

Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Systèmes Informatiques

Présentée et soutenue par :
Hoang-Nam CHU

le : jeudi 1 septembre 2011

Titre :
Test et Evaluation de la Robustesse de la Couche Fonctionnelle d'un Robot Autonome
Test and Evaluation of the Robustness of the Functional Layer of an Autonomous Robot

---

Ecole doctorale :
Systèmes (EDSYS)

Unité de recherche :
LAAS-CNRS

Directeur(s) de Thèse :
M. David POWELL
M. Marc-Olivier KILLIJIAN

Rapporteurs :

M. Bernard ESPIAU
M. Pedro-Joaquin GIL VICENTE

Membre(s) du jury :
M. Jean ARLAT, Président
M. Bernard ESPIAU
M. Pedro-Joaquin GIL VICENTE
M. Jean-Paul BLANQUART
Mme. Dominique SEGUELA
M. François Félix INGRAND

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE TOULOUSE
délivré par l'**Institut National Polytechnique de Toulouse (INP Toulouse)**

École Doctorale : Systèmes (EDSYS)
Spécialité : Systèmes Informatiques

présentée et soutenue

par

Hoang-Nam CHU

le 1 septembre 2011

Test et Evaluation de la Robustesse de la Couche Fonctionnelle d'un Robot Autonome

Test and Evaluation of the Robustness of the Functional Layer of an Autonomous Robot

Directeurs de thèse :
M. David POWELL
M. Marc-Olivier KILLIJIAN

JURY
M. Jean ARLAT, Président
M. Bernard ESPIAU
M. Pedro-Joaquin GIL VICENTE
M. Jean-Paul BLANQUART
Mme. Dominique SEGUELA
M. François Félix INGRAND

Avant-propos

Les travaux de thèse présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je tiens à remercier Raja Chatila, Jean-Louis Sanchez, et Jean Arlat, directeurs successifs du LAAS - CNRS, pour m'avoir accueilli au sein de ce laboratoire. Je remercie également Karama Kanoun, responsable du groupe TSF, de m'avoir permis d'accomplir mes travaux dans ce groupe.

Je veux remercier mes deux encadrants, David Powell et Marc-Olivier Killijian. Avec eux, j'avais des réunions, voire les débats, très enrichissants où je pouvais beaucoup apprendre de leur compétences scientifiques. Un merci spécial à eux pour leur conseils, leur questions difficiles et pour le soutien qu'ils m'ont apporté durant les années de thèse. Leurs critiques ont sans doute beaucoup contribué à ces travaux.

Je remercie les personnes qui ont consacré leur temps à la lecture de ce mémoire et m'ont fait l'honneur de participer à mon jury :

- M. Jean Arlat, directeur de recherche CNRS au LAAS, Toulouse ;
- M. Jean-Paul Blanquart, ingénieur d'études à l'Astrium, Toulouse ;
- M. Bernard Espiau, directeur de recherche INRIA à INRIA Rhône-Alpes, Grenoble ;
- M. Felix Ingrand, chargé de recherche CNRS au LAAS, Toulouse ;
- Mme. Dominique Seguela, chargé de mission scientifique au CNES, Toulouse ;
- M. Pédro-Joaquín Gil Vicente, professeur à l'Université Polytechnique de Valence, Espagne.

Le sujet de thèse dont il est question dans ce mémoire a pour origine le projet MARAE *(Méthode et Architecture Robuste pour l'Autonomie dans l'Espace)*, financé par la Fondation de Recherche pour l'Aéronautique et l'Espace (FNRAE) de 2007 à 2010 et réalisé en collaboration entre le LAAS-CNRS, Verimag et Astrium. J'exprime ma reconnaissance aux personnes avec qui j'ai travaillé dans ce projet, en particulière Saddek Bensalem pour m'avoir accueilli au laboratoire Verimag afin de travailler sur la simulation; Jean Arlat et Jean-Paul Blanquart pour leurs conseils; Félix Ingrand, Imen Kahloul et Lavindra da Silva qui m'ont aidé à comprendre les logiciels de commande de robots. C'est grâce à leur support que mon travail a pu être couronné de succès.

Mes remerciements vont naturellement à l'ensemble des membres du groupe TSF: permanents, doctorants et stagiaires avec lesquels j'ai partagé ces quatre années. Je tiens à remercier mes chers amis du groupe TSF: Mirunaesca qui m'encourage toujours avec un grand sourire ; Amina, ma voisine d'en-face, qui m'a appris la prononciation des mots arabes (dur dur) ; Molivier toujours prompt à taquiner ses "enfants" du bureau ; mes voisins Maximus, Rim, Miguel

et Fernando du bureau 10 qui ont créé une ambiance pleine de joies ; Lily et Damien qui ont apporter des rires aux jeunes TSF ; Roxana, Irina, Quynh Anh, Anthony, Kossi, Jimmy, Robert, Ivan et Yann qui m'ont encouragé tout au long de ce travail. Je n'oublierai jamais tous les bons moments que j'ai passé avec vous. Un merci particulier à Benji et Fernando "le geek" pour leurs réponses quand j'étais confrontés à des problèmes techniques.

Un grand merci à mes amis vietnamiens, à mes frères Tung, Thach, Van et Thành pour leur aide précieuse durant ces trois ans, y compris dans les moments difficiles.

Merci à papa, à maman, à mon grand frère et
à Nam Phuong pour leur soutien moral.

Contents

Résumé étendu **v**

Introduction **1**

1 Dependability and Robustness in Autonomous Systems **5**

1.1 Autonomous systems 5

1.1.1 Autonomy 5

1.1.2 Autonomous system architecture 6

1.2 Dependability 7

1.2.1 Key concepts 8

1.2.2 Techniques for dependability 9

1.3 Robustness in autonomous systems 12

1.3.1 Robustness definitions 12

1.3.2 System-level robustness 12

1.3.3 Robustness of the functional layer 13

1.4 Software design tools for the LAAS architecture functional layer 14

1.4.1 The $G^{en}oM$ environment 15

1.4.2 The *BIP* framework 17

1.4.3 The MARAE context: *BIP* applied to $G^{en}oM$ 19

1.5 Conclusion 21

2 Robustness Testing: State Of The Art **23**

2.1 Types of robustness testing 24

2.1.1 Load robustness testing 24

2.1.2 Input robustness testing 24

2.2 Robustness testing methods based on input domain models 26

2.2.1 Fault injection methodology 26

2.2.2 FUZZ 27

2.2.3 BALLISTA 29

2.2.4 MAFALDA . . . 30
2.2.5 DBench . . . 32
2.3 Robustness testing methods based on behaviour models . . . 34
2.3.1 Rollet-Fouchal approach . . . 34
2.3.2 Saad-Khorchef approach . . . 36
2.4 Hybrid robustness testing . . . 39
2.5 Conclusion . . . 40

