test de systèmes mobiles composés d'un ensemble nœuds, nous souhaitons tester des *fonctionnalités* du système à partir des spécifications de celui-ci. Nous nous placerons donc aux niveaux *Système*, *Spécification* et *Fonctionnel* de la classification présentée Figure 2.3.

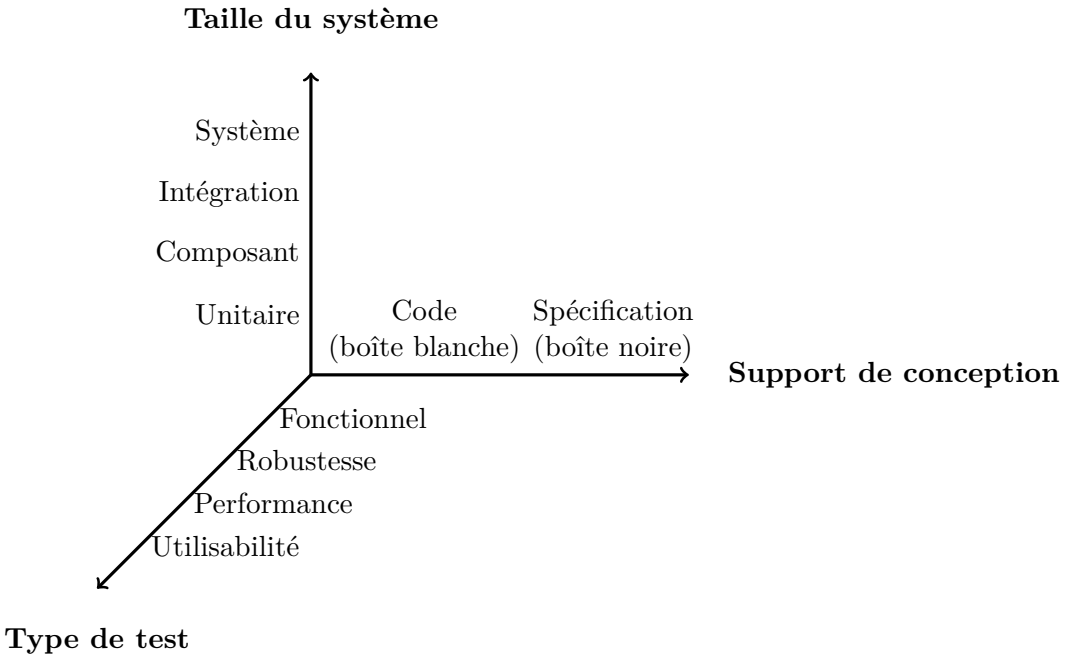


FIGURE 2.3 – Classifications des techniques de test

2.3 Test de systèmes informatiques mobiles et ubiquitaires

2.3.1 Les particularités des systèmes mobiles et leurs implications sur le test

En prenant en compte les deux dimensions de l'informatique ubiquitaire : la mobilité et un environnement instrumenté, le Tableau 2.1 établit une liste des caractéristiques spécifiques des applications mobiles et ubiquitaires par rapport aux caractéristiques des applications traditionnelles ainsi que les implications au niveau du test de ces systèmes [Muccini 2012].

2.3.1.1 Connectivité

Une des caractéristiques particulières des systèmes mobiles est la connectivité, c'est aussi une des plus critiques. Les applications se connectent typiquement à des réseaux mobiles et des réseaux pair à pair. Ces réseaux n'ont pas forcément les mêmes caractéristiques en terme de vitesse, sécurité et fiabilité, il est donc important de tester ces systèmes pour chacun des moyens de communication (WiFi, bluetooth, NFC, RFID ...) disponibles et dans différents environnements.

Type d'application		Particularité	Implications sur le Test
Ubiquitaire	Mobile	Connectivité	Fiabilité, performance, sécurité et tests fonctionnels
		Ressources limitées	Performances et contrôle du fonctionnement
		Autonomie	Contrôle de la consommation d'énergie
		Interactions utilisateur	Test de l'interface homme-machine
		Nouveaux systèmes d'exploitation	Compatibilité et test de l'OS
		Nouveaux langages de programmation	Compatibilité des outils de test
		Diversité de plate-forme d'exécution	Couverture différente
	Pervasive	Sensibilité au contexte	Test des fonctionnalités fonctionnelles et non fonctionnelles dépendantes du contexte
		Adaptation	Exactitudes des adaptations

TABLE 2.1 – Particularités des applications mobiles et implications sur le test

2.3.1.2 Ressources limitées

Les systèmes mobiles deviennent de plus en plus puissants mais leurs ressources restent encore loin de celles disponibles sur un ordinateur de bureau ou même un ordinateur portable. Ces ressources doivent être surveillées en permanence pour éviter toute dégradation des performances. Cela limite d'autant plus l'utilisation de test en ligne qui risque de consommer trop de ressources et en plus d'influencer les résultats.

2.3.1.3 Autonomie

En plus des ressources de calcul disponibles, la plupart des systèmes mobiles fonctionnent grâce à une source d'énergie finie, le plus souvent une batterie. Chaque application et chaque communication réseau consomme de l'énergie. Il faut donc surveiller la consommation d'énergie et adapter le fonctionnement du système en fonction du niveau de batterie restant. Par exemple il peut être nécessaire de désactiver les applications non prioritaires ou de limiter les communications réseau ou même de réduire la fréquence d'actualisation des valeurs provenant des différents capteurs pour économiser de l'énergie.

2.3.1.4 Interactions utilisateur

La majorité des dispositifs mobiles n'est pas équipée de souris voire même de clavier contrairement à ce que l'on retrouve sur la majorité des systèmes classiques. Ils sont plutôt équipés d'écrans tactiles voire même de reconnaissance vocale, ce qui oblige à revoir la gestion des interactions entre l'utilisateur et l'application. Par exemple une saisie grâce à un écran tactile peut être impactée par le niveau d'utilisation des ressources du dispositif qui fait varier le temps de réponse. De plus en plus d'interfaces tactiles détectent en même temps plusieurs points de pression, ce qui implique une multitude de scénarios possibles dans la saisie d'une information. Toutes ces spécificités des interfaces d'entrées et sorties des systèmes mobiles demandent des techniques de test adaptées.

En plus des contraintes provenant des dispositifs d'entrées et sorties des systèmes mobiles, la plupart des applications ayant une interface utilisateur doivent appliquer des règles de mise en page spécifique au système d'exploitation mobile utilisé. Malgré cela, sur des dispositifs différents, l'application peut ne pas avoir le même rendu et se comporter de manière différente [Hu 2011], ce qui oblige à tester une même application sur un ensemble représentatif de plate-formes d'exécution. Pour chacune des plate-formes, il faut aussi tester dans différentes circonstances de charges système et mémoire par exemple.

2.3.1.5 Nouveaux systèmes d'exploitation

Beaucoup de dispositifs mobiles utilisent des systèmes d'exploitation spécialement conçus pour les systèmes mobiles mais ces systèmes évoluent très rapidement. De nouvelles versions sont diffusées régulièrement, elles n'assurent pas tout le temps la rétrocompatibilité. Ces mises à jours régulières obligent à vérifier les applications pour différentes versions du système d'exploitation. De plus certaines fautes peuvent être dues au système d'exploitation.

2.3.1.6 Nouveaux langages de programmation

Avec les nouveaux systèmes d'exploitation viennent de nouveaux langages de programmation tels qu'*Objective-C* pour *iOS* et *JAVA* avec ses librairies spécifiques pour *Android*. Ces langages gèrent la mobilité et les spécificités des systèmes mobiles mais il est par contre nécessaire de développer des outils capables de vérifier ces nouveaux langages. Il faut aussi adapter les outils d'analyse des fichiers binaires créés avec ces nouveaux langages.

2.3.1.7 Diversité de plate-forme d'exécution

Les plate-formes d'exécutions utilisées dans les systèmes mobiles sont beaucoup moins standardisées que sur les systèmes traditionnels. Il existe une multitude d'architectures matérielles différentes comme illustré dans [Gavalas 2011]. À cause de cette diversité, il est difficile de prévoir le comportement d'une application sur tous les dispositifs où elle va pouvoir être exécutée.

2.3.1.8 Sensibilité au contexte

Les applications sensibles au contexte apportent de nouveaux types de fautes qui n'existaient pas dans les systèmes traditionnels et qui sont liés aux informations contextuelles et aux réactions du système à ces informations. Ces types de fautes complexifient le test car il faut prendre en compte l'environnement dans lequel évolue le système ainsi que les interactions avec celui-ci. Cela nécessite de nouvelles méthodes de test adaptées à ces problématiques.

Les informations contextuelles proviennent le plus souvent d'un ensemble de sources différentes telles que des capteurs ou d'autres équipements à proximité. Ces informations sont plus ou moins fiables en fonction de leurs sources, elles peuvent même être contradictoires. Il devient donc nécessaire de détecter les incohérences dans ces informations avec des techniques comme celles proposées par [Lu 2008]. Il est aussi nécessaire d'avoir des méthodes permettant d'identifier les informations contextuelles pertinentes pour le système à tester et de générer des tests utilisant ces informations [Wang 2007, Sama 2010].

2.3.1.9 Adaptation

Les systèmes ubiquitaires sont couramment équipés de fonctionnalités d'adaptation. Elles utilisent pour cela les informations contextuelles à disposition du système pour choisir automatiquement le mode de fonctionnement adapté à l'environnement dans lequel le système évolue. Les données contextuelles étant en constante évolution, il est nécessaire de mettre en place des techniques permettant au système de changer de mode de fonctionnement automatiquement en fonction de celle-ci.

En plus de ces particularités, plusieurs problèmes se posent pour pouvoir tester des systèmes mobiles. Premièrement il est difficile de faire du test en ligne, car le système est distribué sur plusieurs nœuds. Le fait que le système soit composé d'un ensemble de nœuds pose aussi des problèmes d'observabilité. Comment avoir une vue globale du système ? Pour observer de façon globale un système composé d'un ensemble de nœuds, la solution couramment utilisée repose sur la mise en œuvre d'une plate-forme de simulation où l'ensemble des communications entre les nœuds est maîtrisé et où il est possible d'avoir une horloge globale ce qui permet de résoudre le problème de l'oracle.

Les systèmes mobiles apportent aussi des problèmes techniques pour mettre en œuvre les tests. Il est en effet relativement difficile de mettre en place des tests à grande échelle en conditions réelles. Les tests en conditions réelles se limiteront souvent à un petit ensemble de nœuds. Par exemple dans [Cavalli 2009], seulement 4 nœuds sont utilisés pour tester un protocole de communication.

Ces types de tests sont limités et une grande partie des tests devra être réalisé à l'aide de simulations. Ce qui permettra en plus d'avoir des expérimentations reproductibles. Pour simuler un système mobile, plusieurs briques sont nécessaires. Il faut un simulateur de mobilité, un simulateur de réseau et un simulateur de contexte.

2.3.2 Simulation de systèmes mobiles

Pour tester des systèmes informatiques mobiles la simulation est très souvent utilisée du fait de la complexité, des conditions d'observabilité et des coûts d'un test en conditions réelles. Une architecture de plate-forme de simulation utilisant un simulateur de mobilité, un simulateur réseau et un simulateur de contexte est présentée Figure 2.4. Dans ce type d'architecture, le simulateur est utilisé pour générer une trace de mobilité, le simulateur de réseau utilise cette trace pour gérer la distribution des messages, le simulateur de contexte quant à lui contrôle l'expérimentation et distribue les informations contextuelles et de localisations au support d'exécution.

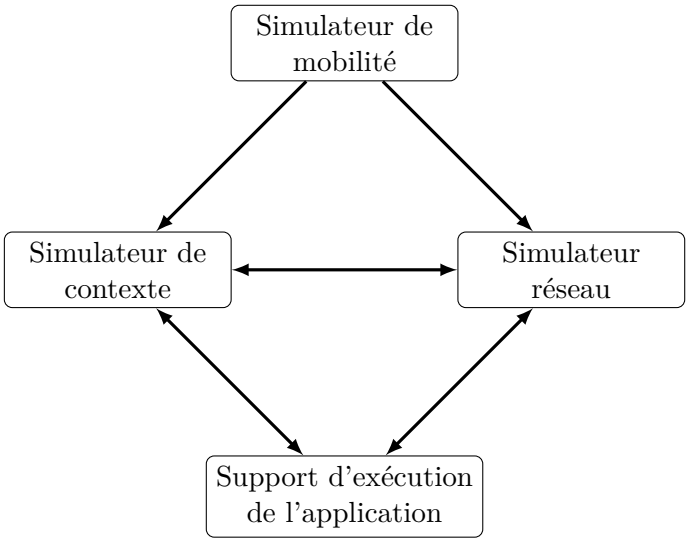


FIGURE 2.4 – Architecture d'une plate-forme d'exécution

2.3.2.1 Simulation de la mobilité

Pour pouvoir réaliser une simulation, il faut donc avoir des informations de mobilité qui seront soit enregistrées depuis une situation réelle comme par exemple d'une expérimentation sur la mobilité [Killijian 2013], soit simulées depuis un outil de gestion de la mobilité tels que GMSF [1] [Baumann 2008] ou IMPORTANT [2] [Bai 2003a].

Ces outils peuvent utiliser différents modèles de mobilité, les plus courants sont le modèle *Manhattan* représentant un quadrillage de rues sur lequel se déplacent les nœuds et le modèle à point de passage aléatoire *Random Waypoint* où la position du nœud est calculée de façon aléatoire par rapport à un point d'origine. De nombreux autres modèles existent, il est même possible d'utiliser des informations provenant de logiciels de SIG [3] permettant une simulation de déplacement sur des cartes réelles

1. Generic Mobility Simulation Framework
2. Impact of Mobility Patterns On RouTing in Ad-hoc NeTworks
3. Systèmes d'Information Géographique

ou même la prise en compte des feux de croisement et du trafic sur la vitesse de déplacement des nœuds.

2.3.2.2 Simulation réseau

Une méthode est de simuler le réseau. Dans le cas de systèmes mobiles les réseaux de type Ad-hoc sont très utilisés. Un outil de simulation de réseau permet de gérer la perte de paquets et la puissance du signal radio de façon transparente pour le nœud testé. Ce type de simulation a été expérimenté dans [Teodosiu 2010].

La mobilité rend extrêmement complexe le test de systèmes car la modification de leurs positions et de la connectivité réseau peut entraîner des changements imprévisibles dans les informations contextuelles.

2.3.2.3 Simulation du contexte

Un simulateur de contexte est un outil pouvant créer et gérer des informations contextuelles demandées par les applications. Ce type de simulateur est utilisé pour développer, visualiser et évaluer des applications mobiles dépendantes du contexte de localisation. Certains simulateurs sont construits à partir de moteurs graphiques de jeux vidéo et sont utilisés pour simuler une vue 2D ou 3D de l'environnement physique, par exemple Ubiwise [Barton 2003]. Les applications véhicule-à-véhicule peuvent utiliser des simulateurs de trafic (comme STRAW [Choffnes 2005]) qui simulent les mouvements dans une zone routière. Il existe également des simulateurs non spécialisés, qui génèrent des événements de localisations génériques, comme l'outil GLS [Sanmugalingam 2002].

2.3.3 Différentes approches du test de systèmes mobiles

La définition du test de systèmes mobiles dans la littérature couvre en réalité différents types de systèmes. Cela peut être une application qui est mobile et donc peut migrer de nœuds en nœuds. Cela peut aussi consister à tester localement une application sans prendre en compte les cas où plusieurs dispositifs communiquent. Dans le cadre de cette thèse, nous nous intéressons au test d'applications mobiles reposant sur un ensemble de nœuds communicants.

2.3.3.1 Test d'une application mobile

Le cas des *Applications Mobiles* est particulier car on souhaite tester une seule application mais sur les différents réseaux sur lesquels elle pourra être connectée. Pour tester cela, une technique développée dans [Satoh 2003a] consiste à simuler la mobilité physique de l'application par la mobilité logique de l'application. Cette mobilité logique consiste à migrer l'application au fur et à mesure de son exécution sur des dispositifs qui sont chacun connectés à un réseau différent. Plusieurs développements du même auteur utilisent cette technique [Satoh 2003b, Satoh 2004] sur des études de cas.

2.3.3.2 Test d'un seul nœud

Dans le cadre des applications distribuées sur plusieurs nœuds, il est quand même possible de tester chaque nœud individuellement. Si on se place dans le cas du test d'un seul nœud ou plutôt de test qu'on pourrait qualifier de test *locaux*, les problèmes provenant du test de systèmes mobiles n'ont plus lieu d'être, on se retrouve dans le cas du test de système classique (i.e. non mobile).

2.3.4 Description de scénarios de Test

Les langages de scénarios graphiques, comme les Message Sequence Charts [ITU-T 2011] ou les Diagrammes de Séquences UML [Omg 2011], permettent de décrire des interactions dans un système distribué. La description peut servir différents buts : capture d'exigences [Kugler 2007], spécification d'objectifs de test (c'est-à-dire, d'interactions à couvrir par des cas de test) [Grabowski 1993], conception de cas de test [Pickin 2004] ou rétro-conception de traces d'exécution [Briand 2006]. Typiquement, on dessine la ligne de vie de chaque participant à l'interaction et on fait apparaître les ordres partiels des communications. Les langages offrent différents opérateurs pour décrire des ordres complexes, tels que des opérateurs de choix, d'itération, de parallélisme et de séquencement. Certains introduisent des modalités pour distinguer les comportements potentiels et nécessaires, tels les Live Sequence Charts (LSC) [Damm 2001] ou la version 2 des Diagrammes de Séquences [Omg 2011]. Les communications sont généralement point à point, avec un émetteur et un récepteur pour chaque message. La topologie de connexion n'est pas un élément central et n'est pas supposée changer durant l'interaction.

Dans un système mobile, le mouvement des nœuds conduit à une topologie de connexion intrinsèquement instable. Les liens avec les autres nœuds mobiles et les nœuds fixes d'infrastructure peuvent être établis ou détruits selon la localisation. De plus, des nœuds peuvent apparaître ou disparaître à tout moment, par exemple lorsqu'un utilisateur allume ou éteint un dispositif portable, lorsque le dispositif se met en veille ou que sa batterie est épuisée. Il est donc souhaitable que les scénarios prennent explicitement en compte la configuration spatiale des nœuds et son évolution au cours de l'interaction.

2.4 Conclusion

Les avancées technologiques du monde sans fil ont conduit au développement des systèmes informatiques mobiles. Leurs caractéristiques propres telles que la dynamicité de la structure du système, les communications avec des partenaires inconnus dans le voisinage et la dépendance vis-à-vis du contexte posent de nouveaux défis au processus de développement des applications mobiles.

Chapitre 3

TERMOS : un langage de scénario pour les systèmes mobiles

Sommaire

Ce chapitre présente une approche de test fondée sur l'utilisation de scénarios permettant de décrire les comportements attendus ou interdits (ou à vérifier) d'une application mobile. La validation de systèmes basée sur des scénarios n'est pas une nouveauté mais l'originalité de nos travaux est qu'ils s'appliquent à une classe spécifique des systèmes distribués : les systèmes à mobilité physique. Cette approche nous a conduits à la création d'un langage de spécification dédié et dénommé *TERMOS* pour *TEst Requirement language for MObile Settings*. Les premiers principes de ce langage ont été posés dans le cadre du projet européen *HIDENETS*. Ce langage a été défini afin de pouvoir décrire des scénarios d'applications basées sur des communications sans fil fiables [Waeselynck 2007, Huszrel 2008] en se basant sur la notation UML.

Notre contribution a été :

— de rassembler les différents éléments de ces travaux qui intègrent le traitement des différentes vues descriptives,

— de finaliser la définition de notre langage,

— de mettre en œuvre le langage.

L'objectif principal de *TERMOS* est de permettre l'évaluation d'une trace d'exécution à partir d'une description à base de scénarios. Le concept général suivi pour le test d'applications dans un contexte de mobilité nécessite deux éléments : une plate-forme d'exécution et un environnement logiciel pour traiter les données provenant de celle-ci. La Figure 3.1 schématise cette structure. Étant donné que nous sommes dans le cadre de systèmes communicants et mobiles, la plate-forme d'exécution est composée d'un simulateur de contexte (pour gérer la mobilité), d'un simulateur réseau (pour gérer l'aspect communication) et d'un support d'exécution (pour exécuter l'application à tester). Elle doit permettre de contrôler, d'observer et de collecter des données lors de l'exécution de l'application cible (*SUT* ou System Under Test). La trace recueillie est ensuite traitée afin de vérifier si les comportements décrits dans les scénarios se produisent.

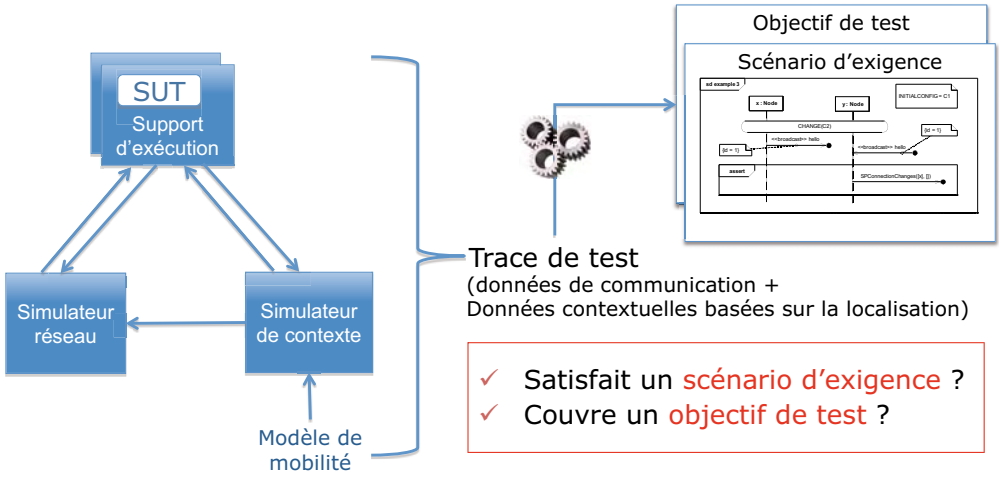


FIGURE 3.1 – Motivations du langage *TERMOS*

Pour atteindre cet objectif, *TERMOS* se base sur trois grandes idées. Premièrement, le langage doit aider à capturer uniquement les détails pertinents pour une exigence donnée. Deuxièmement, les exigences devront être le plus simple possible. Pour cela, il est préférable qu'une exigence complexe soit décomposée en plusieurs exigences plus petites. Enfin, chaque scénario correspondant à une exigence représentera une vérification indépendante, c'est-à-dire qu'il ne pourra pas y avoir de description hiérarchique de scénarios.

3.1 Une approche de test fondée sur la description de scénarios

3.1.1 Définitions préliminaires

Test : L'objectif de ce langage est d'évaluer une trace par rapport à un comportement partiel spécifié dans un scénario. Un test correspond à l'exécution de cette évaluation. Chaque test produit un résultat permettant de classer les traces en trois catégories : *valides*, *invalides* et *inconclusives*.

Scénario : Un scénario correspond à la spécification d'un comportement partiel d'un système. Les scénarios peuvent être de différents types en fonction du type de comportement qu'ils décrivent. Nous pouvons représenter un *scénario d'exigence* ou un *objectif de test*.

Scénario d'exigence : Un scénario d'exigence exprime une propriété attendue du système sous test. Cette propriété doit être satisfaite par toutes les exécutions du système. Ce scénario représente une abstraction du comportement attendu à partir de propriétés de haut niveau.

Dans ce type de scénario, on veut pouvoir exprimer :

— des modalités de type « possibilité/obligation ». Par exemple, nous allons avoir une séquence de deux fragments de scénarios, le premier étant possible et le second obligatoire, pour représenter une exigence de type : « chaque fois que A se produit alors B doit suivre ».

— des traces partielles, ne représentant qu'un sous-ensemble des entités du système et un sous-ensemble des messages échangés.

Objectif de test : Un objectif de test indique un fragment de comportement que nous souhaitons observer au moins une fois durant le test. Il se focalise sur :

— des traces partielles.

— des classes de comportement, plusieurs cas de test peuvent couvrir un objectif de test.

Trace d'exécution : Une trace est une séquence d'événements qui mène à des changements d'état du système.

Dans notre cas, les événements qui nous intéressent sont les événements observables du système, principalement les envois et réceptions de messages. Pour différencier un message d'un autre dans la trace, chaque message contient un identifiant unique. Grâce à cela, une trace concrète peut-être par exemple un tuple contenant *(!m,destinataire,id)* ou *(?m,émetteur,id)*, où m est le nom du message envoyé ou reçu et *id* est un identifiant unique. Il est à noter qu'un message émis peut-être reçu de multiples fois. Par exemple, dans le cas des messages diffusés par émission

radio dans le voisinage, l'identifiant est utilisé pour relier l'événement d'émission aux événements de réception de ce message.

Dans le cas de systèmes ubiquitaires, une trace ne contient pas seulement des événements de communication. Elle contient aussi des informations sur le contexte dans lequel évolue le système, par exemple les données GPS de localisation des nœuds qui le composent.

3.1.2 Approche de test

Les langages de scénarios sont utilisés dans le cadre de différentes activités de tests, que ce soit pour la capture d'exigences [Kugler 2007], la spécification d'objectifs de tests [Grabowski 1993], la conception des cas de tests [Pickin 2003] ainsi que dans l'analyse des traces d'exécution [Briand 2006]. Leur popularité est due à leur syntaxe conviviale qui facilite leur compréhension tout en ouvrant la porte aux traitements formels, ce qui nécessite une notation possédant une sémantique précise.

L'approche générale décrite dans la Figure 3.1 montre notre approche basée sur du test passif avec des traitements qui consistent à :

— Vérifier qu'une *trace d'exécution* satisfait un *scénario d'exigences.*

— Vérifier qu'une *trace d'exécution* couvre un *objectif de test*

Le principe du test passif [Cavalli 2009] est de laisser tourner l'application et ensuite vérifier la trace d'exécution obtenue plutôt que de chercher à produire quelques cas de test ciblés.

La mise en place de ces traitements suppose que le langage de scénario utilisé possède une sémantique bien définie. Selon le type d'activité de test considéré, différentes variantes d'un langage peuvent être utilisées, par exemple différents profils de diagrammes de séquence *UML*. En effet, les scénarios présents sur la Figure 3.1 n'utilisent pas nécessairement les mêmes ensembles d'éléments du langage. Par exemple, une trace d'exécution, puisqu'elle ne contient que des événements ayant eu lieu, ne contiendra jamais d'opérateurs permettant d'effectuer un choix entre deux comportements (fragment *alt* du langage *UML* par exemple).

3.2 Spécificités de la modélisation de systèmes mobiles

Dans le but de valider le fonctionnement de systèmes mobiles, il est nécessaire dans un premier temps de modéliser leurs comportements afin de mieux analyser les événements. Pour cela, il est avant tout important de bien comprendre quelles sont les spécificités des systèmes mobiles.

De tels systèmes où les nœuds se déplacent, engendrent nécessairement une topologie de communication instable et donc dynamique. Des liens avec les autres nœuds mobiles et fixes sont établis et détruits en fonction de leur position les uns par rapport aux autres. L'exemple donné sur la Figure 3.2 illustre bien cela. Sur la partie gauche de la figure, nous pouvons constater que les nœuds B et D sont, dans un

premier temps, à portée de communication de C (indiquée par le cercle pointillé) et donc potentiellement liés. Puis, dans un deuxième temps (partie droite de la figure), le nœud B s'éloigne de C et le nœud A s'approche de C. Nous obtenons ainsi de nouveaux liens et une nouvelle topologie du réseau. Il faut aussi tenir compte du fait que des nœuds peuvent apparaître et disparaître si par exemple ils sont allumés, éteints, s'ils sont à cours d'énergie ou encore en phase d'arrêt, ce qui aura bien entendu un impact sur la topologie du réseau. Il est donc impératif/important de tenir compte de la configuration spatiale des nœuds et de son évolution dans le temps dans notre langage de scénario.

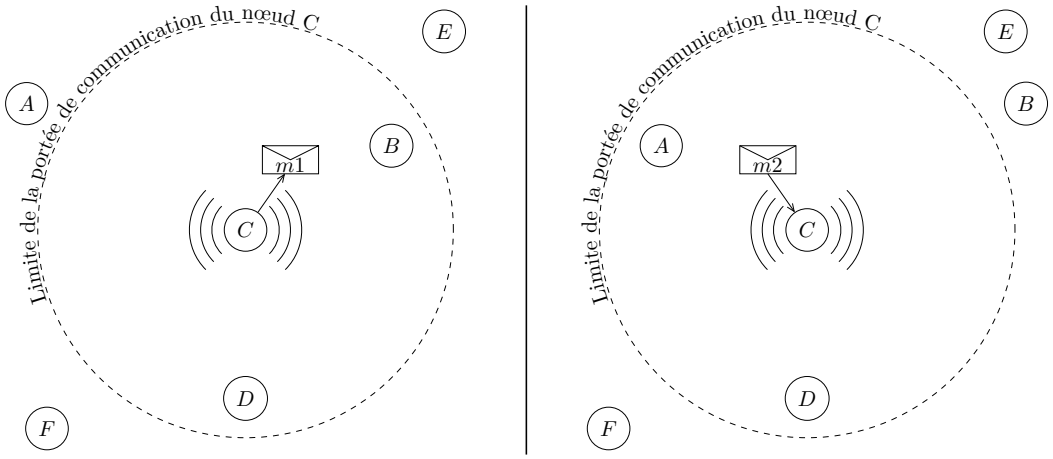


FIGURE 3.2 – Exemple de diffusion d'un message dans le voisinage

Un nœud mobile doit être capable de communiquer au moyen d'un réseau sans fil et souvent dans un environnement où il n'a pas de connaissance pré-établie des nœuds qui l'entourent. Un mode de communication naturel pour un tel système est la diffusion locale en émettant régulièrement un message dans tout le voisinage environnant. Quiconque se trouve alors à portée de communication peut recevoir et réagir à ce message. Par exemple, la diffusion locale est utilisée comme une étape de base pour la couche de découverte dans les applications basées sur la mobilité (découverte de groupe pour des services d'appartenance de groupe, découverte de route pour les protocoles de routage, etc.). Cette diffusion locale est schématisée sur la Figure 3.2 par des ondes autour du nœud C. Elle est bien sûr effective sur tous les nœuds mobiles même si ce n'est pas représenté. La prise en compte de cet aspect dans le langage de scénario est indispensable pour des services dynamiques.

L'utilisation des langages de scénarios graphiques est courante pour décrire les interactions dans les systèmes distribués. Des exemples de ces langages sont *Message Sequence Charts (MSC)* [ITU-T 2011], *UML Sequence Diagrams (UML SD)* [Omg 2011] et *Live Sequence Charts (LSC)* [Klose 2003]. Les nœuds participants sont représentés par des lignes de vie et des ordres partiels liés aux événements de communication peuvent être mis en évidence.

Pour représenter des scénarios plus complexes, ces langages proposent des opérateurs qui permettent de faire des opérations de choix, de répétition, de parallélisme et de séquence. Certains langages permettent aussi de distinguer les comportements obligatoires et possibles. Ceci est un point intéressant étant donné que nous souhaitons exprimer des modalités de type «possibilité/obligation» dans un scénario d'exigence. Concernant les communications, une vue point-à-point est habituellement adoptée dans ces langages, avec un émetteur et un récepteur pour chaque message. Cependant, la topologie de connexion sous-jacente n'est pas prise en compte car elle n'est pas supposée changer au cours de l'évolution des interactions qui est représentée.

Notre choix s'est porté sur le langage UML SD pour lequel de nombreux outils existent. Néanmoins, les diagrammes de séquence UML ne permettent pas de représenter fidèlement des interactions dans un contexte de mobilité. Pour pouvoir prendre en compte les spécificités des systèmes mobiles (voir chapitre 2), nous avons fait le choix d'étendre ce langage.

Étudions maintenant comment représenter ces spécificités afin de voir de quelle manière nous pouvons les formaliser pour les intégrer dans un langage de scénarios graphiques comme le langage UML.

Vue spatiale : La vérification étant basée sur les événements, notre méthode est d'abord basée sur une vue événementielle du comportement du système. Pour décrire le comportement de systèmes mobiles, nous avons besoin de pouvoir représenter plus de choses que des événements entre des nœuds : il est nécessaire de savoir qui peut communiquer avec qui. Les possibilités de communication dépendent de la configuration spatiale des nœuds, par exemple, s'il y a des nœuds voisins susceptibles de recevoir une émission radio. C'est le cas sur la Figure 3.2 (partie gauche) où le nœud C ne peut communiquer qu'avec les nœuds B et D qui sont à portée de communication. Ceci dit, la position des nœuds les uns par rapport aux autres n'est pas le seul critère possible et il serait possible de tenir compte d'un autre critère. Par exemple, nous pourrions vouloir vérifier les communications entre nœuds proches et partageant une certaine caractéristique (par exemple, possédant le même système d'exploitation).

Au cours d'un scénario, ces relations spatiales sont amenées à évoluer, c'est pourquoi il est possible d'avoir une séquence de configurations spatiales. Sur la Figure 3.2, nous avons deux configurations spatiales. Une configuration où un nœud C communique avec les nœuds B et D puis, un peu plus tard, ce même nœud C communique avec un nœud A (qui s'est rapproché de C) et D mais plus avec B (qui s'est éloigné de C). Il nous faut introduire une notation spécifique capable de décrire les relations spatiales entre chaque dispositif mobile mais aussi leurs évolutions.

Sur la Figure 3.3, nous avons un exemple de représentation de la vue spatiale possible avec une séquence de deux configurations spatiales induites par le mouvement des nœuds. Elle est composée de deux graphes non orientés, successifs dans le temps, composés des nœuds intéressants pour notre scénario. Nous avons sur la

configuration $C1$ trois nœuds : C et B sont à portée de communication mais pas A encore trop loin à un instant donné. Un peu plus tard, nous avons la configuration $C2$ avec trois nœuds : C et A qui peuvent communiquer ensemble mais pas avec B.

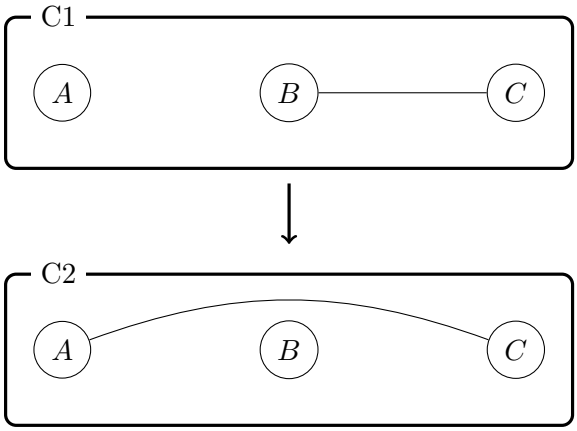


FIGURE 3.3 – Exemple de représentation de la vue spatiale

Événements de changement de configuration : Dans la notation UML, les événements entre les différents objets d'un système sont représentés graphiquement au moyen de diagrammes de séquence. Chaque objet y est représenté par une ligne de vie verticale représentant son évolution au cours du temps.

Si nous représentons l'évolution de systèmes mobiles avec un tel langage graphique, il devient nécessaire de tenir compte de la topologie du système à chaque instant du scénario décrit. Pour cela nous introduisons un type d'événement global qui est le changement de configuration. Il permet de lier une configuration spatiale à la vue événementielle.

Sur la Figure 3.2, C envoie un message $m1$ à B et plus tard il reçoit un message $m2$ de A. Pour recevoir un message, il faut être à portée de communication : la configuration spatiale détermine les comportements possibles.

La Figure 3.3 illustre le changement de configuration nécessaire qui permet de décrire sur la Figure 3.4 que le message $m2$ est envoyé après ce changement, car avant, A n'est pas connecté à C. Le rectangle gris qui schématise ce changement couvre toutes les lignes de vie car c'est un changement de configuration globale qui reflète une propriété globale du monde physique que les nœuds subissent. Cela introduit le fait qu'avant ce changement, le scénario se déroule dans le cadre de la configuration spatiale n°1 et après dans celui de la configuration n°2.

De plus, plusieurs relations d'ordre apparaissent du fait de ce changement de configuration :

— l'émission de $m1$ précède celle de $m2$,

— la réception de $m1$ précède l'émission de $m2$,

— la réception de $m1$ précède celle de $m2$

Tout événement d'émission et de réception de message s'effectue dans une configuration spécifiée.

Remarque : Dans tous les cas, il y a peut-être une communication radio.

Nous distinguons :

— la communication point-à-point, avec un destinataire explicite,

— la diffusion locale (*broadcast*, qui consiste à envoyer un message sans destinataire auquel tous les nœuds qui le reçoivent peuvent réagir.

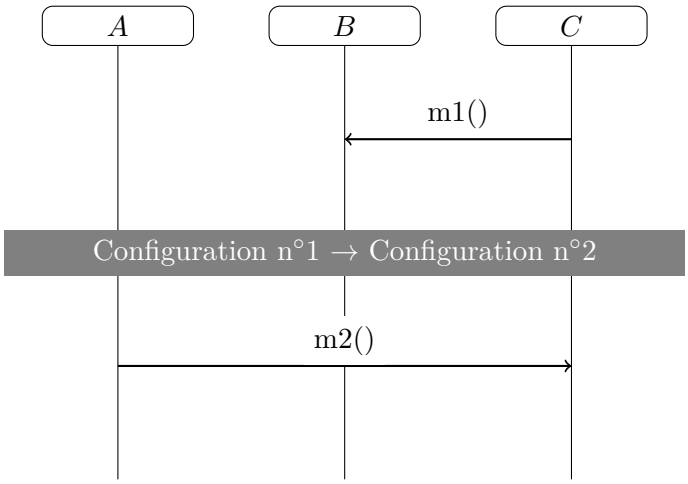


FIGURE 3.4 – Exemple de synchronisation entre les vues

Diffusion dans le voisinage : En plus d'une communication classique point-à-point, les systèmes mobiles utilisent naturellement la communication par diffusion locale. Dans ce type de communication, un nœud diffuse un message dans son voisinage sans destinataire explicite. Quiconque se trouve à portée de communication peut écouter et ainsi réagir au message. Par exemple, la diffusion locale est utilisée comme une étape de base pour la couche de découverte dans les applications basées sur la mobilité (découverte de groupe pour des services d'appartenance de groupes, découverte de chemin dans les protocoles de routage, etc.). La réception de ce type de message permet d'initier les échanges entre les nœuds. Sur la Figure 3.2, C ne peut envoyer un message $m1$ à B qu'après avoir découvert son entourage (composé de B et D dans un premier temps).

Par défaut, les langages de scénario graphiques n'intègrent pas la gestion des événements de communication par diffusion. Un message ne peut avoir qu'un émetteur et un récepteur alors qu'un message de broadcast implique un seul émetteur pour un ou plusieurs récepteurs. Pour prendre en compte les messages diffusés dans nos scénarios, il aurait fallu représenter plusieurs messages (appelés *hello* sur la Figure 3.5) partant d'un même émetteur vers tous les récepteurs potentiels ce qui n'aurait pas le sens voulu. Sur la partie gauche de la Figure 3.5, le nœud C émet un même message *hello* reçu par B et D et donc représenté deux fois à deux instants

différents. Il faut donc introduire un moyen de représenter ce concept de diffusion dans les diagrammes de séquence en ayant, par exemple, un seul message partant de l'émetteur et un message arrivant à chaque récepteur concerné comme sur la partie droite de la Figure 3.5.

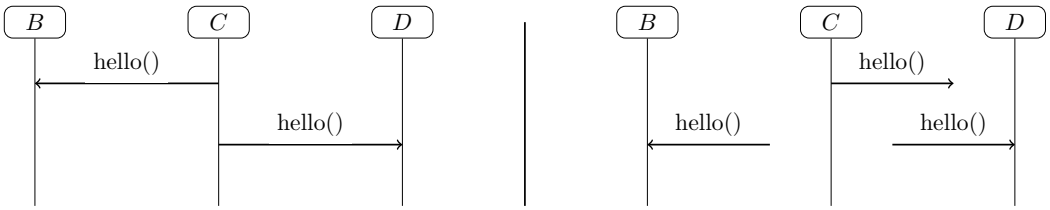


FIGURE 3.5 – Exemple de message broadcast dans un scénario

3.3 Objectifs de TERMOS

L'objectif du langage est de nous permettre de décrire le comportement d'un système mobile en prenant en compte les spécificités des systèmes mobiles et en intégrant les concepts décrits précédemment :

— gérer la topologie du système, à l'aide d'une représentation explicite des liens entre les différents nœuds qui le composent ainsi que l'évolution de celle-ci dans le temps,

— prendre en compte les spécificités liées au mode de communication des nœuds entre eux comme par exemple la diffusion radio de messages aux nœuds voisins.

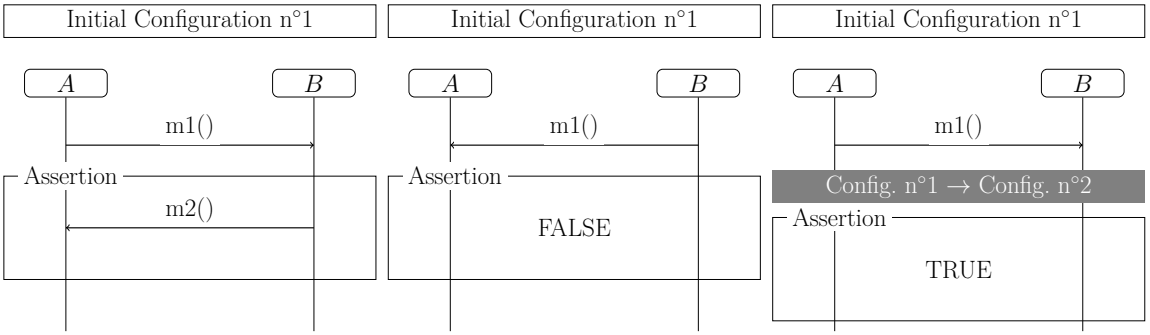
Le nom *TERMOS* signifie *Test Requirement language for Mobile Settings*.

Afin de représenter un scénario dans un cadre de mobilité, nous introduisons notre langage avec deux vues connectées : une vue spatiale décrivant les configurations topologiques des nœuds et une vue événementielle (classique comme pour tout langage de scénario graphique) qui décrit l'ordre des événements à observer.

3.3.1 Représentation des scénarios mobiles avec TERMOS

Les scénarios graphiques peuvent être utilisés dans différentes phases d'un processus de développement, de la capture des exigences aux phases terminales de test. Notre travail se focalise sur l'utilisation des scénarios pour analyser des traces d'exécution de systèmes mobiles. Nous considérons trois classes de scénarios comme illustrées Figure 3.6.

Les exigences positives capturent des propriétés « clef » invariantes de la forme : à chaque fois qu'une certaine interaction arrive dans la trace, alors une interaction spécifique doit toujours arriver. Exemple : « si un message $m1$ est émis de A vers B alors il faut qu'un message $m2$ de B à A soit émis » (cf. Figure 3.6a).



(a) Exigence positive (b) Exigence négative (c) Objectif de test

FIGURE 3.6 – Classes de scénarios de test

Les exigences négatives décrivent les comportements interdits qui ne devraient jamais arriver dans la trace. Exemple : « B n'émet jamais de message $m1$ vers A dans la configuration $C1$ » (cf. Figure 3.6b).

Les objectifs de test décrivent les comportements à couvrir par le test, c'est-à-dire que nous voulons que ces comportements apparaissent au moins une fois dans la trace. Exemple : « nous voulons tester le cas où un message $m1$ est émis de A vers B avant un changement de configuration spatiale » (cf. Figure 3.6c).

3.3.2 Analyse d'un scénario TERMOS

Pour automatiser l'analyse des traces de test à partir de ces scénarios, il est nécessaire que le langage TERMOS intègre les spécificités liées aux systèmes mobiles mais aussi celles liées à ces classes de scénarios avec une notation possédant une sémantique formelle.

Dans les deux premières classes, nous retrouvons les modalités de type « possibilité/obligation » typiques des scénarios d'exigence à représenter. L'analyse doit s'arrêter dès que la modalité d'obligation est vérifiée. Dans la troisième classe, l'analyse doit s'arrêter dès que le comportement à couvrir est observé.

Pour cela, un scénario TERMOS doit se terminer avec un fragment qui représente une assertion. Tout ce qui arrive avant le fragment d'assertion représente un comportement potentiel, alors que son contenu est obligatoire. C'est le contenu de ce fragment qui caractérise la classe du scénario. Une exigence négative contient seulement un invariant faux. Puisque l'assertion *FALSE* n'est pas possible, cela signifie que le comportement avant le fragment d'assertion ne devrait jamais se produire. Un objectif de test contient seulement un invariant *TRUE* qui doit être vérifié chaque fois que le fragment d'assertion est atteint.

Une exigence positive se termine par un fragment dont le contenu est différent d'un simple invariant vrai ou faux. En effet, nous ne souhaitons pas exprimer des modalités du type « si il y a eu ce comportement alors c'est faux » ou « si il y a eu ce comportement alors c'est bon ». Nous souhaitons exprimer des modalités du type « si il y a eu ce comportement alors il faut qu'il se produise cet autre

comportement ».

Les scénarios TERMOS sont interprétés comme des motifs de comportement génériques qui peuvent être trouvés dans divers sous-ensembles de traces au cours de l'exécution du test. Sur la Figure 3.6a, A et B sont des identifiants symboliques. Ce qui fait que l'interprétation de ce scénario devient : « dans la trace d'exécution, chaque fois que deux nœuds présentent dans une configuration spatiale $C1$ et que le nœud correspondant à A envoie un message $m1$ à un nœud correspondant à B, alors le nœud correspondant à B doit répondre par un message $m2$ ».

À un instant au cours de l'exécution du test, nous pouvons avoir deux instances possibles de la configuration $C1$:

— une avec les nœuds physiques $n1$ et $n2$ de la trace correspondant respectivement à A et B,

— une avec les nœuds physiques $n1$ et $n3$ de la trace correspondant respectivement à A et B.

À un autre instant ultérieur, $n1$ pourrait jouer le rôle de B dans une autre instance de $C1$.

Pour un scénario donné, l'analyse de la trace de test doit donc se faire en deux étapes :

1. Déterminer quels nœuds physiques de la trace correspondent à la séquence de configurations spatiales du scénario et à quel moment dans la trace.
2. Analyser l'ordre des événements de communications et de changements de configuration pour chacune des correspondances spatiales identifiées à l'étape précédente.

L'étape 1 relève d'un problème d'appariement de graphes. Il faut apparier les configurations abstraites définies dans des scénarios d'exigence (Figure 3.3) et les configurations observées dans une trace d'exécution. Il faut décider quel nœud de la trace peut jouer le rôle décrit dans les scénarios. À partir de leurs types et de leurs connexions, les nœuds abstraits doivent être associés aux nœuds concrets de la trace. Cependant, usuellement il y a plusieurs correspondances possibles. De plus, la correspondance ne doit pas prendre en compte seulement une configuration mais les changements dans une séquence de configurations. Dans [Nguyen 2009] et [Nguyen 2010], Minh Duc Nguyen explique comment la recherche d'isomorphisme de sous-graphe a été utilisée pour chercher toutes les instances des configurations spatiales du scénario dans une trace. Pour exécuter cette tâche, il a développé une méthode et un outil appelé *GrapSeq* qui peut raisonner sur une séquence de graphes de configuration abstraits et concrets, et qui peut donner l'ensemble des correspondances et des valuations possibles entre les nœuds abstraits et concrets.

Les traces obtenues peuvent ensuite être évaluées pour vérifier les exigences. En utilisant l'ensemble des configurations trouvé, les messages de la trace obtenue doivent être analysés, si leurs types, paramètres et ordre sont conformes à ceux spécifiés dans les scénarios. Cela nécessite une sémantique formelle précise pour la vue événementielle de TERMOS, pour que cela rende possible la vérification des

traces. Il existe de nombreux choix sémantiques pour interpréter les diagrammes de séquence UML donc il faut bien faire attention à sélectionner un ensemble cohérent d'options qui permettra de donner un sens non ambigu aux diagrammes.

3.4 Syntaxe du langage

3.4.1 Syntaxe de la vue spatiale

Comme nous l'avons vu, la vue spatiale peut contenir plusieurs configurations spatiales qui vont se succéder au cours du scénario. Chaque configuration possède un nom différent. Une configuration est un graphe étiqueté, où les sommets représentent les nœuds du système et les arêtes représentent les connexions entre les nœuds. Les arêtes peuvent avoir un attribut précisant le type de connexion entre deux nœuds.

Une configuration peut être définie en spécifiant son nom et les nœuds qui y sont impliqués. Ainsi, si une configuration contient des nœuds *n1*, *n2* et *n3*, le scénario qui se réfère à cette configuration doit contenir les lignes de vie représentant ces trois nœuds. Sur l'exemple de la Figure 3.3, il y a trois nœuds A, B et C présents dans les deux configurations spatiales donc il y aura trois lignes de vie dans le diagramme de séquence associé.

Soit une séquence de deux configurations spatiales Ci et Cj, le nombre de lignes de vie Nb_lignes présente dans la vue événementielle est défini par :

$$Nb\_lignes = Card(Ci) + Card(Cj) - Card(Ci \cap Cj)$$

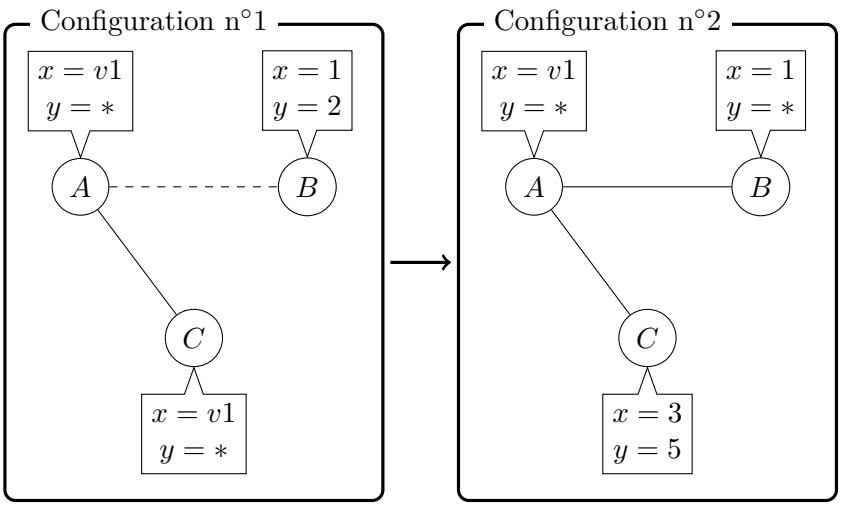


FIGURE 3.7 – Exemple de configurations spatiales

Les nœuds d'une configuration spatiale ont des identifiants symboliques car ils peuvent correspondre à différents nœuds dans une trace concrète. Ils peuvent avoir des attributs qui peuvent également être définis dans la vue spatiale. Prenons l'exemple de la Figure 3.7, nous avons des nœuds avec deux attributs x et y chacun. Ces attributs peuvent avoir trois types de valeurs :

— Une valeur entière constante (les valeurs 1 et 2 pour le nœud B de la configuration n°1).

— Un nom symbolique : la valeur n'est pas spécifiée mais si plusieurs nœuds de la configuration possèdent un attribut ayant comme valeur le même nom symbolique, alors les attributs de ces nœuds sont considérés égaux. C'est le cas de A et B qui ont un attribut ayant pour valeur $v1$, ce qui signifie que x de A est égal à x de C. Ce nom variable représente alors une constante globale symbolique qui doit demeurer stable dans la configuration.

— Un joker qui signifie que la valeur de cet attribut est ignorée. Par exemple, C possède un attribut y que l'on ignore d'abord, puis qui doit prendre la valeur 5 dans la configuration suivante.

Pour spécifier les paramètres ou la qualité de la connexion entre les nœuds, les connexions peuvent être étiquetées par des valeurs constantes ou un joker.

3.4.2 Formalisation de la vue événementielle

La vue événementielle représente l'ensemble des événements que l'on souhaite observer dans un scénario. La syntaxe de la vue événementielle qui a été proposée pour capturer les différents scénarios dans le langage *TERMOS* est dérivée de la syntaxe des diagrammes de séquence *UML*.

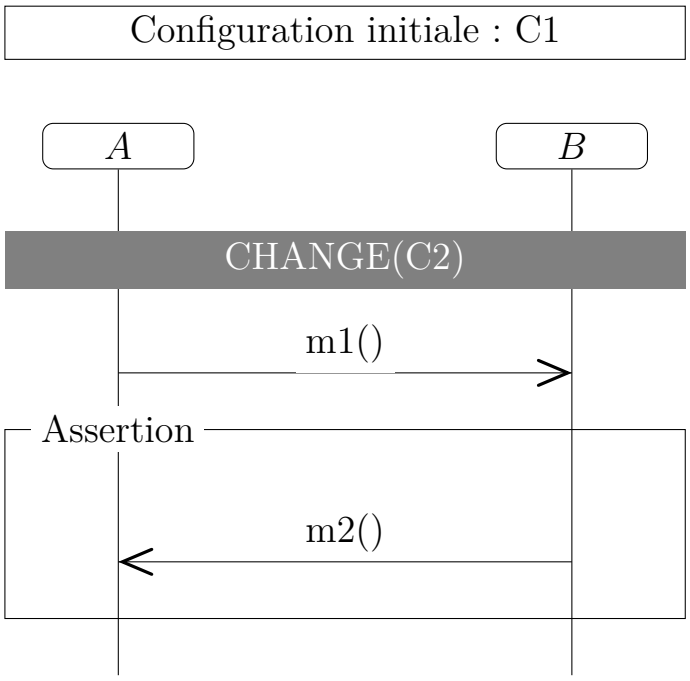


FIGURE 3.8 – Exemple de diagramme de séquence

Un diagramme de séquence (exemple Figure 3.8) permet de représenter des interactions entre des éléments (internes ou externes) d'un système selon un point

de vue temporel logique. Chaque objet possède un axe vertical appelé ligne de vie où le temps s'écoule de haut en bas. Les objets interagissent en s'échangeant des messages. Le diagramme définit des ordres partiels pour les événements représentés. L'ordre des événements de communication dépend de leur position sur la ligne de vie, des opérateurs dans lesquels ils apparaissent (ex : un opérateur de parallélisme *par*) et des autres règles sémantiques (un message est toujours reçu après son émission). Cette représentation permet ainsi d'illustrer un scénario d'exécution avec tous les éléments participants.

Nous allons présenter les choix que nous avons effectués à partir de la syntaxe et de la sémantique de ces diagrammes.

3.4.2.1 Choix de conception

Quand on développe un nouveau langage, la syntaxe et la sémantique doivent s'adapter aux objectifs de haut niveau du langage. L'objectif premier d'un scénario TERMOS est de décrire divers éléments de vérification et d'utiliser ces scénarios pour vérifier les traces d'exécution. Ainsi, le langage a été conçu pour déterminer si une trace d'exécution vérifie une propriété ou non, en limitant le non-déterminisme, et en fournissant une sémantique opérationnelle qui puisse être implémentée dans un outillage logiciel.

Étant donné que TERMOS est basé sur les diagrammes de séquence UML 2.0, une première étape a été d'étudier les possibilités offertes par les diagrammes de séquence. Les travaux correspondants se sont basés sur une analyse détaillée des sémantiques formelles existantes pour les diagrammes de séquence *UML* [Micskei 2010]. Il en ressort que de nombreux choix sémantiques existent même pour des diagrammes simples, comme par exemple le choix d'interpréter un diagramme comme une interaction partielle ou complète. Il a été aussi montré que si le diagramme utilise des opérateurs plus complexes, qui peuvent exprimer des choix alternatifs ou des négations, alors il devient plus difficile de décider si une trace d'exécution donnée vérifie un scénario. Pour adapter les diagrammes de séquence aux besoins de TERMOS, les travaux se sont basés sur l'analyse de [Micskei 2010], et ont proposé de retirer certains éléments et de rajouter plusieurs restrictions syntaxiques.

Interprétation d'une interaction basique Dans TERMOS, les traces d'exécution peuvent être classées dans trois catégories : valide, invalide, non conclusive. Dans un scénario d'exigence ou d'objectif de test, seule la partie significative du comportement est représentée (sous-ensemble de nœuds, sous-ensemble de messages). Par conséquent, un scénario TERMOS ne décrit qu'une interaction partielle. Cela signifie qu'il peut y avoir un passé et un futur à la trace d'exécution. En plus des messages décrits dans le scénario (qui sont ceux qui nous intéressent), la trace d'exécution peut contenir d'autres messages qui s'entrelacent avec les précédents.

Introduction des fragments combinés (CombinedFragments) Un fragment combiné représente des articulations d'interactions. Il est défini par un opéra-

Opérateurs permettant une représentation plus compacte du diagramme	
alt	représente un choix de comportement.
opt	représente un choix de comportement où l'exécution de son contenu est optionnelle.
break	représente une rupture dans le scénario, son contenu est exécuté à la place de la suite du scénario.
loop	représente une boucle qui sera répétée un nombre prédéfini de fois.
Opérateurs modifiant l'ordre partiel des événements	
par	représente une exécution parallèle des opérandes qu'il contient.
seq	représente une relation d'ordre faible entre les opérandes.
strict	représente une relation d'ordre strict entre les opérandes.
critical	représente une zone critique, cela signifie que la trace de cette zone ne peut contenir que les occurrences spécifiées dans la zone.
Opérateurs modifiant la relation de conformité	
neg	représente une trace qui est désignée comme invalide.
assert	représente les seules traces valides.
ignore	représente l'existence possible d'autres types de message non représentés dans le *CombinedFragment*.
consider	seuls les messages spécifiés peuvent se produire dans le *CombinedFragment* considéré.

TABLE 3.1 – Opérateurs pouvant être dans un *CombinedFragment*

teur et des opérandes. L'opérateur conditionne la signification du fragment combiné. Il existe douze opérateurs définis dans la notation UML 2.0, cf. Tableau 3.1. Les fragments combinés permettent de décrire des diagrammes de séquence de manière compacte. Ils peuvent faire intervenir l'ensemble des entités participant au scénario ou juste un sous-ensemble. Les fragments alternatifs (avec opérateur *alt*), les compositions parallèles (avec opérateur *par*) ou encore les négations (avec opérateur *neg*) peuvent être exprimées avec les fragments combinés. Cependant, en UML il n'y a pas de mécanisme de synchronisation entre les lignes de vie à l'entrée comme à la sortie des fragments. Cela pourrait être problématique dans le cas des exigences de tests, car aucun point de synchronisation ne peut être sélectionné pour évaluer les différentes gardes par exemple. Sur la Figure 3.9, le fragment alternatif *alt* coupe les trois lignes de vie mais l'intersection entre le fragment et une ligne de vie n'est pas un point de synchronisation. La vérification de la garde y ne peut se produire qu'après la réception du message c tandis que la vérification de la garde x peut être faite même avant l'envoi de c. La portée des opérateurs n'est pas très claire non plus.

Pour ces raisons, dans TERMOS, nous interprétons l'entrée et la sortie des fragments combinés comme un point de synchronisation à toutes les lignes de vie

participantes. Cette interprétation non standard n'est pas inhabituelle, comme cela a été montré dans [Harel 2008a] ou [Cavarra 2004].

L'étude détaillée des opérateurs pouvant être utilisés dans un fragment combiné a permis de classer ces opérateurs en trois catégories en fonction de l'impact qu'ils ont sur l'ordre des événements contenus dans le fragment : les opérateurs permettant une représentation plus compacte du diagramme, les opérateurs modifiant l'ordre partiel des événements et les opérateurs modifiant la relation de conformité. La fonctionnalité réalisée par chaque opérateur est détaillée dans le Tableau 3.1. Les choix des opérateurs autorisés dans le langage *TERMOS* sont basés sur cette étude.

Seuls *alt*, *opt*, *par*, *assert* et *consider* sont conservés avec des restrictions syntaxiques.

Restriction sur les fragments alternatifs L'ordre entre les éléments des diagrammes sont encodés dans un formalisme basé sur des états. Un point important est la manière de manipuler les fragments alternatifs dans les diagrammes.

Sur la Figure 3.9, la synchronisation à l'entrée du *alt* signifie que la garde doit être évaluée après la réception de *c*. Dans un scénario d'exigence représentant un comportement partiel, il se peut que des événements de communication non représentés soient entrelacés avec ceux représentés. Pour être plus précis, il est possible que des événements se produisent entre la réception de *c* et l'envoi de *a* ou *b*. Que se passe-t-il si ces événements modifient la valeur évaluée par la garde ? De la même façon, des événements non représentés peuvent avoir une influence sur des valeurs évaluées dans un invariant d'état (*StateInvariant*). Différentes stratégies peuvent être envisagées pour décider quand évaluer un prédicat [Micskei 2010], elles vont de l'évaluation au plus tôt à une évaluation à un instant choisi de manière arbitraire.

Pour résoudre les problèmes d'évaluation de prédicats, une restriction sur les prédicats apparaissant dans les gardes et les invariants d'état est imposée. Les prédicats locaux ne peuvent se référer qu'à :

— des paramètres des messages envoyés ou reçus par cette ligne de vie,

— des variables des nœuds de la ligne de vie cible dans la configuration spatiale courante.

Les prédicats globaux quand à eux ne peuvent se référer qu'à :

— des paramètres des messages envoyés ou reçus par l'une des lignes de vie impliquées,

— des variables des nœuds de l'une des lignes de vie impliquées dans les configurations spatiales antérieures ou courantes.

Tout ceci permet de garantir que les messages et les nœuds non représentés ne peuvent pas changer la valeur d'un prédicat.

Dans TERMOS, pour rendre possible une vérification, il est nécessaire d'avoir un point de synchronisation global quand toutes les lignes de vie évaluent les gardes et choisissent une alternative.

Il peut aussi se poser un problème de non-déterminisme dû à la construction des diagrammes de séquence. Il est possible par exemple d'avoir un fragment *alt* avec plus d'une garde à vrai. Nous ne voulons qu'une forme déterministe pour l'évaluation d'une garde. Pour résoudre ce problème, il a été choisi de transformer l'opérateur *alt* en une construction si-alors-sinon.

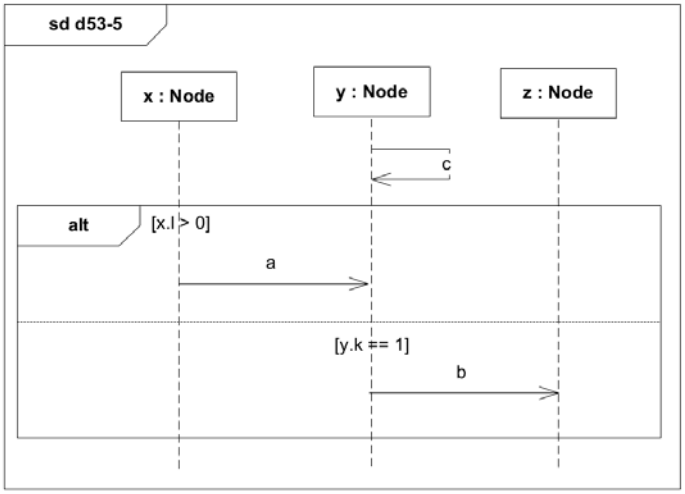


FIGURE 3.9 – Pas de synchronisation à l'entrée d'un fragment

Restriction sur les opérateurs de conformité Les opérateurs de conformité (*assert*, *neg*, *consider*, *ignore*) modifient la catégorie d'une trace telle que valide, invalide ou non conclusive. Leur utilisation est très restreinte afin que la vérification de la trace reste réalisable.

Un diagramme ne peut avoir qu'un seul bloc *assert* à la fin et doit couvrir toutes les lignes de vie. La négation *neg* pose beaucoup de problèmes d'interprétation [Micskei 2010], et n'est donc pas utilisée. Elle est remplacée par un prédicat global FALSE dans un fragment *assert*, comme dans un scénario d'exigence négative.

Les opérateurs *consider* et *ignore* modifient l'alphabet des messages qui sont autorisés à s'entrelacer avec ceux qui sont décrits dans le diagramme. L'opérateur emphignore permet de ne pas tenir compte de messages non décrits. Dans TERMOS, l'interprétation par défaut est que tout message non décrit peut s'entrelacer avec ceux décrits sans modifier le sens du scénario, par conséquent l'opérateur *ignore* est inutile. L'opérateur *consider* permet de réduire le nombre de traces valides en indiquant les messages qui ne peuvent pas s'entrelacer avec ceux décrits. Il s'agit d'une interprétation non standard de cet opérateur.

En UML, il est possible d'imbriquer différents opérateurs. Mais l'imbrication des opérateurs de conformité pose beaucoup de problèmes d'interprétation. Par exemple, comment interpréter un diagramme où un fragment interdisant un message $m1$ (via un *consider*) est mis en parallèle avec un fragment faisant apparaître $m1$?

Pour éviter cela dans TERMOS, les imbrications sont fortement limitées (cf. Tableau 3.2). Un seul niveau d'imbrication est autorisé pour des opérateurs de conformité. Un fragment *assert* ne peut être imbriqué que dans un fragment *consider* de premier niveau. De plus le type d'imbrication (qui avec qui) est restreint : par exemple, nous ne pouvons pas avoir deux *assert* imbriqués l'un dans l'autre. De fait, le diagramme ne peut contenir qu'un opérateur *assert*, placé en bas des lignes de vie.

	alt	opt	par	assert	consider
alt	✓	✓	✓	✓	✓
opt	✓	✓	✓	✓	✓
par	✓	✓	✓	✓	✓
assert	✗	✗	✗	✗	✓ [a]
consider	✗	✗	✗	✓ [b]	✗

a. Consider doit être dans le niveau principal du diagramme
b. Assert doit être dans le niveau principal du diagramme

TABLE 3.2 – L'opérateur de la ligne peut-il être imbriqué dans celui de la colonne ?

3.4.2.2 Connexions entre les diagrammes de séquence

Pour connecter des diagrammes de séquence entre eux, il existe des connexions particulières appelées *gates*. Une *gate* peut être vue comme un moyen de transmettre des informations entre différents diagrammes de séquence. Il s'agit d'un point de connexion relatif à un message, ce message a les extrémités situées sur deux fragments d'interaction différents [Omg 2011]. En fonction du type de fragment dans lequel elle est insérée, la *gate* joue un rôle différent :

— *formal gates* dans une *interaction*,

— *actual gates* dans une *interaction use*,

— *expression gates* dans un *combined fragment*.

Les scénarios d'exigence doivent rester simples, pour cela tous les types de *Gates* sont interdites. Cela signifie qu'un scénario ne peut pas faire de référence à un autre scénario par des échanges de messages et que les scénarios sont analysés individuellement.

3.4.2.3 Broadcast

Les diagrammes de séquences *UML* se concentrent sur les communications point à point ; il n'y a aucun élément dédié à la représentation de la diffusion ou du multicast. Ceci représente un inconvénient majeur pour la représentation de la diffusion locale, comme pour illustrer une communication avec des partenaires inconnus dans

un voisinage local. Pour représenter les diffusions, on utilise alors les messages trouvés/perdus comme illustré Figure 3.10. Les messages perdus sont des messages qui n'ont pas un récepteur explicite. De même, les messages trouvés n'ont pas un émetteur explicite. Les messages perdus et trouvés offrent une flexibilité pour représenter des comportements partiels, notamment quand il n'est pas nécessaire de connaître l'émetteur ou le récepteur d'un événement de communication. Pour distinguer les diffusions des messages trouvés/perdus usuels, une étiquette *broadcast* leur est assignée. Une valeur est attachée aux messages correspondant à la diffusion, afin que chaque événement de réception puisse être jumelé à l'événement d'émission qui l'a causé.

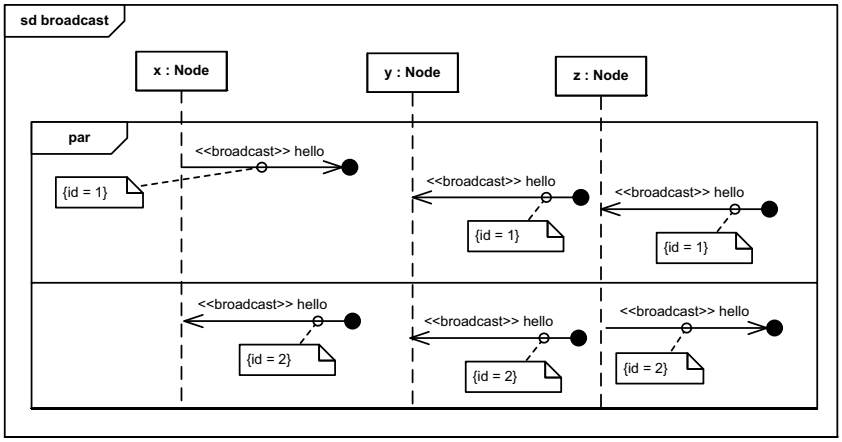


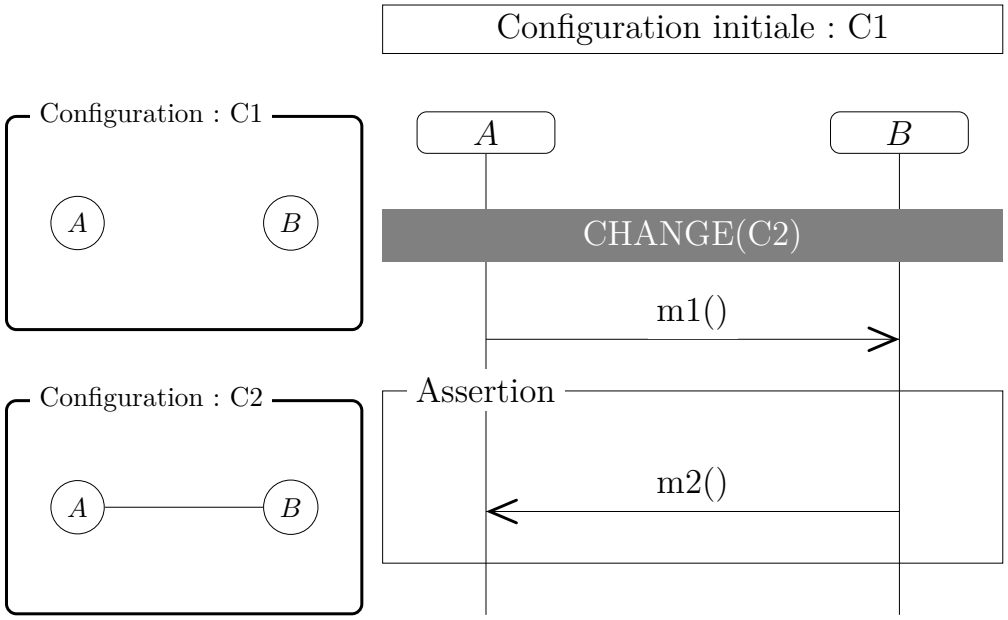
FIGURE 3.10 – Exemple de messages diffusés

3.4.3 Intégration de la vue spatiale dans la vue événementielle

La vue événementielle d'un scénario utilise un diagramme de séquence *UML* avec les extensions apportées par *TERMOS* pour tenir compte explicitement des configurations spatiales définies dans la vue spatiale.

Les changements de configuration sont représentés par des événements globaux qui impliquent une synchronisation globale recouvrant toutes les lignes de vie et de la forme *CHANGE(nom_configuration)*. Un événement de type *Configuration Change* ne peut être imbriqué dans aucun autre opérateur excepté dans un *consider* de niveau principal. Les changements de configuration sont décidés par l'environnement. Ils impliquent toutes les lignes de vie au même instant.

De cette manière, le diagramme peut être décomposé en plusieurs fragments, où chaque fragment correspond à une configuration spatiale bien définie. Cela permet de mieux voir quels événements de communication sont observés dans telle ou telle configuration. Les prédicats (gardes des opérateurs *alt*, invariants d'états) peuvent ainsi faire référence aux variables des configurations courantes ou passées (les variables étiquetées de nœud). Pour l'élément graphique de changement de configuration, le symbole de *Continuation* a été réutilisé.



(a) Vue spatiale (b) Vue événementielle

FIGURE 3.11 – Scénario *TERMOS*

La Figure 3.11 représente un scénario *TERMOS* complet. Il est composé d'une vue événementielle (Figure 3.11b) et d'une vue spatiale (Figure 3.11a) contenant deux configurations spatiales. Les liens entre les deux vues sont matérialisés par la configuration initiale du scénario *C1* ainsi que par l'événement de changement de configuration *CHANGE(C2)*.

Sur cet exemple nous avons deux fragments associés à deux configurations distinctes. Nous avons un fragment dans lequel les nœuds sont dans la configuration initiale $C1$ et un autre fragment à partir de l'événement de changement de configuration *CHANGE(C2)*. Le deuxième fragment est composé de deux sous-fragments (avant le *assert* et dans le *assert*) mais les événements sont observés dans la configuration $C2$.

3.5 Analyse de l'ordre des événements

3.5.1 Sémantique

La sémantique de *TERMOS* est inspirée de la sémantique proposée pour les *Live Sequence Charts* (*LSC*) et, plus spécifiquement celle définie par Klose [Klose 2003]. L'approche consiste à construire un automate à partir du diagramme de séquence, les états de l'automate sont déterminés par les coupes du diagramme. De façon informelle, une coupe est destinée à représenter un état global caractérisé par les événements qui ont eu lieu jusqu'alors, il est donc significatif de raisonner sur le passé et le futur de cet état. Les transitions de l'automate représentent les relations

de succession entre les coupes. L'approche de Klose a été étendue pour incorporer les éléments *UML* absents du *LSC*, par exemple, les fragments combinés *alt* et *par*, mais aussi pour prendre en compte la mobilité des nœuds avec la gestion des *broadcast* ainsi que des changements de configuration. La construction de l'automate diffère de cette approche sur quelques aspects :

— La sémantique de Klose encode les ordres partiels dans des automates de Büchi, construits pour pouvoir traiter des traces infinies. Puisque nous ne traitons que des traces finies, nous nous contentons d'automates standards.

— Klose utilise un traitement divisé en deux étapes : le pré-automate qui, pour nous, représente l'ensemble des événements avant le bloc *assert* et l'automate qui représente les événements dans le bloc *assert*. La sémantique de TERMOS permet de ne construire qu'un seul automate pour l'ensemble du diagramme avec la totalité des événements (avant et après le *assert*).

— Klose autorise de multiples événements à se produire en même temps tandis que nous évaluons les événements les uns après les autres, selon une sémantique d'entrelacement.

En ce qui concerne les deux derniers points, les choix pour TERMOS sont similaires à ceux faits pour la variante *UML* du *LSC*, appelée *MSD* [Harel 2008b]. Cette variante autorise aussi la capture d'événements entrelacés dans un seul automate. À ces différences près, la définition de la sémantique de TERMOS suit de près les étapes identifiées par Klose.

Premièrement, le diagramme est analysé. Ses blocs de construction élémentaires ainsi que leurs ordonnancements sont identifiés. Ensuite, l'automate est construit en utilisant la structure créée à l'étape précédente. Comme précisé dans la section 3.4, le diagramme doit respecter un ensemble de règles pour être bien formé. Le lien avec la vue spatiale s'effectue par le biais des événements de changement de configuration.

3.5.2 Construction de l'automate

L'objectif est de valider des traces d'exécution à partir de scénarios d'exigence décrits selon deux vues (spatiale et événementielle). Le traitement de ces deux vues est effectué selon deux méthodes différentes : résolution de problème d'appariement de graphes avec GraphSeq pour la vue spatiale et vérification à base d'un automate pour la vue événementielle. Dans cette partie, nous allons présenter une description haut-niveau de la sémantique de la vue événementielle de TERMOS.

Nous allons illustrer le principe en nous basant sur le scénario de la Figure 3.11. C'est un scénario relativement simple qui ne contient aucun fragment alternatif ou de composition parallèle.

La construction de l'automate symbolique correspondant à la vue événementielle implique deux étapes principales :

1. un pré-traitement
2. un algorithme de déroulement

Une présentation de haut niveau de cette construction est donnée ci-après.

Pré-traitement Le diagramme de séquence doit être analysé pour extraire ses éléments atomiques (appelés *atomes*) et les relations d'ordres qu'il y a entre eux.

Définition 1 *Les blocs de construction élémentaires d'un scénario* TERMOS *sont appelés des* atomes. *Les éléments suivants représentent des* atomes *:*

- *les débuts des lignes de vie,*
- *les fins des lignes de vie,*
- *les événements d'émission et de réception de messages,*
- *les invariants d'état,*
- *les changements de configuration spatiale,*
- *l'entrée et la sortie des fragments combinés,*
- *les gardes.*

Par exemple, les éléments atomiques apparaissant sur la ligne de vie a de la Figure 3.11b sont : le début de la ligne de vie, le changement de configuration, l'instant d'émission de $m1$, le point d'entrée du fragment *assert*, l'instant de réception de $m2$, le point de sortie du fragment *assert*, la fin de la ligne de vie.

Les atomes sont regroupés en classes qui capturent les événements simultanés, comme par exemple toutes les lignes de vie qui entrent dans un *assert* au même instant. Les relations de conflit et de précédence entre les classes peuvent ainsi être calculées. Sur la Figure 3.11b, l'émission de $m1$ précède sa réception sur la ligne de b, mais aussi précède l'entrée dans le fragment *assert*. Les relations de conflit concerneraient les atomes localisés dans différents opérandes d'un fragment combiné *alt*, cas non illustré ici.

Algorithme de déroulement La construction d'un automate symbolique basé sur un scénario TERMOS est réalisé à l'aide d'un algorithme de dépliage. Le principe de celui-ci est de dérouler progressivement l'ensemble des classes des atomes du diagramme jusqu'à ce qu'elles aient toutes été traitées. Un état de l'automate est un état global du scénario qui représente la progression de toutes les lignes de vie.

L'algorithme démarre dans un état initial avec le début des lignes de vie non déroulées et dans la configuration spatiale initiale. Ensuite, il utilise les relations de précédence et de conflit pour rechercher les classes d'atomes autorisées et calculer les états suivants. Chaque transition est étiquetée par un label en fonction de la classe non déroulée courante. Un label peut être une expression d'événement, consommant un événement de la trace à qui elle correspond, ou un prédicat qui doit être évalué sans consommer un événement. Les deux types de labels peuvent impliquer des variables, et la consommation d'un événement peut déclencher des actions de mise à jour. Si l'analyse de la trace atteint un état où aucune transition ne peut être franchie, alors l'automate s'arrête et renvoie un verdict qui dépend de la catégorie de l'état.

Cet algorithme appliqué au scénario décrit graphiquement dans la Figure 3.11 permet de produire l'automate présenté Figure A.4.

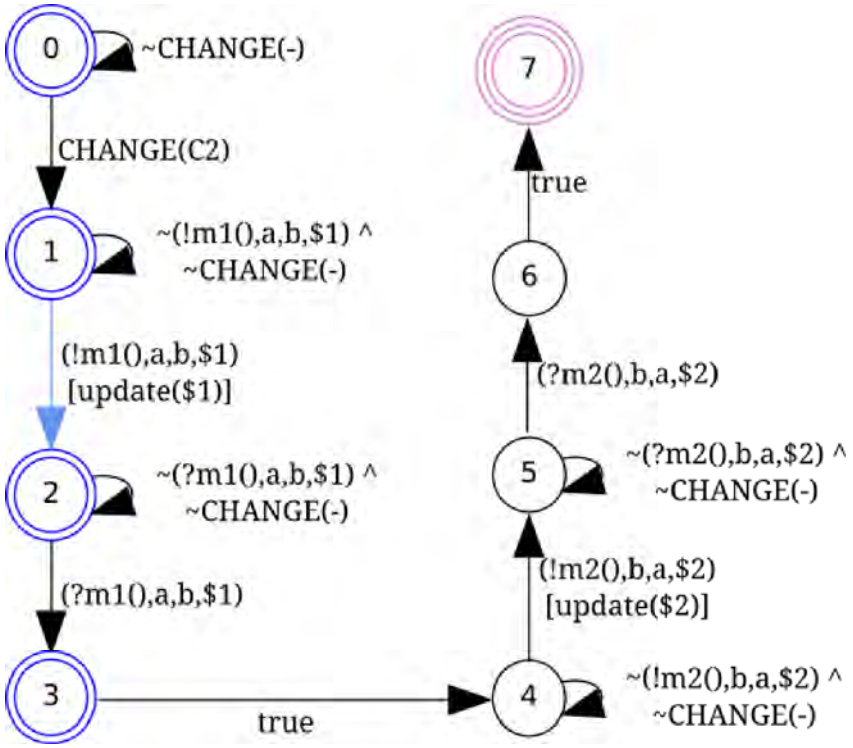


FIGURE 3.12 – Automate correspondant au scénario de la Figure 3.11

L'automate de la Figure A.4 possède trois catégories d'états :

1. les états d'acceptation triviale : représentés par un double cercle sur la figure, ils sont utilisés pour indiquer les traces qui ne présentent pas de comportement potentiel avant un *assert*. Si l'automate s'arrête sur un de ces états alors le verdict est *non conclusif*.
2. les états de rejet : représentés par un simple cercle sur la figure, ils indiquent si la trace est invalide. Si l'automate s'arrête sur un de ces états alors le verdict est *invalide*.
3. les états d'acceptation stricte : représentés par un triple cercle, ils indiquent que la trace a atteint avec succès la fin du fragment *assert*. Si l'automate s'arrête sur un de ces états alors le verdict est *valide*.

Les états sont étiquetés par un identifiant et l'ensemble des variables qui est en train d'être évaluée. Par exemple, dans l'état initial 0, les seules variables valuées sont celles de la configuration courante $C1$. Nous connaissons donc l'identité des nœuds qui jouent le rôle de a et b.

Les transitions qui sortent des états 0 et 1 possèdent des labels qui illustrent des expressions d'événements. La transition de 0 à 1 consomme un changement de configuration de $C1$ à $C2$. La transition réflexive (transition qui transite d'un état vers lui-même) sur 0 consomme n'importe quel événement qui n'est pas un changement de configuration, c'est-à-dire qu'elle peut consommer n'importe quel

événement de communication.

La transition de 1 vers 2 consomme l'émission du message $m1$. L'expression de l'événement $(!m1(), a, b, \$1)$ est un quadruplet où $!m1$ représente l'émission du message, a est le nœud à l'origine de cet événement, b est le nœud recevant cet événement, et \$1 est un identifiant de message symbolique. Étant donné que \$1 est une variable libre dans l'état 1, elle peut être associée à n'importe quel identifiant généré par les fonctions d'instrumentation sur la plate-forme de test. La transition vers l'état 2 met à jour cette variable avec *update*(\$1), et donc \$1 n'est plus libre pendant l'attente de l'événement de réception.

Les transitions qui ne sont pas étiquetées par des expressions d'événement ne consomment pas d'événements de la trace. Les exemples des transitions $3 \rightarrow 4$ et $6 \rightarrow 7$ correspondent à l'entrée et à la sortie d'un fragment *assert*.

Une fois que le scénario a été transformé en un automate, il peut être utilisé pour évaluer des traces en attribuant des verdicts basés sur le type de l'état sur lequel l'automate s'est arrêté après avoir évalué la trace.

3.5.3 Vues spatiale et événementielle combinées

L'analyse de la vue événementielle d'un scénario *TERMOS* produit un automate symbolique contenant des variables pouvant dépendre des configurations spatiales. Pour cela, il est nécessaire d'instancier l'automate avec les configurations concrètes provenant de la trace d'exécution pour pouvoir vérifier celui-ci. L'instanciation de l'automate dépend donc de l'étape préliminaire de valuation des configurations concrètes par l'outil *GraphSeq* (présenté en détails section 6.1) qui est en charge d'apparier les nœuds des configurations spatiales avec les nœuds concrets de la trace d'exécution. Son but est d'analyser une trace d'exécution pour identifier la séquence de configurations spatiales recherchée pour le scénario et de retourner la fenêtre temporelle pendant laquelle la séquence se produit ainsi que l'ensemble des nœuds concrets identifiés. L'automate sera vérifié pour chaque résultat fournis par *GraphSeq*.

La vérification, qui indique si une trace satisfait ou pas un scénario, est exécutée dans les conditions suivantes :

— L'analyse démarre dans un état où le système est dans une configuration concrète qui correspond à la configuration initiale du scénario.

— Les valeurs concrètes des variables provenant de la configuration spatiale sont connues, par exemple, les identifiants des nœuds participant au scénario.

— La trace inclut les événements de changements de configuration.

3.6 Conclusion

Les caractéristiques spécifiques des systèmes mobiles engendrent des comportements dynamiques qui rendent complexes les activités de test et de vérification. Nous avons choisi d'aborder le problème en décrivant un langage formel graphique

qui permette de capturer les interactions clefs de ces systèmes. Dans ce chapitre, nous avons présenté ce langage appelé *TERMOS*, qui permet de représenter des scénarios d'exigence et d'objectifs de test. TERMOS est le cœur de nos travaux. Ce langage est une extension des diagrammes de séquence UML permettant de prendre en compte les spécificités des comportements d'une application mobile. Il permet la représentation des interactions entre les différents dispositifs mobiles qui composent ce type de systèmes.

TERMOS contient plusieurs éléments importants pour représenter des scénarios mobiles : une vue spatiale pour abstraire le mouvement et les créations/disparitions de nœuds par une séquence de graphes étiquetés ; une vue événementielle qui inclut les événements de changements de configuration et les événements de communication comme la diffusion locale.

TERMOS permet de spécifier des propriétés de sous-ensembles de nœuds pour des motifs prédéfinis de configurations spatiales. Ces propriétés sont liées à l'ordre partiel entre les communications et les événements de changements de configuration. Ces propriétés se retrouvent sous trois formes : les exigences positives, les exigences négatives et les objectifs de test.

Enfin, pour implémenter ce langage formel, nous avons présenté une sémantique qui combine l'appariement des graphes et une sémantique opérationnelle basée sur les états pour les éléments des diagramme de séquences UML.

Chapitre 4

Conception et architecture

Sommaire

Bien que les grands principes du langage *TERMOS* aient été définis avant cette thèse [Huszrel 2008], aucune intégration de celui-ci dans un environnement de développement n'existait. Ce chapitre présente les choix de conception et d'architecture qui ont permis la mise en œuvre du langage *TERMOS* ainsi que son utilisation pour le test de systèmes mobiles.