3 Functional Layer Robustness Testing Framework 43
3.1 System model . . . 44
3.1.1 Interface protocol . . . 44
3.1.2 Notation . . . 45
3.2 Property types and enforcement policies . . . 47
3.3 System observation . . . 48
3.3.1 Behaviour categories . . . 48
3.3.2 Property oracles . . . 49
3.3.3 Discussion . . . 68
3.4 Conclusion . . . 68

4 Application to Dala Rover 69
4.1 Dala application . . . 69
4.2 Specified safety properties . . . 71
4.2.1 PEX(*module*): Precondition for EXec request . . . 72
4.2.2 AIB(x): Activity x Interrupted By . . . 72
4.2.3 PRE(x): activity x PREceded by . . . 73
4.2.4 EXC(x, y): mutual EXClusion between activities x and y . . . 74
4.3 Testing environment . . . 74
4.3.1 Overview . . . 74
4.3.2 Workload . . . 76
4.3.3 Faultload . . . 78
4.4 Readouts . . . 79
4.4.1 System logging . . . 79
4.4.2 *Trace Analyzer*: system observation tool . . . 80
4.5 Measures . . . 81
4.6 Results . . . 83
4.6.1 Per trace results . . . 83

4.6.2 Per request results 91
4.7 Conclusion 93

5 Conclusion and Future Work **95**

Bibliography **99**

Résumé étendu

Les systèmes autonomes couvrent un large spectre d'applications, allant des animaux-robots de compagnie et les robots aspirateurs jusqu'aux robots guides de musée, les robots d'exploration planétaire et, dans un avenir proche, les robots de services domestiques. Comme les systèmes autonomes sont appliqués à des tâches de plus en plus critiques et complexes, il est essentiel qu'ils exhibent une fiabilité et une sécurité-inoccuité suffisantes dans toutes les situations qu'ils peuvent rencontrer.

Par la fiabilité, nous voulons dire que le système autonome peut atteindre ses objectifs malgré des conditions endogènes ou exogènes défavorables, telles que les fautes internes et les situations environnementales adverses. La fiabilité face aux situations environnementales adverses (parfois appelé la robustesse (au niveau système)) est une condition particulièrement importante pour les systèmes autonomes, qui doivent être capables de fonctionner dans les environnements partiellement inconnus, imprévisibles et éventuellement dynamiques.

Par la sécurité-innocuité, nous entendons que le système autonome ne devrait ni causer du mal à d'autres agents (particulièrement des humains) dans son environnement ni endommager ses propres ressources critiques. La protection de sa propre intégrité est en fait une condition préalable pour qu'un système autonome soit fiable : si ses ressources critiques ne sont plus utilisables, aucun raisonnement automatisé, aussi poussé soit-il, ne permettra au système d'atteindre ses objectifs.

Cette thèse traite de l'évaluation d'un type particulier de mécanisme de sécurité pour un robot autonome, mis en œuvre dans son logiciel de commande. Le mécanisme visé a pour objectif la mise en vigueur d'un ensemble de *contraintes de sécurité-innocuité* qui définissent les comportements dangereux ou incohérents qui doivent être évités. Des exemples des contraintes de sécurité-innocuité sont, par exemple, qu'un robot mobile ne devrait pas se déplacer à grande vitesse si son bras est déployé, ou qu'un satellite d'observation planétaire ne devrait pas démarrer ses propulseurs si l'objectif de son caméra n'est pas protégé.

Les mécanismes visant à faire respecter les contraintes de sécurité-innocuité sont mis en œuvre dans la plus basse couche de logiciel de commande du robot (appelé ici la *couche fonctionnelle*), qui communique directement avec le matériel de robot. Typiquement, une telle couche de logiciel réalise des fonctions de base permettant la commande du matériel du robot et fournit une interface de programmation vers la couche supérieure (que nous l'appellerons, pour l'instant, la *couche d'application*). Spécifiquement, les clients de la couche fonctionnelle (situés à la couche d'application) peuvent émettre des requêtes pour initialiser des modules, mettre à jour leurs structures de données internes, ou lancer et arrêter diverses *activités*, telles que : faire tourner les roues de robot à une vitesse donnée, déplacer le robot aux coordonnées indiquées tout en évitant des obstacles, etc.

Les clients de la couche d'application peuvent créer des comportements plus complexes en envoyant des requêtes asynchrones pour lancer et arrêter des activités à la couche fonc-

tionnelle. Nous considérons que les contraintes de sécurité-innocuité de haut niveau sont exprimées en termes de *propriétés de sûreté* qui placent des restrictions sur les instants auxquels les activités de la couche fonctionnelle peuvent s'exécuter. Par exemple, une propriété de sûreté peut requérir une exclusion mutuelle entre les activités x et y. Ainsi, si un client de la couche d'application envoie une requête pour x pendant que y s'exécute, la propriété d'exclusion mutuelle doit être mise en vigueur, par exemple, en rejetant la requête pour x.

Dans cette thèse, nous définissons une méthode pour évaluer l'efficacité de tels mécanismes. Nous abordons le problème sous l'angle du *test de robustesse*, où la robustesse est défini comme "le degré selon lequel un système, ou un composant, peut fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes" [IEE90]. De notre perspective, une entrée invalide est une requête de la couche d'application qui peut mettre en danger une propriété de sûreté si elle est exécutée dans l'état actuel de la couche fonctionnelle. Nous nous intéressons en particulier à l'invalidité dans le domaine temporel (c'est-à-dire, des requêtes envoyées au "mauvais moment"). Cependant, notre approche peut facilement être étendue pour prendre en compte l'invalidité dans le domaine des valeurs (par exemple, des paramètres de requêtes incorrects).

Nous adoptons une approche de test aléatoire basée sur l'injection de fautes, par laquelle un grand nombre de cas de test sont produits automatiquement en mutant une séquence d'entrées valides. Notre approche de test permet l'évaluation de robustesse dans le sens que nous pouvons fournir des statistiques descriptives, par rapport à la population des cas de test, sur le comportement robuste ou non du système sous test (dans notre cas, une implémentation de couche fonctionnelle). De plus, nous suivons une approche de test boîte noire, qui ne considère pas les détails internes du système sous test. Ainsi, un ensemble de cas de test généré en utilisant notre approche peut être appliqué comme un benchmark (ou "étalon") de robustesse pour comparer différentes implémentations d'une même couche fonctionnelle.