Dans un premier temps, nous présentons l'architecture de la plate-forme de test et de l'outillage logiciel que nous avons conçus afin de mettre en œuvre cette méthode de test consistant à spécifier un scénario puis vérifier celui-ci sur une trace d'exécution. Dans un second temps, nous présentons les formats de fichier permettant des échanges entre les différentes étapes de vérification d'un scénario. Enfin, nous présentons l'environnement de développement choisi pour la mise en œuvre du langage et comment nous avons intégré l'ensemble des éléments de notre architecture afin d'être capable de vérifier un scénario.

4.1 Architecture générale

Lors de la conception du processus de test d'applications mobiles, nous avons fait le choix de répartir en trois grandes activités les étapes nécessaires à la réalisation des tests. Il s'agit de la **collecte de traces**, de la **spécification du scénario** et de la **vérification du scénario**. La **collecte de traces** se déroule habituellement sur

une plate-forme de simulation. Il est aussi possible d'utiliser des traces provenant d'un environnement réel. La **spécification du scénario** se déroule au sein d'un atelier *UML*. L'activité **vérification du scénario** quant à elle, attend des fichiers provenant des deux activités précédentes ce qui implique qu'elles aient été réalisées préalablement. Cette structure est présentée dans la Figure 4.1. Elle montre l'ensemble des éléments composant notre processus de test ainsi que les connexions entre eux. Dans la suite de cette section, nous présentons en détail chacune des trois activités.

4.1.1 Spécification du scénario

L'étape intitulée *Spécification du scénario* peut être découpée en trois phases qui consistent à spécifier le scénario à l'aide de diagrammes *UML*, à vérifier que celui-ci est bien formé syntaxiquement puis finalement à le transformer dans des formats exploitables. L'ensemble de cette étape de spécification du scénario est présenté dans les paragraphes suivants et est détaillé au sein du chapitre 5.

Spécifier un scénario en *UML* : Cette phase est réalisée par un opérateur qui saisit un ensemble de diagrammes *UML* dans un éditeur *UML* afin de décrire le scénario qu'il souhaite vérifier. Cet éditeur intègre les spécificités des systèmes mobiles. Il a été spécialisé à l'aide de profils *UML* ainsi que d'une palette contenant les objets couramment utilisés dans les scénarios permettant de simplifier la procédure de création de scénarios.

Vérifier qu'un scénario est bien formé : L'utilisation d'un profil *UML* ne permet pas à lui seul de garantir qu'une paire de diagrammes pour la vue spatiale et la vue événementielle représente un scénario *TERMOS* bien formé. Nous avons défini un ensemble de vérifications syntaxiques afin de garantir la bonne forme du scénario. Une partie de ces vérifications est directement issue des règles de construction d'un scénario qui ont été présentées dans le chapitre 3, d'autres proviennent des extensions au langage *TERMOS* que nous avons proposées, enfin d'autres vérifications sont liées à l'atelier *UML* dans lequel le langage *TERMOS* a été mis en œuvre.

Transformation du scénario : Lorsqu'un scénario a été vérifié et est correct, deux fichiers sont générés. Le premier appelé *Motif* représente la séquence de configurations spatiales à rechercher par l'outil *GraphSeq* dans les traces collectées lors de l'exécution. Le second contient l'*automate* représentant le scénario complet sous la forme d'un ensemble d'états et de transitions. Cette transformation est réalisée à l'aide de l'algorithme présenté dans le chapitre 3 et détaillé dans l'annexe A.

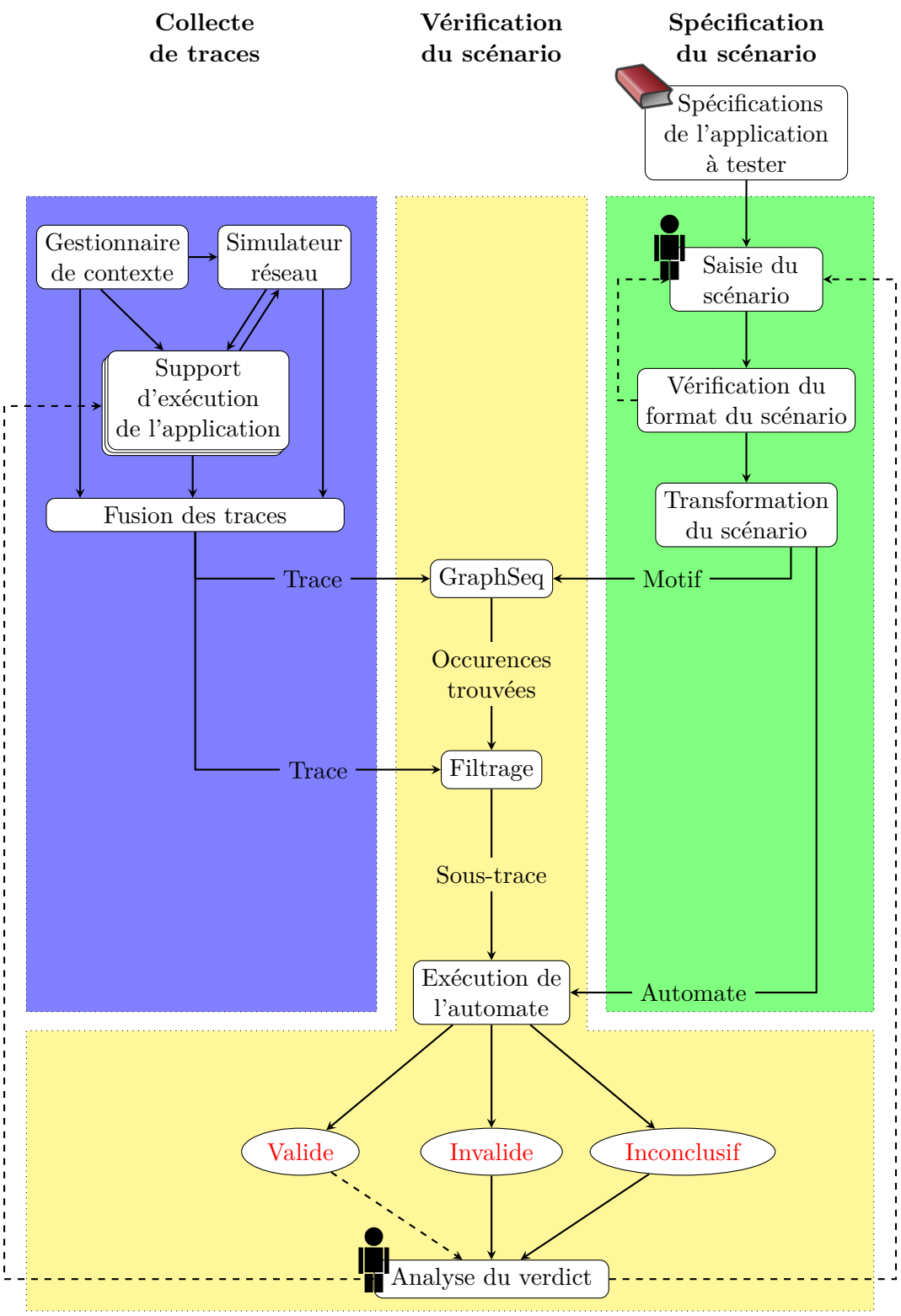


FIGURE 4.1 – Schéma du processus de test

4.1.2 Collecte de traces

Afin de pouvoir vérifier un scénario et donner un verdict de test, il est nécessaire de collecter des traces d'exécutions du système. Cette étape est complexe et nécessite de pouvoir contrôler l'exécution du système sous test sur un ensemble de nœuds mobiles, de contrôler les positions relatives de chacun d'eux ainsi que l'ensemble des communications entre eux afin de pouvoir simuler le système et collecter des traces. Elle est réalisée sur une plate-forme de simulation contenant (1) un gestionnaire de contexte chargé de coordonner la distribution des informations contextuelles à l'ensemble des nœuds, (2) un simulateur réseau chargé de contrôler les flux entre les nœuds et (3) un support d'exécution instrumenté de l'application à tester pour chacun des nœuds. L'ensemble de ces éléments participe à la collecte de traces en enregistrant les événements qu'ils ont à gérer. Ces événements doivent pouvoir être ordonnés les uns par rapport aux autres, et pour cela il est nécessaire d'avoir soit une référence temporelle globale ou bien un ordre partiel avec estampillage comme proposé par les auteurs de [Hallal 2006]. Un dernier outil est chargé de collecter ces données et de les fusionner en une seule trace où les événements apparaissent entrelacés de manière chronologique. La mise en œuvre de cette plate-forme de simulation est fortement dépendante de l'application à tester. Par exemple la capture des événements de communication peut être réalisée par le simulateur de réseau ou bien par une instrumentation de l'application à tester. L'ensemble de cette phase de collecte de traces est illustrée sur une étude de cas dans le chapitre 7.

4.1.3 Vérification du scénario

La *Vérification du scénario* consiste à analyser les traces avec des outils développés en concluant par un verdict (valide, non valide, inconclusif). Elle se déroule en quatre étapes, les fonctionnalités réalisées par chacune d'elles sont introduites dans les paragraphes suivants. Leur fonctionnement est présenté en détail dans le chapitre 6.

GraphSeq : La première étape de vérification du scénario consiste à rechercher la séquence de configurations spatiales du scénario dans la trace d'exécution grâce à l'outil *GraphSeq*. La séquence de configurations spatiales est écrite dans la vue spatiale du scénario TERMOS. Une version initiale de *GraphSeq* [Nguyen 2009] a été développée préalablement à cette thèse. Cette version bien que fonctionnelle sur des graphes de taille limitée n'a pas supporté le passage à l'échelle avec le traitement de séquence de graphes contenant plusieurs centaines de graphes. Tout en gardant son algorithme de base, nous avons modifié *GraphSeq* afin qu'il soit capable de passer à l'échelle en améliorant certains points comme notamment : le changement de la structure interne de données, l'ajout d'un prétraitement sur les motifs de graphes.

Filtrage : A partir des nœuds impliqués dans le scénario ainsi que des résultats d'appariement fournis par l'outil *GraphSeq*, la trace d'exécution est filtrée pour

créer une sous-trace par occurrence de la vue spatiale trouvée par *GraphSeq*. En effet, chaque vue spatiale peut apparaître plusieurs fois dans une trace. Cette sous-trace est enregistrée dans le format *etrace* défini dans la suite de ce chapitre. Cette sous-trace ne contient que les changements de configurations identifiés par *GraphSeq* comme étant les changements présents dans le scénario, et que les messages émis et reçus par des nœuds identifiés par *GraphSeq*.

Exécution de l'automate : L'automate généré lors de l'étape de transformation du scénario est exécuté sur chacune des sous-traces générées par l'étape de filtrage. Chaque exécution d'un automate retourne un *verdict* déterminé par l'état dans lequel l'exécution se termine. Pendant cette étape, l'ensemble des événements qui a déclenché le franchissement d'une transition est mémorisé pour l'étape suivante qui consiste à analyser le verdict.

Analyse du verdict La vérification d'un scénario peut conduire à plusieurs milliers d'exécutions de l'automate en fonction du nombre d'occurrences du motif trouvé par *GraphSeq* ainsi que du nombre de franchissements de transitions initiales. Devant la quantité de données à analyser, nous avons créé un outil aidant l'utilisateur à analyser ces *verdicts*. Il compile l'ensemble des *verdicts* afin d'avoir une vue globale de la validation. Par exemple, combien de validations ont conduit à un même *verdict* et combien de validations se sont terminées dans le même état. Pour chaque validation, il est possible de visualiser l'ensemble des événements qui a conduit à l'état final. Si le scénario a permis de déceler la présence d'une faute, cette étape d'analyse est là pour aider l'opérateur à déterminer l'origine de celle-ci et permettre son élimination, et ce, qu'elle provienne d'une erreur de spécification du scénario de test ou bien de l'implémentation testée.

4.2 Validation de l'outillage

L'ensemble des éléments que nous avons mis en œuvre dans cette plate-forme de test, fait l'objet de vérifications. Ces vérifications prennent différentes formes en fonction de la fonctionnalité à tester. Cela peut être une génération automatique de vecteurs d'entrée pour vérifier *GraphSeq*, ou bien des tests unitaires pour vérifier une grammaire, voire la conception d'un ensemble de scénarios *TERMOS* pour vérifier l'application des règles de bonnes formes des scénarios.

4.2.1 GraphSeq

L'outil d'appariement de séquence de graphes *GraphSeq* est testé à l'aide de séquences de graphes générées aléatoirement. Pour cela une séquence de graphes motifs est générée. Ensuite, à partir de la séquence de graphes motifs, une séquence de graphes concrets est générée en ajoutant de façon aléatoire des nœuds et des liaisons à la séquence motif. Cela permet de garantir que le motif recherché sera bien présent à l'intérieur de la séquence de configurations concrètes. Une vérification

automatique à partir d'un ensemble de séquences permet de contrôler le fonctionnement correct de l'application *GraphSeq* et surtout sa non régression lors d'une éventuelle modification de son code.

4.2.2 Grammaire des prédicats

La grammaire des prédicats qui sera présentée en détail au chapitre 5 est vérifiée à l'aide d'un ensemble de tests unitaires. Chaque opérateur est associé à au moins deux tests, un valide et un invalide. À chaque modification de la grammaire, l'ensemble des tests est exécuté pour s'assurer de la non régression de celle-ci. S'agissant d'une grammaire, il est impossible de tester l'ensemble des combinaisons d'opérateurs possibles car cet ensemble est infini, mais il est par contre possible de tester individuellement chaque opérateur sur des expressions simples. Ces tests permettent d'assurer la couverture de l'ensemble de la grammaire.

4.2.3 Vérification syntaxique

Les vérifications syntaxiques appliquées à chaque scénario avant la génération de l'automate sont elles aussi vérifiées. Pour celles-ci, nous avons fait le choix de créer manuellement un ensemble de scénarios de test. Pour chaque vérification, un scénario valide et un invalide ont été créés. La vérification se fait actuellement manuellement sur chaque scénario de test et consiste à exécuter l'ensemble des vérifications syntaxiques sur chaque scénario et analyser les retours d'erreurs. Il est prévu d'automatiser ces vérifications pour être capable de les intégrer dans le processus de développement.

4.3 Choix de représentations des données

Les composants de notre plate-forme doivent être capables de communiquer entre eux. Pour cela, il est nécessaire de pouvoir représenter les données à échanger. Nous avons créé des formats de fichiers dédiés. Le logiciel *GraphSeq* étant existant, ses formats d'échange de données ont été conservés. *GraphSeq* utilise deux fichiers représentant des séquences de graphes : le premier contenant la description de séquences de graphes motifs à rechercher et le second contenant tous les graphes présents dans la trace d'exécution. Il produit donc un fichier contenant toutes les occurrences trouvées des séquences de graphes motifs.

Pour ce qui est des autres données telles que la représentation des automates ou bien des traces d'exécution, une représentation à l'aide d'un balisage de type *XML* a été mis en œuvre. Ce type de balisage permet une lecture aisée que ce soit par un programme ou par un utilisateur. Le format *XML* permet aussi d'étendre un format existant en ajoutant des balises, par exemple, cela sera utile pour déclarer des formats de messages adaptés à chaque étude cas. Grâce à l'utilisation de fichiers *XSD* (*XML Schema Definition*), il est possible de spécifier de manière formelle la structure de chaque format de fichier *XML* et de vérifier automatiquement si un

fichier a une structure conforme au format. Dans la suite de ce manuscrit, nous ne reviendrons plus sur les formats d'échange de données mis en œuvre, ils sont donc décrits en détail dans ce chapitre.

4.3.1 Représentation des automates

Un automate est représenté par un ensemble d'états, de transitions et de variables associés aux états. Chaque automate contient une liste d'états auxquels sont liées des variables. Il contient aussi une liste de transitions contenant chacune l'état de départ, l'état d'arrivée ainsi que la condition pour que cette transition puisse être franchie. Afin de pouvoir stocker l'automate dans un fichier, nous avons utilisé un format de fichier *XML* dont la structure est détaillée Figure 4.2.

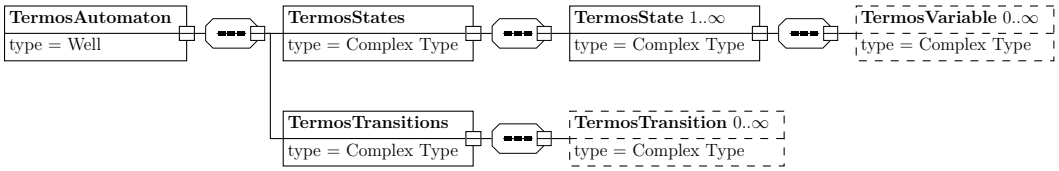


FIGURE 4.2 – Schéma XML pour les automates

Les éléments *TermosState*, *TermosVariable* et *TermosTransition* peuvent contenir des attributs non représentés dans la figure Figure 4.2. Ils sont présentés dans le Tableau 4.1.

Le franchissement d'une transition entre deux états peut être conditionné par la validité d'une expression. Pour cela, nous avons créé un langage dédié permettant de mettre en œuvre des opérations de comparaison entre deux variables numériques (=,>,>=...), des opérations sur des ensembles (*includesAll*,*excludesAll*), des opérations entre un ensemble et une variable (*includes*,*excludes*) ainsi que des opérations logiques (*and*,*or*,*xor*,*not*) permettant de composer une expression avec les autres opérations. Ce langage repose sur un sous-ensemble d'opérateurs *OCL*. Sa grammaire a été codée à l'aide de l'outil *ANTLR* et a permis de générer un compilateur permettant d'évaluer ces expressions.

4.3.2 Représentation des traces d'exécution

Une trace d'exécution doit contenir aussi bien les événements de communication se produisant sur chaque dispositif mobile composant le système à tester que les informations concernant la topologie du système. Dans le but de simplifier la création de traces d'exécution grâce à l'instrumentation d'un *Système sous test* ou à l'utilisation d'un outil de capture de trafic réseau, mais aussi d'avoir une trace compréhensible par un opérateur, nous avons choisi de créer un format de trace dédié. Ce format que nous appelons *etrace* utilise un balisage de type *XML*. Il a été pensé pour être extensible et indépendant de l'étude de cas servant à générer la trace.

Attribut	Description
TermosVariable	
name	Nom de la variable
value	Valeur affectée à la variable
type	Type de la variable
instanciate	Nom de l'élément dont la variable est une instance
TermosState	
stateId	Identifiant unique de l'état
stateKind	Type de l'état (*Valide*, *Invalide*, *Inconclusif*)
TermosTransition	
initFlag	Attribut optionnel permettant d'identifier le premier événement concret apparié à un événement du scénario
from	Identifiant de l'état d'origine
to	Identifiant de l'état d'arrivée
label	Expression qui doit être valide pour que la transition soit franchissable

Table 4.1 – Définition des attributs des éléments d'un automate

Il se compose d'un ensemble de balises et d'attributs obligatoires ainsi que de balises et d'attributs optionnels. La Figure 4.3 détaille l'organisation générale du format de fichier ainsi que les balises obligatoires à l'exception de la balise *Event* qui elle, est présentée dans la Figure 4.4.Un fichier au format *etrace* comporte un nom, un ensemble de nœuds avec un nom et des paramètres, une configuration initiale composée de nœuds et de liens entre eux et pour finir d'un ensemble d'événements. Les événements comportent tous un horodatage. Ils peuvent être de trois types différents. Il peut s'agir d'un changement de configuration, dans ce cas, l'événement contiendra l'ensemble de nœuds et des liens entre eux. Il peut s'agir d'un événement d'envoi ou de réception de message. Dans ce cas, l'événement contient un identifiant de message, l'identité de l'émetteur ainsi que celle du récepteur (seulement si elle est connue) et le contenu du message. Le contenu d'un message étant dépendant de l'étude de cas, il n'est pas défini ici. Il devra par contre être aussi mis sous la forme de balise *XML* pour permettre son utilisation dans les prédicats par exemple.

4.4 Choix de l'environnement de mise en œuvre du langage

Afin d'être utilisable le langage *TERMOS* doit être mis en œuvre dans un atelier *UML*. Lors des différentes tentatives de mises en œuvre, plusieurs ateliers ont été

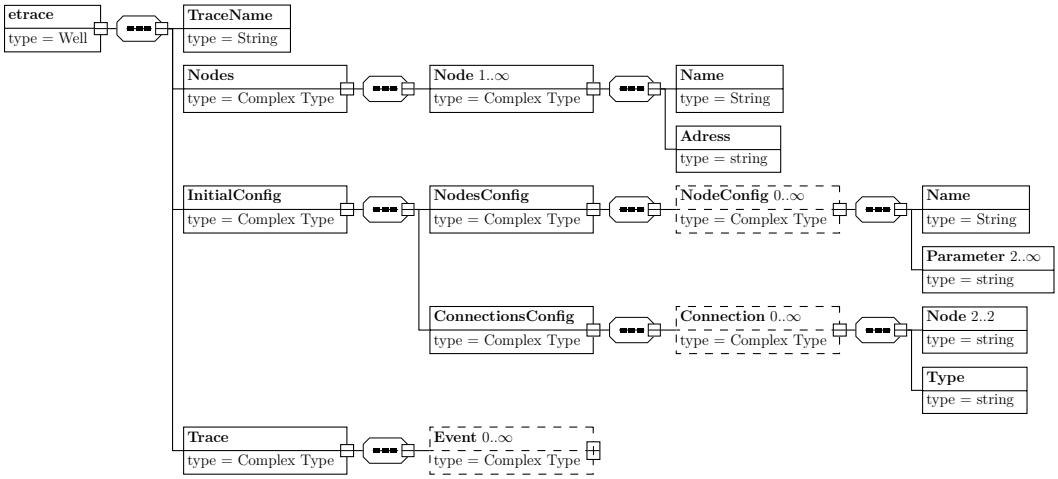


FIGURE 4.3 – Schéma XML général etrace

testés, néanmoins, tous avaient une base commune : la plate-forme *Eclipse*[1]. Des tests ont été menés sur *UML2Tools*[2], *IBM Rational Software Architect*[3], *TOPCASED*[4] et *Papyrus*[5].

Le choix de la plate-forme de développement sur laquelle développer notre langage est fondé non seulement sur les contraintes du langage *TERMOS* mais aussi sur les possibilités d'extensions de la plate-forme. Notre choix s'est porté vers *Papyrus* qui est un outil respectant entièrement le standard *UML 2* défini par l'*OMG* et qui dispose d'une architecture extensible aussi bien directement en *UML* avec la gestion des profils que sous la forme de plugins *Eclipse* pour ajouter des fonctionnalités nécessitant des traitements complexes.

Comme précisé dans le chapitre 3, les scénarios *TERMOS* doivent pouvoir représenter des messages diffusés à un ensemble de destinataires. La représentation graphique la plus adaptée de ces messages dans un diagramme de séquence est celle des messages perdus et trouvés, représentant respectivement des messages sans émetteur et sans récepteur. Lors du choix de l'environnement, seul *Papyrus* mettait en œuvre ce type de messages grâce à son respect des spécifications *UML 2*. Afin de différencier les messages diffusés des messages perdus et trouvés nous avons ajouté un stéréotype *broadcast* qui sera appliqué sur les messages diffusés. Le stéréotype contient en plus un attribut servant d'identifiant de message qui permet de relier plusieurs événements entre eux que ce soit un événement d'émission à plusieurs événements de réception ou seulement un ensemble d'événements de réception entre eux. La gestion des diffusions a aussi été complétée par l'intégration aux règles de vérifications de bonne forme des scénarios de vérifications concernant l'utilisation de ces messages diffusés.

1. http://www.eclipse.org/
2. http://wiki.eclipse.org/MDT-UML2Tools
3. http://www.ibm.com/software/products/fr/ratisoftarch
4. https://www.polarsys.org/
5. http://www.eclipse.org/papyrus/

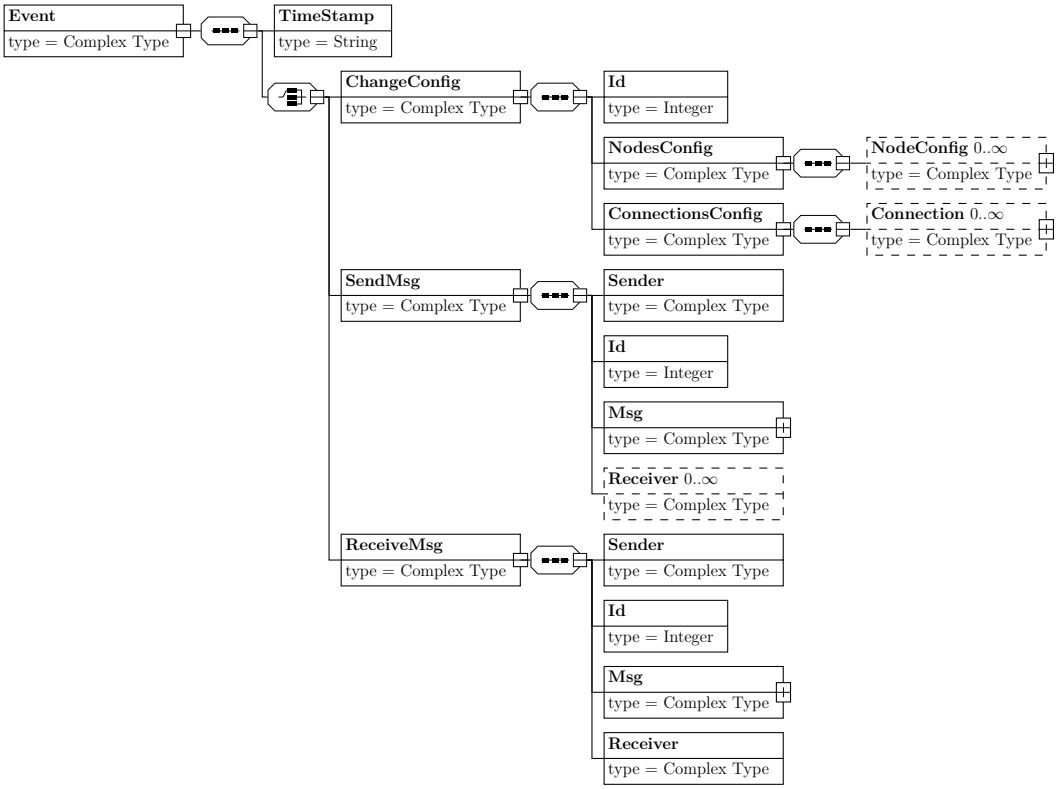


FIGURE 4.4 – Schéma XML de la balise événement

Papyrus répond à tous nos besoins d'extensibilité. De plus il est développé et maintenu principalement par un laboratoire français, le *CEA-List* et est distribué sous licence libre *EPL 1.0* (*Eclipse Public License 1.0*). Ce qui permet à toute personne souhaitant utiliser *TERMOS* de ne pas avoir de frais de licences mais aussi et surtout de modifier le logiciel pour l'adapter à ses besoins, contrairement à l'utilisation de logiciels commerciaux. De plus, depuis la sortie en 2014 de la version *1.0* qui est considérée comme la première version stable, *Papyrus* est sorti de la phase d'incubation pour devenir un projet supporté par la fondation *Eclipse* ce qui est un gage de pérennité de la plate-forme.

4.5 Conclusion

Dans ce chapitre, nous avons présenté l'architecture générale de notre méthode ainsi que sa décomposition en trois grandes étapes qui sont la **spécification du scénario**, la **collecte de traces d'exécutions** et la **vérification du scénario sur les traces**. Les étapes **spécification du scénario** et **vérification du scénario sur les traces** font l'objet chacune d'un chapitre dédié dans la suite de ce manuscrit. L'étape **collecte de traces d'exécutions** sera quant à elle illustrée à l'aide de l'étude de cas car elle est dépendante de celle-ci. Nous avons décrit les stratégies de vérifications mises en œuvre pour contrôler le fonctionnement correct de notre

plate-forme de test. Nous avons aussi présenté les formats de fichier que nous avons créés afin de connecter les différents éléments de cette plate-forme entre eux, nous avons aussi abordé le choix de l'atelier *UML Papyrus* pour mettre en œuvre notre méthode de test.

Chapitre 5

Spécifier un scénario

Sommaire

Ce chapitre présente l'ensemble des étapes nécessaires à la spécification d'un scénario *TERMOS*. Il résume l'ensemble des problématiques que nous avons traitées afin d'être capable de réaliser ces étapes.

Dans un premier temps, nous abordons les problèmes liés à l'édition graphique de scénarios *TERMOS* que ce soit pour la définition de la vue spatiale ou pour la définition de la vue événementielle. Nous avons fait le choix d'utiliser le même éditeur pour représenter la vue spatiale et la vue événementielle ce qui permet de mettre en œuvre des liens entre les deux vues plus facilement. Cette intégration a été possible grâce à la spécialisation de l'éditeur *UML Papyrus* à l'aide de profils *UML*. L'*UML* est un des langages les plus utilisés dans le domaine de la spécification du fonctionnement d'un système. Dans certains cas, l'*UML* est trop général et il est nécessaire d'avoir un langage spécifique au domaine (*Domain Specific Language DSL*). Pour répondre à cette problématique, l'*UML* propose des mécanismes d'extensions. En particulier, les profils *UML* permettent de spécialiser le langage pour l'adapter à la modélisation d'un domaine spécifique [Fuentes-Fernández 2004]. A l'aide d'un profil *UML* dédié à *TERMOS*, nous sommes capables de prendre en compte les spécificités des systèmes mobiles. Il permet par exemple de décrire une topologie entre les nœuds sous la forme de diagrammes d'objets. Il permet aussi d'intégrer à un diagramme de séquence les spécificités du langage *TERMOS*. Le

profil intègre les contraintes syntaxiques de bonne définition des scénarios. Nous avons choisi d'implémenter ces contraintes par des fonctions de vérification écrites en Java, plutôt que par des contraintes *OCL*.

Dans un second temps nous abordons le problème lié à la gestion de scénario complexe avec la possibilité de tester le contenu des messages. Ce problème a nécessité l'introduction d'une grammaire dédiée permettant la gestion des prédicats. Ces prédicats peuvent être utilisés dans des gardes de fragments combinés de type *alt* ou bien dans des prédicats globaux de type *StateInvariant*.

Enfin le dernier problème était de réussir à transformer un scénario graphique décrit en *UML* vers des fichiers spécifiques permettant la validation de traces. L'utilisation du profil *UML* pour *TERMOS* a permis d'automatiser la phase de transformation d'un scénario graphique vers une séquence de graphes contenant la succession de topologies utilisée durant le scénario ainsi qu'un automate contenant la liste exhaustive des états et transitions qui peuvent être parcourus lors de la vérification du scénario. La transformation n'est appliquée qu'à des scénarios bien formés : elle est précédée par un appel de l'ensemble des fonctions de vérification que nous avons définies, et qui sont récapitulées en fin de ce chapitre.

5.1 Définition de la vue spatiale

Une vue spatiale est composée de **configurations spatiales**, elle-même composées de **nœuds** et de **liens** reliant les nœuds entre eux. Afin de représenter des instances de ces éléments dans un diagramme *UML*, nous utilisons un diagramme d'*objets UML*. Le choix du diagramme d'objets plutôt qu'un diagramme de classes est lié au besoin de pouvoir représenter des instances d'un objet. En fonction de l'étude de cas, il peut être nécessaire de différencier certains nœuds par exemple des nœuds mobiles et des nœuds fixes. Il est possible de définir des attributs de nœuds ainsi que des types de nœuds. Pour cela, il est nécessaire de définir les types de nœuds utilisables ainsi que leurs attributs à l'aide d'un diagramme de classe avant de créer un scénario *TERMOS*. Le profil *UML* permettant la mise en œuvre de la vue spatiale au sein de *Papyrus* est représenté dans la Figure 5.1. Les éléments qui le composent sont décrit ci-dessous.

Configuration : La classe *package* représente une configuration spatiale contenant un ensemble de nœuds ainsi que des liens entre ces nœuds. Le stéréotype qui pourra être appliqué à un *package* afin qu'il représente une configuration a été nommé *termosConfiguration*.

Nœud : Un nœud au sens de *TERMOS* est une instance d'un objet, il est représenté dans un diagramme d'objets par une *instanceSpecification* ayant le stéréotype *termosNode*. Les nœuds présents dans les configurations spatiales d'un scénario pourront ensuite être présents dans la vue événementielle de celui-ci. Les types des nœuds sont définis par des classes, permettant de spécifier leurs attributs. Lors de

l'instanciation des classes dans les configurations, les attributs des noeuds peuvent recevoir des valeurs concrètes ou symboliques. Lorsqu'aucune valeur n'est spécifiée, l'attribut correspondant peut être ignoré (c'est donc équivalent à la valeur joker présentée dans le chapitre précédent).

Liens de communication : Comme les nœuds sont représentés par des *instanceSpecification*, il parait normal d'utiliser des objet de type *instanceSpecificationLink* pour représenter les liens de communication entre eux. Un stéréotype général pour définir un lien a été nommé *termosConnection*. Il est possible de typer plus précisément les liens avec la création d'un profil dédié à chaque étude de cas pour, par exemple, spécifier un attribut indiquant la qualité de la connexion.

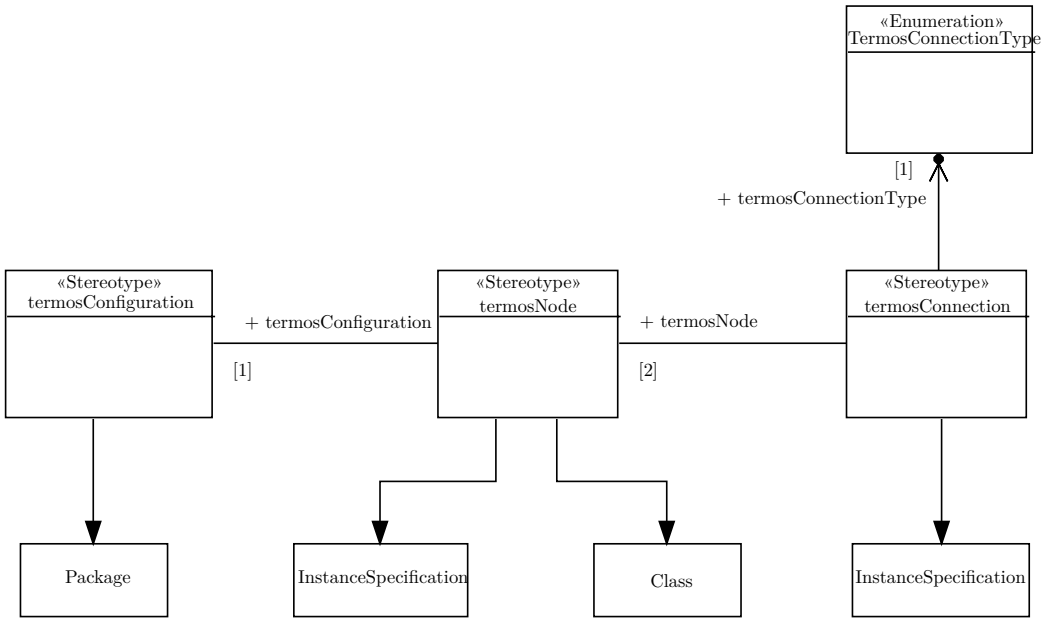


FIGURE 5.1 – Gestion des configurations spatiales

Un exemple de définition de configuration est donné en Figure 5.2. Les nœuds de cette configuration sont des instances de la classe $mobileNode$ et ont deux attributs. L'instance $n1$ a son premier attribut à une valeur concrète, et son deuxième est ignoré. Pour l'instance $n2$, les deux attributs sont considérés avec des valeurs symboliques. L'appariement de graphes déterminera leur valeur concrète, par exemple en appariant $n2$ avec un nœud réel dont le premier attribut est à 1 et le second à 0. Dans la vue événementielle, le comportement défini pourra dépendre explicitement de ces valeurs, en introduisant des prédicats sur z et $myVar$ (par exemple, une garde $z > myVar$).

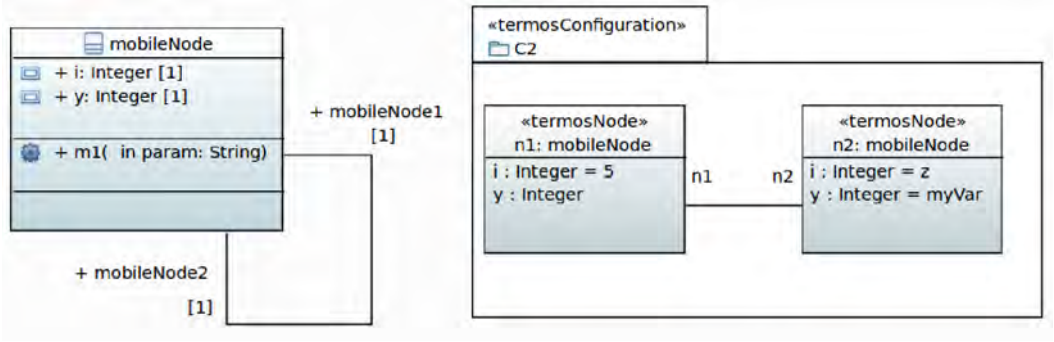


FIGURE 5.2 – Exemple de définition de configuration spatiale.

5.2 Définition de la vue événementielle

La vue événementielle est un diagramme de séquence. Nous avons spécialisé les diagrammes de séquences pour intégrer la gestion des configurations spatiales, de la configuration initiale et des changements de configurations ; mais aussi la gestion des messages diffusés à plusieurs destinataires. Nous avons aussi tenu compte des restrictions apportées à l'utilisation des opérateurs au sein du diagramme.

Pour qu'un diagramme de séquence puisse être considéré comme un scénario *TERMOS* il faut lui appliquer le stéréotype *termosScenario* présenté Figure 5.3. Ce stéréotype permet d'ajouter un attribut *initialConfiguration* de type *termosConfiguration* au diagramme permettant de spécifier la configuration initiale du scénario.

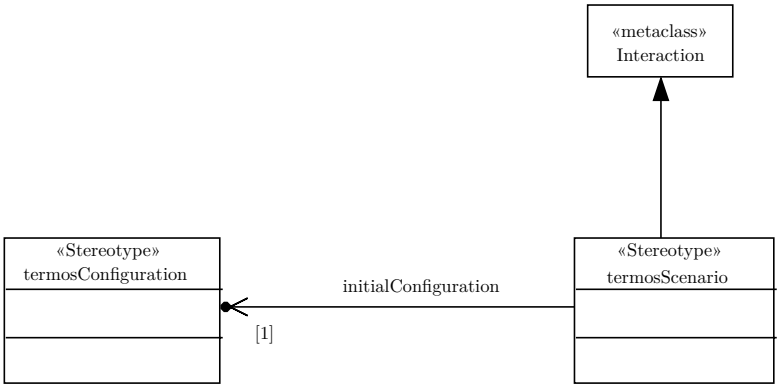


FIGURE 5.3 – Stéréotype appliqué à un diagramme de séquence et configuration initiale liée à celui-ci

Le langage *TERMOS* utilisant deux vues pour représenter un scénario, nous devons nous assurer que les liaisons entre ces deux vues sont bien spécifiées et cohérentes. Pour qu'un diagramme de séquence représente la vue événementielle d'un scénario *TERMOS*, il faut contrôler que le stéréotype *termosScenario* lui soit bien appliqué et qu'une configuration spatiale initiale soit bien sélectionnée. Cette configuration initiale doit être sélectionnée dans la liste des configurations spatiales

disponibles, c'est à dire les *pakages UML* ayant le stéréotype *termosConfiguration* d'appliqués.

Tous les événements de changement de configuration doivent également référencer des *termosConfiguration*. Les lignes de vie du diagramme ne peuvent alors correspondre qu'à des *termosNode* apparaissant dans au moins une des configurations spatiales du scénario.

Des vérifications sont également effectuées sur les messages apparaissant dans le scénario. A l'aide des liens définis dans les configurations spatiales, il est possible de déterminer si un message pourra être échangé entre deux nœuds. Un message représenté dans un scénario doit pouvoir être échangé, c'est à dire que la topologie courante du système doit avoir un chemin possible entre l'émetteur et le récepteur du message. Cette règle ne s'applique qu'aux messages pour lesquels la vue événementielle montre à la fois un événement d'envoi et un (ou plusieurs) événement(s) de réception. Nous n'effectuons pas cette vérification pour les messages trouvés (sans événement d'envoi) ou perdus (sans événement de réception) : comme le scénario n'implique qu'un sous-ensemble de nœuds du système, les messages perdus ou trouvés sont possibles et permettent notamment de représenter des communications avec le reste du système.

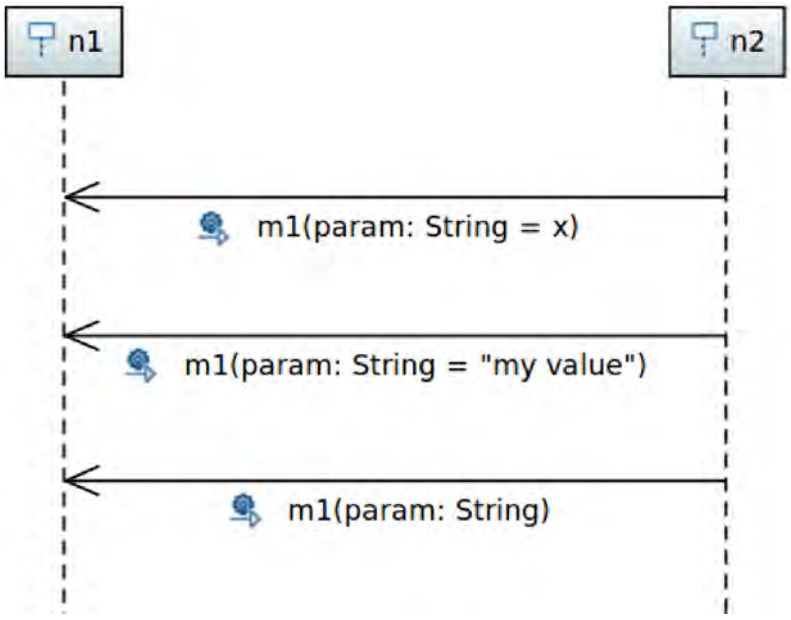


FIGURE 5.4 – Extrait d'une vue événementielle contenant des messages

Lors de la spécification des classes de nœuds, l'utilisateur a déclaré non seulement les attributs de ces nœuds, mais également les opérations offertes, c'est-à-dire les messages qu'ils peuvent traiter. Par exemple, dans la Figure 5.4, les nœuds peuvent traiter des messages $m1$. Nous vérifions que tout message reçu correspond bien à une opération déclarée. Les opérations peuvent avoir des paramètres, et les messages spécifiés dans le diagramme peuvent soit ignorer ces paramètres, soit leur

donner des valeurs concrètes ou symboliques. La Figure 5.4 illustre les différentes possibilités. Comme les valeurs symboliques sont susceptibles d'être ensuite utilisées dans des prédicats, nous vérifions que les types des paramètres des opérations sont compatibles avec les types pris en compte dans notre grammaire de prédicats. Cette grammaire sera présentée dans un paragraphe ultérieur (section 5.3). Les types actuellement pris en compte sont les types Booléen, entier, chaîne de caractère, ainsi que les ensembles d'éléments de ces types.

Dans ce qui suit, nous détaillons plus particulièrement deux points délicats pour la définition de la vue événementielle : l'édition d'éléments *TERMOS* non standards dans *UML*, et les restrictions sur l'utilisation des opérateurs standards dans les diagrammes de séquence.

5.2.1 Représentation d'éléments non standards

Le langage *TERMOS* introduit plusieurs éléments non standards en *UML*, qui ne sont donc a priori pas permis dans les éditeurs. Un premier problème concerne la représentation de points de synchronisation globaux introduits par les événements de **changements de configurations** et l'évaluation de prédicats globaux (par exemple l'assertion *FALSE* d'un scénario d'exigence négative). Le deuxième élément non standard est la gestion des **messages diffusés**. En *UML*, il n'est pas prévu qu'un événement d'envoi d'un messages puisse être lié à plusieurs événements de réception.

5.2.1.1 Changements de configuration et prédicats globaux

Un événement de changement de configuration est un événement global d'un scénario, il doit donc couvrir l'ensemble des lignes de vie de celui-ci. C'est un événement de synchronisation du scénario et de l'automate qui en découle. De même, nous voulons pouvoir représenter des synchronisations sur des prédicats globaux, comme les assertions *TRUE* et *FALSE* terminant un scénario, ou plus généralement des prédicats testant les valeurs symboliques définies dans le scénario. Pour cela, plusieurs possibilités ont été envisagées.

La première solution a été d'utiliser un élément de type *StateInvariant*, qui est un élément simple encapsulant un label. C'est l'élément prévu dans *UML* pour représenter l'évaluation d'un prédicat, et le contenu du label nous permettait de différencier les éléments *TERMOS* de synchronisation. Un label de forme emph-CHANGE(xx) correspondait à un changement de configuration, et tout autre label devait respecter la syntaxe des prédicats *TERMOS*. En *UML* standard, le *StateInvariant* ne s'applique qu'à une seule ligne de vie. Nous avions modifié sa définition dans l'éditeur papyrus pour pouvoir l'appliquer à toutes les lignes de vies.

Malheureusement, cette solution s'est avérée non portable lors des évolutions de Papyrus. Dans les versions les plus récentes, la modification de l'extension spatiale d'un élément ne peut plus se faire facilement dans sa définition. L'éditeur permet à l'utilisateur de modifier manuellement le nombre de lignes de vie couvertes par une

instance d'élément (par exemple, par un *StateInvariant* particulier), via un menu, mais la modification est ensuite mal gérée graphiquement. Nous avons dû chercher une solution plus satisfaisante.

Nous avons cherché un élément syntaxique *UML* qui couvre toutes lignes et qui contienne seulement un label. La *continuation* répond à ce besoin. En *UML* standard, elle est typiquement utilisée pour faciliter la représentation de fragments *Alt* complexes, en définissant des labels de branchement. Ce type de facilité n'est pas autorisé dans *TERMOS*, nous sommes donc libres d'en détourner l'usage. La représentation graphique est visuellement la même que pour les *StateInvariant*, avec des contraintes syntaxiques différentes : la *continuation* ne peut être utilisée que dans un opérateur, et doit couvrir toutes les lignes de vies couvertes par cet opérateur. Comme opérateur englobant, nous avons choisi le *Seq*, qui est un opérateur non utilisé dans *TERMOS*. Dans la solution actuelle, les prédicats globaux et changements de configurations sont ainsi représentés par une *continuation* dans un *Seq* qui couvre toutes les lignes de vie. La *continuation* peut aussi être utilisée directement dans un *Assert* pour les prédicats globaux. Lors du passage à cette solution, nous avons également introduit un stéréotype *configChange*, pour mieux différencier le changement de configuration par rapport à un prédicat global. L'événement est maintenant explicitement associé à une $termosConfiguration$ (la nouvelle configuration). Plutôt que de rentrer un label textuel *CHANGE (xx)*, l'utilisateur doit simplement préciser la valeur de l'attribut $configuration$. Ce profil est représenté sur la Figure 5.5. Pour des raisons de rétrocompatibilité, le profil inclut à la fois la solution actuelle et la solution antérieure basée sur les *StateInvariant* avec des labels textuels.

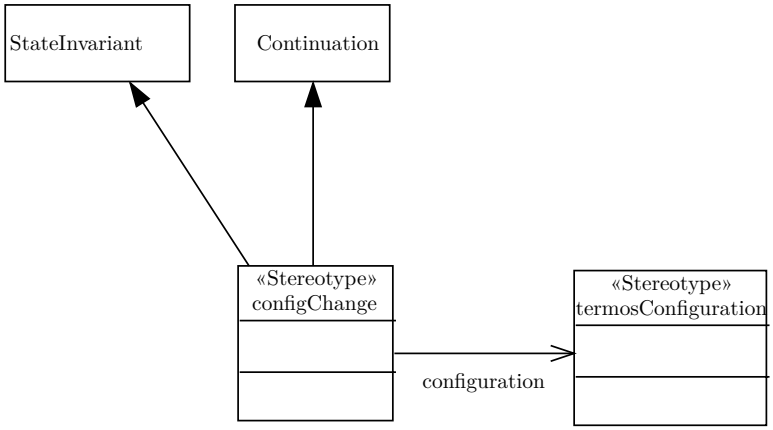


FIGURE 5.5 – Stéréotype d'un changement de configuration

Il est important de noter que dans *TERMOS*, un événement de changement de configuration du scénario doit correspondre à un événement concret dans le système réel. Par exemple, l'attribut d'un nœud change, ou son type de connexion avec un autre nœud change, ou encore un nouveau nœud apparaît. Un "changement" d'une configuration Ci vers cette même configuration Ci ne pourrait être apparié

à aucun événement concret, et reviendrait à définir un scénario systématiquement inconclusif. Pour écarter ce cas trivial, nous vérifions que tous les changements de configuration présents dans le diagramme ont une configuration d'arrivée différente de celle de départ.

5.2.1.2 Message diffusé

La diffusion d'un message implique un événement d'émission suivi par un ou *plusieurs* événements de réception. Ce cas n'est pas prévu dans *UML*, qui met l'accent sur des communications point-à-point. Conformément à la proposition qui avait été faite lors des premiers travaux sur *TERMOS*, nous avons fait le choix de représenter les messages diffusés dans le voisinage d'un nœud à l'aide de messages perdus et trouvés. Rappelons que ces deux types de messages sont des messages classiques pour lesquels soit l'émetteur soit le récepteur n'est pas spécifié. Dans *TERMOS*, ils sont autorisés et permettent de représenter des communications avec l'environnement des nœuds du scénario. Nous avons défini une autre utilisation de ces messages, que l'on distingue de la première par l'application d'un stéréotype *broadcast* montré en Figure 5.6.

Techniquement, au niveau de *UML*, il n'y a pas de type particulier pour les messages perdus et trouvés. Ils ont simplement le type *message*, et selon les cas, ces messages ne sont associés à aucun *sendEvent* ou aucun *receiveEvent*. Graphiquement, une des deux extrémités du message ne touche aucune ligne de vie. Le stéréotype *broadcast* est donc défini génériquement pour des messages, et nous avons dû ajouter une vérification que les messages ayant ce stéréotype sont bien des messages perdus ou trouvés.

Un exemple de broadcast est donné sur la Figure 5.7, où le nœud *n2* diffuse un message à ses voisins *n1*, *n3* et *n4*. La représentation de cette diffusion utilise quatre éléments UML : un message perdu pour l'envoi, et trois messages trouvés pour la réception multiple. L'attribut *id* ajouté par l'application du stéréotype *broadcast* permet de faire le lien entre eux : en attribuant le même *id* à tous ces éléments, l'utilisateur spécifie qu'ils correspondent au même message. Notons que l'identifiant n'est pas visible sur la figure (l'éditeur n'affiche pas cet attribut), mais l'attribut est bien défini dans le diagramme *UML*. Les vérifications associées assurent que l'utilisateur a rentré un *id* pour tout élément ayant le stéréotype *broadcast* et que chaque événement d'envoi comporte un identifiant différent. On vérifie aussi que les éléments ayant le même *id* correspondent à la même instance de message.

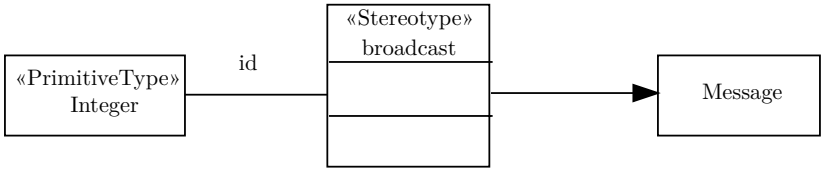


FIGURE 5.6 – Stéréotype appliqué à un message diffusé

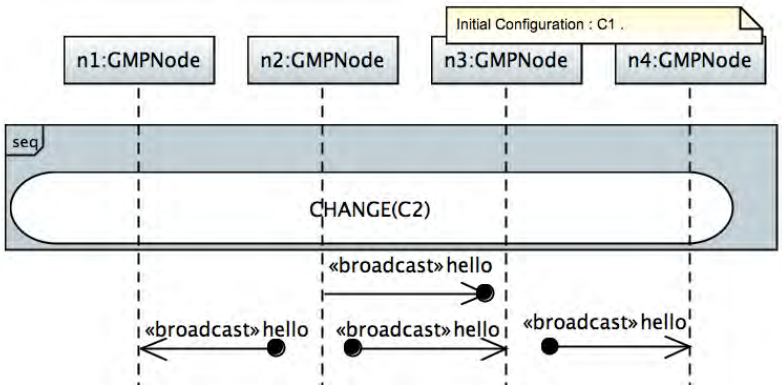


FIGURE 5.7 – Extrait d'un scénario utilisant des messages diffusés

5.2.2 Contraintes syntaxiques sur les opérateurs standard

Pour adapter les diagrammes de séquences à la représentation d'interactions au sein de systèmes mobiles, certains éléments ont été retirés et certaines contraintes ont été ajoutées. Ces modifications sont résumées dans le Tableau 5.1.

Type de changement	**Description du changement**
Suppression	Les éléments suivants ont été supprimés : *Events*, *Gate*, *PartDecomposition*, *GeneralOrdering*, *Continuation* et *ExecutionSpecification*.
Suppression	Les opérateurs suivants ont été supprimés : *Strict*, *Loop*, *Ignore*, *Neg*, *Break* et *Critical*
Contrainte	L'opérateur *Seq* n'est utilisé qu'avec une *Continuation* à l'intérieur et couvre toutes les lignes de vie
Contrainte	Seuls les opérateurs *Alt* et *Opt* peuvent avoir des gardes.
Contrainte	L'opérateur *Alt* est déterministe, de type si-alors-sinon.
Contrainte	Les opérateurs suivants ne peuvent avoir qu'une seule opérande : *Opt*, *Assert* et *Consider*.
Contrainte	Les opérateurs *Consider* et *Assert* doivent couvrir toutes les lignes de vie.
Contrainte	Il doit y avoir un *Assert* à la fin du diagramme.
Contrainte	Si un prédicat global *FALSE* est utilisé, il doit être le seul élément dans le *Assert*.
Contrainte	Les imbrications d'opérateurs de conformité sont limitées voir Tableau 3.2.
Contrainte	Un changement de configuration ne peut être que dans le fragment principal du diagramme ou imbriqué dans un *consider* à condition que celui-ci soit dans le fragment principal du diagramme.

TABLE 5.1 – Résumé des changements apportés aux opérateurs *UML* par le langage *TERMOS*

Ces contraintes n'étant pas intégrées dans le profil *UML TERMOS*, nous avons

codé des vérifications syntaxiques du scénario permettant de vérifier qu'elles sont bien respectées. Parmi ces vérifications, nous en détaillons deux pour lesquelles nous présentons la mise en œuvre de la vérification en java au sein de *Papyrus*. Ces deux exemples nous permettent d'illustrer l'utilisation de l'interface de programmation (*API*) fournie par l'environnement *UML*.

***Assert* à la fin d'un scénario :** Pour être capable de produire un verdict, un scénario doit comporter un fragment combiné de type *Assert*. Deux cas sont possibles. Soit le *Assert* est dans le fragment principal du diagramme, et il en constitue le dernier élément. Soit il est imbriqué dans un fragment *Consider* lui-même de premier niveau, et il constitue le dernier élément du *Consider*. L'Algorithme 5.1 montre comment cette vérification a été mise en œuvre au sein de *Papyrus*. Le paramètre d'entrée encode le diagramme UML, il est de type *Interaction*. La structure est navigable. Une interaction contient des *fragments*, qui peuvent être des événements de communication, des *StateInvariant* ou des *CombinedFragment* (i.e., des opérateurs). Les opérandes d'un *CombinedFragment* sont eux-même des interactions. Dans le code montré, le premier appel de l'opération *getFragments()* retourne la liste des fragments de premier niveau du diagramme. Si cette liste n'est pas vide, on vérifie la présence d'un *Assert* ou d'un *Consider*, ce dernier cas donnant lieu à l'appel d'une autre fonction qui vérifie le contenu du *Consider*.

Algorithme 5.1 Vérification de la présence d'un fragment combiné *assert* à la fin d'un scénario

```
protected boolean checkOneAssertAtTheEnd(Interaction aInteraction) {
  int frags = aInteraction.getFragments().size();
  if(frags<1){
    printViewPart(MessageSeverity.ERROR,0,"Assert","SyntaxError: No assertion
        ↪ found at the bottom of the interaction.");
    return false;
  }
  CombinedFragment cf;
  InteractionFragment IntFrag = aInteraction.getFragments()
      .get(frags - 1);
  if (IntFrag instanceof CombinedFragment) {
    cf = (CombinedFragment) IntFrag;
    if (cf.getInteractionOperator().getValue() == InteractionOperatorKind.
        ↪ ASSERT) {
      printViewPart(MessageSeverity.INFO,0,"Assert","ASSERT operator found at
          ↪ the end of the diagram.");
      return true;
    }
    else if (cf.getInteractionOperator().getValue() ==
        ↪ InteractionOperatorKind.CONSIDER) {
      return checkOneAssertInConsider(cf);
    }
  }
  printViewPart(MessageSeverity.ERROR,0,"Assert","SyntaxError: No assertion
      ↪ found at the bottom of the interaction.");
  return false;
}
```

Restriction de l'utilisation des *alt* : La syntaxe des fragments combinés de type *alt* a été légèrement adaptée au langage *TERMOS* afin de limiter les cas d'indéterminisme. Les fragments *alt* doivent se présenter comme une structure *si-alors-sinon*. Il est donc nécessaire de vérifier qu'une seule garde est spécifiée et que la seconde est vide ou utilise l'opérateur *else*. L'Algorithme 5.2 montre comment cette vérification a été mise en œuvre au sein de *Papyrus*. La fonction montrée est appelée dans le cadre d'un parcours récursif de la structure du diagramme, chaque fois qu'un fragment combiné est rencontré. Dans le code, on peut voir des exemples d'opérations applicables à ce type de fragment, comme *getOperands()*, ou encore des opérations applicables à ses opérandes comme *getGuard()*.

Algorithme 5.2 Vérification de la bonne utilisation des fragments combinés de type *alt*

```
public boolean checkALTCombinedFragment(CombinedFragment cf) {
  int type = cf.getInteractionOperator().getValue();
  if (type == InteractionOperatorKind.ALT) {
    if ( cf.getOperands().size() > 2 ) {
      printViewPart(MessageSeverity.ERROR, 0, "Fragments", "Syntax Error: "
          ↪ More than 2 InteractionOperand declared in an ALT : "+ cf .
          ↪ getLabel());
    } else {
      //Check first InteractionOperand ( If - THEN )
      InteractionOperand firstInteractionOperand = cf.getOperands().get(0);
      if (firstInteractionOperand.getGuard().getSpecification().isComputable
          ↪ ()) {
        printViewPart(MessageSeverity.ERROR, 0, "Fragments", "Syntax Error: "
            ↪ The first InteractionOperand ALT must have an Interaction
            ↪ constraint declared : "+ cf .getLabel());
      }
      if ( cf.getOperands().size() == 2 ) {
        // Check second InteractionOperand ( IF - THEN - ELSE )
        InteractionOperand secondInteractionOperand = cf.getOperands().get(1)
            ↪ ;
        if ( ! secondInteractionOperand.getGuard().getSpecification().
            ↪ isComputable() ) {
          printViewPart(MessageSeverity.ERROR, 0, "Fragments", "Syntax Error:
              ↪ The second InteractionOperand ALT must NOT have an
              ↪ Interaction constraint : "+ cf .getLabel());
        }
      }
    }
  }

  return false;
}
```

Les restrictions syntaxiques ci-dessus visent à faciliter la vérification des traces de test, en évitant par construction les diagrammes ayant une interprétation ambigüe ou indéterministe. Néanmoins, ces restrictions sont insuffisantes pour garantir cet objectif. L'expressivité conférée à *TERMOS* par la prise en compte de prédicats va nécessiter d'autres vérifications pour détecter les cas problématiques, comme nous le verrons par la suite.

5.3 Une grammaire pour écrire des prédicats

L'incorporation de prédicats est nécessaire pour représenter des scénarios intéressants. Le scénario dont la vue spatiale est représentée sur la Figure 5.8 fournit un exemple simple de besoin de prédicats. Dans le cadre du test d'un service d'appartenance de groupe, on souhaite vérifier que le nœud $n1$ est bien présent dans la liste des membres reçue dans le message $GroupChangeMessage$.

Dans les principes généraux de *TERMOS* (chapitre 3) figurait la possibilité d'exprimer des prédicats sur les variables du scénario, qui incluent les attributs symboliques des nœuds et les paramètres symboliques des messages. Cependant, cet aspect du langage n'avait pas été précisé. Nous présentons ci-dessous la gestion des prédicat que nous avons définie et implémentée dans l'outillage *TERMOS*.

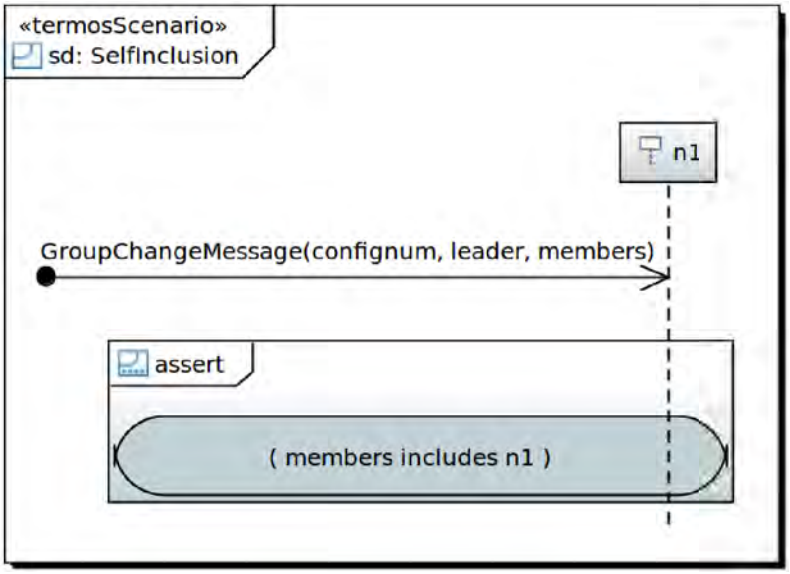


FIGURE 5.8 – Scénario contenant un prédicat

5.3.1 Syntaxe

La syntaxe des prédicats est fondée sur un sous ensemble de la syntaxe du langage *OCL*. Il permet d'effectuer des opérations sur deux types de variables : les variables de type de base comme les entiers, et les variables de type ensembliste comme un ensemble de chaînes de caractères. Les opérations possibles sont réparties en trois catégories en fonction du type de variable. Elles sont listées dans le Tableau 5.2.

Opérations de comparaison numérique : Les opérations permettant de comparer deux valeurs numériques sont =, <>, <, >, <=, >=.

Opérations de comparaison ensembliste : Les opérations ensemblistes sont réparties en deux sous catégories : les comparaisons entre deux ensembles et les comparaison entre un ensemble et une variable. Dans les deux cas, deux opérations sont disponibles, l'*inclusion* et l'*exclusion*.

Opérations logiques : Les opérations logiques sont utilisées pour combiner ensemble des résultats de comparaisons, que ces comparaisons soient numériques ou ensemblistes. Les opérateurs à disposition de l'utilisateur sont *and*, *or*, *xor*, *not*.

Le choix des opérateurs pourrait bien sûr être étendu, de même que les types des variables pris en compte.

5.3.2 Mise en œuvre

La gestion des expressions a été mise en œuvre grâce au langage *ANTLR* [Parr 2013], ce langage permet d'écrire une grammaire et de générer un compilateur permettant l'analyser une expression respectant cette grammaire. L'ensemble des opérateurs présenté dans le Tableau 5.2 a été intégré dans cette grammaire dont le contenu est décrit dans l'Algorithme 5.3.

ANTLR crée un compilateur à partir de la grammaire. Ce compilateur est capable d'interpréter une expression et de l'évaluer. Le compilateur ainsi généré permet de découper une expression en un arbre syntaxique et de parcourir celui-ci de façon hiérarchique. Chaque morceau de l'expression est associé à une fonction correspondant à la catégorie d'opération décrite dans la grammaire. Chaque opération se résume à évaluer une expression composée d'un ou deux opérandes et d'un opérateur puis à renvoyer un booléen correspondant au résultat de l'opération.

5.3.3 Vérification

Les vérifications relatives aux prédicats se déroulent en trois étapes.

Une première étape s'effectue avant la génération de l'automate qui encode les ordres partiels du scénario. Elle consiste simplement à vérifier que le prédicat respecte la grammaire *ANTLR*.

La deuxième s'effectue lors de la génération de l'automate. Elle vérifie que le prédicat est évaluable, c'est-à-dire que toutes les variables apparaissant dans le prédicat ont reçu une valeur au moment de l'utilisation du prédicat. La mise à jour des valeurs dépend des chemins suivis dans l'automate. Par exemple, la valeur d'un paramètre de message n'est connue que si une transition précédente a consommé un événement d'émission ou de réception de ce message. L'algorithme de construction de l'automate gère un historique des variables connues dans chaque état, ce qui permet de vérifier à la volée qu'un prédicat sur une transition sortante est évaluable.

La troisième étape s'effectue lors de l'utilisation de l'automate pour vérifier une trace d'exécution. Elle vise à détecter les cas ambigüs qui ne sont pas couverts par les restrictions syntaxiques et les vérifications précédentes. La Figure 5.9 montre un exemple. Le scénario respecte toutes les règles syntaxiques. Dans l'état atteint

Algorithme 5.3 Grammaire du langage de prédicats

```
/*
 * Grammaire des expressions dans TERMOS
 */
grammar TERMOS;

NAME : [a-zA-Z_]+('a'..'z' | '0'..'9' | '.' | ':' | ',' | '->')*;
DECIMAL : '-'?[0-9]+('.'[0-9]+)? ;

/*** Gramatical Rules ***/

prog: LPAREN? log_expr RPAREN?;

/* Logical Expression */
log_expr
        : log_expr ('and'|'or'|'xor') log_expr  # LogicalExpression
        | NOT log_expr                          # LogicalExpressionNot
        | rel_expr                              # ComparisonExpression
        | ens_expr                              # EnsemblistExpression
        | '(' log_expr ')'                      # LogicalExpressionInParen
        | log_entity                            # LogicalEntity
        ;

/* Relative Expression */
rel_expr
        : numeric_entity ('='|'<>'|'<'|'>'|'<='|'=>') numeric_entity
        ;

/* Ensemblist Expression */
ens_expr
        : varList ('includes'|'excludes') var
        | varList ('includesAll'|'excludesAll') varList
        ;

/* Logical Entity (variable or boolean) */
log_entity
        : (TRUE | FALSE)
        ;

/* Numeric Entity (variable or integer) */
numeric_entity
        : DECIMAL
        | var
        ;

/* Variable */
varList
        : '[' NAME ']'
        ;

/* Variable */
var
        : NAME
        | '(' NAME ')'
        ;
```

Opérateur	Description
Comparaisons numériques	
int = int	Opérateur d'égalité
int <> int	Opérateur d'inégalité
int < int	Opérateur d'infériorité
int > int	Opérateur de supériorité
int <= int	Opérateur d'infériorité ou d'égalité
int >= int	Opérateur de supériorité ou d'égalité
Comparaisons ensemblistes	
set **includesAll** set	Opérateur d'inclusion : est-ce que l'ensemble des éléments contenu dans une première variable sont présents dans une seconde variable ?
set **excludesAll** set	Opérateur d'exclusion : est-ce qu'aucun des éléments contenu dans une première variable n'est présent dans une seconde variable ?
set **includes** var	Opérateur d'inclusion : est-ce qu'une variable est contenue dans l'ensemble des éléments d'une seconde variable ?
set **excludes** var	Opérateur d'exclusion : est-ce qu'une variable n'est pas contenue dans l'ensemble des éléments d'une seconde variable ?
Opérations logiques	
bool **and** bool	Opérateur et logique
bool **or** bool	Opérateur ou logique
bool **xor** bool	Opérateur ou exclusif logique
not bool	Opérateur de négation logique (ne s'applique qu'à une seule variable)

TABLE 5.2 – Opérateurs mis en œuvre dans le langage de prédicats

à la sortie du fragment *Par*, les deux variables ont reçu une valeur, qui sont donc disponibles pour l'évaluation du prédicat. Pourtant, le verdict final est ambigü : le comportement spécifié par le *Par* est en fait indéterministe, selon que l'on décide d'apparier tel ou tel message avec tel ou tel opérande du fragment. Les vérifications lors de l'exécution de l'automate prennent en compte ces cas. A chaque pas d'exécution, nous calculons l'ensemble des transitions franchissables, et s'il y a plus d'une possibilité de transition l'analyse de la trace est stoppée. L'utilisateur est informé que l'analyse est non concluante du fait de la détection d'un cas de non déterminisme. Cette ultime étape de vérification sera décrite dans le chapitre 6.

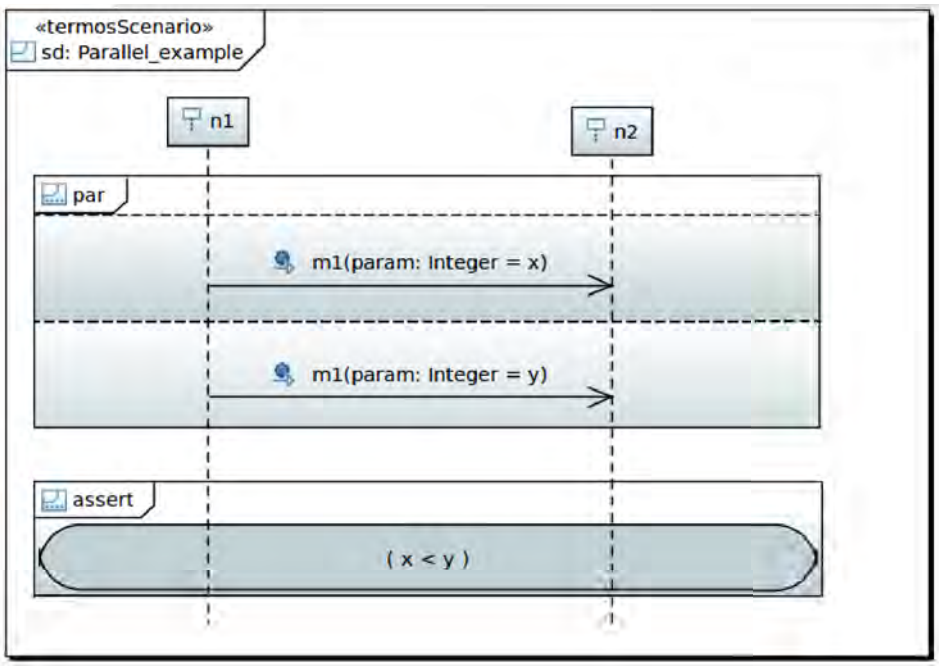


FIGURE 5.9 – Exemple de scénario ambigü

5.4 Transformation du scénario

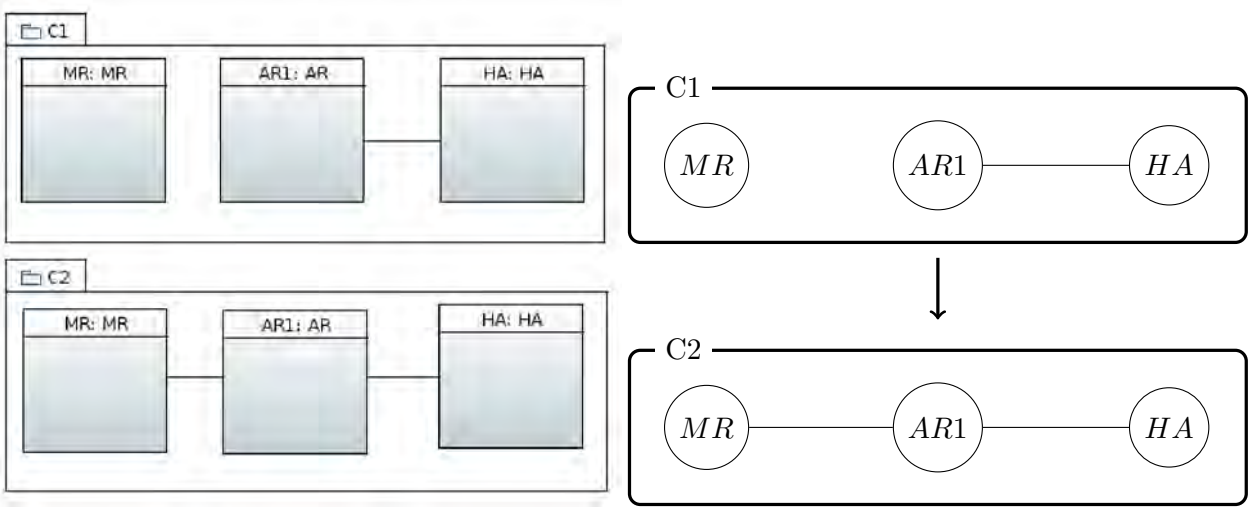
La vérification d'un scénario sur une trace d'exécution ne se fait pas directement sur les diagrammes *UML* mais sur deux éléments générés à partir du scénario. Il s'agit d'une séquence de configurations spatiales représentée par une séquence de graphes et d'un automate représentant l'ensemble des événements pouvant se dérouler dans le scénario. Une étape de transformation est nécessaire pour traduire un scénario *TERMOS* vers ces éléments nécessaires à la vérification du scénario sur des traces d'exécutions. En particulier, la construction de l'automate est réalisée à l'aide de l'algorithme de déroulement du scénario décrit en annexe, dans la section A.2.

La construction de l'automate ne sera pas lancée, ou sera interrompue, si le scénario est mal formé. Nous terminons par un récapitulatif des vérifications effectuées, et présentons les rapports retournés à l'utilisateur en cas d'échec ou de succès de la transformation du scénario.

5.4.1 Création de la séquence de configurations spatiales

GraphSeq utilise en entrée une séquence de graphes appelée séquence motif et recherche dans une séquence concrète si elle se produit. Pour être capable de générer cette séquence motif, il faut connaître l'ensemble des configurations spatiales du scénario ainsi que l'ordre dans lequel elles doivent se produire. Grâce à l'utilisation du profil *UML TERMOS*, il est aisé de parcourir un modèle *UML* pour en extraire les éléments suivants nécessaires à la construction d'une séquence motif :

— Les configurations spatiales ont le stéréotype *termosConfiguration*,



(a) Représentation *TERMOS* (*UML*) (b) Séquence de graphes

FIGURE 5.10 – Transformation d'une vue spatiale en séquence de graphe

— Les nœuds ont le stéréotype *termosNode*,

— Les liens entre les nœuds ont le stéréotype *termosLink*,

— Le diagramme de séquence représentant la vue événementielle a le stéréotype *termosScenario*, il contient un attribut identifiant la configuration initiale du scénario. Les autres changements de configurations sont extraits des événements *CHANGE(xx)* présents dans la vue événementielle.

A l'aide de l'ensemble de ces informations, il est possible de créer pour chaque scénario, la séquence de configurations spatiales que l'outil *GraphSeq* devra identifier. La Figure 5.10 illustre cette transformation avec à gauche, la représentation en *UML* et à droite, une représentation graphique de la séquence de graphe qui sera utilisée par *GraphSeq*. Dans cette partie droite, les noms des instances de *termosNode* sont ajoutés comme premier attribut des nœuds des graphes. Du point de vue de *GraphSeq*, MR, $AR1$, HA seront des identifiants symboliques, qui seront appariés à des identifiants concrets de nœuds du système testé.

5.4.2 Création de l'automate

La sémantique de la vue événementielle est donnée par la construction d'un automate symbolique, dont les variables dépendent des configurations spatiales et des paramètres de messages. Les principes généraux de l'algorithme de construction ont été présentés dans le chapitre 3. On trouvera une présentation plus détaillée en annexe de ce mémoire et dans [Micskei 2013], avec les deux étapes de pré-traitement et de déroulement. Dans ce qui suit, nous mettons l'accent sur deux points : l'utilisation de l'*API UML* pour implémenter l'algorithme, et les traitements spécifiques à la gestion des variables et des prédicats. Nous nous appuyons sur l'exemple de la Figure 5.11 pour illustrer ces derniers aspects : le scénario comporte un message avec

un paramètre symbolique x, et un prédicat comparant ce paramètre symbolique à un attribut symbolique de nœud $n2\_y$.

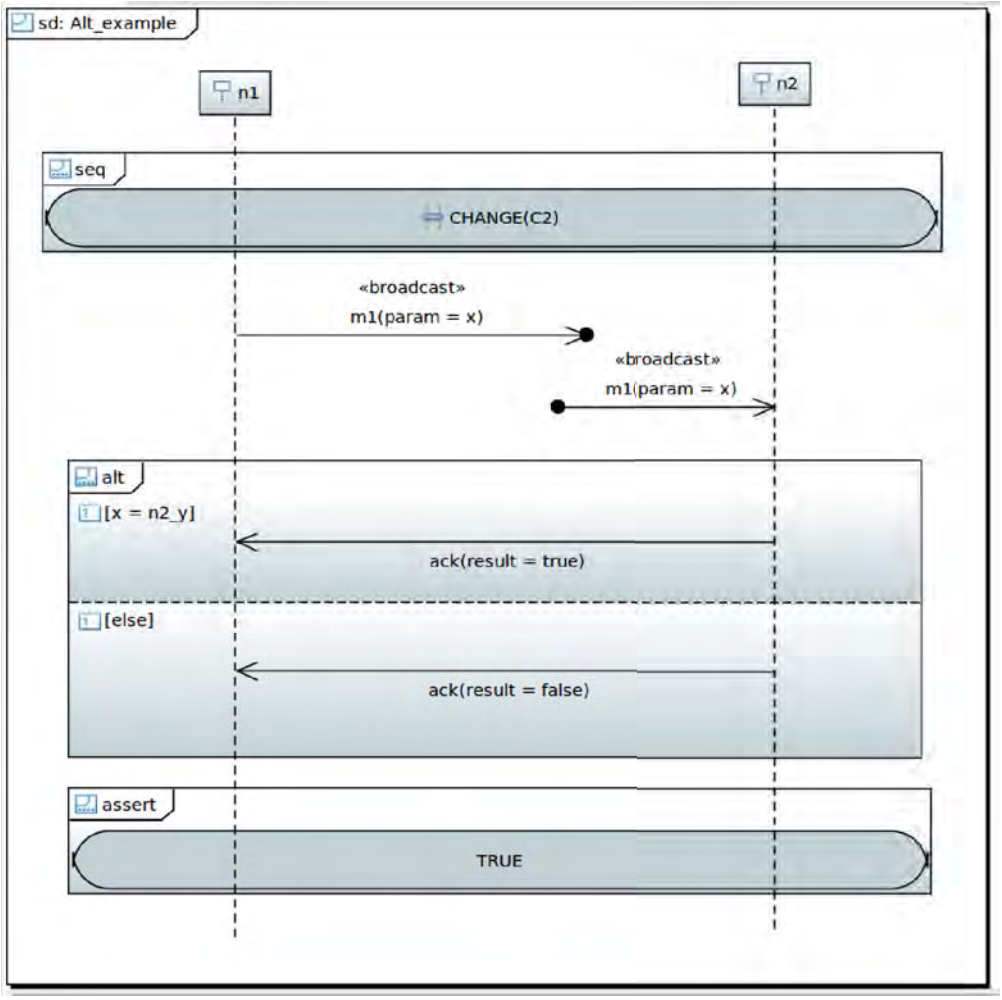


FIGURE 5.11 – Exemple de scénario utilisant un *alt*

5.4.2.1 Prétraitement

Le prétraitement consiste à extraire les *atomes* apparaissant sur les lignes de vie, à les regrouper en classes et à calculer les relations de précédence et de conflit entre les classes. Les atomes sur la ligne de vie $n1$ de la figure sont : le début de la ligne de vie, le changement de configuration, l'émission de $m1$, l'entrée dans le *Alt*, etc. Le calcul des relations entre atomes s'appuie sur un étiquetage des atomes avec un label de *position*, qui tient compte à la fois de sa position physique sur la ligne de vie et de son niveau d'imbrication dans les opérateurs du diagramme. Cette position s'écrit sous la forme d'un chemin. La position $4.Alt(2).0$ est attribuée au premier (0) atome à l'intérieur du deuxième opérande du *Alt*. L'entrée dans le *Alt* avait la position 3, donc tout ce qui est à l'intérieur du *Alt* a une position préfixée

par 4. La sortie du *Alt* aura la position 5.

L'extraction des atomes et de leur position s'effectue par un parcours récursif de la structure de donnée *Interaction* correspondant au diagramme. Le parcours utilise les fonctions de l'*API* telles que :

— *getFragments()*, qui permet de récupérer une liste ordonnée des fragments contenus dans une interaction ;

— *getInteractionOperator()*, qui détermine l'opérateur d'un fragment de type *CombinedFragment* ;

— *getOperands()*, qui permet de récupérer la liste des opérandes d'un fragment de type *CombinedFragment* ; ces opérandes sont des interactions elles-mêmes susceptibles de contenir des fragments simples ou combinés.

L'implémentation de l'extraction tient compte de la spécificité de *TERMOS*. Par exemple, le parcours du *Seq* et de sa *continuation* imbriquée ne donnera qu'un seul atome par ligne de vie, correspondant au changement de configuration sur cette ligne. L'entrée et la sortie d'un opérateur ne sont pas des éléments *UML*, mais nous créons les atomes correspondants chaque fois que le parcours rencontre un fragment combiné. La garde d'un opérande *Alt* n'appartient à aucune ligne de vie, c'est un attribut de l'opérande, et nous l'ajoutons à chaque ligne de vie couverte par le *Alt*, comme premier atome dans cet opérande.

En plus de leur position, les atomes que nous créons possèdent d'autres attributs, et notamment un pointeur sur l'élément *UML* qui a justifié leur création. Ce lien avec les éléments du diagramme est ensuite exploité pour identifier les classes d'atomes et calculer leurs relation. Ainsi, on pourra déterminer que les atomes d'entrée dans le *Alt* des deux lignes de vie peuvent être regroupés dans la même classe, car ils pointent sur le même fragment *UML*. De même, le calcul des conflits pourra déterminer que deux atomes figurent dans des opérandes différents du même *Alt*. Le calcul des relations de précédence requiert de pouvoir apparier les atomes d'émission et de réception d'un même message. Pour un message *UML* standard, l'appariement se fait facilement, en appelant la fonction $getMessage()$ associée à la structure pointée. Pour un message de stéréotype *Broadcast*, il faut non seulement récupérer le message impliqué, mais aussi consulter son attribut d'identifiant rentré par l'utilisateur.

A l'issue du prétraitement, toutes les relations ont été calculées, et le déroulement des classes d'atomes peut commencer.

5.4.2.2 Déroulement des classes d'atomes

Le déroulement des classes d'atomes consiste à construire un automate à partir des classes d'atomes calculées lors de la phase de prétraitement. L'état initial de l'automate correspond à l'ensemble des classes qui n'ont pas de relations de précédence avec les autres classes, il s'agit des sommets des lignes de vie. On suppose que le système est dans la configuration spatiale initiale. Ensuite, en partant de cet état initial, l'algorithme recherche les classes d'atomes qui sont prêtes à être

déroulées, c'est-à-dire que tous leurs prédécesseurs ont été traités. Chaque classe d'atomes prête donne lieu à la création d'une transition sortante de l'état courant. En partant de l'état initial, l'algorithme calcule les états qui le succèdent jusqu'à ce que l'ensemble des classes d'atomes ait été traité.

Les labels des transitions entre deux états de l'automate contiennent typiquement une condition décrivant des événements de communication ou des changements de configuration. Le label de la transition peut avoir en plus une condition de garde ou une contrainte. S'il existe de nouvelles variables évaluées, alors le label de la transition contient une mention explicite de mise à jour. Si la transition contient une garde ou une contrainte, une vérification de l'accessibilité des variables nécessaires à son évaluation est effectuée.

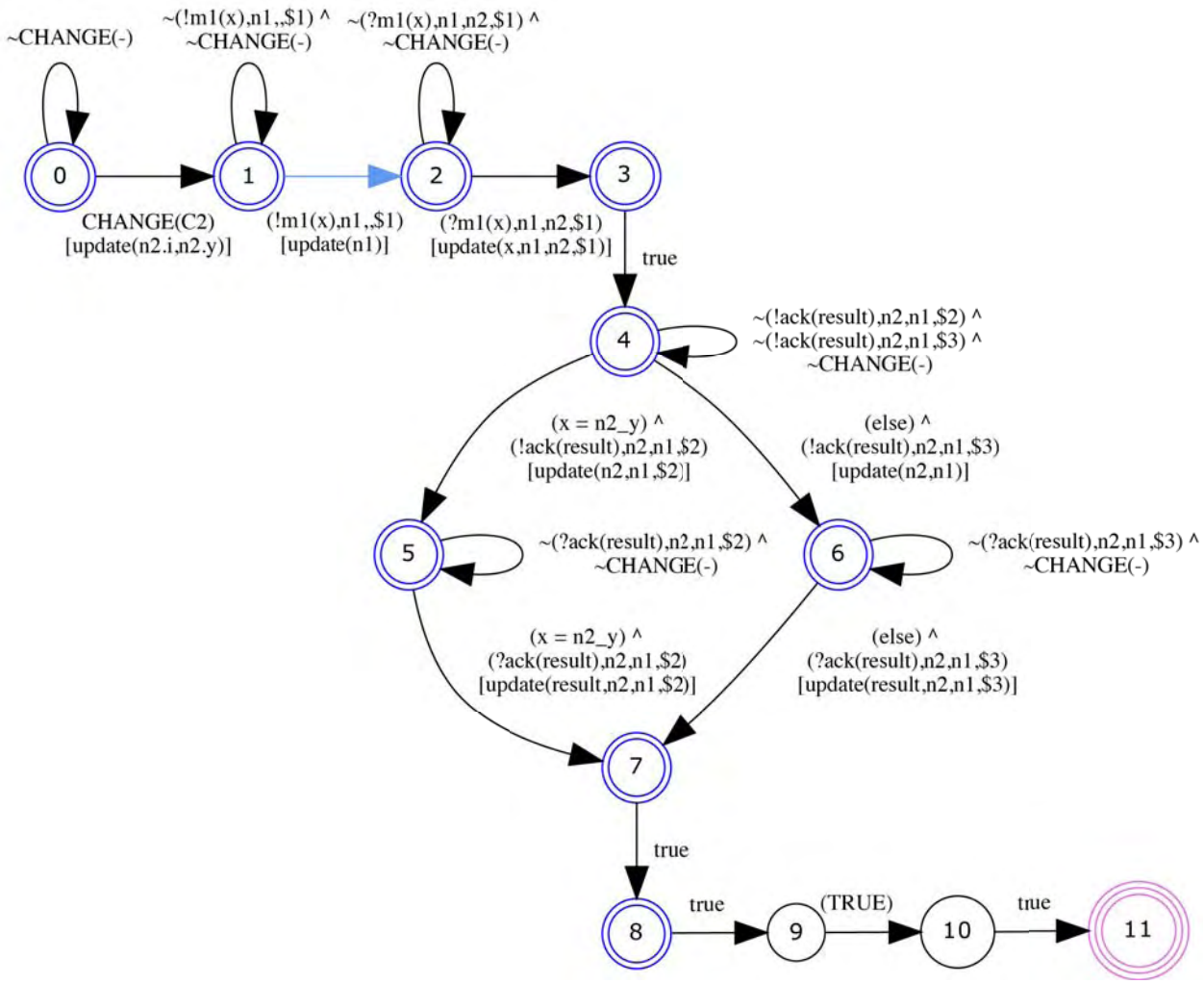


FIGURE 5.12 – Automate généré à partir du scénario de la Figure 5.11

Des rebouclages sont ajoutés sur les états à partir du moment où il existe une transition sortant de l'état contenant un événement provenant de la trace, que ce soit un événement de communication ou un changement de configuration. Ce rebouclage

permet aux états qui en possèdent d'être capables lors de la validation de trace de consommer des événements non représentés tout en restant dans l'état courant en attendant que les événements recherchés se produisent.