Notre mémoire est structuré en quatre chapitres principaux, que nous résumons ci-après, avant de donner nos conclusions générales et des suggestions d'orientations futures.

Sûreté de fonctionnement et robustesse dans les systèmes autonomes

Ce chapitre introductif présente les principaux concepts des systèmes autonomes et de la sûreté de fonctionnement. La sûreté de fonctionnement des systèmes autonomes est analysée sous l'angle de la robustesse dans les architectures hiérarchiques de système autonome. On présente enfin la couche fonctionnelle d'une telle architecture hiérarchique, qui sera le point de focalisation de la théorie, des méthodes et des outils développés par la suite.

Plusieurs travaux ont essayé de définir et de caractériser le degré d'autonomie d'un système. *Fogel* [MAM+00] a décrit l'autonomie d'un système comme "la capacité de générer ses propres buts sans aucune instruction de l'extérieur" qui signifie que ces systèmes peuvent effectuer, de leur propre volonté, des actions d'apparence intelligente. *Clough* in [Clo02] souligne également cette notion de systèmes "ayant leur propre volonté" et propose quatre capacités essentielles pour y parvenir : perception, conscience de la situation, prise de décision et coopération.

Huang [Hua07] a proposé la définition suivante d'un système autonome, qui caractérise particulièrement un système robotique autonome interagissant avec des humains :

> L'autonomie est la capacité propre d'un système inhabité de détecter, percevoir, analyser, communiquer, planifier, établir des décisions, et agir, afin d'atteindre des objectifs assignés par un opérateur humain (...). [L'autonomie] est caractérisée par les missions que le système est capable d'effectuer, les environnements au sein desquels les missions sont effectuées, et le degré d'indépendance qui peut être autorisée dans l'exécution des missions.

Cette définition de l'autonomie souligne deux facteurs importants dans les systèmes autonomes : la capacité de prise de décision, obtenue par des diverses techniques d'intelligence artificielle ; et la capacité de faire face à la difficulté, l'incertitude et l'évolution de l'environnement du système. Ce dernier aspect est lié à la notion de la *robustesse*.

Le concept de *robustesse* se retrouve dans de nombreux contextes différents. Avizienis *et al.* [ALRL04] définissent la robustesse comme "la sûreté de fonctionnement par rapport aux fautes externes". Dans le contexte des systèmes autonomes, Lussier [Lus07] définit la robustesse comme *"la délivrance d'un service correct en dépit de* ***situations adverses*** *dues aux incertitudes vis-à-vis de l'environnement du système (telles qu'un obstacle inattendu)"*. Dans le domaine du génie logiciel, la robustesse est définie comme "*le degré selon lequel un système, ou un composant, peut fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes*". Cette définition est assez générale et recouvre les deux premières définitions : "des fautes externes" peuvent affecter le système comme des entrées invalides; "des situations adverses" peuvent être considérées comme des conditions environnementales stressantes.

La "robustesse" est souvent citée comme une caractéristique essentielle des systèmes autonomes, en raison de l'emphase accordée à la capacité de tels systèmes à travailler dans un environnement dynamique, partiellement inconnu et non-prévisible. Cependant, l'incertitude de l'environnement peut donner lieu à des événements imprévus pouvant mettre le système en danger. Il est donc nécessaire d'équiper les systèmes autonomes avec des mécanismes de protection appropriés, tels que des moniteurs de sécurité en charge de détecter et réagir face à des situations dangereuses [RRAA04, GPBB08, OKWK09], ou des mécanismes visant à empêcher l'exécution d'actions qui pourraient violer des contraintes de sécurité [ACMR03].

Ces derniers mécanismes sont particulièrement pertinents dans les systèmes autonomes hiérarchiques car, pour que les procédures délibératives implantées aux couches supérieures puissent s'exécuter en un temps raisonnable, les modèles sous-jacents doivent rester les plus abstraits possibles, en ignorant des détails de bas niveau, tels, en particulier, l'état des contrôleurs matérielles et d'autres modules de la couche la plus basse (appelée *couche fonctionnelle*). Par conséquence, il peut arriver qu'un client de la couche fonctionnelle (par exemple, un contrôleur d'exécution de plan) émette des requêtes pour des actions dont l'exécution serait incohérente voire dangereuse dans l'état courant de la couche fonctionnelle. Des mécanismes de protection doivent donc être mis en place, dans la couche fonctionnelle où à l'interface de celle-ci, afin d'empêcher la prise en compte de telles requêtes "invalides". L'assurance d'une telle protection peut donc être vue comme un problème de robustesse.

Les travaux décrits dans cette thèse ont pour objectif l'évaluation de l'efficacité de tels mécanismes de protection. Nous abordons cette évaluation sous l'angle du *test de robustesse* : une entrée invalide est une requête qui peut mettre en danger une propriété de sécurité. Nous focalisons tout particulièrement sur l'invalidité dans le domaine temporelle, c'est-à-dire, des requêtes envoyées de façon inopportune. Cependant, notre approche peut facilement être étendue pour aborder l'invalidité dans le domaine des valeurs (par exemple, des requêtes avec des paramètres incorrects).

Notre travail s'inscrit dans le cadre du projet MARAE[1] dont l'objectif était de développer une architecture robuste pour les systèmes autonomes à l'aide, d'une part, de l'environnement $G^{en}oM$ du LAAS [FHC97] pour le développement de couches fonctionnelles modulaires et, d'autre part, la méthodologie BIP de Vérimag [BBS06] pour la modélisation de programmes temps réel hétérogènes. Nous visons en particulier à évaluer la robustesse d'une couche fonctionnelle construite à l'aide de $G^{en}oM$ étendu par BIP, qui fournit un cadre systématique pour la mise en place de protections contre des requêtes invalides. Nous souhaitons aussi comparer cette robustesse à celle fournie par une couche fonctionnelle de référence, construite uniquement à l'aide de $G^{en}oM$, qui ne fournit que quelques mécanismes de protection élémentaires.

Etat de l'art du test de robustesse

Nous présentons dans ce chapitre un état de l'art du test de robustesse. Nous distinguons deux types principaux de test de robustesse : le test de la robustesse vis-à-vis de la charge de travail et le test de la robustesse vis-à-vis des entrées. Ensuite, nous examinons différentes approches pour le test de robustesse vis-à-vis des entrées, que nous classifions dans deux larges catégories : les approches basées sur des modèles de domaine d'entrée et les approches basées sur des modèles de comportement. Une troisième catégorie hybride est considérée, dans laquelle les jeux de test sont produits en utilisant un modèle de domaine d'entrée et les verdicts de robustesse sont livrés par un oracle formalisé comme un ensemble d'invariants sur le comportement observable du système.