L'automate symbolique, formé par l'ensemble des états, des transitions et des variables générés par le déroulement des classes d'atomes, est finalement exporté vers le format de fichier *XML* défini au chapitre 4 afin d'être utilisé lors de la phase de vérification de trace. Ce format est une traduction directe de la structure de données de l'automate symbolique. Une version graphique allégée pour rester lisible est aussi générée.

Les étapes de prétraitement et de déroulement appliquées au scénario présenté dans la Figure 5.11 produisent l'automate présenté dans la Figure 5.12. Sur cette figure, nous voyons clairement apparaître les états correspondant au fragment *alt* avec les deux branches parallèles.

5.4.3 Vérification de la forme du scénario

Avant cette étape de transformation, il est nécessaire de vérifier que le scénario soit bien formé. Un ensemble de règles a été défini afin de garantir que les diagrammes *UML* d'un scénario *TERMOS* représentent un scénario valide pour lequel nous pourrons construire un automate. Pour chacune des règles, nous avons mis en œuvre des vérifications permettant de contrôler leurs applications sur les scénarios.

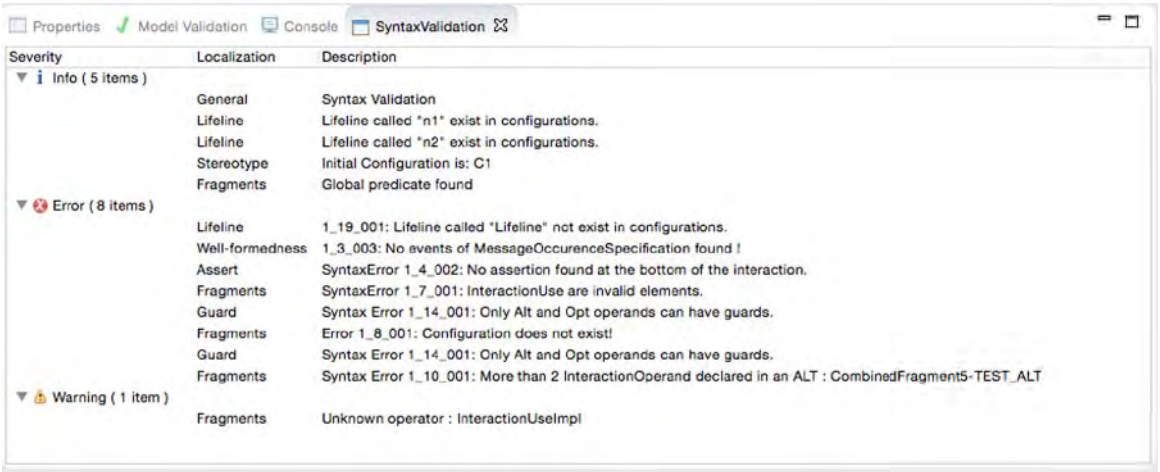


FIGURE 5.13 – Panneau de gestion des vérifications syntaxiques d'un scénario

Ces vérifications ont été présentées tout au long de ce chapitre et sont résumées au sein du Tableau 5.3. Avant la génération d'un automate, les vérifications réalisables sur le scénario sont exécutées et l'ensemble des résultats est présenté à l'utilisateur. Ces vérifications peuvent retourner un avertissement ou une erreur en fonction de la gravité du problème détecté et si il est bloquant ou non pour la suite des vérifications. Afin d'aider l'utilisateur à localiser et corriger les problèmes présents dans un scénario, une vue synthétique de l'ensemble des résultats est créée. Un

Vérification	Description
Vérifications avant la génération de l'automate	
Contraintes sur les opérateurs *UML*	Ensemble des règles indiquées dans le Tableau 5.1
Cohérence avec le diagramme de classes	Vérification que les éléments utilisés dans le scénario (type de nœuds, attributs, opérations...) sont conformes a leur déclaration dans le diagramme de classes.
Cohérence des vues	Vérifie que les lignes de vies présentes dans la vue événementielle correspondent bien à un nœud présent dans au moins une configuration spatiale utilisée dans le scénario.
Configuration initiale	Le diagramme de séquence a le stéréotype *TermosScenario* et la configuration initiale est bien sélectionnée.
Messages	Lors d'un échange de message, vérifie que la topologie courante permet la communication entre l'émetteur et le récepteur.
Format des opérateurs *alt*	Vérification que les gardes de l'opérateur *alt* correspondent à une structure *si-sinon*.
Changements de configurations	Vérification que deux changements de configurations successifs n'utilisent pas la même configuration spatiale.
Messages diffusés	Contrôle l'utilisation des messages diffusés, stéréotype *broadcast* appliqué aux messages et contrôle des identifiants de messages.
Gardes et prédicats conformes	Vérifie que les expressions des gardes et des prédicats respectent la grammaire.
Vérification lors de la génération de l'automate	
Variables accessibles	Lorsqu'une garde ou un prédicat est rencontré, vérifier que les variables nécessaires à son évaluation soient bien accessibles à cet état de l'automate.
Vérification en ligne lors de l'exécution de l'automate	
Déterminisme	L'utilisation des prédicats peut entraîner l'automate dans des cas de non déterminisme. Si deux transitions peuvent être franchies simultanément, on arrête l'exécution de l'automate.

Table 5.3 – Résumé des vérifications faites à un scénario

exemple de cette vue est donné dans la Figure 5.13 où chaque erreur est matérialisée par une ligne avec le niveau de l'erreur précisé. Un message détaille la gravité de celle-ci ainsi que sa localisation dans le diagramme. Une partie des vérifications ne peut être exécutée que pendant la génération de l'automate pour la vérification des

variables accessibles. Une autre partie sera effectuée pendant l'exécution de l'automate pour la vérification consistant à détecter si des cas de non déterminisme se produisent.

5.4.4 Rapport de génération

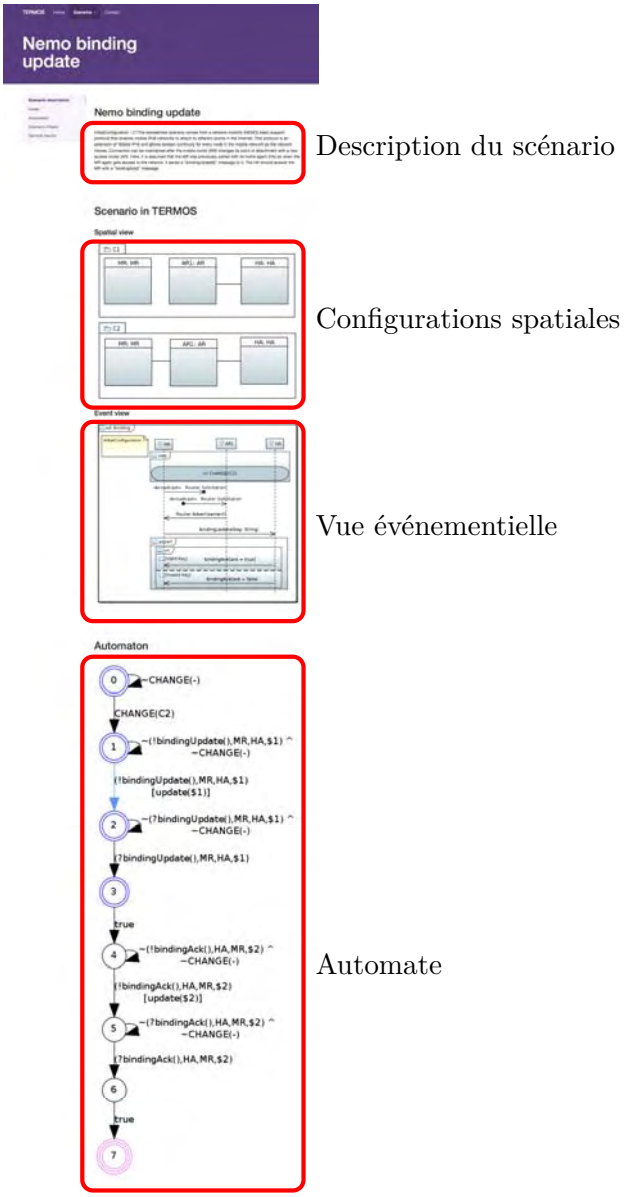


FIGURE 5.14 – Exemple de rapport de génération d'un scénario

En plus des formats nécessaires à la vérification d'un scénario, un ensemble de fichiers peut être exporté depuis l'application *TERMOS*. Pour chaque scénario une page web contenant l'ensemble des vues d'un scénario, son automate ainsi qu'un texte de description sont générés. Un exemple de rapport de génération est présenté

dans la Figure 5.14. L'ensemble du contenu de la page provient des informations saisies par l'utilisateur dans *TERMOS* ou bien des fichiers générés par *TERMOS*. Grâce à cela, il est possible de créer un site web contenant l'ensemble des scénarios de test d'une étude de cas. Cette représentation sous forme de site web permet de communiquer avec des utilisateurs sans qu'ils aient besoin d'installer l'ensemble de la plate-forme *UML* pour consulter les scénarios. Il sera possible lors de la phase de vérification des scénarios sur des traces de compléter chaque page de scénario avec les résultats de vérification.

5.5 Conclusion

Dans ce chapitre, nous avons présenté l'ensemble des étapes permettant la spécification d'un scénario *TERMOS*. La première consiste à décrire le comportement d'un système à l'aide de scénarios *TERMOS* composés d'une vue spatiale et d'une vue événementielle. Cette étape a conduit à la mise en œuvre d'un profil *UML* afin de spécialiser l'éditeur *Papyrus* pour la saisie de scénarios *TERMOS*. Afin d'étendre les possibilités du langage *TERMOS*, une grammaire a été créée pour être capable de gérer des prédicats dans les scénarios.

La deuxième étape consiste à contrôler la bonne forme du scénario. Nous avons pour cela mis en œuvre une série de vérifications pour s'assurer que les règles de construction des scénarios sont respectées. Les résultats de ces vérifications sont ensuite mis en forme et présentés à l'utilisateur directement dans la plate-forme d'édition *UML* pour permettre une correction des scénarios. Pour faciliter les échanges des scénarios avec des personnes non équipées d'un environnement *UML*, nous avons aussi créé une fonctionnalité d'exportation des scénarios dans un format portable (pages *HTML*) du scénario permettant sa lecture sur tout support.

Enfin, après l'étape de vérification syntaxique, le scénario est transformé pour obtenir une représentation de celui-ci sous forme d'automate pouvant être vérifié automatiquement sur des traces d'exécution.

Chapitre 6

Analyser un scénario

Sommaire

Comme nous l'avons vu, un scénario pour le test d'applications mobiles comporte deux vues : une spatiale et une événementielle. L'analyse de celui-ci s'effectue en plusieurs étapes. La première consiste à rechercher les nœuds concrets présents dans une trace d'exécution pouvant correspondre aux nœuds symboliques présents dans un scénario. Cette première étape est réalisée à l'aide de l'outil *GraphSeq* qui permet de trouver l'ensemble des occurrences présentes dans une trace d'exécution où une séquence de configurations spatiales recherchée se manifeste. Le fonctionnement de *GraphSeq* ainsi que les améliorations que nous lui avons apportées sont détaillés dans la section 6.1.

À partir de ces occurrences, un filtrage de la trace est effectué pour chacune d'entre elles afin de produire une sous-trace qui ne contiendra que les nœuds identifiés ainsi que les événements se produisant entre les bornes temporelles de l'occurrence.

La section 6.2 précise comment est exécuté l'automate représentant le scénario sur l'ensemble des sous-traces. À chaque état de l'automate est associé un verdict, ce qui permet à la fin de la sous-trace de déterminer quel est le verdict de l'exécution.

La dernière étape présentée en section 6.3 consiste à compiler l'ensemble des résultats des exécutions de l'automate et de permettre une analyse graphique de ceux-ci.

6.1 GraphSeq pour l'analyse de la vue spatiale

La description de nos scénarios comporte une vue spatiale qui est représentée à l'aide d'une séquence de graphes dans le langage *TERMOS*. Son traitement formel fait appel à des algorithmes d'appariements de graphes. L'outil *GraphSeq* (*Graph matching tool for Sequences of configurations*) [Nguyen 2009] met en œuvre ces algorithmes afin d'obtenir tous les nœuds d'une trace qui correspondent à la séquence de graphes. *GraphSeq* utilise comme entrée deux séquences de graphes avec des nœuds symboliques :

— Une séquence $C_1, \ldots, C_m$ de m graphes provenant de la description du scénario, appelée *motif de graphe*,

— Une séquence $CC_1, \ldots, CC_n$ de n graphes extraits de la trace d'exécution, appelée *graphes de configuration concrète*.

GraphSeq compare les deux séquences et retourne toutes les occurrences de la séquence motif. Les graphes contenus dans les séquences ont un identifiant permettant de les différencier et contiennent des nœuds qui ont un identifiant unique, auxquels peuvent être associées des étiquettes. Les étiquettes dans un motif peuvent être une constante, une variable ou bien un joker. Dans le cas des configurations concrètes par contre, vu qu'il s'agit d'une trace d'exécution, les étiquettes ne peuvent être que des constantes. Ces étiquettes sont prises en compte par l'algorithme d'appariement de *GraphSeq*.

Pour calculer les occurrences, la comparaison entre les séquences concrètes et les séquences de motifs est effectuée en deux étapes. La première compare les paires de graphes C_i et CC_j pour déterminer toutes les instances de C_i apparaissant comme un sous-graphe de CC_j. Techniquement, la comparaison recherche des isomorphismes de sous-graphe. Ce problème est déjà bien étudié dans la litérature [Ullmann 1976] [Messmer 2000] et nous utilisons un outil existant prenant en compte les variables étiquetées appelé *Graph Matching and Transformation Engine* (*GMTE*) [Guennoun 2006]. Ensuite, une seconde étape s'appuie sur les paires identifiées à l'étape précédente afin de déterminer toutes les instances de la séquence de motifs.

Dans cette section, nous présentons le principe de fonctionnement de *GraphSeq* puis l'ensemble des évolutions que nous avons mis en place afin de rendre plus performant cet outil notamment pour son passage à l'échelle.

6.1.1 Principes de fonctionnement

Chaque nœud du graphe est conceptuellement composé de tuples d'étiquettes $< l_1, \ldots, l_k >$, où l_1 est l'identifiant symbolique du nœud et $l_2, \ldots, l_k$ sont ses attributs contextuels optionnels. L'implémentation actuelle de *TERMOS* considère seulement deux attributs optionnels, donc ici $k = 3$, ce qui fait que nous manipulerons des triplets. La syntaxe d'un motif de graphe est donnée en exemple dans la Figure 6.1.

Considérons les triplets $< x, v1, * >$, $< y, 1, 2 >$ et $< z, v1, * >$ de la Figure 6.1. Les identifiants symboliques x, y et z sont des étiquettes de nœuds, chacun pouvant

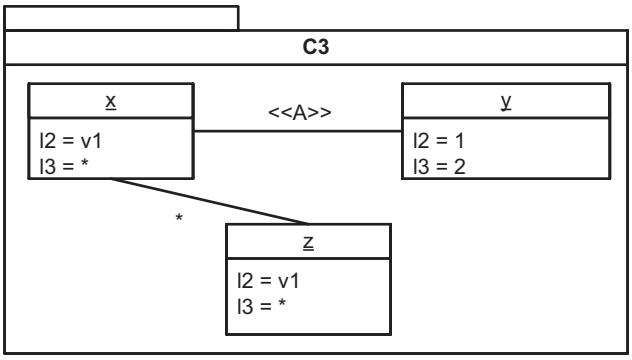


FIGURE 6.1 – Une configuration TERMOS avec différentes formes d'étiquettes

être apparié à un identifiant de nœud concret. Par exemple, si nous souhaitons analyser des traces où les nœuds sont identifiés par des adresses réseau IP, l'identifiant x peut être représenté par l'adresse IP d'un nœud. L'étiquette $v1$ est une variable qui indique que les nœuds concrets jouant le rôle de x et z ont leur premier attribut à une valeur non spécifiée $v1$ mais identique, qui sera déterminée par l'instance spécifique trouvée. Le deuxième attribut des nœuds x et z est un *joker* ($*$), c'est-à-dire que la valeur peut être quelconque ou ignorée. Le nœud y quand à lui ne contient que des attributs constants.

De la même façon que pour les nœuds, les arcs liant les nœuds entre eux peuvent être étiquetés pour indiquer un type de connexion, comme sur l'exemple de la Figure 6.1 où un arc est étiqueté $<< A >>$. Il est aussi possible d'utiliser une valeur *joker* ($*$) qui comme pour les nœuds, indique des valeurs que nous pouvons ignorer. Par exemple, nous ignorons le type de connexion entre x et z, mais il est nécessaire que x soit connecté à z et que y soit déconnecté de z.

Une occurrence du motif est représentée par une structure de données composée de deux champs :

— une table d'*index* qui représente la fenêtre temporelle $[Date\_de\_début, Date\_de\_fin]$ de l'occurrence pour chaque graphe du motif.

— une table *val* qui affecte une valeur concrète à chaque variable du motif.

Les Figures 6.2 et 6.3 donnent un aperçu visuel d'une occurrence. Une instance de C_1 apparaît comme un sous-graphe de la configuration concrète CC_2. Ce sous-graphe n'est pas affecté par le changement de configuration concrète du système à CC_3. Le changement suivant affecte cette partie du système et conduit à apparier CC_4 à une instance de C_2, puis CC_5 à une instance de C_3. Cette instance de C_3 persiste jusqu'au changement de configuration concrète du système pour CC_8 qui termine l'occurrence. L'encodage des dates est représenté Figure 6.3 dans le tableau *index*. Par exemple, si la vue événementielle du scénario montre un événement de communication *msg* pendant la configuration spatiale C_3, alors l'outil recherche le message *msg* dans la sous-trace du système comprise entre la date du changement $CC_4 \rightarrow CC_5$ et celle du changement $CC_7 \rightarrow CC_8$. De plus, l'évaluation des identifiants de nœuds symboliques indique quels nœuds concrets sont émetteur et lesquels

sont récepteur du message msg.

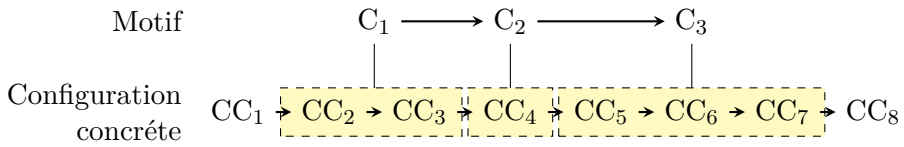


FIGURE 6.2 – Exemple d'une occurrence d'une séquence de graphe

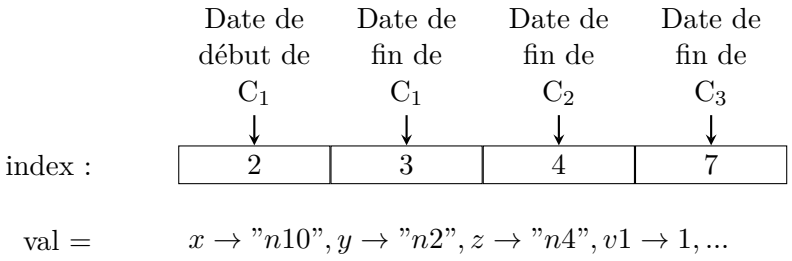


FIGURE 6.3 – Représentation de l'occurrence donnée en exemple Figure 6.2

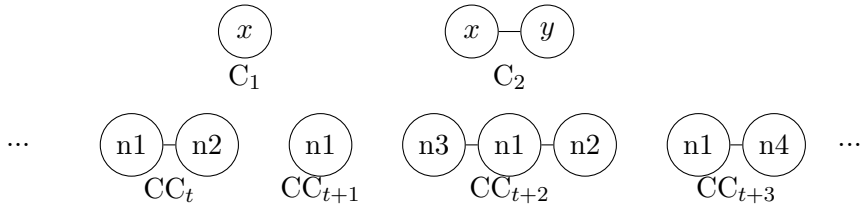
En général, plusieurs instances du motif C_i peuvent apparaître dans la configuration concrète, ce qui induit beaucoup d'occurrences potentielles. Par exemple, supposons que CC_2 contienne plusieurs instances du premier motif C_1. Chacun d'entre eux à leur tour offrent plusieurs possibilités pour les instances ultérieures de C_2, avec différentes dates de transition et différentes valeurs pour les nouvelles variables. Une variable valuée dans une configuration de la séquence garde la même valeur tout au long de la séquence, par exemple l'identifiant du nœud concret correspondant à x ne peut pas changer durant le scénario. La clef est d'explorer toutes les possibilités en gardant une valuation cohérente des variables tout au long du scénario, et d'identifier correctement les transitions entre les différents graphes composant le motif.

Une présentation détaillée de l'algorithme de *GraphSeq* peut être trouvée à la section 3.3 du rapport technique D5.3 du projet HIDENETS [Huszrel 2008]. Une vue de plus haut niveau est donnée dans [Nguyen 2010] ainsi que la formalisation des propriétés attendues d'une occurrence. Ces propriétés sont résumées ci-dessous.

Supposons que M soit un match retourné par *GraphSeq*, et que $[s_i, e_i]$ soit la fenêtre temporelle pour chaque motif P_i comme indiqué dans $M.index$. De façon évidente, si le match est correct, chaque configuration concrète $C_{si}, ..., C_{ei}$ doit contenir l'instance de P_i déterminée par la valuation $M.val$.

Propriété 1 Les occurrences du motif doivent exister dans les configurations concrètes.
$\forall i \in [0, m-1], \forall k \in [s_i, e_i]$, il existe une fonction f telle que : $MatchGraph(ValuateGraph(P_i, M.val), C_k) = (f, \emptyset)$

Par exemple, dans la Figure 6.3, l'instance de C_1 déterminée grâce à la valuation val doit apparaître comme un sous-graphe aussi bien de CC_2 que de CC_3, l'instance de C_2 doit apparaître comme un sous-graphe de CC_4, etc.



(a) Séquence de graphes avec introduction de nouveaux nœuds

Occurence n°1 incorrecte par rapport à la propriété 2	index :	t	$t+1$	$t+2$
	val =	$x \to$ "$n1$", $y \to$ "$n2$"		
Occurence n°2 correcte	index :	$t+1$	$t+1$	$t+2$
	val =	$x \to$ "$n1$", $y \to$ "$n2$"		
Occurence n°3 incorrecte par rapport à la propriété 3	index :	$t+1$	$t+1$	$t+2$
	val =	$x \to$ "$n1$", $y \to$ "$n3$"		

(b) Exemple d'occurrences correctes et incorrectes

FIGURE 6.4 – Exemple de motifs et d'occurrences

Soit F_Id_i l'ensemble des valeurs concrètes déterminé par $M.val$ pour les identifiants symboliques qui n'apparaissent pas dans P_i, mais sont présent dans certains P_j pour $j \neq i$.

Propriété 2 Une configuration concrète ne peut pas contenir de nœuds interdits. $\forall i \in [0, m-1], \forall k \in [s_i, e_i]$, C_k n'a pas de sommet étiqueté par un identifiant dans F_Id_i.

Cette propriété concerne les nœuds qui apparaissent et disparaissent dans la séquence de graphes recherchés. La Figure 6.4a montre un exemple où un nœud y apparaît dans C_2 (la figure utilise une vue simplifiée du graphe recherché avec uniquement les identifiants de nœuds). Pour que y soit considéré comme un nouveau nœud, il ne doit pas apparaître dans les configurations concrètes appariées à C_1. Dans la Figure 6.4b, l'occurrence n°1 est incorrecte, bien que la Propriété 1 soit satisfaite, le nœud $n2$ n'est pas nouveau et ne peut pas jouer de rôle de y. En revanche l'occurrence n°2 est, quant à elle, correcte.

Propriété 3 La fenêtre temporelle d'une occurrence est maximale.
$\forall i \in [0, m-1]$, soit $P'_i = ValuateGraph(P_i, M.val)$. La fenêtre temporelle $[s_i, e_i]$ est maximale, c'est-à-dire :

— $s_i = 0$, ou il n'existe pas d'homomorphisme entre P'_i et C_{S_i-1}, ou C_{S_i-1} contient un somment avec un identifiant dans F_Id_i.

— $e_i = 0$, ou il n'existe pas d'homomorphisme entre P'_i et C_{E_i+1}, ou C_{E_i+1} contient un somment avec un identifiant dans F_Id_i.

Dans la Figure 6.4b, l'occurrence n°3 est incorrecte parce que la date de départ de C_1 devrait être à t, ou même avant si le nœud $n1$ existait déjà dans CC_{t-1}. L'occurrence n°2, quant à elle, est correcte, l'occurrence ne peut pas débuter avant à cause du nœud interdit n_2. Cette propriété implique que les dates de début et de fin correspondent réellement à des changements de configuration du système. Un scénario avec deux motifs successifs identiques ne donnera aucun résultat d'appariement. Les dates contenues dans la table *index* sont déterminées par des événements concrets comme un changement dans la connexion entre deux nœuds, l'introduction ou la disparition d'un nœud, le changement d'un attribut d'un nœud ou n'importe quelle combinaison de ces événements.

Quand un scénario implique l'introduction de nouveaux nœuds, l'appariement peut devenir compliqué. Par exemple, dans la Figure 6.4a, si le nœud concret $n1$ joue le rôle de x, la date de fin de C_1 peut être $t+1$ (avant l'introduction des nouveaux nœuds $n2$ et $n3$), mais aussi $t+2$ (avant l'introduction de $n4$) ou encore plus tard. En fonction du nœud nouveau choisi pour jouer le rôle de y, la date de début de l'occurrence de C_1 peut aussi varier pour prendre en compte le fait que y est un nœud interdit. De plus, la séquence de motifs peut contenir d'autres configurations C_3 ... C_m, et chaque choix, pour y, exige une exploration distincte pour compléter les correspondances entre les nœuds.

6.1.2 Amélioration de la gestion des ressources

L'implémentation de *GraphSeq* représente environ 2000 lignes de code *C++* sans inclure la fonctionnalité d'appariement de graphes provenant de *GMTE*. Afin de vérifier le fonctionnement de cet outil, nous avons développé un outil qui produit de façon aléatoire des séquences de graphes C_i et CC_j qui, par construction, contiennent au moins une fois le motif recherché. Cet outil a démontré une grande utilité pour déboguer *GraphSeq* ainsi que pour les tests de non régression après avoir apporté des changements au code.

La version initiale de *GraphSeq* se présente comme un démonstrateur de l'algorithme d'appariement de séquence de graphes. L'utilisation de l'outil de test a permis d'étudier la capacité de *GraphSeq* à traiter des séquences de différentes tailles. Ce démonstrateur, bien que fonctionnel sur des graphes de taille raisonnable, a rapidement montré ses limites. Ces limites n'étaient aucunement liées à l'algorithme d'appariement de graphes mais plutôt à sa mise en œuvre principalement à sa gestion de la mémoire.

Dans l'objectif de permettre le traitement de « grands » graphes, nous avons effectué une étude détaillée du code de *GraphSeq*. Elle a notamment mis en évidence le besoin de remplacer l'ensemble des structures de données statiques du logiciel par des structures dynamiques plus à même de gérer des quantités de données importantes. Par exemple, le nombre de configurations concrètes était limité à 300 graphes et le nombre de configurations motifs était limité à 5 graphes. Une analyse des opérations de gestion de mémoire a aussi été réalisée à l'aide de l'outil *Valgrind*[1]. Le module *Memcheck* de *Valgrind* permet de tracer l'ensemble des allocations et libérations d'espace mémoire. Il rapporte les erreurs relatives à l'utilisation de la mémoire notamment l'utilisation de pointeurs non initialisés, la non libération de mémoire. Grâce à ces résultats, l'ensemble des fuites mémoire détectées a été éliminé.

Une fois les nouvelles structures de données mises en place, la taille des graphes pouvant être traités par l'application *GraphSeq* n'est plus limitée que par les ressources disponibles sur le calculateur utilisé.

6.1.3 Évaluation de l'efficacité de l'appariement et amélioration des performances

La fonctionnalité bas niveau de comparaison de paires de graphe est un problème de recherche d'isomorphisme de sous-graphes, il est connu pour être *NP-complet* [Goldreich 2008]. Le pire cas d'exécution est quand tous les nœuds du motif peuvent être appariés avec tous les nœuds concrets, ce qui conduit à une complexité exponentielle à la taille du graphe motif.

Ensuite, il y a la partie séquentielle du raisonnement. Le pire cas est quand toutes les transitions $C_i \rightarrow C_{i+1}$ introduisent des nouveaux nœuds : le programme doit explorer toutes les combinaisons possibles pour ces nouveaux nœuds dans les motifs successifs. Cette fois, nous avons une complexité exponentielle à la longueur de la séquence de graphes motifs.

En revanche, la taille et le nombre de configurations concrètes sont bien moins critiques pour les performances. Une première expérimentation avec *GraphSeq* avait consisté à analyser des traces de mobilité correspondant à des enregistrements de durées comprises entre 5 et 10 minutes. Cela correspond à quelques centaines de configurations concrètes contenant entre 15 et 25 nœuds [Nguyen 2010].

Les motifs recherchés sont relativement petits avec moins de 5 nœuds et moins de 5 graphes successifs mais cette taille de configuration est réaliste car elle est similaire à celle utilisée pour tester des protocoles de routage [Devarapalli 2001, Maltz 2001, Cavalli 2004, Medidi 2004, Hu 2005].

D'après l'analyse de complexité précédente, *GraphSeq* ne pourra pas supporter de séquence de graphe motif trop grande qui conduirait à une explosion combinatoire. Dans la pratique néanmoins *GraphSeq* est parfaitement utilisable pour les cas pour lesquels il a été conçu. Un scénario représenté graphiquement ne comporte qu'un nombre relativement faible de nœuds, il en est de même pour le nombre de

1. http ://valgrind.org/

configurations. On peut considérer que les configurations comporteront rarement plus de 10 nœuds et qu'il y aura rarement plus de 5 configuration spatiales.

Les scénarios graphiques se focalisent typiquement sur les interactions qui impliquent quelques entités. Une représentation graphique excède rarement la dizaine de nœuds dans un scénario. Dans tous les cas, une représentation graphique ne peut pas accueillir un grand nombre de lignes de vie et rester lisible facilement. De même, on s'attend à ce que le nombre de motifs successifs dans les scénarios reste faible. En outre, le pire cas où chaque nœud du modèle peut être mis en correspondance avec chaque nœud concret est peu susceptible de correspondre à un scénario d'application significatif. Il est raisonnable de supposer que les nœuds de modèle possèdent des caractéristiques discriminantes tels que des attributs spécifiques ou bien des liens spécifiques avec d'autre nœuds, atténuant ainsi le problème lié à la combinatoire.

Pour améliorer les performances dans de tels cas, nous avons ajouté un prétraitement à *GraphSeq*. Il consiste à ordonner les nœuds dans le motif, de sorte que les nœuds les plus discriminants soient appariés en premier. Reprenons l'exemple de la Figure 6.1. Supposons que l'ordre d'exploration des nœuds par *GraphSeq* soit d'envisager x puis y et z. Le nœud y a des attributs connus ainsi qu'une connexion connue à un autre nœud du motif. Il est, de fait, plus discriminant que x ou z, qui comportent des valeurs d'attributs et de liens joker. Si le motif est réarrangé pour que y soit identifié en premier, il se peut qu'il reste moins de possibilités à explorer à cause du nombre moins important d'occurrences de y et du plus faible nombre de nœuds ayant le bon type de liaison avec y.

Cette fonctionnalité est réalisée à l'aide d'une fonction d'objectif détaillée dans l'Algorithme 6.1 dont le but est de classer les nœuds en fonction du nombre d'attributs connus ainsi que du nombre de connexions qu'ils possèdent. Chaque nœud a un poids qui dépend du nombre de variables déjà valuées, soit parce qu'il s'agit d'une constante ou bien parce que la variable a déjà été valuée dans un graphe précédent de la séquence. Le poids du nœud dépend aussi du nombre de liaisons spécifiées (*i.e.* non joker) avec les autres nœuds du graphe. Pour chaque graphe de la séquence, l'ordre des nœuds dépend de cette fonction d'objectif.

La fonctionnalité a été évaluée à l'aide de séquences de graphes générées aléatoirement [André 2013a]. La taille et le nombre de graphes ont été réglés en fonction de quatre configurations expérimentales $S1 \rightarrow S4$. Le Tableau 6.1 présente les résultats de l'exécution de *GraphSeq* sur ces 4 séquences. La première colonne du tableau indique les paramètres du générateur de graphes. Par exemple, le quadruplet $(5, 5, 5..35, 700..2100)$ signifie :

— le motif généré comporte 5 nœuds ;

— la longueur de la séquence motif est de 5 graphes ;

— les graphes de configurations concrètes contiennent un nombre de nœuds compris entre 5 et 35 ;

— la longueur de la séquence de configurations concrètes est comprise entre 700 et 2100 graphes.

Algorithme 6.1 Calcul du poids d'un nœud apparaissant dans le motif P_i

```
Fitness = 0;

// Reward the node if optional labels are discriminating
foreach optional label li do
   if li is a constant value or a variable that appeared in a previous
       ↪ pattern then
      Fitness += 2;
   endif
endfor

// Reward the node if its connection types with other nodes are
    ↪ discriminating
foreach other node nk of the pattern do
   if connection to nk is not a don't care then
      Fitness += 1;
   endif
endfor
```

Ces configurations sont plus grandes que celles de la première expérimentation mentionnée dans [Nguyen 2010]. Pour chaque configuration, nous avons généré 20 paires de séquences de graphes C_i,(motif) et CC_j(concrète) et nous avons exécuté *GraphSeq* avec et sans la fonctionnalité de réarrangement du motif. Cette expérimentation a été réalisée sur un calculateur équipé d'un processeur cadencé à $2.26Ghz$ et comportant $48Go$ de mémoire vive. Certaines exécutions ont dû être abandonnées et cela pour deux raisons :

— lorsqu'une exécution consommait plus de 90% de la mémoire vive ;

— lorsque la durée de l'exécution dépassait trois heures.

Le Tableau 6.1 donne le nombre d'exécutions abandonnées que ce soit à cause de la mémoire notée M ou bien à cause de la durée d'exécution notée T. Les durées des exécutions qui se sont terminées correctement peuvent être comparées. Le Tableau 6.1 donne les durées moyennes, médianes ainsi que l'écart-type que nous avons observés. La valeur élevée de l'écart type indique que pour une configuration de génération donnée, la difficulté du problème d'appariement varie d'une façon importante. De plus, la moyenne et la médiane peuvent être assez différentes, ce qui indique que les valeurs ne sont pas distribuées suivant une loi normale. Nous avons observé une moyenne et une médiane inférieures lorsque l'optimisation est activée. Pour déterminer si l'optimisation est statistiquement significative, nous avons effectué un test de symétrie des répartitions. Nous avons choisi le *Wilcoxon T test* [Bertrand 2011] car une distribution normale des résultats ne peut être supposée. Ce test permet d'obtenir la valeur-p (ou *p-value*) reportée dans le Tableau 6.1 qui est la probalilité d'obtenir le même valeur du test si l'hypothèse que notre optimisation n'apporte pas d'amélioration du temps de calcul était vraie. -

Il n'y a eu aucun cas où l'exécution s'est terminée correctement sans optimisation alors que la même exécution avec optimisation a été abandonnée. Pour les plus grandes séquences de motifs ($S2$ et $S4$), l'optimisation a amené un gain important permettant de terminer une recherche alors que celle-ci a dû être abandonnée dans la

TABLE 6.1 – Exécutions avec des séquences aléatoires de graphes

Paramètres	**Exécutions abandonnées**		**Durée exécutions [s]** μ (σ) [*median*]		**p-Value**
	optimisation		optimisation		
	sans	avec	sans	avec	
S1 (5, 5, 5..35, 700..2100)	M : 2 T : 0	M : 0 T : 0	1110.29 (2335.47) [140.14]	382.55 (1369.86) [18.72]	$< 10^{-5}$
S2 (5, 10, 10..40, 700..2100)	M : 7 T : 3	M : 0 T : 0	511.82 (635.93) [226.50]	213.18 (31.96) [207.76]	0.037
S3 (10, 5, 5..35, 1200..3600)	M : 0 T : 2	M : 0 T : 1	909.22 (1786.19) [43.38]	259.93 (799.54) [39.68]	0.001
S4 (10, 10, 10..40, 1200..3600)	M : 6 T : 8	M : 0 T : 0	281.92 (396.61) [95.67]	47.07 (6.16) [47.33]	0.031

version sans optimisation de *GraphSeq*. Les exécutions terminées ont une durée plus faible avec optimisation et, la différence de durée est statistiquement significative : l'hypothèse que l'optimisation n'apporte pas d'amélioration peut être rejetée avec un niveau de confiance de 95% pour chacun des 4 jeux de configurations expérimentales.

Nous avons aussi observé un gain pour des scénarios d'une étude de cas présentée dans [André 2013a]. Nous rapportons un cas où l'exécution de *GraphSeq* est plus de 160 fois plus rapide avec optimisation en ne durant que 40 secondes au lieu d'un peu moins de deux heures sans optimisation. Cela nous permet de conclure que la version optimisée de *GraphSeq* est efficace pour traiter les cas ciblés en pratique par *TERMOS*. En effet, ce sont des scénarios avec des motifs réduits, peu susceptibles de correspondre aux pires cas.

6.2 Validation des traces

Grâce aux résultats fournis par *GraphSeq*, nous sommes capable d'extraire de la trace d'exécution une sous-trace par occurrence de la séquence de configuration spatiale. Cette sous-trace est issue d'un filtrage de la trace d'exécution où seul les événements concernant les nœuds identifiés par *GraphSeq* sont présents et les chan-

gements de configurations sont étiquetés avec le nom de la configuration spatiale qu'ils représentent. L'étape suivante dans la vérification d'un scénario consiste à vérifier l'automate représentant le scénario sur chaque sous-trace.

6.2.1 Vérification d'un automate sur une trace

La validation d'une trace d'exécution peut être lancée à partir du moment où nous avons un automate et une trace à vérifier. Les actions permettant la validation sont présentées dans l'Algorithme 6.2. Le fichier de traces est traité événement par événement. Chaque événement doit déclencher le franchissement d'une transition de l'automate à partir de l'état en cours.

TERMOS ayant été conçu pour éviter les non-déterminismes, il ne devrait y avoir qu'une seule transition autorisée dans chaque état de l'automate, et si ce n'est pas le cas, une erreur est reportée à l'utilisateur. L'état suivant la transition venant d'être franchie est conservé pour le traitement du prochain événement de la trace.

Algorithme 6.2 Algorithme général de la vérification d'un automate

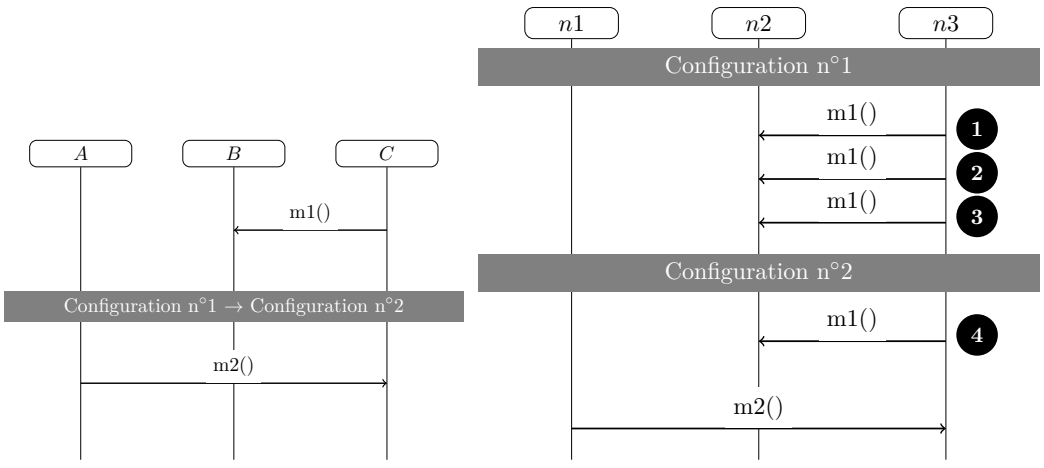
```
1  Input :      trace file
                automaton
3
   Output :     set of verdicts
5
   Variables :  detectedInitMsg : set of message mapped to initial transition
7               results : set of verdicts
                traceEvents : set of events
9
   begin
11
   InitValidateTrace()
13
   if detectedInitMsg.size() > 0 then
15      foreach detectedInitMsg as msg do
            ValidateTrace(msg.timestamp)
17      endfor
   else
19      // We need at least one verdict even if no initial event was detected
        ValidateTrace(null)
21 endif

23 end
```

Au début de la validation représentée par la fonction *ValidateTrace* de l'Algorithme 6.3, l'automate se trouve dans l'état initial. Lorsque la vérification arrive à court d'événements à traiter, le résultat de la validation est déterminé grâce à l'état courant de l'automate. Si l'automate est dans un état de rejet, la validation a échoué (verdict de test *invalide*). Si l'automate est dans un état d'acceptation, la validation a réussi. Il existe deux types d'état d'acceptation, les acceptations triviales (le verdict de test est *non conclusif*) et les acceptations strictes (le verdict de test est *invalide*).