En partant de la définition de robustesse du IEEE Std. 610-12, 1990 [IEE90]) déjà mentionnée, on peut distinguer deux aspects importants à considérer dans le test de robustesse : des *entrées invalides* et des *conditions environnementales stressantes*. Les *entrées invalides* sont des fautes externes (du point de vue de la frontière de système [ALRL04]) qui peuvent être injectées directement à l'interface du système sous test (ou SUT, *System Under Test*), ou dans d'autres systèmes qui interagissent avec lui. L'*environnement* est l'ensemble des conditions de fonctionnement du système (tel son environnement physique, sa charge de travail, etc.).

Le test d'un système en présence d'entrées invalides nous conduit à la notion du "test de robustesse vis-à-vis des entrées" (*input robustness testing*). En ce qui concerne le test de robustesse par rapport aux conditions environnementales stressantes, nous focalisons sur l'influence de l'environnement sur les *entrées fonctionnelles* du système, autrement dit, nous n'étudions pas les facteurs environnementaux physiques, tels que la température, la pression, les rayonnements, etc. Nous interprétons donc le "stress" de l'environnement en termes du charge de travail soumis au système, d'où vient la notion du "test de robustesse vis-à-vis de la charge de travail" (*load robustness testing*).

Nous définissons le *test de robustesse vis-à-vis de la charge de travail* comme l'ensemble des techniques qui peut être utilisé pour évaluer le comportement d'un SUT au delà des conditions de charge nominales. Parmi ces techniques, on peut nommer notamment *le test de stress*, un test au cours duquel les testeurs soumettent le SUT à une charge de travail exceptionnellement élevée afin d'évaluer sa capacité à travailler correctement dans des conditions de forte charge, voire de surcharge; et *le test d'endurance*, où le testeur soumet une charge importante de travail sur une longue durée pour voir si le SUT est capable de supporter une telle activité

[1]MARAE (Méthode et Architecture Robustes pour l'Autonomie dans l'Espace), était un projet national partiellement soutenu par la *Fondation Nationale de Recherche en Aéronautique et l'Espace* (FNRAE) et regroupant le LAAS-CNRS, Vérimaq et EADS Astrium.

intense sans dégradation des performances.

Le *test de robustesse vis-à-vis des entrées* se concentre sur l'examen du comportement d'un système face à des entrées invalides. Nous définissons deux types d'entrées invalides: des *entrées invalides en valeur*, qui sont des entrées qui ne sont pas décrites dans les spécifications du SUT ou dont un ou plusieurs paramètres prennent des valeurs hors du domaine spécifié ; et des *entrées invalides en temps*, qui sont des entrées spécifiées mais dont l'occurrence n'est pas prévue à l'état actuel du système. Nous identifions ainsi deux types de test de robustesse d'entrée : *le test de robustesse en valeur* et *le test de robustesse en temps*, qui représentent respectivement les cas ou le testeur essait de faire échouer le SUT en soumettant respectivement des entrées invalides en valeur et des entrées invalides en temps.

Plusieurs approches pour le test de robustesse ont été proposées, qui peuvent être classées dans deux larges catégories selon qu'elles s'appuient sur des *modèles du domaine d'entrée* ou sur des *modèles de comportement*. Un troisième catégorie d'approche, appelée *hybride*, combine des aspects des deux premières.

Test de robustesse basé sur des modèles du domaine d'entrée

Les méthodes basées sur les modèles du domaine d'entrée effectuent le test de robustesse par rapport aux *entrées invalides* qui sont générés en analysant des spécifications des entrées de système ou en mutant des entrées valides. Ces approches sont basées habituellement sur l'injection aléatoire de fautes et se concentrent sur *l'évaluation* de la robustesse (les défauts de robustesse sont néanmoins détectés par une sorte d'effet de bord). L'évaluation s'appuie sur une caractérisation des comportements observés et des mesures statistiques correspondantes.

Le test de robustesse basé sur des modèles du domaine d'entrée est ainsi étroitement lié à l'injection de faute, qui simule des fautes et des erreurs afin d'observer l'impact qu'ils ont et le comportement du système [Voa97]. Dans [AAA+90], *Arlat et al.* ont proposé **FARM**, un modèle conceptuel permettant de décrire une campagne d'injection de fautes au moyen de 4 éléments : en entrée, un ensemble de fautes **F** et un ensemble d'activations (ou activités) **A** du système ; en sortie, un ensemble de relevés **R** et un ensemble de mesures **M**. Le comportement du système est observé et sauvegardé dans un relevé r qui caractérise le résultat d'expérience. L'ensemble **R** de relevés est traité pour déduire un ou plusieurs membres d'un ensemble de mesures **M** qui caractérisent la sûreté de fonctionnement du système considéré.

Parmi les différentes approches décrites dans la littérature, nous présentons ici les travaux du projet DBench (Dependability Benchmarking) (`http://www.laas.fr/DBench`) comme un exemple typique de cette catégorie d'approches.

Le but d'un benchmark (ou "étalon") de sûreté de fonctionnement est de fournir une méthode générique et reproductible pour caractériser le comportement d'un système informatique en présence des fautes. *Kanoun et al.* [KCK+05] définissent un benchmark de sûreté de fonctionnement pour les systèmes d'exploitation et l'appliquent à six versions de Windows et à quatre versions de Linux. Cette étude cherchait à mesurer la robustesse de Windows et de Linux en présence des fautes en injectant des entrées incorrectes à l'OS (*operating system*) via son API (*application programming interface*). Un profil d'exécution de benchmark et une mesure de benchmark sont définis pour tester la robustesse de ces dix OS. Le profil d'exécution du benchmark se compose d'un ensemble de charges de travail générées par PostMark et de paramètres corrompus d'appels systèmes utilisés comme ensemble de fautes. Pendant l'exécution, les fautes sont injectées sur les appels système en les interceptant, en corrompant leurs paramètres, puis en les re-insérant dans le système.

Les résultats de test sont regroupés en se basant sur l'observation de l'état du système :

- **SEr** : Un code d'erreur est renvoyé.
- **SXp** : Une exception est levée.
- **SPc** : Le système est dans l'état de panique. L'OS n'assure plus ses services aux applications.
- **SHg** : Le système se bloque.
- **SNS** : Aucun des résultats précédents n'est observé.

La robustesse des différents systèmes est évaluée et comparée au moyen de statistiques sur les proportions des résultats de test tombant dans chacune de ces catégories.

Test de robustesse basé sur des modèles de comportement

Les approches basées sur des modèles de comportement s'appuient sur un modèle formel du SUT pour définir à la fois des cas de test et un oracle permettant de juger si le SUT est robuste. Ces méthodes se concentrent sur *la vérification* de la robustesse.

Quand un modèle de comportement du SUT existe qui spécifie le comportement robuste en présence d'entrées invalides, le test de robustesse peut être exprimé comme un test de conformité. Plusieurs travaux ont visé la transformation d'une spécification de comportement nominal en une spécification de comportement robuste, et puis la définition de cas de test pour vérifier la conformité du SUT avec cette dernière spécification afin de donner une conclusion quant à la robustesse du système [TRF05, SKRC07, FMP05].

Comme exemple typique de cette catégorie d'approches, nous résumons ici les travaux de *Saad-Khorchef et al.*[SKRC07, Saa06], qui ont proposé une approche de test de robustesse basée sur un modèle de comportement pour tester les protocoles de communication. Il s'agit d'un des rares travaux qui prennent en compte l'invalidité des entrées dans les deux domaines : valeur et temps. Cette approche utilisent deux spécifications du système : une *spécification nominale*, qui décrit le comportement normal du système ; et une *spécification augmentée*, qui décrit les comportements acceptables du système en présence d'entrées invalides. Les testeurs produisent des cas de test des spécifications augmentées et les utilisent pour activer le SUT. Les résultats de test sont évalués vis-à-vis cette spécification pour donner le verdict de robustesse.

La spécification nominale est donnée au moyen d'un modèle IOLTS (*Input Output Labelled Transition System*) qui décrit le système par un ensemble d'états, un ensemble de relations de transition entre états et un alphabet d'actions de transition. La méthode de test se compose de deux phases : i) la construction de la *spécification augmentée* en intégrant des "hasards" (des événements non-attendus dans la spécification nominale du système) dans la spécification nominale; ii) la génération de cas de test de conformité à partir de cette spécification augmentée.

Test hybride de robustesse

L'approche hybride pour le test de robustesse sépare la génération des cas de test de la prise de décision quant à la robustesse du SUT pour chaque cas de test. Les cas de test sont

obtenus au moyen de l'injection aléatoire de fautes par rapport à un modèle du domaine d'entrée. Le verdict de test est obtenu au moyen d'un oracle défini à partir d'un modèle de comportement.

Cavalli et al. a proposé dans [CMM08] une approche de test de robustesse de protocoles de communication qui combine l'injection de fautes et *le test passif* [LCH+02], un processus permettant de détecter le comportement incorrect d'un système sous test en observant passivement ses entrées et ses sorties sans interrompre son fonctionnement.

L'approche vise la caractérisation de la robustesse d'une implémentation de protocole de communication face aux fautes pouvant affecter le canal de transmission. Les fautes injectées sont basées sur un modèle des fautes de communication catégorisé en 3 classes : les *fautes d'omission*, qui peuvent être imitées en arrêtant des messages envoyés et reçus par un serveur ; les *fautes "arbitraires"*, qui peuvent être imitées en corrompant des messages reçus par le SUT ; et les *fautes temporelles*, qui peuvent être imitées en retardant la transmission de messages.

Le comportement robuste est exprimé sous forme d'un ensemble des *propriétés invariables* qui spécifient les ordres valides d'entrée et de sortie qu'un système peut produire. L'approche proposée se compose des étapes suivantes: (i) établir un modèle formel du comportement de système ; (ii) définir les propriétés invariables et les vérifier contre le modèle formel ; (iii) définir le modèle de faute et les fautes à injecter ; (iv) équiper le SUT pour l'injection des fautes et la surveillance de son comportement ; (v) exécuter les tests par l'activation du SUT en injectant les fautes et en surveillant son comportement ; (vi) analyser les résultats basés sur les invariants définis et produire un verdict de robustesse.

Cadre conceptuel pour le test de robustesse d'une couche fonctionnelle

Nous présentons ici notre cadre conceptuel pour évaluer le test de robustesse de la couche fonctionnelle d'un système autonome. Nous développons notre approche dans le contexte de l'architecture LAAS, qui est une architecture hiérarchique typique pour les systèmes autonomes.

Notre cadre conceptuel est prévu pour *comparer* différentes implémentations de la même fonctionnalité, avec la même interface de programmation (API), dans un esprit analogue au travaux de DBench. Pour comparer différentes implémentations, les mêmes cas de test doivent être appliqués à chaque implémentation. La génération des cas de test ne peut pas être basé sur un modèle de comportement détaillé d'une implémentation. Au lieu de cela, une approche de test du type *boîte noire* est nécessaire, où seule la spécification de l'interface de programmation est fournie au testeur.

Pour rendre un verdict de robustesse, une possibilité envisagée [CAI+09, CAK+09] était de définir une catégorisation des résultats de test de robustesse, inspiré des travaux sur l'évaluation de la robustesse de systèmes d'exploitation. Cependant, la couche fonctionnelle d'un robot autonome est très différente d'un système d'exploitation, tous les concepts ne sont donc pas transférables. Néanmoins, un concept qui peut être utilisé est celui du blocage du système (comme les catégories SPc et SHg de la classification de résultat de test dans DBench) : une couche fonctionnelle qui se bloque à cause d'une entrée invalide est évidemment non-robuste. Par contre, les retours d'anomalie tels que les signaux d'erreur et les exceptions auraient besoin d'une analyse plus fine.

L'approche que nous proposons ici est d'évaluer la robustesse au niveau du *propriétés de sûreté* que la couche fonctionnelle devrait respecter. Un SUT qui respecte un ensemble spécifique de propriétés de sûreté en présence des entrées invalides sera considéré comme robuste. Puisque nous adoptons une approche de test boîte noire, la décision si le SUT respecte ou non les propriétés de sûreté doit être basée uniquement sur l'observation des requêtes et des réponses traversant son API. Dans ce contexte, le cadre conceptuel proposé ressemble à l'approche hybride de test de robustesse étudié par *Cavalli et al.* [CMM08] pour les protocoles de communication. Néanmoins, les cas de test de robustesse correspondent à des comportements inappropriés des clients de la couche fonctionnelle plutôt qu'aux fautes de communication considérées par *Cavalli et al.*

Nous supposons une architecture en couches avec des modules à la couche fonctionnelle basée sur l'architecture LAAS et des clients de modules dans une couche supérieure abstraite (la *couche d'application*) (Figure 1). Les clients peuvent envoyer des requêtes aux modules de la couche fonctionnelle pour les initialiser, les contrôler, et pour lancer et arrêter des *activités*.