Certaines transitions ont la particularité d'être étiquetées comme transitions initiales d'un automate, c'est-à-dire qu'elles sont franchissables à tout instant. A chaque franchissement de ce type de transition, une nouvelle validation de l'auto-

Algorithme 6.3 Détail de fonctions de l'algorithme de vérification d'un automate

```
// Parse all the trace looking for initial events of the automaton
begin InitValidateTrace()
    Load traceEvents form trace file
    foreach traceEvents as evt do
        if isInitMsg(evt) then
            detectedInitMsg.add(evt)
        endif
    endfor
end InitValidateTrace

// Execute the automaton on the trace starting from initTimeStamp
begin ValidateTrace(timeStamp initTimeStamp)
    Initialize the automaton (Initial state, reset all variables...)
    Load node list and nodes attributes
    Load traceEvents form trace file starting at initTimeStamp

    currentAutomatonState = initialAutomatonState

    foreach traceEvents as evt do
        if isConfigurationChange(evt) then
            // Process event, update currentAutomatonState
            currentAutomatonState = processChange(evt, currentAutomatonState)
        elseif isMsg then
            // Process event, update currentAutomatonState
            currentAutomatonState = processMsg(evt, currentAutomatonState)
        endif

        if enabledTransitions.size() == 0 then
            break
        endif
    endfor

    results.add(currentAutomatonState)

end ValidateTrace
```



(a) Exemple de scénario (b) Exemple de trace

FIGURE 6.5 – Exemple de transition initiale

mate, indépendante de la validation en cours, est déclenchée et produira un verdict. La détection de ces franchissements est effectuée pas la fonction *InitValidateTrace* de l'Algorithme 6.3. Par exemple, le message $m1$ de la Figure 6.5a est une transition initiale. Lors de la phase de vérification de ce scénario sur la trace représentée Figure 6.5b, les événements ❶, ❷ et ❸ déclenchent le franchissement de la transition initiale correspondant au message $m1$. Par contre, l'événement ❹ ne peut pas déclencher le franchissement de la transition initiale car l'événement global de changement de configuration spatiale *Configuration n° 1* → *Configuration n° 2* s'est produit avant lui. Cet exemple de trace, conduira donc à trois vérifications indépendantes.

6.2.2 Prise en compte des prédicats

Les transitions d'un automate peuvent comporter des prédicats écrits d'après la grammaire présentée au chapitre précédent. Ces conditions doivent être évaluées lorsque l'état courant de l'automate est l'état source de la transition.

Cette évaluation se déroule en plusieurs étapes. Une première parcourt l'ensemble de l'expression pour en retirer la liste des variables utilisées dans celle-ci et vérifier que ces variables sont définies pour l'état courant de l'automate.

La deuxième étape consiste à décomposer l'expression en un arbre pour en extraire et évaluer les opérations basiques. Ensuite, les résultats de ces opérations sont combinés avec les autres opérations présentes dans l'arbre, en remontant jusqu'à la racine de l'arbre afin d'obtenir la valuation de l'expression. Prenons par exemple l'expression suivante :

$$
\begin{array}{l}
(\\
\quad (\\
\quad\quad (m1.members\ includes\ x)\ and\ (m2.members\ includes\ x) \\
\quad) \\
\quad and \\
\quad (\\
\quad\quad (m1.members\ includesAll\ m2.members) \\
\quad\quad or \\
\quad\quad (m2.members\ includesAll\ m1.members) \\
\quad) \\
)
\end{array}
$$

Sa décomposition sous forme d'un arbre syntaxique provenant de la grammaire décrite au chapitre 5 est présenté sur la Figure 6.6. À partir de cette représentation, on évaluera chaque expression en partant des extrémités de l'arbre pour remonter jusqu'à sa base. Le résultat final est un booléen indiquant si la transition de

l'automate est franchissable ou pas.

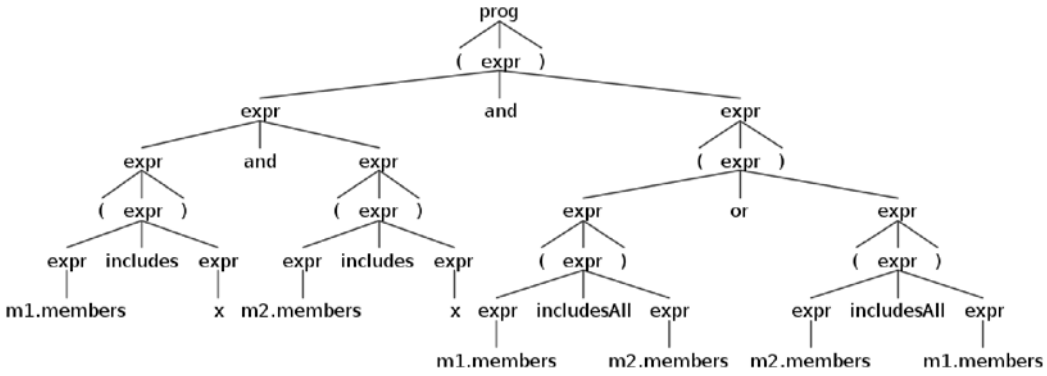


FIGURE 6.6 – Exemple d'arbre représentant une expression

Grâce à l'intégration des prédicats dans nos scénarios, nous sommes maintenant capables d'évaluer des scénarios plus complexes nécessitant l'accès au contenu des messages échangés entre les nœuds d'un scénario.

6.3 Aide à l'analyse par visualisation graphique

La vérification d'un scénario sur une trace d'exécution donne la plupart du temps un ensemble de résultats de validation, du fait du nombre d'occurrences trouvées par *GraphSeq* et du nombre de fois où l'automate a été vérifié. Cet ensemble de résultats, résumé à un ensemble de verdicts et aux informations de validation comme présentées dans la Figure 6.7, est souvent difficile à interpréter à cause du nombre de résultats. Nous avons donc décidé de développer une interface graphique permettant une analyse des résultats de validation par l'utilisateur.

6.3.1 Vue générale

À l'aide de cette vue graphique, il est plus aisé pour l'utilisateur d'analyser en détail la succession d'événements qui a conduit à un verdict. En fonction de son analyse, il pourra en déduire les changements à apporter à l'application pour qu'elle respecte les spécifications ou bien ceux à apporter aux spécifications.

La Figure 6.8, présente une vue de l'ensemble de l'outil d'analyse des verdicts dans l'environnement *Eclipse*. Cette vue est composée de trois panneaux que nous allons décrire dans les paragraphes suivants. Un premier contenant l'ensemble des résultats provenant des exécutions de l'automate. Un zoom du contenu de ce panneau est présenté sur la Figure 6.7, il contient le verdict, le nom de la trace, l'identifiant de l'état final ainsi que l'horodatage du premier événement de la trace traitée. Lorsque l'utilisateur sélectionne une ligne de verdict de ce panneau les deux autres panneaux s'affichent. L'un contient l'ensemble des transitions franchies par l'automate et le second contient la trace.

Verdict		Final Stat	Start Time
▼ ACCEPT_STRINGENT	13		
▼ at state 6	13	6	
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:04 245
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:05 243
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:05 248
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:06 358
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:06 386
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:16 255
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:16 257
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:34 282
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:34 331
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:38 252
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:38 256
ACCEPT_STRINGENT	/home/aleclerc/sources/traces-14-07-07/	6	08:23:41 270
▼ ACCEPT_TRIVIAL	1		
▼ at state 2	1	2	
ACCEPT_TRIVIAL	/home/aleclerc/sources/traces-14-07-07/	2	08:23:41 272

FIGURE 6.7 – Extrait des verdicts obtenus lors d'une validation de trace

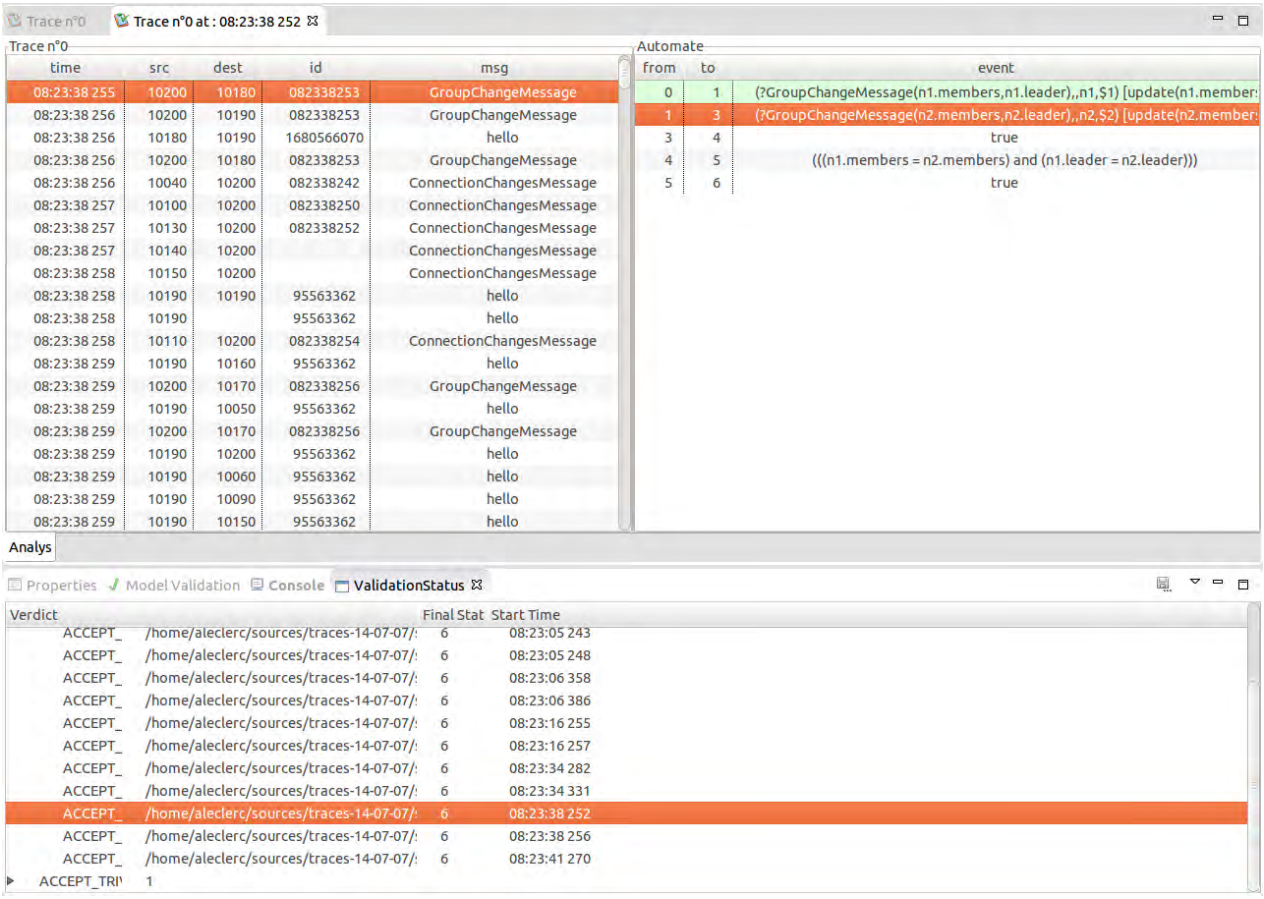
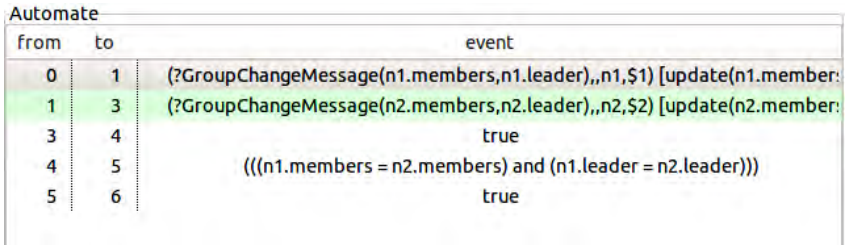


FIGURE 6.8 – Vue globale de l'outil d'aide à l'analyse de résultats

6.3.2 Visualisation des transitions franchies

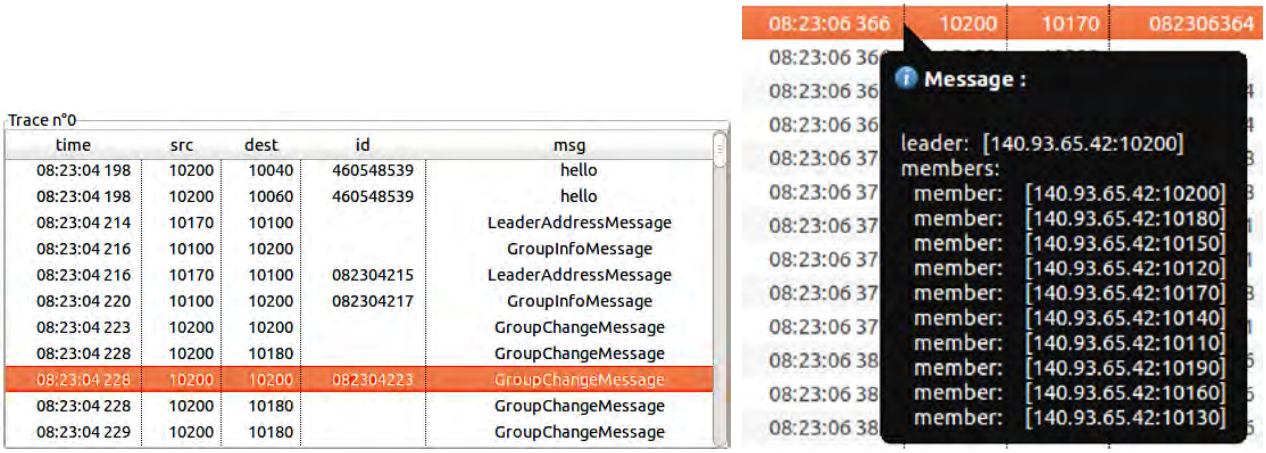
Pour aider à analyser ces résultats, notre plate-forme comporte un outil permettant de représenter chaque résultat de validation de manière détaillée. Une première partie de l'outil permet, pour une exécution donnée, de représenter l'ensemble des transitions de l'automate franchies pour arriver à l'état final et donc au verdict. Les transitions franchies sont représentées sous la forme de trois colonnes, état source, état destination et étiquette de la transition. Cette représentation sous forme de liste de transitions est celle qui reste la plus compréhensible quelle que soit la taille de l'automate. Un exemple sur un automate est présenté Figure 6.9. Pour un automate simple comme celui-ci, il aurait été plus parlant d'utiliser une vue graphique de l'automate en mettant en surbrillance le chemin parcouru par la validation. En effet, la validation de scénarios plus complexes entraîne la génération d'automates bien trop grands pour être lisible sur un écran.


Automate

from	to	event
0	1	(?GroupChangeMessage(n1.members,n1.leader),,n1,$1) [update(n1.member:
1	3	(?GroupChangeMessage(n2.members,n2.leader),,n2,$2) [update(n2.member:
3	4	true
4	5	(((n1.members = n2.members) and (n1.leader = n2.leader)))
5	6	true

FIGURE 6.9 – Détail des transitions franchies d'un automate

6.3.3 Visualisation des événements déclencheurs


Trace n°0

time	src	dest	id	msg
08:23:04 198	10200	10040	460548539	hello
08:23:04 198	10200	10060	460548539	hello
08:23:04 214	10170	10100		LeaderAddressMessage
08:23:04 216	10100	10200		GroupInfoMessage
08:23:04 216	10170	10100	082304215	LeaderAddressMessage
08:23:04 220	10100	10200	082304217	GroupInfoMessage
08:23:04 223	10200	10200		GroupChangeMessage
08:23:04 228	10200	10180		GroupChangeMessage
08:23:04 228	10200	10200	082304223	GroupChangeMessage
08:23:04 228	10200	10180		GroupChangeMessage
08:23:04 229	10200	10180		GroupChangeMessage

(a) Détail d'une trace (b) Exemple d'une infobulle

FIGURE 6.10 – Présentation d'une trace

Pour chacune des transitions franchies, cet outil permet d'analyser l'événement déclencheur de celle-ci. Par exemple l'événement sélectionné par l'utilisateur dans

la Figure 6.9 est surligné dans la Figure 6.10a. Cette figure présente l'ensemble de la trace. Par défaut, cela affiche les informations communes à chaque message (l'horodatage, la source, la destination, l'identifiant et le type du message) et le volume des informations spécifiques à chaque type de messages sont représentées dans une infobulle lors de la sélection d'un événement (Figure 6.10b).

6.4 Conclusion

Dans ce chapitre, nous avons présenté les modifications apportées à un outil existant, *GraphSeq*, pour permettre son passage à l'échelle. Nous avons aussi présenté les différentes étapes de la vérification d'un scénario sur une trace d'exécution d'une application. Un scénario se produisant la plupart du temps plusieurs fois sur une trace d'exécution, à l'issue de cette vérification nous obtenons autant de verdicts qu'il y a eu d'exécutions de l'automate sur la trace. Grâce à l'analyse des verdicts effectuée par l'utilisateur, il est possible de revenir sur les spécifications et le code du système sous test afin de corriger le comportement du système.

Chapitre 7

Démonstrateur

Sommaire

Nous sommes maintenant capable de réaliser l'ensemble des étapes de conception et de vérification d'un scénario sur une trace d'exécution d'une application mobile. Pour valider tous ces travaux, il faut collecter des traces d'exécutions d'une application. Cette étape nécessite une plate-forme d'exécution capable de contrôler l'exécution de l'application sous test. Ce chapitre présente dans un premier temps la plate-forme d'exécution mise en œuvre pour collecter des traces d'exécution. Dans un second temps, nous étudions une étude de cas : un protocole d'appartenance de groupe. Les propriétés de ce protocole seront présentées et traduites sous la forme de scénario *TERMOS*. Puis, à l'aide de la plate-forme de test, nous simulerons l'exécution de ce protocole afin de collecter des traces d'exécutions et vérifier les scénarios sur celles-ci.

7.1 Plate-forme d'exécution

La plate-forme expérimentale doit offrir des facilités pour observer et collecter des données lors de l'exécution de l'application testée. La trace recueillie comporte à la fois des données relatives aux messages échangés par les nœuds composant le système mais aussi des données contextuelles utilisées par chaque nœud pour adapter leur comportement. Il est important pour TERMOS d'avoir une trace contenant

la position des nœuds à un instant précis. La plate-forme doit aussi permettre de contrôler l'environnement virtuel dans lequel évolue le système que ce soit au niveau des informations obtenues par des capteurs instrumentant l'environnement ou au niveau des communications avec les autres nœuds composant le système. En effet, un nœud ne doit pas communiquer avec un autre qui est hors de portée de communication donc il faudra un moyen de contrôler cela.

Dans les sections suivantes, nous présentons l'architecture de la plate-forme de test que nous avons mise en place mais aussi différentes solutions envisageables pour avoir une plate-forme de simulation et de collecte de traces qui soit la moins invasive possible pour le système sous test. Une solution ne nécessitant pas de modification du système sous test permettrait de garantir que l'application testée est non modifiée. Cela permettrait en plus une mise en place des tests plus rapide que ce soit pour tester une nouvelle version de la même application ou une nouvelle application sur la même plate-forme.

7.1.1 Architecture générale

Dans le cadre des applications mobiles, les nœuds communiquent en point-à-point ou par diffusion. Dans tous les cas, le message émis sera reçu si le récepteur est à portée de communication de l'émetteur. De plus, le temps de transmission d'un message dépend de la distance entre les nœuds. Il faut donc que notre plate-forme comporte des composants qui gèrent la position et la communication.

Vu d'un haut niveau d'abstraction, notre plate-forme de test de systèmes mobiles est composée de trois composants essentiels représentés Figure 7.1 :

— Le *Gestionnaire de contexte* contrôle le contexte. Dans notre cas, le contexte correspond aux positions relatives de chaque nœud en fonction d'un modèle de mobilité. La position et le mouvement des nœuds sont fournis par le gestionnaire de contexte au simulateur réseau et au support d'exécution.

— Le *Simulateur réseau* contrôle les communications entre les nœuds à l'aide des informations de mobilité fournies par le *Gestionnaire de contexte*. Le simulateur utilise cette position pour contrôler, en fonction du medium de communication mis en œuvre, si le message émis peut être distribué et si oui, à qui.

— L'application à tester s'exécute sur plusieurs nœuds mobiles simultanément. Le *Support d'exécution de l'application* est donc composé de plusieurs *Supports d'exécutions de l'application* représentant chacun un nœud. Le message à émettre est transmis par le support d'exécution de l'application au simulateur réseau.

7.1.2 Gestionnaire de contexte

Le contexte dans le cadre des systèmes mobiles est au minimum composé des informations de mobilité de chaque nœud membre du système. Comme présenté

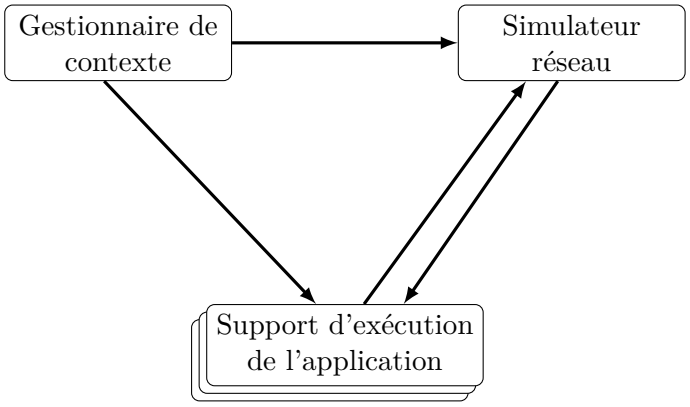


FIGURE 7.1 – Vue schématique de la plate-forme de test

dans le chapitre 2, le contexte peut contenir beaucoup d'autres informations permettant au système d'adapter son comportement en fonction de l'environnement dans lequel il évolue. La gestion du contexte peut être décomposée en deux étapes, la génération et la simulation de l'environnement.

Génération de l'environnement Cette première étape peut être réalisée hors ligne, elle consiste à produire une trace de mobilité ainsi que l'ensemble des informations contextuelles nécessaires à l'application à tester. La trace de mobilité peut provenir d'un simulateur de mobilité ou de traces réelles.

Un simulateur de mobilité génère une trace à l'aide de modèles de mobilité. Dans notre étude de cas, nous avons utilisé le simulateur issu du projet *Impact of Mobility Motifs On Routing proTocol in the mobile Ad hoc NeTworks* (*IMPORTANT*) [Bai 2003b]. Les traces réelles proviennent de captures effectuées lors d'une expérimentation en conditions réelles avec des véhicules, des personnes, des animaux,etc.

La production des traces de mobilité hors ligne est importante si nous souhaitons reproduire les expériences. En effet, si les nœuds mobiles étaient reliés en temps réel à des positions de personnes dans une foule par exemple, il ne serait pas possible de reproduire l'expérience.

Simulation de l'environnement La deuxième partie consiste pendant l'exécution du système à fournir en « temps réel » les informations de position au simulateur réseau ainsi que les informations contextuelles nécessaires à chaque nœud. Cette partie est aussi responsable de la coordination de la simulation entre l'ensemble des *supports d'exécution* et le *simulateur réseau* notamment en fournissant une horloge de référence globale, ce qui permettra d'entrelacer les événements de communication dans une seule trace. Le *gestionnaire de contexte* est aussi utilisé pour produire une trace contenant les événements *globaux* relatifs aux informations contextuelles telles que le changement de type de liaison entre deux nœuds (connectés ou pas, par exemple). Cela permettra d'entrelacer cette trace avec celle contenant les événements de communication pour faire de la vérification avec TERMOS.

7.1.2.1 Modèles de mobilité

Le fonctionnement d'un simulateur de mobilité repose sur un modèle de mobilité qui est le plus souvent paramétrable. Par exemple, le simulateur *IMPORTANT* que nous avons utilisé dispose de quatre modèles de mobilité que nous allons présenter.

Modèle *Random Waypoint* Ce modèle est celui qui est le plus couramment utilisé en recherche. Sa mise en œuvre se résume à choisir de façon aléatoire pour chaque nœud une destination ainsi qu'une vitesse de déplacement. Une fois que le nœud a atteint sa destination, il se met en pause pour une durée aléatoire puis choisit une nouvelle destination et une nouvelle vitesse de déplacement. Ce processus est répété indéfiniment jusqu'à arriver à la fin de la simulation.

Modèle *Freeway* Le modèle *Freeway* émule le comportement du mouvement de véhicules sur une autoroute. Certaines limitations sont introduites dans ce modèle par rapport au modèle *Random Waypoint*.

— Les dispositifs mobiles ne peuvent pas changer de voie.

— La vitesse d'un dispositif est dépendante de sa vitesse précédente.

— Si deux dispositifs se trouvent sur la même voie à une distance l'un de l'autre inférieure à une distance de sécurité, la vitesse du dispositif en seconde position ne peut excéder celle du premier.

Elles imposent une dépendance spatiale et temporelle entre les positions successives d'un nœud ainsi que des restrictions aux mouvements des nœuds.

Modèle *Manhattan* Le modèle *Manhattan* reprend le principe du modèle *freeway* en l'adaptant à une structure urbaine contenant uniquement des rues perpendiculaires ou parallèles les unes aux autres. Ce modèle utilise une carte contenant un ensemble de rues. À chaque intersection, le dispositif mobile peut tourner à droite, à gauche ou aller tout droit. Les relations entre dispositifs mobiles sont les mêmes que pour le modèle *freeway*.

Modèle *Reference Point Group* Ce modèle est basé sur un déplacement en groupe des dispositifs. Chaque groupe contient un meneur qui détermine le mouvement du groupe. À l'initialisation du modèle les autres membres du groupe sont répartis de manière uniforme dans le voisinage du meneur. À chaque instant, les autres membres du groupe ont leurs paramètres de vitesse et de direction dérivés de façon aléatoire de ceux du meneur.

7.1.3 Simulateur réseau

Pour notre plate-forme, nous avons fait le choix d'utiliser un simulateur de réseau au lieu de connecter directement entre eux les dispositifs mobiles simulés. Ce choix permet de répondre à deux problèmes : la gestion des communications et l'instrumentation de l'application.

Gestion des communications La simulation du réseau permet de gérer les communications entre les nœuds en se basant sur les informations provenant du *gestionnaire de contexte*, par exemple si deux nœuds sont trop éloignés l'un de l'autre, ils ne peuvent échanger aucun message. Il est aussi possible de mettre en place des délais de transfert en fonction de l'environnement dans lequel évolue le système ou même d'introduire des pertes de paquets si nous voulons être le plus proche possible d'un comportement global réel.

Instrumentation de l'application Les outils de simulation réseau peuvent être utilisés pour capturer l'ensemble des communications entre les nœuds du système. Intégrer la capture de trace au niveau du simulateur réseau permettrait d'être moins intrusif dans le système à tester mais nécessite de savoir décoder le type et le contenu de chaque message.

Pour répondre à ces deux points, plusieurs techniques sont utilisables, certaines dépendent des choix de mise en œuvre du *support d'exécution de l'application*.

7.1.4 Support d'exécution de l'application

Pour vérifier un scénario *TERMOS*, il est nécessaire d'avoir une trace d'exécution contenant les échanges de messages entre les différents nœuds de l'application à tester. *TERMOS* s'attache à ne prendre en compte que les messages entre les nœuds. Aucun événement interne à un nœud ne peut être utilisé dans un scénario. Grâce à cela, la collecte de traces peut être faite de différentes manières plus ou moins intrusives au niveau de l'application à tester.

Dans le cas du démonstrateur, nous avons utilisé une méthode basée sur une instrumentation de l'application sous test en raison de sa simplicité de mise en œuvre. Cette méthode nécessite une instrumentation manuelle de l'application à tester afin de simuler les interactions de l'application avec le monde extérieur.

Une deuxième méthode, plus complexe à mettre en œuvre, sera par contre plus adaptée lorsqu'il sera nécessaire d'industrialiser la gestion des tests sur un ensemble d'applications.

Instrumentation nécessitant une modification de l'application sous test Il est possible d'instrumenter directement l'application mais cela nécessite de modifier légèrement son fonctionnement pour être capable d'intercepter les événements d'émission et de réception de messages. Cette modification impacte directement l'application et peut introduire des fautes dans le système, par contre elle est relativement simple à mettre en place contrairement aux autres solutions. De plus, cette méthode permet d'avoir un accès direct aux formats des messages. Cela permet d'enregistrer dans la trace d'exécution les données dans un format structuré sans nécessiter de traitement supplémentaire.

La plupart du temps, le système sous test, ne comporte pas que des liens réseaux. Il a aussi besoin de connaître l'environnement dans lequel il évolue, par

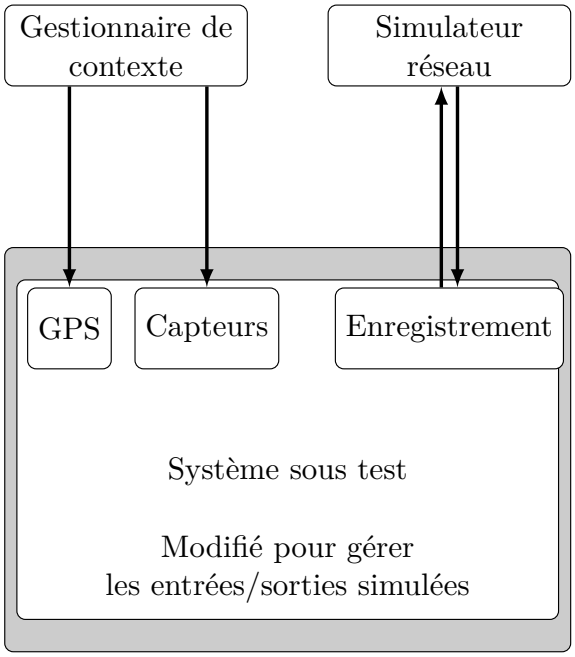


FIGURE 7.2 – Système sous test instrumenté

exemple en utilisant un *GPS* ou des capteurs. Il est donc nécessaire de simuler ces périphériques pour fournir au système l'ensemble des informations nécessaire à son fonctionnement. Ces informations doivent pouvoir être contrôlées par le gestionnaire de contexte. Pour cela, de la même manière que pour la gestion des communications réseau, il est possible de modifier l'application pour qu'elle puisse recevoir directement les informations contextuelles. Cette architecture est présentée Figure 7.2.

Instrumentation préservant l'application sous test de modifications Pour être le moins intrusif au niveau de l'application, il faudrait gérer l'enregistrement des messages au niveau du réseau. Ce fonctionnement nécessite une parfaite isolation des nœuds entre eux afin de garantir que l'ensemble des communication réseau est bien enregistré. Il est aussi nécessaire de connaître les spécifications du format des messages pour être capable de décoder et de stocker correctement les messages.

Pour gérer les informations contextuelles, il faudrait que l'environnement d'exécution simule le comportement de capteurs réels. Par exemple un *GPS*, est utilisé comme une source d'information de position et de temps. Il utilise la plupart du temps une liaison série pour envoyer périodiquement aux applications un message contenant l'heure et les coordonnées auxquelles il se trouve. Il suffirait donc de simuler un périphérique connecté par une liaison série à l'application sous test pour fournir les informations contextuelles à celle-ci. Cette architecture est représentée Figure 7.3.

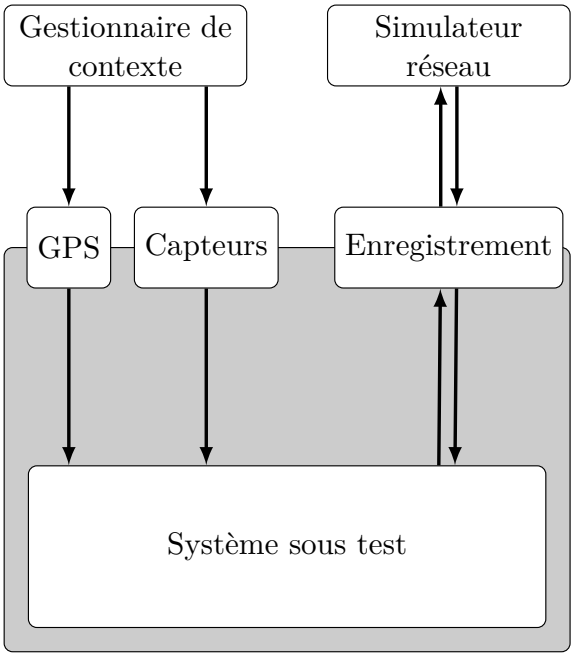


FIGURE 7.3 – Support d'exécution instrumenté

7.2 Étude d'un protocole d'appartenance de groupe : le GMP

Nous avons mis en œuvre l'ensemble de notre méthode de test, sur un protocole d'appartenance de groupe dans les réseaux ad hoc. Un protocole d'appartenance de groupe est un service de base des systèmes distribués tolérant aux fautes. Son but est de maintenir une vue cohérente des membres du groupe.

Le *GMP* étudié ici a été proposé par [Huang 2004], il est destiné à une utilisation sur des nœuds mobiles formant des réseaux ad hoc. Sous certaines hypothèses sur l'environnement, le protocole étudié vise à offrir un service de groupe dépendant de la distance entre les nœuds.

7.2.1 Principe

La fonctionnalité principale du *GMP* est de maintenir une vue cohérente des membres du groupe à chaque instant. Dans un système mobile, les liens de communication entre les nœuds sont instables. Les nœuds peuvent être déconnectés ou reconnectés en fonction de leurs mouvements ou de leur niveau de batterie par exemple.

Pour adapter dynamiquement les groupes de nœuds, l'implémentation étudiée repose sur la notion de distance de sécurité (*safe distance*). Cette distance de sécurité est calculée pour donner suffisamment de temps à deux nœuds appartenant au même groupe, s'éloignant l'un de l'autre à leur vitesse de déplacement maximum, de partitionner le groupe correctement avant de perdre la possibilité de communiquer

l'un avec l'autre. Cette notion est illustrée sur la Figure 7.4. Autour de chaque nœud est représentée la portée de communication en trait plein et la distance de sécurité en trait pointillé. Grâce à cette notion de distance de sécurité, un lien entre deux nœuds peut être caractérisé de trois manières différentes :

safe distance **:** par exemple, sur la Figure 7.4 les nœuds A et B se trouvent à une distance l'un de l'autre inférieure à la distance de sécurité. La liaison entre eux est donc de type *safe*.

communication range **:** par exemple, sur la Figure 7.4 les nœuds C et D se trouvent à une distance l'un de l'autre supérieure à la distance de sécurité et inférieure à la portée de communication. La liaison entre eux est donc de type *communication range*.

disconnected **:** les nœuds éloignés d'une distance supérieure à la portée de communication sont considéré comme déconnectés l'un de l'autre.

Une quatrième catégorie, intitulée *any*, regroupant l'ensemble des trois précédentes permet dans un scénario *TERMOS* de représenter un lien pouvant être d'un type quelconque.

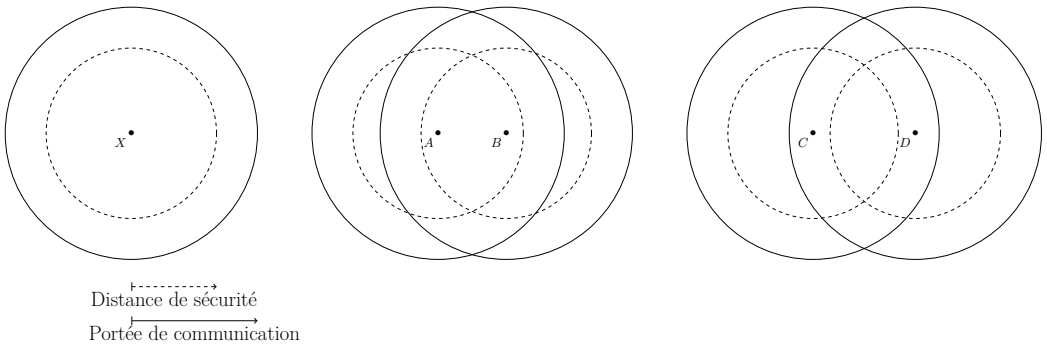


FIGURE 7.4 – Notion de distance de sécurité

7.2.2 Propriétés

Le service d'appartenance de groupe est spécifié par la définition de variables d'état locales à chaque nœud, et par les propriétés que ces variables d'état doivent satisfaire. Deux variables d'état sont :

— $group(p, t)$: l'identifiant du groupe d'un nœud p à un instant local t. L'identifiant du groupe est composé de l'identifiant du leader et d'un numéro de séquence qui croît à chaque changement de groupe,

— $mem(p, t)$: la vue locale de l'ensemble des nœuds membres du groupe du nœud p à un instant local t.

Le numéro de séquence de l'identifiant de groupe permet de définir une relation d'ordre sur les groupes successifs auxquels le nœud p appartient. Un groupe est appelé g si son identifiant est g. Un groupe g' est un successeur du groupe g s'il existe un membre p de g qui quitte g pour rejoindre g'. On note $succ(g, p)$ le successeur du groupe g pour le nœud p. D'une manière similaire, $pred(g, p)$ indique le prédécesseur du groupe g pour le nœud p.

Propriété		Description
Auto inclusion	AI	Un nœud appartient toujours à sa propre vue du groupe. $p \in mem(p, t)$
Monotonicité locale	ML	Sur chaque nœud, l'identifiant du groupe courant croît à chaque changement de groupe. $pred(g, p) < g < succ(g, p)$
Accord sur les membres du groupe	AM	Si deux nœuds ont le même identifiant de groupe, ils ont la même vue du groupe. $group(p, t_p) = group(q, t_q) \Rightarrow mem(p, t_p) = mem(q, t_q)$
Vue initiale du groupe	VIG	Un nœud est le seul membre de sa propre vue quand il est créé. $mem(p, t_{init}) = p$
Justification de changement de groupe	JCG	Le successeur du groupe g pour le nœud p est soit un sur-ensemble strict, soit un sous-ensemble strict du groupe g.
Cohérence de vues à la délivrance de message	CVD	Si un nœud p envoie un message applicatif m_{pq} à un nœud q à une date t et q est dans $mem(p, t)$ alors le message m_{pq} sera délivré à q à une date t_0, et $mem(q, t_0) = mem(p, t)$.
Fusion condition-nelle des groupes	FCG	Si deux groupes g_1 et g_2 satisfont le critère de fusion à un instant T et s'ils le satisfont assez longtemps (au moins pendant une constante de temps T_c) alors ils fusionneront en un seul groupe. Soit p un membre de g_1 et q un membre de g_2 : $\exists t_p, t_q \in [T, T + T_c], mem(q, t_q) = mem(p, t_p)$
Scission condition-nelle du groupe	SCG	Un groupe se divise seulement si c'est nécessaire.

Table 7.1 – Les propriétés du protocole GMP

Le service d'appartenance du groupe est alors caractérisé par les huit propriétés présentées dans le Tableau 7.1, qui doivent être satisfaites lorsque les hypothèses du protocole sont vérifiées. On trouvera une description plus détaillée de ces propriétés

dans la section 2 de l'article [Huang 2004]. Afin de préparer le test du *GMP*, nous classons ces propriétés en deux catégories :

— les propriétés qui peuvent être vérifiées localement sur un nœud (AI, ML, VIG, JCG),

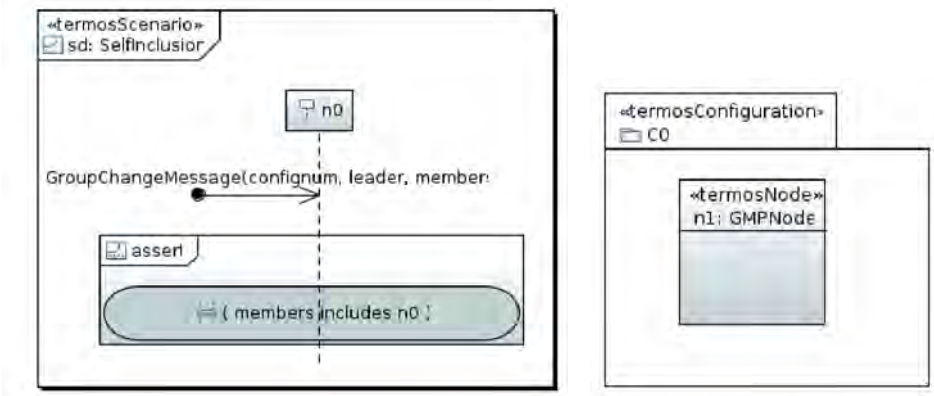
— les propriétés qui doivent être vérifiées par des relations entre au moins deux nœuds (AM, CVD, FCG, SCG).

7.3 Scénarios *TERMOS* des propriétés du *GMP*

Pour pouvoir vérifier les propriétés du *GMP* listées dans le Tableau 7.1, nous avons crée un scénario *TERMOS* pour chacune d'entre elles.

Auto Inclusion : A tout instant, un nœud doit forcément être présent dans sa vue du groupe. Pour ce scénario représenté sur la Figure 7.5, la vue spatiale a un intérêt limité puisqu'elle représente un seul nœud. En effet la phase de recherche des configurations spatiales ne sera pas discriminante et générera des sous-traces contenant l'ensemble des événements.

L'exigence positive peut s'écrire : A chaque fois qu'un message de type *groupChange* est réceptionné par un nœud, alors la liste des nœuds présente dans le message doit contenir le nœud.

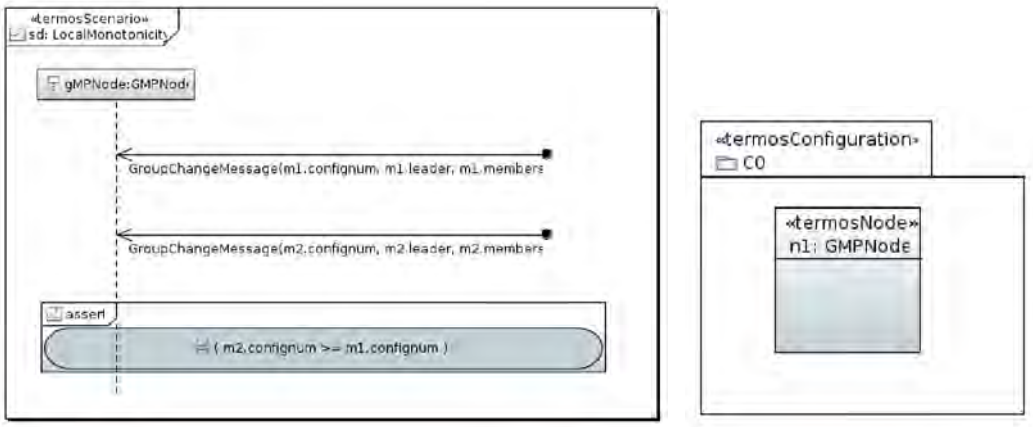


(a) Vue événementielle (b) Vue spatiale

FIGURE 7.5 – Scénario *TERMOS* de la propriété *Self Inclusion*

Monotonicité locale : L'identifiant de groupe d'un nœud ne peut évoluer que de façon incrémentale. Le scénario de la Figure 7.6 représente un nœud et vérifie que le numéro de séquence $confignum$ de deux messages $m1$ et $m2$ de type *GroupChange* reçus successivement par ce nœud évolue bien de manière incrémentale.

L'exigence positive peut s'écrire : A chaque fois qu'un nœud réceptionne des messages de type *groupChange*, alors le numéro de séquence reçu ne peut évoluer que de manière croissante.