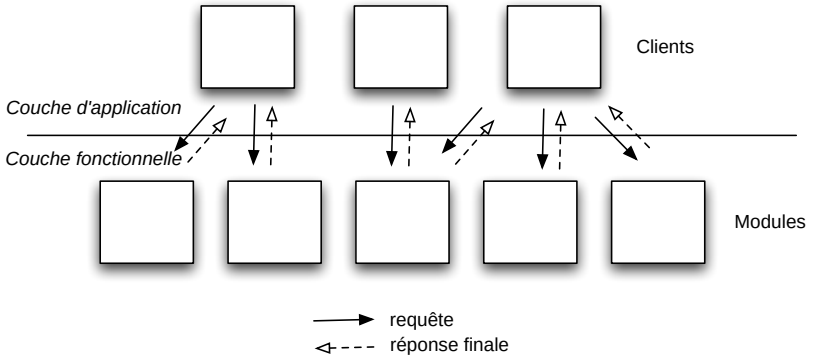


Figure 1: L'architecture du système

Pour chaque requête d'exécution, le module appelé renverra une *réponse finale* qui indique le résultat de l'activité demandée. On peut distinguer trois scénarios :

1. *Exécution normale* : la requête est acceptée par le module appelé, et la réponse finale indique le résultat de l'activité demandée. De façon nominale, l'activité renvoie *ok* comme la réponse finale, mais le programmeur peut spécifier d'autre raisons pour la terminaison de l'exécution dépendant de l'application.

2. *Requête rejetée* : la requête est rejetée par le module demandé, et la réponse finale indique la cause de ce rejet (par exemple, des paramètres incorrects).

3. *Exécution interrompue* : la requête est acceptée par le module appelé, mais un client envoie une autre requête d'exécution au même module. Dans ce cas, le module peut interrompre la première activité pour démarrer une nouvelle activité. La réponse finale à la première requête indique la cause de l'interruption.

Nous définissons la robustesse de la couche fonctionnelle en termes d'un ensemble de *propriétés de sûreté* que la couche fonctionnelle devrait respecter malgré des requêtes asynchrones envoyées par ses clients. Pour se protéger contre des requêtes envoyées au "mauvais moment", la couche fonctionnelle peut soit les rejeter, soit les mettre dans une file d'attente, soit forcer un changement de son état interne (par exemple, en interrompant une de ses activités courantes) afin de pouvoir les accepter.

Nous identifions les types suivants des propriétés des base. Pour une explication plus détaillée de ces propriétés, le lecteur devra se référer au paragraphe 3.2 du corps du mémoire :

Propriété *Pré-condition* $PC[x, C_{PRE}]$: une activité du type x est autorisée si une condition spécifiée C_{PRE} est vraie à l'instant que x est demandé.

La propriété $PC[x, C_{PRE}]$ peut être mise en vigueur :

- en rejetant les requêtes pour x ou les mettant en file d'attente tant que C_{PRE} est fausse, ou
- en forçant C_{PRE} à vraie (si cela est possible) afin de pouvoir accepter les requêtes pour x.

Propriété *Démarrage Exclue* (Excluded Start) $ES[x, y]$: une activité du type x est autorisée si il n'y a aucune activité en cours du type y à l'instant que x est demandé.

La propriété $ES[x, y]$ peut être mise en vigueur en rejetant les requêtes pour x ou en les mettant en file d'attente tant que l'activité y est en cours d'exécution.

Propriété *Exécution Exclue* (Excluded Execution) $EE[x, y]$: une activité du type x peut s'exécuter tant qu'il n'y a pas de requête pour une activité du type y.

La propriété $EE[x, y]$ peut être mise en vigueur en interrompant l'activité x en cours, afin de servir la requête pour y.

Propriété *Exclusion* $EX[x, y]$: une activité du type x est exclue par une activité du type y.

La propriété $EX[x, y]$ peut être mise en vigueur en rejetant les requêtes pour x ou les mettant en file d'attente tant que l'activité y est en cours d'exécution, et en interrompant l'activité x en cours afin de pouvoir servir la requête pour y. Notons que $EX[x, y] \equiv ES[x, y] \wedge EE[x, y]$.

Propriété *Exclusion Mutuelle* (Mutual Exclusion) $MX[x, y]$: une activité de type x et une activité de type y ne peuvent pas s'exécuter en même temps.

La propriété $MX[x, y]$ peut être mise en vigueur soit en rejetant une requête pour x (respectivement y) ou la mettant en file d'attente tant que l'activité y (respectivement x) est en cours d'exécution, soit en interrompant l'activité en cours du type x (respectivement y), afin de servir la requête pour y (respectivement x).

Nous considérons deux politiques de mise en vigueur pour $MX[x, y]$:

- *rejet mutuel*, noté $x \,{}^{R}\!\rightleftarrows_{R}\, y$, qui favorise l'activité en cours. La requête qui arrive en dernier sera rejetée ou mise en file d'attente. Avec cette politique de mise en vigueur, on dénote la propriété par $MX_R[x, y]$.
- *interruption mutuelle*, notée $x \,{}_{I}\!\leftrightarrows^{I}\, y$, qui favorise la nouvelle requête. La requête qui arrive en dernier interrompt l'activité en cours. Avec cette politique de mise en vigueur, on dénote la propriété par $MX_I[x, y]$.

Pour évaluer la robustesse du SUT, nous adoptons une approche de test passif. Nous observons le SUT et nous l'évaluons de façon totalement indépendante de son activation et de la génération des cas de test, au niveau de trace de requêtes et de réponses traversant son API. Notre but est de définir un oracle qui peut classer le comportement du SUT vis-à-vis d'un ensemble de propriétés de sûreté.

Pour chaque propriété P, nous classons le comportement du système pour chaque requête pertinente selon les résultats suivants :

vrai négatif (TN, *true negative*) : l'exécution de la requête est autorisée parce qu'elle ne met pas en danger la propriété P ; aucune invocation de la mise en vigueur de propriété (comportement correct) ;

vrai positif (TP, *true positive*): l'exécution de la requête est interdite parce qu'elle met en danger la propriété P ; la mise en vigueur de propriété est invoquée (comportement correct) ;

faux négatif (FN, *false negative*): l'exécution de la requête est interdite parce qu'elle met en danger la propriété P ; cependant, la mise en vigueur de la propriété n'est pas invoquée (comportement incorrect) ;

faux positif (FP, *false positive*): l'exécution de la requête est autorisée parce qu'elle ne met pas en danger la propriété P ; cependant, la mise en vigueur de la propriété est invoquée (comportement incorrect) ;

autre positif (op, *other positive*): l'exécution de la requête est interdite parce qu'elle met en danger une autre propriété $P' \neq P$, ce qui conduit à l'invocation de la mise en vigueur de la propriété P' au lieu de celle de P ;

non applicable (na): P n'est pas applicable à la requête considérée ;

trace tronquée (ω) : la fin de la trace est atteinte sans pouvoir établir une conclusion quant au comportement du système vis-à-vis de P.

Cas d'étude: le robot Dala

Notre cadre conceptuel a été appliqué à un cas d'étude, que nous présentons ici : la couche fonctionnelle du robot Dala qui, dans un scénario d'exploration planétaire, est actuellement utilisé au LAAS pour des expériences de navigation. Cette couche doit respecter des propriétés de sûreté pour protéger le robot de combinaisons d'activités qui pourraient conduire à un comportement dangereux. Dans notre cadre conceptuel, le test de robustesse active les mécanismes de mise en vigueur des propriétés de la couche fonctionnelle de Dala au moyen de mutations d'un script de mission d'exploration. Un script ainsi muté comporte des requêtes potentiellement *invalides dans le domaine temporel*, qui peuvent mettre en danger les propriétés de sûreté. Les traces d'exécution, contenant les requêtes et les réponses interceptées à l'interface de la couche fonctionnelle, sont ensuite traitées par un analyseur de trace afin d'évaluer la robustesse de la couche fonctionnelle.

Nous introduisons d'abord la couche fonctionnelle du robot Dala (notre système sous test), et les propriétés de sûreté qui doivent être mise en œuvre par celui-ci. Ensuite, nous présentons notre environnement de test de robustesse, qui est une application du cadre conceptuel FARM [AAA+90], et les caractéristiques de la campagne de tests qui a été menée. Enfin, nous présentons et analysons les résultats du cas d'étude.

Le système sous test est une configuration de la couche fonctionnelle du robot Dala comportant cinq modules, dont certains communiquent directement avec le matériel du robot :

- Rflex : commande des roues et odométrie ;
- Sick : capteur laser de distance ;
- Aspect : carte 2D de l'environnement ;

- Ndd : navigation et évitement d'obstacle ;
- Antenna : communication (simulée) avec un orbiteur.

La couche fonctionnelle se compose d'un ensemble de modules qui communiquent directement avec le matériel de robot (Figure 2). La "couche d'application" qui envoie des requêtes aux modules est ici une couche exécutive implémentée dans Open-PRS [ICAR96]).

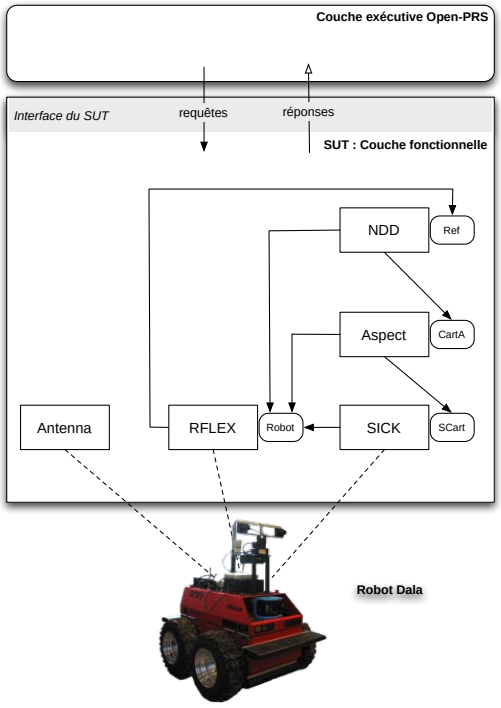


Figure 2: Le système sous test: la couche fonctionnelle du robot Dala

Nous définissons quatre familles de propriétés de sûreté pour le robot Dala. À chaque famille de propriétés correspond une des propriétés de base de la section précédente.

- PEX(*module*) - *Precondition for EXec request*: Pour chaque module, il doit y avoir au moins une requête d'initialisation qui termine avec succès avant que le module puisse traiter une requête d'exécution de type quelconque.
  Propriété de base: Pré-condition $PC[x, C_{PRE}]$
- AIB(x) - *Activity x Interrupted By*: Des activités du type x doivent être inactives ou être interrompues si une quelconque requête d'un type qui domine le type x est reçue.
  Propriété de base: Exécution Exclue $EE[x, y]$
- PRE(x) - *activity x PREceded by*: Une activité de type x ne peut pas être exécutée avant qu'un ensemble spécifié d'activités ait terminé avec succès.
  Propriété de base: Pré-condition $PC[x, C_{PRE}]$
- EXC(x, y) - *mutual EXClusion between activites x and y*: Des activités de types x et y ne peuvent pas s'exécuter en même temps ; priorité à la requête la plus récente.
  Propriété de base: Exclusion mutuelle par interruption $MX_I[x, y]$

Les quatre familles de propriétés de Dala sont détaillées à la Section 4.2.

La figure 3 illustre notre environnement de test, qui est une application du cadre conceptuel FARM [AAA+90] (cf. §2.2.1). La procédure de test se compose des étapes suivantes :

1. Création manuelle d'un *script d'or* qui définit une mission du robot d'exploration (l'ensemble d'activations **A**).

2. Génération d'une base de *scripts mutés* (l'ensemble de fautes **F**) en appliquant une procédure de *mutation* au script d'or.

3. Soumission de l'ensemble des scripts mutés à OpenPRS afin d'activer les mécanismes de robustesse du SUT.

4. Sauvegarde des traces d'exécution dans une *base de traces* (l'ensemble de relevés **R**).

5. Utilisation de l'*analyseur de traces* pour analyser et traiter la *base de traces* afin d'obtenir des verdicts de robustesse du SUT (une extension de l'ensemble de relevés **R**).