(a) Vue événementielle (b) Vue spatiale

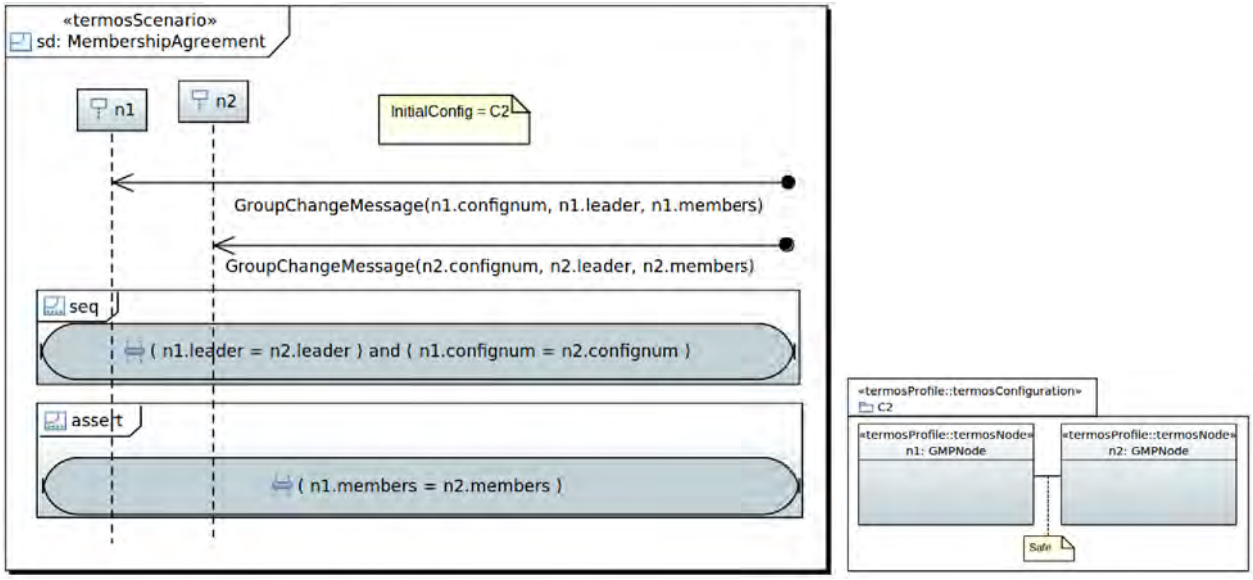
FIGURE 7.6 – Scénario *TERMOS* de la propriété *Local Monotonicity*

Accord sur les membres du groupe : La propriété d'*Accord sur les membres du groupe* permet de spécifier que si deux nœuds ont le même identifiant de groupe alors ils doivent avoir la même vue du groupe. Pour traduire cette propriété dans le langage *TERMOS*, nous avons considéré deux nœuds $n1$ et $n2$ connectés l'un avec l'autre par une liaison de type *safe* (Figure 7.7b). D'après les propriétés du protocole, ces deux nœuds devraient alors être membres du même groupe à partir du moment où ils auront reçu un message de type *GroupChange* leur indiquant leur nouveau groupe. Ce message provient du *leader* du groupe qui peut être $n1$, $n2$ ou bien d'un nœud non représenté. L'expéditeur du message n'étant pas important pour cette propriété, il n'est pas spécifié dans la vue événementielle du scénario (Figure 7.7a). À partir du moment où les deux nœuds ont reçu un message de type *GroupChange*, il nous est possible de vérifier le contenu des informations reçues par les deux nœuds. Ces informations contiennent l'identifiant du groupe composé de deux éléments : le *leader* et un identifiant numérique incrémental. Si ces deux éléments sont identiques dans les messages reçus par les nœuds $n1$ et $n2$ alors ils doivent avoir la même liste de membres.

L'exigence positive peut s'écrire : A chaque fois que deux nœuds à *safe distance* l'un de l'autre réceptionnent des messages de type *groupChange* contenant le même leader et le même numéro de séquence, alors ils doivent avoir la même liste de membres.

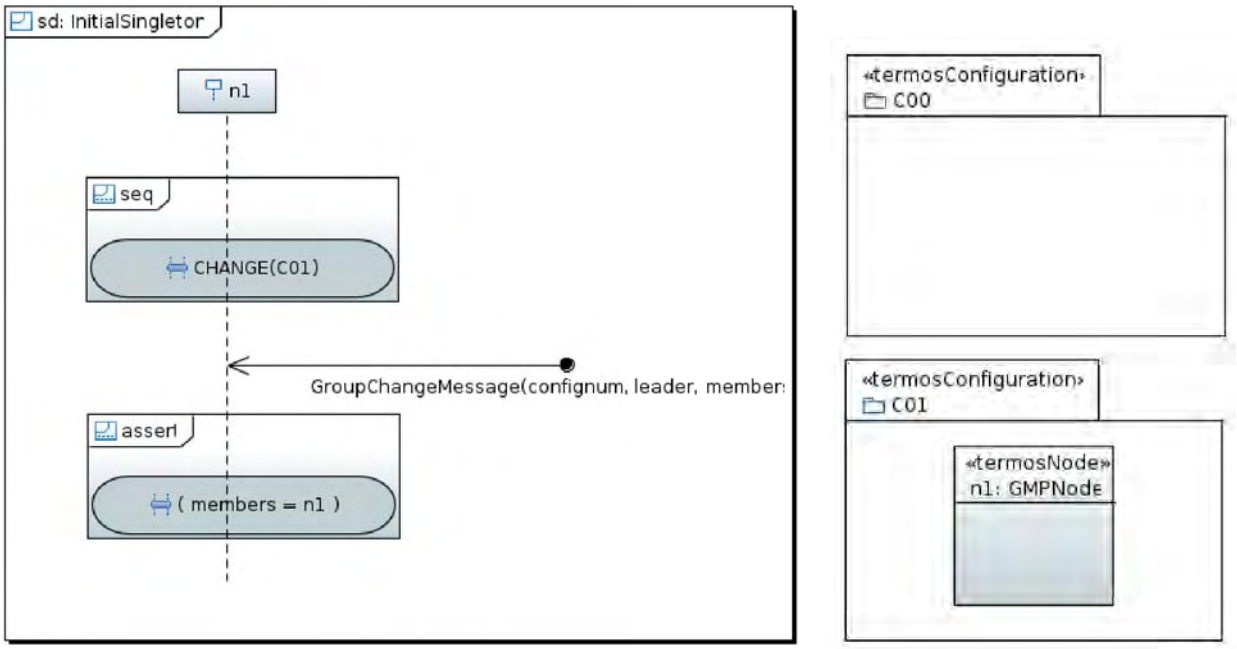
Vue initiale du groupe : À la création d'un nœud, celui-ci crée un groupe dans lequel il est le seul membre. Pour décrire cette propriété sous forme de scénario, Figure 7.8, nous utilisons deux configurations spatiales. Une première, nommée $C00$ où le nœud n'existe pas et une deuxième, nommée $C01$ dans laquelle le nœud existe.

L'exigence positive peut s'écrire : A chaque fois qu'un nœud est crée, alors il crée un groupe dont il est le seul membre.



(a) Vue événementielle (b) Vue spatiale

FIGURE 7.7 – Scénario *TERMOS* de la propriété *Membership Agreement*



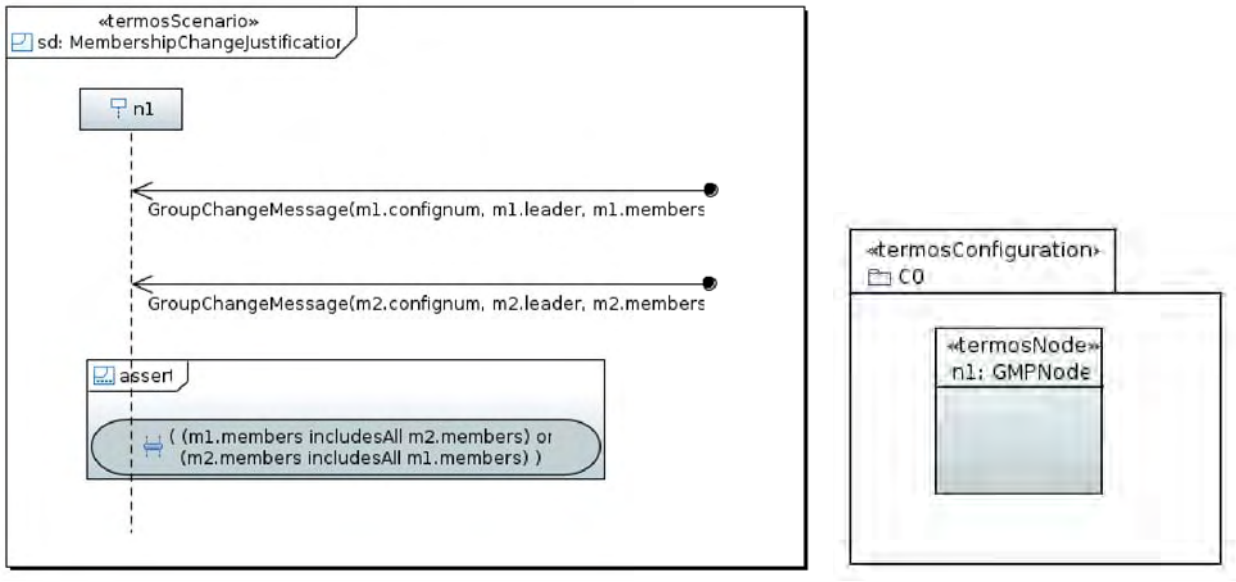
(a) Vue événementielle (b) Vue spatiale

FIGURE 7.8 – Scénario *TERMOS* de la propriété *Initial Singleton*

Justification de changement de groupe : Lors d'un changement de groupe, le nouveau groupe ne peut être qu'un sous-ensemble strict du groupe précédent ou bien un sur-ensemble strict du groupe précédent. Cela se traduit par le scénario

représenté Figure 7.9 où un nœud reçoit deux messages *GroupChange* $m1$ et $m2$, les listes de membres contenues dans ces deux messages ne peuvent être que des sur-ensembles stricts l'un de l'autre.

L'exigence positive peut s'écrire : A chaque fois qu'un nœud réceptionne des messages de type *groupChange*, alors la liste des membres de deux messages successif ne peut être qu'un sur-ensemble strict ou un sous-ensemble strict l'un de l'autre.



(a) Vue événementielle (b) Vue spatiale

FIGURE 7.9 – Scénario *TERMOS* de la propriété *Membership Change Justification*

7.4 Scénarios illustrant des comportements particuliers

En plus des propriétés définies dans la spécification du *GMP*, nous avons construit des scénarios illustrant des comportements particuliers du système. Par exemple une scission de groupe qui ne se déroule pas correctement et conduit à une vue incohérente du groupe, ou bien une opération de fusion et de scission concurrentes d'un même groupe qui peut elle aussi conduire à une vue incohérente du groupe.

L'exigence négative de ce scénario ne peut s'écrire sous forme textuelle, elle signifie qu'on souhaite ne jamais observer cet entrelacement de message.

Fusion/scission de groupe concurrentes : Un groupe composé des nœuds $n2$, $n3$ et $n4$ se divise en deux avec d'un coté $n2$ et $n4$ et de l'autre $n3$. Simultanément, le nœud $n1$ fusionne avec le groupe formé par $n2$ et $n4$. Ce scénario est représenté à l'aide du langage *TERMOS* dans la Figure 7.10

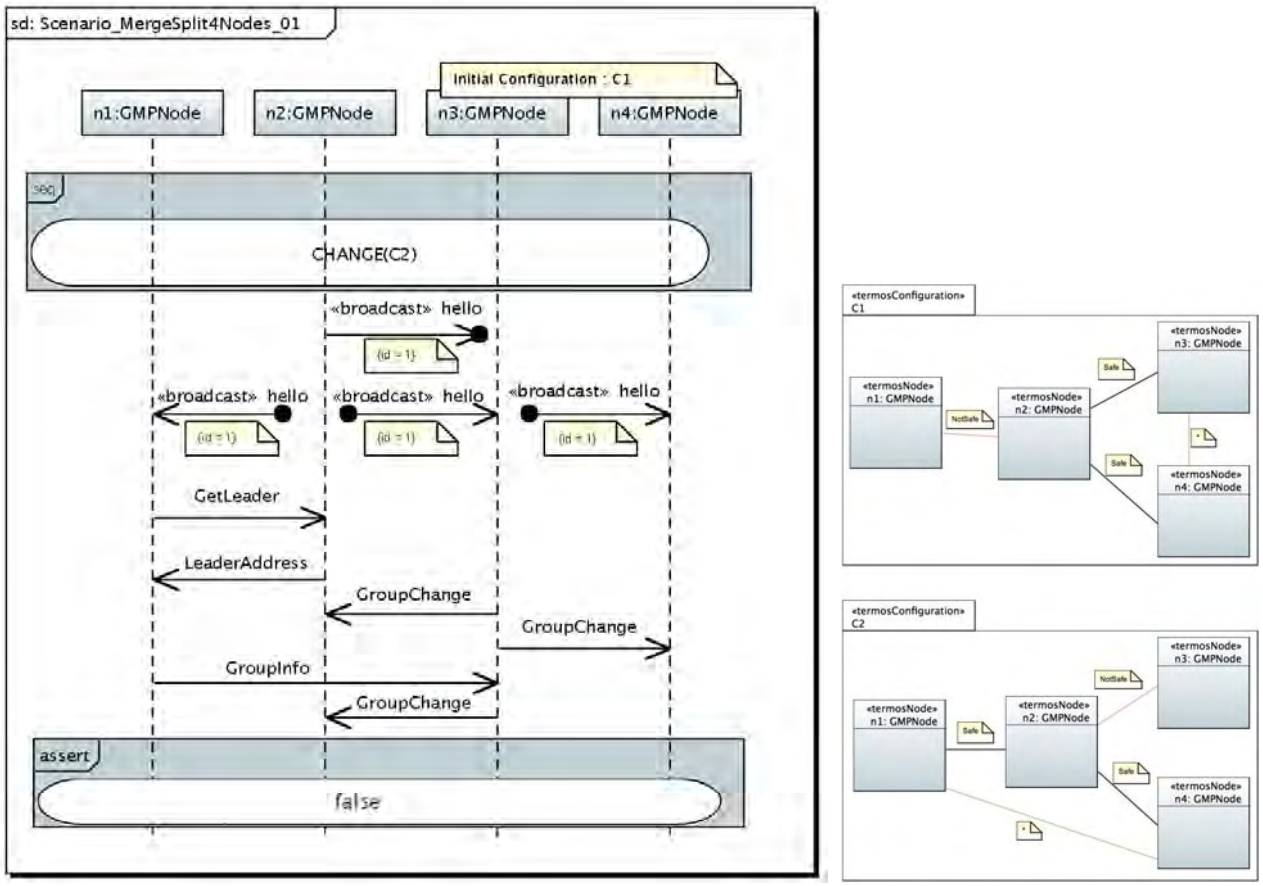
L'exigence négative de ce scénario signifie qu'on souhaite ne jamais observer cet entrelacement de messages.

Scissions de groupe concurrentes : Trois nœuds ($n1$, $n2$ et $n3$) font partie du même groupe. Lorsque la configuration $C2$ apparaît, les trois nœuds s'éloignent les uns des autres et ne sont plus à distance de sécurité. Une scission se déclenche pour former trois groupes distincts. Le scénario vérifie si les nœuds $n2$ et $n3$ ont le même identifiant de configuration mais des groupes différents. C'est-à-dire s'ils n'ont pas le même leader et s'ils n'ont aucun membre du groupe en commun. Les configurations spatiales de ce scénario ont une certaine symétrie. Les occurrences des vues spatiales comporteront l'ensemble des permutations possibles entre les trois nœuds. Ce scénario est présenté Figure 7.11.

L'exigence positive de ce scénario signifie : A chaque fois que trois nœuds s'éloignent l'un de l'autre et ne sont plus à distance de sécurité une scission se déclenche, alors ils doivent avoir des leader différents et n'avoir aucun membre en commun.

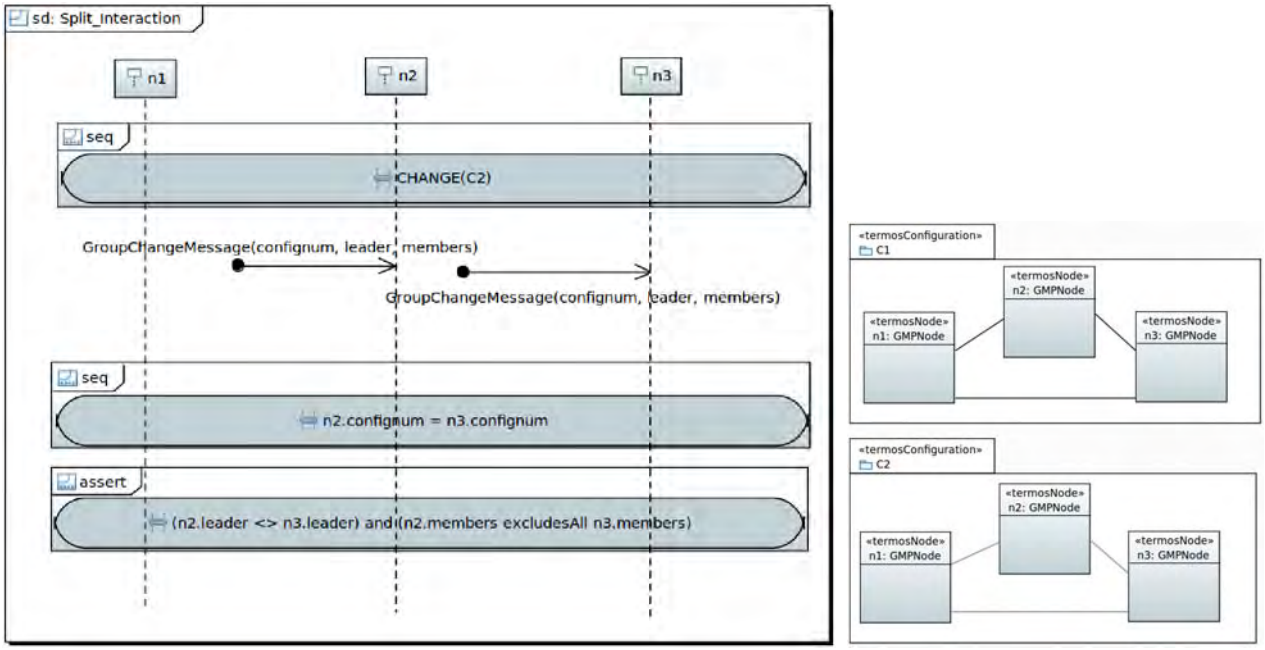
Scission incohérente : Deux nœuds $n1$ et $n2$ sont dans le même groupe. Soudain, sans changement de configuration, $n1$ reçoit un message lui indiquant une nouvelle composition du groupe dans laquelle le nœud $n2$ n'est pas présent alors qu'ils sont à distance de sécurité l'un de l'autre. Ce scénario est présenté Figure 7.12

L'exigence négative de ce scénario signifie que lorsque deux nœuds sont à distance de sécurité l'un de l'autre et sont dans le même groupe, on ne souhaite jamais observer de scission du groupe.



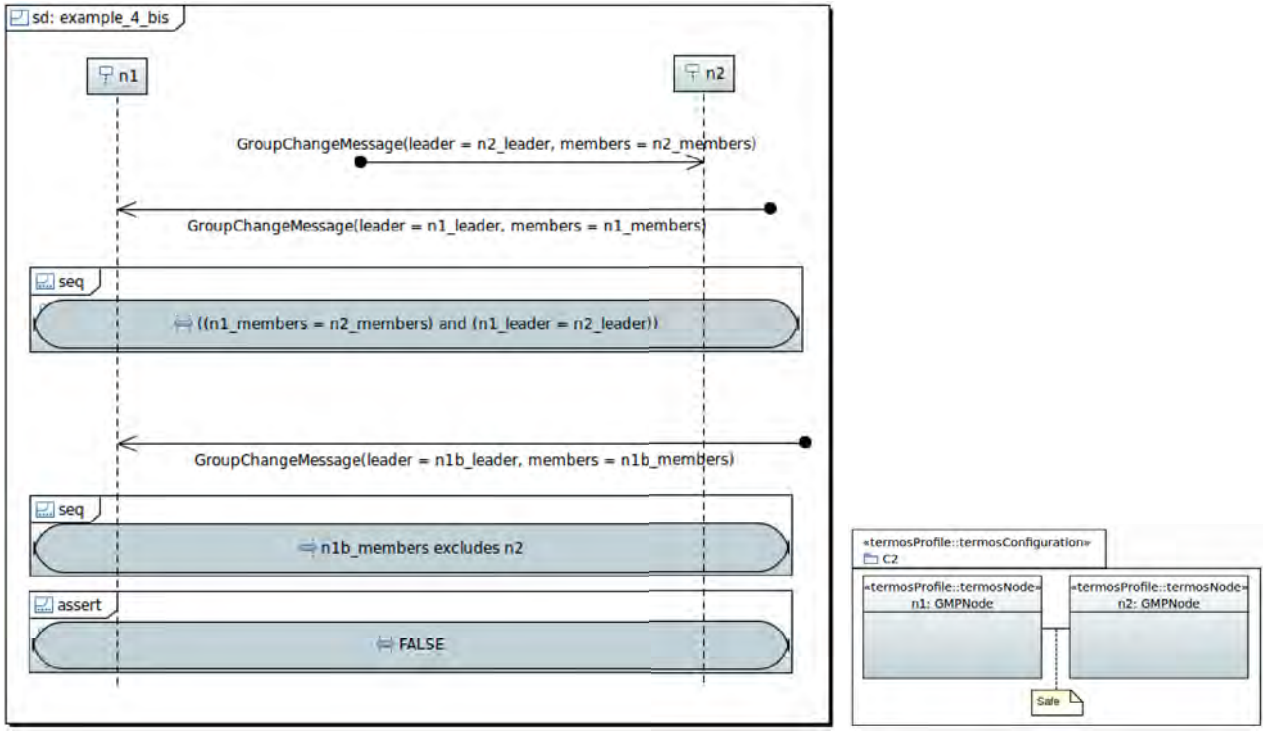
(a) Vue événementielle (b) Vue spatiale

FIGURE 7.10 – Scénario d'une fusion/scission concurrente



(a) Vue événementielle (b) Vue spatiale

FIGURE 7.11 – Scénario d'une scission concurrente



(a) Vue événementielle (b) Vue spatiale

FIGURE 7.12 – Scénario d'une scission incorrecte

7.5 Expérimentation

7.5.1 Plate-forme de l'étude de cas

L'implémentation du *GMP* étudié est réalisée dans *LIME*, un intergiciel pour les systèmes mobiles [1]. Cette implémentation n'est pas triviale : chaque nœud se compose d'environ 4000 lignes de code Java, contient 22 classes et implique 6 threads concurrents. Nous pouvons constater que ce n'est pas un exemple simple, et qu'il présente un réel intérêt du point de vue de la validation.

La version fournie du *GMP* permet l'exécution de plusieurs nœuds sur un même support physique d'exécution, ce qui limite les contraintes de mise en œuvre de la plate-forme de test. Chaque nœud comporte un générateur aléatoire de position permettant de simuler le déplacement des nœuds appelé *FakeGPS*. Afin de simuler le fonctionnement d'un ensemble de nœuds équipé de cette implémentation du *GMP*, nous avons repris l'architecture de la plate-forme de test initialement présentée dans la Figure 7.1 et nous l'avons adaptée au *GMP* comme présenté dans la Figure 7.13. Nous avons remplacé le *FakeGPS* pas un module recevant les coordonnées *GPS* directement du gestionnaire de contexte. Cette version permettant déjà les échanges de messages sans nécessiter de simulateur réseau, nous avons seulement connecté les nœuds sur un même réseau.

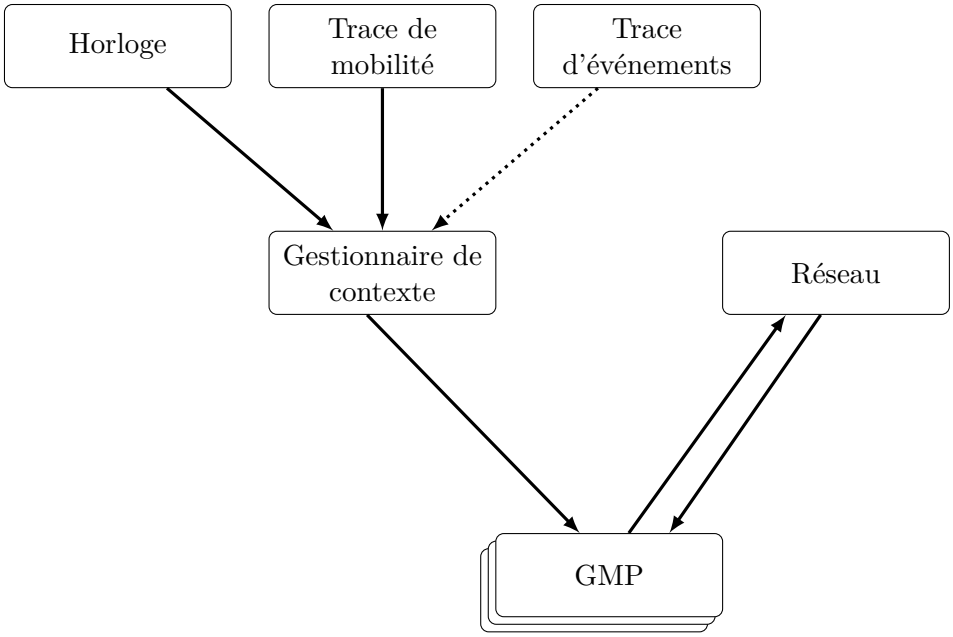


FIGURE 7.13 – Vue schématique de la plate-forme de test du *GMP*

De même que pour le remplacement du *FakeGPS*, nous avons mis en place la possibilité d'échanger des informations d'état entre les nœuds et le gestionnaire de contexte. Cela permet de démarrer, arrêter ou mettre en pause un nœud au cours

1. http://lime.sourceforge.net/

de l'exécution pour simuler, par exemple, une perte de réseau ou un niveau de batterie insuffisant pour communiquer. Enfin la collecte de traces d'exécution s'est faite en instrumentant l'ensemble des appels de fonction d'émission et de réception de messages de l'application. Cela permet d'enregistrer une copie de l'ensemble des événements d'émission et de réception de messages dans un fichier de trace.

Grâce à ces modifications du *GMP*, nous sommes capables de contrôler l'exécution de chaque nœuds en contrôlant son état et ses déplacements. Nous sommes aussi capables de capturer l'ensemble des communications entrantes et sortantes de chaque nœud.

Dans l'architecture de la plate-forme de simulation présentée Figure 7.13, le gestionnaire de contexte joue le rôle de chef d'orchestre de la simulation. Il utilise pour cela une trace de mobilité préalablement générée et une trace d'événements contextuels permettant d'agir sur les nœuds pendant la simulation. Le gestionnaire de contexte utilise aussi une horloge pour gérer la simulation.

Nous pouvons désormais simuler le fonctionnement d'un système composé d'un ensemble de nœuds du *GMP* et de capturer des traces d'exécutions de ce système. Par conséquent, nous pouvons vérifier nos scénarios *TERMOS* sur ces traces.

7.5.2 Résultats des tests

Les propriétés et les scénarios présentés dans la section précédente ont été testés sur une exécution du protocole *GMP* composée de 16 nœuds durant 15 minutes. Cette exécution a produit une trace contenant environ 900 changements de configurations spatiales et environ 500 000 événements représentant des émissions et des réceptions de messages. Les résultats sont résumés dans le Tableau 7.2.

Pour chaque scénario, la première étape consiste à rechercher les occurrences de la séquence de configurations spatiales dans la trace. Le nombre d'occurrences trouvé pour chaque scénario est indiqué dans la deuxième colonne du Tableau 7.2. Ensuite, à partir de chaque occurrence, l'automate représentant le scénario est vérifié une ou plusieurs fois. En effet, durant la vérification d'un automate, nous détectons l'occurrence d'événements *initiaux* et chacune déclenche une vérification indépendante du scénario. Le nombre de vérifications effectivement réalisé est indiqué dans la troisième colonne du Tableau 7.2. Chaque vérification conduit à un verdict. Ils sont regroupés en trois catégories : valides (*stringent accept*), non conclusives (*trivial accept*) et invalides (*reject*).

Cette étude de cas nous a permis de mettre en œuvre notre processus de test de bout en bout. Il est important de noter que notre objectif n'est pas de trouver des défauts à un prototype de recherche mais plutôt de vérifier notre méthode sur un exemple non-trivial de systèmes mobiles.

Néanmoins, l'analyse de ce tableau permet de mettre en évidence certains points de notre méthode. Tout d'abord en comparant les deux colonnes de *matchs*, on pourra remarquer l'utilité de la phase de recherche de configurations spatiales à l'exclusion des trois cas où la vue spatiale ne contient qu'un seul nœud. Ensuite, au moins une exécution de l'automate de chaque scénario s'est conclue par un verdict

TABLE 7.2 – Résumé des résultats de validation des scénarios sur une trace contenant environ 500 000 événements.

Tested scenario	**Matchs**		**Accept**		**Reject**
	Spatial	Event	Stringent	Trivial	Reject
Local monotonicity	16	3 608	3 478	16	114
Self inclusion	16	3 608	3 606	0	2
Membership change justification	16	3 608	3 495	16	97
Membership agreement	3 116	36 460	16 186	20 240	34
*Wrong split**	8 768	53 702	0	53 098	604
*Concurrent merge**	2 450	2 487	0	2 336	151
Concurrent split	162	569	52	387	130

invalide (*reject*). Maintenant, et à l'aide des outils d'analyse de traces que nous avons mis en œuvre, l'utilisateur peut procéder à l'analyse de ces cas de verdict afin de détecter des fautes dans l'application testée.

7.6 Conclusion

La validation d'applications mobiles passe par la mise en œuvre d'une plate-forme de test. Cette plate-forme inclut un simulateur de réseau pour assurer une distribution réaliste des messages, un gestionnaire de contexte pour fournir les informations contextuelles (positions dans notre cas) et un support d'exécution pour chaque nœud.

La collecte de trace nécessite une instrumentation de l'application ou du support d'exécution de l'application. Nous avons fait le choix de modifier l'application pour l'instrumenter dans la mesure où elle ne modifiait pas son fonctionnement nominal. Dans une autre mesure il aurait été préférable de modifier le support d'exécution à l'aide d'outils de virtualisation pour créer une plate-forme générique et non dépendante de l'application à tester.

Les résultats obtenus ont montré l'efficacité de notre méthode de validation d'applications mobile à partir de scénarios d'exigences et des traces d'exécutions.

Chapitre 8

Conclusion et perspectives

Nous présentons dans ce chapitre les conclusions sur le travail effectué au cours de cette thèse et les pistes de recherches qui en découlent.

L'objectif de cette thèse était de contribuer à l'élaboration d'une méthode de test de systèmes mobiles. L'approche développée est fondée sur la description de tests à l'aide de scénarios et leurs vérifications sur une trace d'exécution. Un scénario modélise le comportement et les interactions que l'on souhaite observer entre un ensemble de nœuds. Les caractéristiques des systèmes mobiles nous ont conduit à représenter un scénario sous deux points de vue différents et complémentaires. Un premier représente des événements de communications entre les nœuds et un second représente la topologie des liens entre ces nœuds.

Cette méthode de test de systèmes mobiles dont le processus est illustré dans la Figure 4.1 de la page 47 repose sur l'utilisation du langage *TERMOS* (*TEst Requirement language for MObile Settings*) pour lequel une spécification a été réalisée préalablement à nos travaux dans le cadre du projet européen HIDENETS [Huszrel 2008]. Dans le cadre de nos travaux, nous avons finalisé la définition du langage, nous avons étendu les spécifications de ce langage avec la création d'une grammaire permettant la gestion des prédicats dans les scénarios et nous avons mis en œuvre ce langage au sein de l'atelier *UML Papyrus*.

Les travaux présentés dans cette thèse ont fait l'objet de deux publications, une première se focalise sur les optimisations de l'outil *GraphSeq* [André 2013a]. Une deuxième présente plus généralement le langage *TERMOS* et sa mise en œuvre [André 2013b].

L'intégration des outils *GraphSeq* et *TERMOS* nous a permis de réaliser une chaîne complète permettant de spécifier des scénarios de test et de les vérifier sur des traces d'exécutions de façon automatique. Ce travail d'intégration nous a confronté a des problèmes de passage à l'échelle de certains algorithmes présents dans *GraphSeq* lors de tests sur des traces concrètes. Nous avons résolu ce problème en proposant une optimisation de l'algorithme et en modifiant les structures de données. *GraphSeq* pourrait encore être amélioré. Par exemple il serait intéressant d'enlever la limite du nombre d'attributs de nœuds qui est actuellement de deux attributs et de prendre différents types d'attributs. Pour le traitements de graphes de grande taille, il pourrait aussi être envisagé de paralléliser *GraphSeq*.

Nous avons conduit une expérimentation sur un protocole d'appartenance de groupe afin d'évaluer notre approche. Nous avons pour cela créé une plate-forme de simulation incluant un simulateur de réseau pour assurer une distribution réaliste des messages, un gestionnaire de contexte pour fournir les informations de posi-

tions et un support d'exécution pour chaque nœud. Sur chaque nœud une version instrumentée de l'application à tester a été mise en œuvre.

Des perspectives à nos travaux restent à explorer, surtout en ce qui concerne la prise en compte du temps, la prise en compte de données contextuelles plus riches et la conception d'une plate-forme de test.

Une première perspective d'évolution pourrait être la gestion du temps dans les scénarios. Actuellement il n'y a aucune contrainte de durée minimale ou maximale d'un scénario. Cela permettrait dans le cas du *GMP* par exemple de fixer une borne temporelle dans la scission d'un groupe et de détecter des violations du protocole. Cela permettrait aussi de prendre en compte le caractère stable ou instable des configurations spatiales rencontrées.

Une deuxième perspective d'évolution de nos travaux serait d'étendre la prise en compte du contexte dans les scénarios qui se limite pour le moment à des informations de géolocalisation. En effet, les systèmes ubiquitaires s'adaptent à un contexte de plus en plus complexe, dû au niveau d'instrumentation élevé de l'environnement dans lequel ils évoluent.

Enfin un problème auquel nous devrions porter attention concerne l'utilisation d'une plate-forme de test basée sur la simulation. Une simulation ne représentera jamais parfaitement un environnement réel. Dans le cadre de la validation et la vérification d'applications, quelle confiance pouvons-nous accorder à un outil de simulation ? D'un autre coté, dans un environnement réel comment pouvons nous observer le comportement d'un système mobile ?

Annexe A

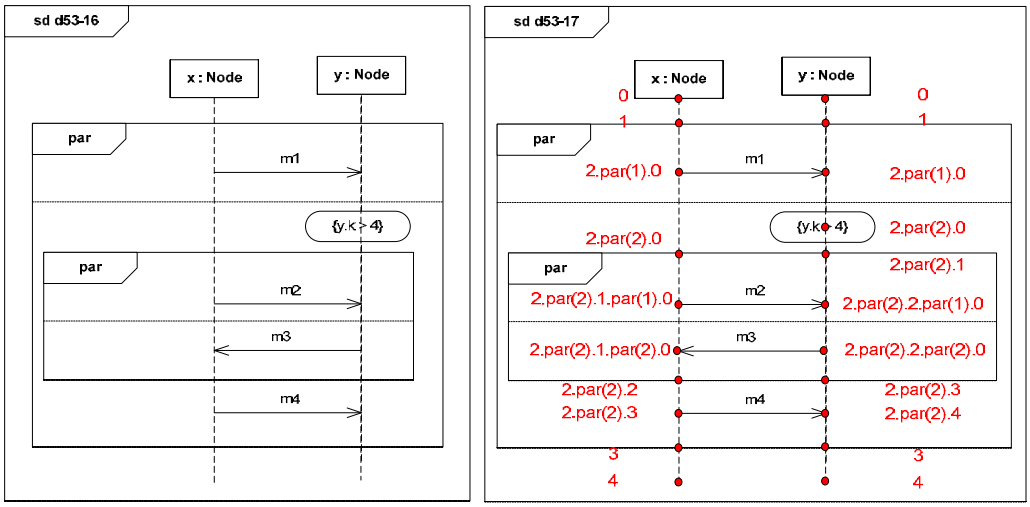
Construction de l'automate

A.1 Analyse du diagramme

Définition 1 *Les blocs de construction élémentaires d'un scénario* TERMOS *sont appelés des* atomes. *Les éléments suivants représentent des* atomes *:*

- *les sommets des lignes de vie, noté $\bot_l$ pour la ligne de vie l,*
- *l'extrémité des lignes de vie, notée $\top_l$ pour la ligne de vie l,*
- *les événements d'émission et de réception de messages,*
- *les invariants d'état,*
- *les changements de configuration spatiale,*
- *l'entrée et la sortie des fragments combinés,*
- *les gardes.*

L'ordre des *atomes* d'une ligne de vie est défini par leurs positions. Lors de l'utilisation de fragments combinés de type *parallel* ou *alternate*, l'ordre des *atomes* représentés sur le diagramme ne signifie plus nécessairement qu'il y a une relation temporelle entre deux *atomes*.



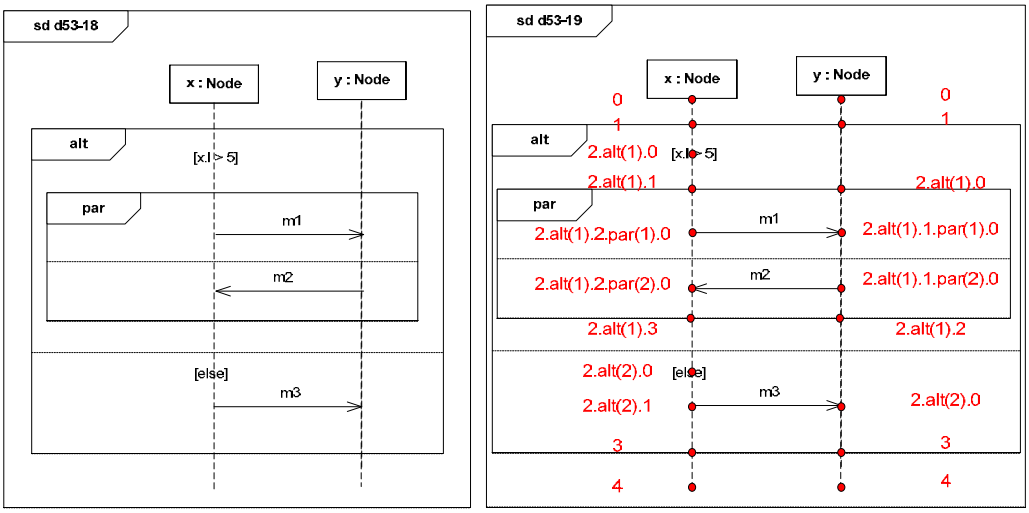
(a) Scénario (b) Atomes

Figure A.1 – Exemple d'assignation d'atomes pour des fragments *par*

Par exemple, sur la Figure A.1a les messages $m2$ et $m3$ sont représentés l'un après l'autre mais puisqu'ils sont dans deux opérandes différentes d'un fragment combiné de type *parallel*, il n'y a pas d'ordre prédéterminé des événements. Ce qui explique que, la position des *atomes* dans une ligne de vie est notée à l'aide d'une expression d'une façon similaire à l'approche utilisée dans [Küster-Filipe 2006].

La Figure A.1a représente un scénario et la Figure A.1b contient ce même scénario annoté avec les positions des *atomes*. L'idée principale de cette notation est la suivante. A l'intérieur du fragment principal ou dans une opérande, chaque *atome* se voit assigné un numéro correspondant à sa position à l'intérieur de l'opérande ou du fragment principal. A l'entrée dans un fragment combiné, l'expression de la position est complétée par une information concernant l'opérande dans laquelle se trouve l'*atome*.

Définition 2 *A chaque* atome *est liée une position sur la ligne de vie. Cette position est de la forme path.id, où path est une chaîne de caractères identifiant le fragment combiné dans lequel l'*atome *se trouve et id est un entier indiquant l'ordre de l'*atome *dans le fragment. Le* path *peut être vide si l'*atome *se trouve dans le fragment principal du diagramme, sinon il est de la forme p.opr(o) où p est la position du fragment combiné dans lequel est l'*atome*, opr est l'opérateur du fragment et o est le numéro de l'opérande dans lequel est l'*atome*.*



(a) Scénario (b) Atomes

FIGURE A.2 – Exemple d'assignation d'atomes pour des fragments *alt*

Par exemple, la Figure A.2 montre comment sont attribués les *atomes* dans le cadre de fragment de type *alt*. Les gardes des opérandes sont regroupées avec l'*atome* suivant de la même ligne de vie pour former un *cluster*. Il faut néanmoins faire attention aux cas où il n'y a pas d'*atome* à l'intérieur de la garde par exemple dans le cas d'un [*else*] vide. Dans ce cas le *cluster* contient seulement la garde. Dans

la spécification *UML* originale, il peut y avoir plusieurs successeurs immédiats à un *atome* sur une même ligne de vie par exemple si l'*atome* est juste avant un *par* ayant plusieurs opérandes. Avec l'introduction d'un *atome* séparé pour le début de chaque fragment, cela n'est plus le cas dans *TERMOS*.

Définition 3 *Si un* atome *a est une garde ayant la position p.i et si il existe un* atome *ayant la position* $p.(i+1)$*, alors les deux* atomes *forment un* cluster*. Tout autre* atome *est un* cluster *avec seulement l'*atome *à l'intérieur.*

Par exemple Figure A.2b, les atomes $2.alt(2).0$ et $2.alt(2).1$ forment un cluster.

Définition 4 *Un* cluster *comportant plusieurs éléments prend comme position le minimum des positions des* atomes *le constituant. On définit la fonction* $location(cl)$ *qui renvoie ce minimum, avec* $min(p.i), p.(i+1)) = p.i$*.*

Par exemple Figure A.2b, le cluster formé par les atomes $2.alt(2).0$ et $2.alt(2).1$ à pour position la position de $2.alt((2).0$.

Plusieurs éléments comme les changements de configuration ou l'entrée dans un fragment combiné recouvrent plusieurs lignes de vie et nécessitent une synchronisation entre elles. Pour cela les *clusters* correspondant à ces éléments doivent être regroupés ensemble, les classes simultanées (*simClasses*) ont été définies pour cela.

Définition 5 *Une classe simultanée ou* simClasse *est un ensemble de* clusters *appartenant à différentes lignes de vie. Les* clusters *représentant les éléments suivants forment ensemble une* simClass*, tout autre* cluster *est une* simClass *:*

- *le début d'un même fragment combiné,*
- *la fin d'un même fragment combiné,*
- *le même changement de configuration,*
- *le même invariant d'état.*

La Figure A.3 résume comment sont définis les *atomes*, *clusters* et *simClasses*. Pour résumer, les *atomes* sont des points sur une ligne de vie ; les *clusters* regroupent les *atomes* simultanés d'une même ligne de vie ; les *simClasses* regroupent les *clusters* qui sont simultanés au niveau de tout le diagramme. Ce qui veux dire que les *simClasses* recouvrant plusieurs lignes de vies représentent une synchronisation entre elles.

Deux relations sont définies entre les *clusters* d'une même ligne de vie : une relation de causalité notée $\prec$ qui définit un ordre partiel entre les différents clusters, et une relation de conflit notée # qui définit quels événements ne peuvent pas apparaître dans une même trace, comme les *atomes* de différents opérantes d'un fragment *alt* par exemple.

Définition 6 (Causalité locale) *Soient* cl_1 *et* cl_2 *deux* clusters *de la ligne de vie l ayant des positions de la forme* $location(cl_1) = p_1.i.p_2$ *et* $location(cl_2) = p_1.j.p_3$ *avec* p_1*,* p_2 *et* p_3 *pouvant être la chaîne vide, alors :*

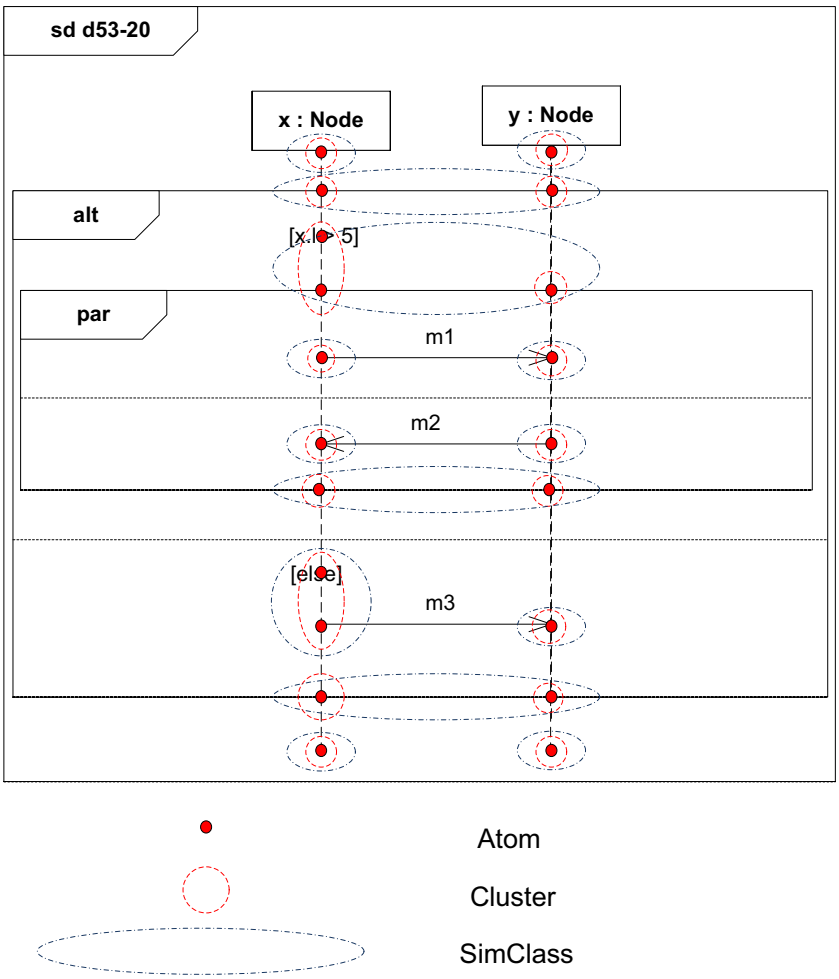


FIGURE A.3 – *Atomes*, *clusters* et *simClasses* sur un diagramme

$$cl_1 \prec cl_2 \ si \ i < j$$

Définition 7 (Conflit local) *Soient* cl_1 *et* cl_2 *deux* clusters *de la ligne de vie* l *ayant des positions de la forme* $location(cl_1) = p_1.alt(i).p_2$ *et* $location(cl_2) = p_1.alt(j).p_3$ *alors :*

$$cl_1 \# cl_2 \ si \ i \neq j$$

Définition 8 *La fonction* predecessors *calcule les prédécesseurs immédiats d'un* cluster *cl sur sa ligne de vie l.*

$$predecessors(cl) := \{cl' \in Clusters(l) | cl' \prec cl \wedge \neg cl'' \in Clusters(l) : cl' \prec cl'' \prec cl\}$$

Par exemple, dans la Figure A.2, le prédécesseur du *cluster* ayant la position $2.alt(1).0$ est le *cluster* ayant la position 1, tandis que les prédécesseurs du *cluster* ayant la position 3 sont les *clusters* ayant les positions $2.alt(2).0$ et $2.alt(1).3$.

Pour gérer la causalité entre des *clusters* de différentes ligne de vie, les événements d'émission et de réceptions de messages sont liés entre eux. Pour cela, un identifiant symbolique unique de la forme $\$i$ où i est un entier est assigné à chaque message. Soit ID l'ensemble des identifiants de messages et soit $MessageSends(sd)$ et $MessagesReceives(sd)$ les ensembles de tous les messages émis et reçus du diagramme *sd*.

Définition 9 *La fonction* messageID *retourne pour chaque* atome *d'émission ou de réception, l'identifiant du message correspondant.*

$$messageID : MessageSend(sd) \cup MessageReceives(sd) \rightarrow ID$$

La fonction *predecessors* est étendue aux *simClasses* et prend alors en compte les relations de causalités induites par la connexion entre les messages émis et reçus. Soit $simClasses(sd)$ l'ensemble des *simClasses* du diagramme *sd*.

Définition 10 *Les prédécesseurs immédiats d'une* simClasses *scl dans le diagramme de séquence sd sont renvoyés par la fonction* prerequisite.

$$prerequisite(scl) := \{scl' \in simClasses(sd) | \exists cl \in scl, \exists cl' \in scl' : cl' \in predecessors(cl) \vee (\exists a \in cl \cap messageReceives(sd), \exists a' \in cl' \cap messagesSend(sd) : messageID(a) = messageID(a'))\}$$

La relation de conflit est étendue aux *simClasses* en utilisant la fonction *conflict*.

Définition 11 *La fonction* conflict *retourne les* simClasses *qui ont des* clusters *qui sont dans un opérande différent du fragment* alt *dans lequel se trouve les* clusters *de la* simClass *scl*.

$$conflict(scl) : simClass \rightarrow \wp(simClass)$$

Nous avons donc $scl_2 \in conflict(scl_1)$ si ces deux *simClasses* contiennent respectivement un *cluster* cl_1 et cl_2 tel que :

- la position du *cluster* cl_1 sur la ligne de vie l_1 est de la forme $prefix_1.k_1.alt(i).suffix_1$. cl_1 est une opérande *alt*.
- la position du *cluster* cl_2 sur la ligne de vie l_2 est de la forme $prefix_2.k_2.alt(j).suffix_2$ avec $j \neq i$.
- les *clusters* $cl'_1 \in clusters(l_1)$ ayant la position $prefix_1.k_1 - 1$ et $cl'2 \in clusters(l_2)$ ayant la position $prefix_2.k_2 - 1$ appartiennent à la même *simClass*.

Notons que la relation de conflit local est un cas particulier de la relation de conflit global en prenant $l_1 = l_2$.

A.2 Construction de l'automate

La construction d'un automate basé sur le scénario d'exigence est réalisé à l'aide d'un algorithme de déroulement. Le principe de celui-ci est de dérouler progressivement l'ensemble des *simClasses* du diagramme jusqu'à ce qu'elles soit toutes traitées.

Un automate symbolique est défini par le tuple $(\Sigma, \mathcal{Q}, q_0, \mathcal{F}_\mathcal{T}, \mathcal{F}_\mathcal{S}, \rightarrow, Var, Def)$ tel que :

- Σ est l'ensemble des labels des transitions. L'utilisation de variables de Var est possible dans les labels.
- $\mathcal{Q}$ est l'ensemble des états
- q_0 est l'état initial de l'automate
- $\mathcal{F}_\mathcal{T} \subseteq \mathcal{Q}$ et $\mathcal{F}_\mathcal{S} \subseteq \mathcal{Q}$ sont deux ensembles disjoints d'états d'acceptation. $\mathcal{F}_\mathcal{T}$ contient les états qui expriment la satisfaction triviale des exigences, alors que $\mathcal{F}_\mathcal{S}$ contient ceux qui expriment une satisfaction stricte des exigences.
- $\rightarrow \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ est l'ensemble des transitions.
- Var est l'ensemble des variables extraits du scénario. Il contient les variables apparaissant dans le vue spatiale ainsi que celles présentes dans la vue événementielle.
- $Def \subseteq \mathcal{Q} \times Var$ est le sous-ensemble des variables définies pour chaque état.

L'algorithme utilise la notion de phase, définie par le tuple $(History, Ready, Cut, Variables)$ telle que :

- $History$ est l'ensemble des *simClasses* qui ont été déroulées,
- $Ready$ est l'ensemble des *simClasses* qui sont prêtes à être déroulées,
- Cut représente la frontière entre les éléments déjà déroulés et ceux actuellement prêt à être déroulés. C'est un tuple de *clusters* $(c_1, ..., c_n)$ où c_i est un *cluster* de la ligne de vie i,
- $Variables$ est l'ensemble des variables qui ont déjà été valuées.

Les phases vont correspondre aux états de l'automate. Un nom unique sera assigné à chaque état à l'aide de la fonction $state(ph : phase)$. Notons qu'on suppose que si une phase est rencontrée plusieurs fois, la fonction retournera le même nom d'état.

La phase initiale considérée par l'algorithme est définie par le tuple $(History_0, Ready_0, Cut_0, Variables_0$ tel que :

- $History_0 = \{\{\{\perp_1\}\}, \{\{\perp_2\}\}, ..., \{\{\perp_n\}\}\}$
- $Ready_0 = \{scl \in simClasses(sd) \mid prerequisite(scl) \subseteq History_0\}$
- $Cut_0 = (\{\perp_1\}, \{\perp_2\}, ..., \{\perp_n\})$
- $Variables_0$ = l'ensemble des variables apparaissant dans la configuration initiale du scénario y compris les identifiants symboliques des nœuds participants au scénario.

Lors de la phase initiale, seuls les sommets des lignes de vies sont déroulés. On suppose que le système est dans la configuration spatiale initiale. L'état $state(phase_0)$ est ajouté à l'ensemble $\mathcal{Q}$ de l'automate ainsi qu'au sous-ensemble $\mathcal{F}_\mathcal{T}$. Partant de la phase initiale, l'algorithme commence à calculer les phases qui succèdent a la phase initiale.

Soit une phase $ph = (History_i, Ready_i, Cut_i, Variables_i)$, la fonction *step* retourne la phase suivante en franchissant une des *simClasses* contenue dans $Ready_i$. La fonction $step(ph, scl)$ retournant $ph' = (History_{i+1}, Ready_{i+1}, Cut_{i+1}, Variables_{i+1})$ est définie de la façon suivante :

- $History_{i+1} = History_i \cup \{scl\} \cup conflict(scl)$, ainsi les *simClasses* en conflit et la *simClasse scl* sont considérées comme déroulées.
- $Ready_{i+1} = \{scl' \in simClasses(sd) \setminus \{\{\{\top_1\}\}, ..., \{\{\top_n\}\}\} \mid prerequisite(scl') \subseteq History_{i+1} \wedge scl' \notin History_{i+1}\}$
- $Cut_{i+1} = \{cl'_1, ...cl'_n\}$ est produit à partir de $Cut_i = \{cl_1, ...cl_n\}$ avec $cl'_j = cl_j$ si la ligne de vie j n'est concernée par aucun des *clusters* de la *simClasse* déroulée. Pour chacune des autres lignes de vies impliquées k, les éléments cl'_k sont remplacés par les *clusters* correspondant de la *simClass* déroulée.
- $Variables_{i+1}$ est l'union des $Variables_i$ avec l'ensemble des nouvelles variables valuées. Il n'existe de nouvelles variables valuées que si ph contient des événements de communication ou de changement de configuration.

La nouvelle phase peut correspondre à un état d'acceptation ou pas. Ceci est géré par l'algorithme que nous présentons à l'aide de la variable *currentMode*. Avant l'entrée dans le fragment combiné *assert*, le mode courant est *AcceptTrivial* et les états générés sont ajoutés à $\mathcal{F}_\mathcal{T}$. A l'entrée du fragment *assert*, le mode passe à la valeur *Reject*. A la sortie du *assert*, le mode passe à *AcceptStringent* et le successeur est ajouté à l'ensemble $\mathcal{F}_\mathcal{S}$.

Les labels des transitions entre deux états de l'automate contiennent typiquement une condition décrivant des événements de communication ou des changements de configuration. Le label de la transition peut avoir en plus une condition de garde ou une contrainte. S'il existe de nouvelles variables évaluées, alors le label de la transition contient une mention explicite de mise à jour. Par exemple, supposons que nous sommes actuellement en train de dérouler une garde $x > 3$ et un événement d'émission de message $(!m(x), n_1, n_2, \$4)$. Supposons de plus que les valeurs de x, n_1 , et n_2 sont actuellement définies, tandis que la valeur de \$4 assignée lors de l'analyse préliminaire ne l'est pas encore. Alors, le label de la transition correspondante sera : $x > 3 \wedge (!m(x), n_1, n_2, \$4)[update(\$4)]$. Ceci pourrait être interprété par : Si $x > 3$ à l'évaluation courante, et l'événement suivant de la trace correspond à $(!m(x), n_1, n_2, \$4)$, alors la transition est permise. L'événement de la trace est alors consommé, et l'évaluation courante est mise à jour par la valeur concrète de l'identifiant du message de cet événement. L'entrée et la sortie des fragments sont simplement représentées par des transitions true.

Des rebouclages doivent être ajoutés sur les états à partir du moment où il existe une transition contenant un événement provenant de la trace que ce soit un

événement de communication ou un changement de configuration. Par exemple si le prochain événement de la trace ne correspond pas à $(!m(x), n_1, n_2, \$4)$, sommes nous autorisé à consommer cet événement et à rester dans le même état ? Inversement s'il correspond, est-ce que l'on a le choix de rester dans ce même état ? La réponse à cette deuxième question est négative, d'où le label du rebouclage $\neg(!m(x), n_1, n_2, \$4)$. La réponse à la question initiale est généralement positive mais peut dans certains cas être négative par exemple lors de l'utilisation de fragment *consider*. Pour rappel, notre interprétation d'un fragment *consider* tel que *consider*$\{m\}$ est la suivante : l'envoi de messages de type m est interdit pour toutes les lignes de vies mais il est autorisé de recevoir ce type de message si l'émetteur ne fait pas partie des lignes de vies pour lesquelles l'envoi est interdit. Le rebouclage n'est donc utile que pour les événements d'envoi. Finalement puisque les traces sur lesquelles seront vérifiées les automates ne sont pas les traces brutes mais le résultat d'un premier traitement à l'aide d'un outil d'appariement de séquence de graphe, seul les changements de configuration attendus seront présents dans la trace. Afin de garantir cela, l'automate interdit explicitement tout changement inattendu de configuration à l'aide de rebouclage de type $\neg CHANGE(-)$.

Algorithme A.1 – Algorithme de déroulage

```
// Initialization
Phases := {Phase0}
Q := {STATE(Phase0)} // set of states
q0 := STATE(Phase0)  // initial state
FT := {STATE(Phase0)}
FS := ∅
currentMode := AcceptTrivial
SelfLoopLabel := "¬CHANGE(−)" // unexpected config changes should never
    ↪ happen
// Unwinding loop
While (Phases ≠ ∅)
  Extract ph = (Historyi, Readyi, Cuti, Variablesi) from Phases
  If (Readyi ≠ ∅)
    SelfLoop := false
    AddedSelfLabel = ""
    For all sc  Readyi
      successor := STEP(ph, sc)
      // compute the label of the triggered transition
      UpdatedVariables := ∅
      Label := ""
      For all cl  sc
        For all a  cl
          Switch a
            Case entering of an assert box:
              // Note: changes the CurrentMode to reject
              If (Label ="") then
                // first atom processed
                currentMode = reject
                Label = "true"
              Endif // Else nothing to do
            Case exiting an assert box:
              // Note: changes the CurrentMode to AcceptStringent
              If (Label ="") then
                // first atom processed
```

```
        currentMode = AcceptStringent
        Label = "true"
      Endif // Else nothing to do
    Case entering of a consider box:
      //Note: changes SelfLoopLabel by forbidding considered send events
      If (Label ="") then
        // first atom processed
        For all considered message name m
          For all symbolic node id li in the current valuation
            build label ll of the form: ¬(!m(−), li, −)
            SelfLoopLabel := SelfLoopLabel conjoined with ll
          End For
        End For
        Label = "true"
      Endif // Else nothing to do
    Case exiting a consider box:
      //Note: changes SelfLoopLabel by discarding the forbidden events
      If (Label ="") then
        // first atom processed
        SelfLoopLabel := "¬CHANGE(−)"
        Label = "true"
      Endif // if not the first atom, nothing to do
    Case entering or exiting a par or alt box:
      If (Label ="") then
        // first atom processed, or other atoms yielded a non empty Label
        Label = "true"
      Endif // if non empty label, no need to conjoin with true
    Case Guard or state invariant:
      Make a label ll with the predicate
      If (ll does not already appear in Label)
        Label := Label conjoined with ll
      Endif
    Case Change in Configuration:
      // Note: change UpdatedVariables to account for new variables in Ci
      // Also, a selfloop is needed. SelfLoopLabel already contains
      // a negated change event
      If (Label ="") then
        // first atom processed
        UpdatedVariables := {variables in new config Ci} \ Variablesi
        SelfLoop := true
        Make a label ll of the form CHANGE(Ci)
        Label = ll
      Endif // if not the first atom, nothing to do
    Case Send or receive event:
      // Note: change UpdatedVariables to account for new variables in the event
      // Also, a selfloop is needed. If SelfLoopLabel does not already
      // contain a negated form of the event (due to a consider),
      // AddedSelfLabel is changed.
      UpdatedVariables := {variables in message parameters or message id} \ Variablesi
      SelfLoop := true
      Make a label ll for the event
      Label := Label conjoined with ll
      If (the atom is receive event, or the atom is a send event that does not appear under the form ¬(!m(−), li, −) in SelfLoopLabel)
```

```
87                  AddedSelfLabel := AddedSelfLabel conjoined with ¬ll
                  Endif // Else the event is already forbidden by an embodying
                      ↪ consider
89              End Switch a
              End For // all atoms of the cluster processed
91          End For // all clusters of the unwound Simclass processed
            // Update the transition set
93          If (UpdatedVariables ≠ ∅)
              Build a label ll of the form [list of updated variables]
95            Append ll to Label
            Endif
97          → := → ∪ {STATE(ph), Label, STATE(successor)}
            // Put successor in automaton states and in Phases
99          Q := Q ∪ {STATE(successor)}
            If (CurrentMode = AcceptTrivial)
101           FT := FT ∪ {STATE(successor)}
            Else if (CurrentMode = AcceptStringent)
103           FS := FS ∪ {STATE(successor)}
            End if
105         Put successor in Phases
        End for // All ready SimClasses processed
107     // Add a self-loop if needed
        If (SelfLoop = true)
109       If (AddedSelfLabel is not empty)
            Build label ll conjoining AddedSelfLabel and SelfLoopLabel
111       Else
            ll = SelfLoopLabel
113       Endif
          → := → ∪ {STATE(ph), ll, STATE(ph)}
115     Endif
      Endif
117 End While
```

L'Algorithme A.1 appliqué au scénario présenté dans la sous-section 3.4.3 (page 37) et décrit graphiquement dans la Figure 3.11 (page 38) permet de produire l'automate présenté Figure A.4.

Les trois types d'états sont représentés d'une manière différente.

— **Reject** un état de rejet est représenté par un cercle simple noir,

— **AcceptTrivial** un état d'acceptation triviale est représenté par un double cercle bleu,

— **AcceptStingent** un état d'acceptation stricte est représenté par un triple cercle rouge.

Le verdict de chaque vérification de l'automate dépend du type d'état dans lequel la vérification s'est terminée. Ce type de représentation graphique n'est utilisée que pour des scénarios de petite dimension mais permet de bien illustrer la transformation mise en œuvre entre le scénario en UML et l'automate.

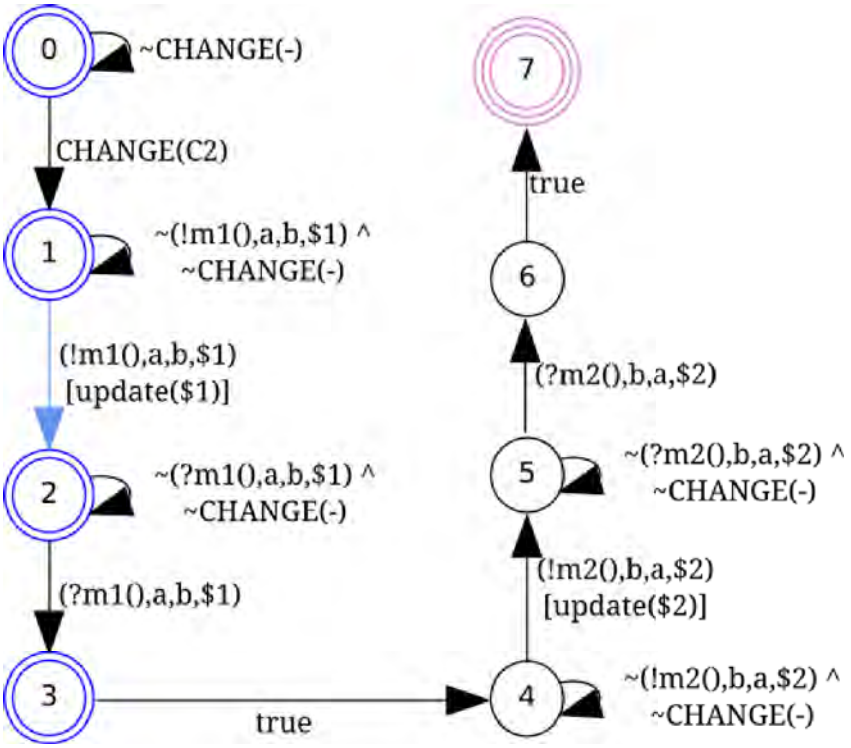


FIGURE A.4 – Automate correspondant au scénario de la Figure 3.11

Bibliographie

[Adrion 1982] W Richards Adrion, Martha A Branstad et John C Cherniavsky. *Validation, verification, and testing of computer software.* ACM Computing Surveys (CSUR), vol. 14, no. 2, pages 159–192, 1982. (Cité en page 11.)

[Ammann 2008] J. Ammann P. et Offutt. Introduction to software testing. Cambridge Univ Pr, 2008. (Cité en page 10.)

[André 2013a] Pierre André, Nicolas Riviere et Hélène Waeselynck. *GraphSeq Revisited : More Efficient Search for Patterns in Mobility Traces.* In Marco Vieira et João Carlos Cunha, editeurs, Dependable Computing - 14th European Workshop, EWDC 2013, Coimbra, Portugal, May 15-16, 2013. Proceedings, volume 7869 of *Lecture Notes in Computer Science*, pages 88–95. Springer, 2013. (Cité en pages 89, 90 et 119.)

[André 2013b] Pierre André, Hélène Waeselynck et Nicolas Rivière. *A UML-Based Environment for Test Scenarios in Mobile Settings.* In 2013 International Conference on Computer, Information and Telecommunication Systems (CITS), 2013. (Cité en page 119.)

[Avizienis 2004] A. Avizienis, J.-C. Laprie, B. Randell et C. Landwehr. *Basic concepts and taxonomy of dependable and secure computing.* vol. 1, no. 1, pages 11–33, 2004. (Cité en page 9.)

[Bai 2003a] F. Bai, Narayanan Sadagopan et A. Helmy. *IMPORTANT : a framework to systematically analyze the Impact of Mobility on Performance of Routing Protocols for Adhoc Networks.* In Proc. INFOCOM 2003. Twenty-Second Annual Joint Conf. of the IEEE Computer and Communications. IEEE Societies, volume 2, pages 825–835, 2003. (Cité en page 16.)

[Bai 2003b] F. Bai, Narayanan Sadagopan et A. Helmy. *IMPORTANT Mobility Generator.* `http://nile.cise.ufl.edu/important/software.htm`, 2003. 2012.06.07. (Cité en page 101.)

[Barton 2003] John J Barton et Vikram Vijayaraghavan. *UBIWISE, a simulator for ubiquitous computing systems design.* Hewlett-Packard Laboratories Palo Alto, â AI HPL-2003-93, 2003. (Cité en page 17.)

[Baumann 1997] Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel et Markus Straβer. *Communication concepts for mobile agent systems.* In Mobile Agents, pages 123–135. Springer, 1997. (Cité en page 8.)

[Baumann 2008] R. Baumann, F. Legendre et P. Sommer. *Generic mobility simulation framework (GMSF).* In Proceeding of the 1st ACM SIGMOBILE workshop on Mobility models, pages 49–56. ACM, 2008. (Cité en page 16.)

[Bellavista 2012] Paolo Bellavista, Antonio Corradi, Mario Fanelli et Luca Foschini. *A survey of context data distribution for mobile ubiquitous systems.* ACM Comput. Surv., vol. 44, no. 4, pages 24 :1–24 :45, Septembre 2012. (Cité en page 9.)

[Bertrand 2011] Frédéric Bertrand et Myriam Maumy. *Tests non paramétriques*. http://www-irma.u-strasbg.fr/~fbertran/enseignement/DUS2_2011/DUS2_CoursNonPara_1.pdf, 06 2011. 2015.06.07. (Cité en page 89.)

[Bettini 2010] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan et Daniele Riboni. *A survey of context modelling and reasoning techniques*. Pervasive and Mobile Computing, vol. 6, no. 2, pages 161 – 180, 2010. Context Modelling, Reasoning and Management. (Cité en page 8.)

[Boytsov 2013] Andrey Boytsov et Arkady Zaslavsky. *Formal verification of context and situation models in pervasive computing*. Pervasive and Mobile Computing, vol. 9, no. 1, pages 98 – 117, 2013. Special Section : Pervasive Sustainability. (Cité en page 10.)

[Briand 2006] Lionel C Briand, Yvan Labiche et Johanne Leduc. *Toward the reverse engineering of UML sequence diagrams for distributed Java software*. Software Engineering, IEEE Transactions on, vol. 32, no. 9, pages 642–663, 2006. (Cité en pages 18 et 22.)

[Cavalli 2004] A. Cavalli, C. Grepet, S. Maag et V. Tortajada. *A validation model for the DSR protocol*. In Proc. 24th Int Distributed Computing Systems Workshops Conf, pages 768–773, 2004. (Cité en page 87.)

[Cavalli 2009] Ana Cavalli, Stephane Maag et Edgardo Montes de Oca. *A passive conformance testing approach for a MANET routing protocol*. In Proceedings of the 2009 ACM symposium on Applied Computing, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM. (Cité en pages 15 et 22.)

[Cavarra 2004] Alessandra Cavarra et Juliana Küster-Filipe. *Formalizing liveness-enriched sequence diagrams using ASMs*. In Abstract State Machines 2004. Advances in Theory and Practice, pages 62–77. Springer, 2004. (Cité en page 34.)

[Choffnes 2005] David R Choffnes et Fabián E Bustamante. *An integrated mobility and traffic model for vehicular wireless networks*. In Proceedings of the 2nd ACM international workshop on Vehicular ad hoc networks, pages 69–78. ACM, 2005. (Cité en page 17.)

[Cubo 2009] J. Cubo, M. Sama, F. Raimondi et D. Rosenblum. *A Model to Design and Verify Context-Aware Adaptive Service Composition*. In Proc. IEEE Int. Conf. Services Computing SCC '09, pages 184–191, 2009. (Cité en page 8.)

[Dadeau 2014] Frédéric Dadeau et Hélène Waeselynck. *Les défis du Test Logiciel-Bilan et Perspectives*. Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel, page 177, 2014. (Cité en page 5.)

[Dai 2004] Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen et Holger Pals. *From Design to Test with UML : Applied to a Roaming Algorithm for Bluetooth Devices*. In TestCom, pages 33–49, 2004. (Cité en page 11.)

[Damm 2001] Werner Damm et David Harel. *LSCs : Breathing life into message sequence charts*. Formal Methods in System Design, vol. 19, no. 1, pages 45–80, 2001. (Cité en page 18.)

[Devarapalli 2001] V. Devarapalli et D. Sidhu. *MZR : a multicast protocol for mobile ad hoc networks*. In Proc. IEEE Int. Conf. Communications ICC 2001, volume 3, pages 886–891, 2001. (Cité en page 87.)

[Fuentes-Fernández 2004] Lidia Fuentes-Fernández et Antonio Vallecillo-Moreno. *An introduction to UML profiles*. UML and Model Engineering, vol. 2, 2004. (Cité en page 57.)

[Gavalas 2011] Damianos Gavalas et Daphne Economou. *Development platforms for mobile applications : Status and trends*. Software, IEEE, vol. 28, no. 1, pages 77–86, 2011. (Cité en page 14.)

[Goldreich 2008] Oded Goldreich. Computational complexity : a conceptual perspective. Cambridge University Press, Cambridge New York, 2008. (Cité en page 87.)

[Grabowski 1993] Jens Grabowski, Dieter Hogrefe et Robert Nahm. *Test case generation with test purpose specification by MSCs*. SDL, vol. 93, pages 253–266, 1993. (Cité en pages 18 et 22.)

[Guennoun 2006] M.K. Guennoun. *Architectures dynamiques dans le contexte des applications à base de composants et orientées service*. PhD thesis, Université de toulouse - UPS, 2006. (Cité en page 82.)

[Hallal 2006] H. H. Hallal, S. Boroday, A. Petrenko et A. Ulrich. *A formal approach to property testing in causally consistent distributed traces*. Formal Aspects of Computing, vol. 18, no. 1, pages 63–83, 2006. (Cité en page 48.)

[Harel 2008a] David Harel et Shahar Maoz. *Assert and negate revisited : Modal semantics for UML sequence diagrams*. Software & Systems Modeling, vol. 7, no. 2, pages 237–252, 2008. (Cité en page 34.)

[Harel 2008b] David Harel et Shahar Maoz. *Assert and negate revisited : Modal semantics for UML sequence diagrams*. Software and Systems Modeling, vol. 7, no. 2, pages 237–252, 2008. (Cité en page 39.)

[Harris Cheheyl 1981] Maureen Harris Cheheyl, Morrie Gasser, George A Huff et Jonathan K Millen. *Verifying security*. ACM Computing Surveys (CSUR), vol. 13, no. 3, pages 279–339, 1981. (Cité en page 9.)

[Hu 2005] Yih-Chun Hu, Adrian Perrig et David B Johnson. *Ariadne : A secure on-demand routing protocol for ad hoc networks*. Wireless Networks, vol. 11, no. 1-2, pages 21–38, 2005. (Cité en page 87.)

[Hu 2011] Cuixiong Hu et Iulian Neamtiu. *Automating GUI testing for Android applications*. In Proceedings of the 6th International Workshop on Automation of Software Test, pages 77–83. ACM, 2011. (Cité en page 14.)

[Huang 2004] Q. Huang, C. Julien et G.C. Roman. *Relying on safe distance to achieve strong partitionable group membership in ad hoc networks*. Mobile

Computing, IEEE Transactions on, vol. 3, no. 2, pages 192–205, 2004. (Cité en pages 105 et 108.)

[Huszrel 2008] Gábor Huszrel, Hélène Waeselynck, Zoltán Égel, András Kövi, Zoltán Micskei, Minh Duc Nguyen, Gergely Pintér et Nicolas Rivière. *Refined design and testing framework, methodology and application result.* Rapport technique, The HIDENETS Project (FP6-STREP-26979), 2008. (Cité en pages 19, 45, 84 et 119.)

[ITU-T 2011] Recommendation Z ITU-T et Z Recommendation. *120 : Message sequence chart (MSC).* ITU-T, Geneva, vol. 27, 2011. (Cité en pages 18 et 23.)

[Killijian 2013] Marc-Olivier Killijian, Matthieu Roy, Gilles Trédan et Christophe Zanon. *SOUK : social observation of human kinetics.* In Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing, pages 193–196. ACM, 2013. (Cité en page 16.)

[Klose 2003] Jochen Klose. *Live sequence charts : A graphical formalism for the specification of communication behavior.* PhD thesis, Univ., Fachbereich Informatik, 2003. (Cité en pages 23 et 38.)

[Kugler 2007] Hillel Kugler, Michael J Stern et E Jane Albert Hubbard. *Testing scenario-based models.* In Fundamental Approaches to Software Engineering, pages 306–320. Springer, 2007. (Cité en pages 18 et 22.)

[Küster-Filipe 2006] Juliana Küster-Filipe. *Modelling concurrent interactions.* Theoretical Computer Science, vol. 351, no. 2, pages 203–220, 2006. (Cité en page 122.)

[Laprie 1996] JC Laprie. *Le guide de la sûreté de fonctionnement.* Cépadues, ISBN : 2854283821, 1996. (Cité en page 9.)

[Lieberman 2000] Henry Lieberman et Ted Selker. *Out of context : Computer systems that adapt to, and learn from, context.* IBM systems journal, vol. 39, no. 3.4, pages 617–632, 2000. (Cité en page 8.)

[Lu 2008] Heng Lu, W. K. Chan et T. H. Tse. *Testing pervasive software in the presence of context inconsistency resolution services.* In Proc. ACM/IEEE 30th Int. Conf. Software Engineering ICSE '08, pages 61–70, 2008. (Cité en page 15.)

[Lyytinen 2002] K. Lyytinen et Y. Yoo. *Issues and Challenges in Ubiquitous Computing.* COMMUNICATIONSOF THE ACM, vol. 45, no. 12, page 63, 2002. (Cité en page 5.)

[Macker 1999] Joseph Macker. *Mobile ad hoc networking (MANET) : Routing protocol performance issues and evaluation considerations.* 1999. (Cité en page 7.)

[Maltz 2001] David B Johnson David A Maltz et Josh Broch. *DSR : The dynamic source routing protocol for multi-hop wireless ad hoc networks.* Computer Science Department Carnegie Mellon University Pittsburgh, PA, pages 15213–3891, 2001. (Cité en page 87.)

[Medidi 2004] S. R. Medidi et K.-H. Vik. *Quality of service-aware source-initiated ad-hoc routing*. In Proc. First Annual IEEE Communications Society Conf. Sensor and Ad Hoc Communications and Networks IEEE SECON 2004, pages 108–117, 2004. (Cité en page 87.)

[Messmer 2000] Bruno T. Messmer et Horst Bunke. *Efficient Subgraph Isomorphism Detection : A Decomposition Approach*. IEEE Trans. Knowl. Data Eng., vol. 12, no. 2, pages 307–323, 2000. (Cité en page 82.)

[Micskei 2010] Z. Micskei et H. Waeselynck. *The many meanings of UML 2 Sequence Diagrams : a survey*. Software and Systems Modeling, pages 1–26, 2010. (Cité en pages 32, 34 et 35.)

[Micskei 2013] Zoltan Micskei. *Languages and frameworks for specifying test artifacts*. PhD thesis, Budapest University of Technology and Economics, 2013. (Cité en page 73.)

[Muccini 2012] H. Muccini, A. Di Francesco et P.exiexi Esposito. *Software testing of mobile applications : Challenges and future research directions*. In Automation of Software Test (AST), 2012 7th International Workshop on, pages 29–35. IEEE, 2012. (Cité en page 12.)

[Nguyen 2009] M.D. Nguyen. *Méthodologie de test de systèmes mobiles : une approche basée sur les scénarios*. PhD thesis, Université de toulouse - UPS, 2009. (Cité en pages 2, 29, 48 et 82.)

[Nguyen 2010] Minh Duc Nguyen, H. Waeselynck et N. Rivière. *GraphSeq : A Graph Matching Tool for the Extraction of Mobility Patterns*. In Proc. Third Int Software Testing, Verification and Validation (ICST) Conf, pages 195–204, 2010. (Cité en pages 29, 84, 87 et 89.)

[Offutt 1999] A. Jefferson Offutt et Aynur Abdurazik. *Generating Tests from UML Specifications*. In UML, pages 416–429, 1999. (Cité en page 11.)

[Omg 2011] Omg. *Unified Modeling Language (UML), Superstructure Specification (Version 2.4.1)*. Rapport technique OMG Document Number : formal/2011-08-06, Object Management Group, `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF`, aug 2011. (Cité en pages 18, 23 et 36.)

[Parr 2013] Terence Parr. The definitive antlr 4 reference. Pragmatic Bookshelf, 2nd édition, 2013. (Cité en page 69.)

[Pickin 2003] Simon Pickin. *Test des composants logiciels pour les télécommunications*. PhD thesis, Rennes 1, 2003. (Cité en page 22.)

[Pickin 2004] Simon Pickin et Jean-Marc Jézéquel. *Using UML sequence diagrams as the basis for a formal test description language*. In Integrated Formal Methods, pages 481–500. Springer, 2004. (Cité en page 18.)

[Sama 2010] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum et Zhimin Wang. *Context-Aware Adaptive Applications : Fault Patterns and Their Automated Identification*. vol. 36, no. 5, pages 644–661, 2010. (Cité en pages 5 et 15.)

[Sanmugalingam 2002] Kumaresan Sanmugalingam et George Coulouris. *A generic location event simulator*. In UbiComp 2002 : Ubiquitous Computing, pages 308–315. Springer, 2002. (Cité en page 17.)

[Satoh 2003a] I. Satoh. *Software testing for mobile and ubiquitous computing*. In Proc. Sixth Int. Symp. Autonomous Decentralized Systems ISADS 2003, pages 185–192, 2003. (Cité en page 17.)

[Satoh 2003b] I. Satoh. *A testing framework for mobile computing software*. vol. 29, no. 12, pages 1112–1121, 2003. (Cité en page 17.)

[Satoh 2004] I. Satoh. *Software testing for wireless mobile computing*. vol. 11, no. 5, pages 58–64, 2004. (Cité en page 17.)

[Satyanarayanan 1996] M. Satyanarayanan. *Fundamental Challenges in Mobile Computing*. In Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '96, pages 1–7, New York, NY, USA, 1996. ACM. (Cité en page 7.)

[Schieferdecker 2003] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski et Axel Rennoch. *The UML 2.0 testing profile and its relation to TTCN-3*. In Testing of Communicating Systems, pages 79–94. Springer, 2003. (Cité en page 11.)

[Schmidt 1999] Albrecht Schmidt, Kofi Asante Aidoo, Antti Takaluoma, Urpo Tuomela, Kristof Van Laerhoven et Walter Van de Velde. *Advanced interaction in context*. In Handheld and ubiquitous computing, pages 89–101. Springer, 1999. (Cité en page 8.)

[Schmidt 2000] Albrecht Schmidt. *Implicit human computer interaction through context*. Personal technologies, vol. 4, no. 2-3, pages 191–199, 2000. (Cité en page 8.)

[Teodosiu 2010] Roxana-Alexandra Teodosiu. The development and evaluation of a platform based on the simulation for the test of mobile systems. Master's thesis, Military Technical Academy of Bucharest, 2010. (Cité en page 17.)

[Tretmans 2004] Jan Tretmans. *Model-based testing : Property checking for real*. In Keynote address at the International Workshop for Construction and Analysis of Safe, Secure and Interoperable Smart devices, 2004. (Cité en page 11.)

[Ullmann 1976] Julian R. Ullmann. *An Algorithm for Subgraph Isomorphism*. J. ACM, vol. 23, no. 1, pages 31–42, 1976. (Cité en page 82.)

[Waeselynck 2007] Hélène Waeselynck, Zoltán Micskei, Minh Duc Nguyen et Nicolas Rivière. *Preliminary testing framework and methodology*. Rapport technique, The HIDENETS Project (FP6-STREP-26979), 2007. (Cité en page 19.)

[Wang 2007] Zhimin Wang, S. Elbaum et D. S. Rosenblum. *Automated Generation of Context-Aware Tests*. In Proc. 29th Int. Conf. Software Engineering ICSE 2007, pages 406–415, 2007. (Cité en page 15.)

[Weiser 1993] Mark Weiser. *Hot topics-ubiquitous computing.* Computer, vol. 26, no. 10, pages 71–72, 1993. (Cité en page 7.)

[Weyuker 1982] Elaine J Weyuker. *On testing non-testable programs.* The Computer Journal, vol. 25, no. 4, pages 465–470, 1982. (Cité en page 11.)

[Winstanley 2012] Christopher Winstanley, Rajiv Ramdhany, François Taïani, Barry Porter et Hugo Miranda. *PAMPA in the wild : a real-life evaluation of a lightweight ad-hoc broadcasting family.* In Proceedings of the 7th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks, page 3. ACM, 2012. (Cité en page 8.)

Test de systèmes ubiquitaires avec prise en compte explicite de la mobilité.

Résumé :

L'objectif de cette thèse est de contribuer à l'élaboration d'une méthode de test de systèmes mobiles. L'approche développée est fondée sur la description de tests à l'aide de scénarios et leurs vérifications sur une trace d'exécution. Un scénario modélise le comportement et les interactions que l'on souhaite observer entre un ensemble de nœuds. Les caractéristiques des systèmes mobiles nous ont conduit à représenter un scénario sous deux points de vue différents et complémentaires. Un premier représente des événements de communications entre les nœuds et un second représente la topologie des liens entre ces nœuds. Notre approche est décomposée en deux étapes : une étape de spécification des cas de tests à l'aide de scénarios et une étape de vérification de ces scénarios sur des traces d'exécutions.

La première consiste à spécifier à l'aide du langage dédié *TERMOS* les cas de test de l'application mobile à vérifier. Ce langage *TERMOS* a été mis en œuvre au sein de l'atelier *UML Papyrus*. À partir des scénarios décrits de manière graphique, nous générons pour chacun d'eux un automate ainsi qu'une séquence de topologie que nous utilisons dans l'étape suivante.

La deuxième étape consiste à vérifier chaque scénario sur des traces d'exécutions provenant de l'application à tester. Pour cela un premier outil recherche les occurrences de la séquence de topologie du scénario dans la trace d'exécution. Pour chacune d'entre elles, l'automate est exécuté et conclut à un verdict. L'analyse de l'ensemble des verdicts d'un scénario permet de détecter les fautes présentes dans le système.

Mots clés : test logiciel, systèmes informatiques mobiles, tests à base de scénarios, sensibilité au contexte

---

Test of ubiquitous systems with explicit consideration of the mobility.

Abstract :

The main objective of this thesis is to contribute to elaborating a mobile system test method. The proposed approach is based on test definition using scenarios and their verification on an execution trace. A scenario modelizes the behavior and the interactions we want to achieve on a set of nodes. Considering the characteristics of mobile systems we represented scenarios from two different but complementary points of view. The first represents communication events between nodes and the second represents the link topology between the nodes. Our approach is composed of two steps : a first step to specify the test cases by using scenarios and a second step to verify these scenarios on execution flows.

The first step consists in using the dedicated *TERMOS* language in order to specify the test cases of the mobile application. The *TERMOS* language has been developed in the *UML Papyrus* workshop. Based on the graphically defined scenarios, we generate an automaton for each one of them, as well as a sequence of topologies which we will be using in the next step.

The second step consists in verifying each scenario by using execution traces from the application to be tested. Therefore a first tool detects scenario topology sequences in the execution flow. For each one of them the automaton is executed and comes out with a verdict. The analysis of all the verdicts of a scenario allows the detection of faults in the system.

Keywords : software testing, mobile computing systems, scanarios based testing, context aware