6. Evaluation de la robustesse (l'ensemble des mesures **M**).

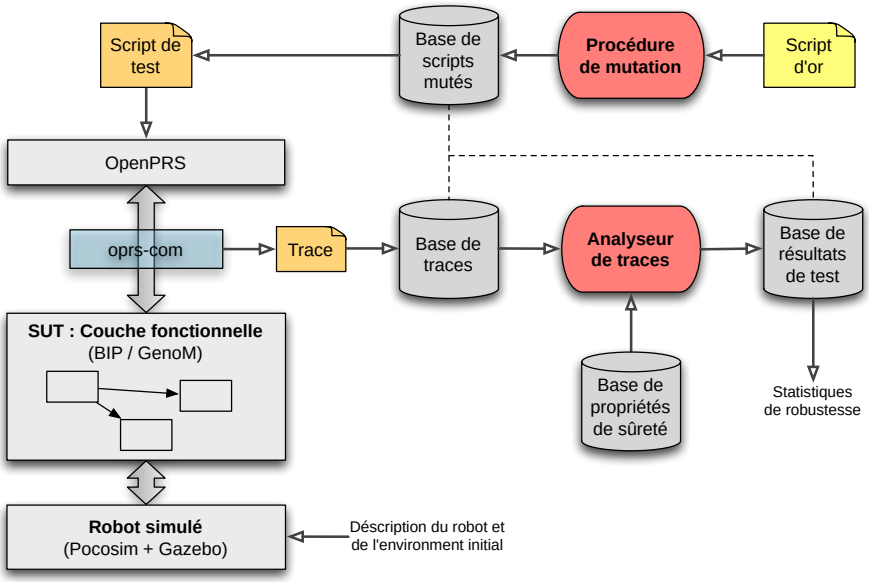


Figure 3: L'environnement de test de robustesse

Le vrai robot Dala est remplacé par un robot simulé car, d'une part, le test doit être automatisé afin d'effectuer un grand nombre de tests et, d'autre part, l'exécution de scripts mutés pourrait avoir des conséquences très dangereuses avec un vrai robot. Une présentation plus détaillée sur l'environnement de test (la mission, la procédure de mutation) ; et l'analyseur de trace se trouve dans la Section 4.3.

L'approche de test de robustesse proposée a été appliquée à trois implémentations différentes de la couche fonctionnelle simplifiée de Dala représentée sur la Figure 2 :

G^en oM : une implémentation mature développée avec l'environnement $G^{en}oM$, qui fournit quelques protections de base qui mettent en vigueur seulement les propriétés des familles PEX and AIB.

BIP-A : une implémentation préliminaire utilisant le cadre conceptuel BIP, avec une grande proportion du code BIP produite automatiquement à partir des descriptions de module de $G^{en}oM$, et des protections supplémentaires générées à partir de connecteurs entre composants BIP.

BIP-B : une implémentation plus aboutie utilisant le cadre conceptuel BIP, avec plusieurs corrections résultant des expériences effectuées sur *BIP-A*.

Nous avons soumis aux SUT les 293 scripts de test qui ont été générés par l'application de la procédure de mutation au script d'or, qui définit une mission du robot d'exploration Dala. Les traces d'exécution de ces tests, contenant les requêtes et les réponses interceptées à l'interface de la couche fonctionnelle, sont traitées par l'analyseur de trace afin d'évaluer la robustesse de la couche fonctionnelle. Une analyse détaillée des résultats de test se trouve dans la Section 4.6. Les principaux résultats sont résumés ici sur les tableaux 1 à 3.

Table 1: Résumé des résultats par trace

	Nombre de traces	Traces bloqués	Traces avec $\geqslant 1$ FN	Traces avec $\geqslant 1$ FP	Traces avec $\geqslant 1$ anomalie	Robustesse de trace (T_{ROB})
$G^{en}oM$	293		74	5	76	74.1%
BIP-A	293	42	40		80	72.7%
BIP-B	293	1	11		12	95.9%

Le tableau 1 donne les nombres de traces d'exécution présentant des anomalies (blocages, faux négatifs, faux positifs) et la proportion T_{ROB} de traces robustes (celles ne présentant aucune anomalie). On constate que l'implémentation *BIP-B* offre une robustesse sensiblement plus élevée que l'implémentation $G^{en}oM$ de référence. Les faux négatifs dans les 11 traces dénombrées ont été analysés manuellement et nous avons conclus qu'il s'agit de verdicts de test incorrects dus à un problème de fausses observations inhérent à notre approche de test "boîte noire" (voir §4.6.1.2)). Après correction manuelle, la robustesse de trace pour *BIP-B* serait de $292/293 = 99,7\%$, avec une seule trace anormale (blocage) qui résulte d'un défaut résiduel dans cette implémentation dû à une erreur lors de la génération du code exécutable.

Table 2: Taux de vrais positifs (%)

	$G^{en}oM$	*BIP-A*	*BIP-B*
PEX	100.0	95.7	99. 7
AIB	99.7	99.8	99.7
PRE	0	0	99.4
EXC	0	100.0	100.0
All	93.1	96.3	99.7

Table 3: Taux de faux positis (%)

	$G^{en}oM$	*BIP-A*	*BIP-B*
PEX	0	0	0
AIB	0.3	0	0
PRE	0	0	0
EXC	0	0	0
All	0.1	0	0

Les tableaux 2 et 3 donnent des taux de vrais et faux positifs résumés au niveau des familles de propriétés, puis globalement, toutes propriétés confondues.

On observe sur le tableau 2 une croissance du taux de vrais positifs avec les implémentations successives à base de BIP, grâce à la possibilité de mise en place de protections supplémentaires offerte par cette approche. En fait, le taux de vrais positifs atteindrait 100% pour *BIP-B*, après correction manuelle des verdicts incorrects dus au problème de fausses observations déjà mentionné.

Sur le tableau 3, une seule anomalie est constatée : un taux de faux négatifs non-nul pour la famille de propriétés AIB dans l'implémentation de référence $G^{en}oM$. Après analyse, il s'avère qu'une caractéristique de $G^{en}oM$ n'a pas été documentée : toute requête d'initialisation d'un module interrompt toute activité en cours sur ce module. Comme cette caractéristique n'est pas documentée, notre oracle pour la famille de propriétés AIB conclut à un faux positif. Il est intéressant à remarquer que l'oracle ne trouve aucun faux positif dans les tests des implémentations à base de BIP car, cette caractéristique n'étant pas documentée, elle a tout simplement était omise de ces nouvelles implémentations !

En conclusion de ce cas d'étude, on peut observer que le test de robustesse s'est montré comme un complément utile à l'approche de développement formel de BIP. Il a permis d'analyser et de comprendre le fonctionnement réel des différentes implémentations. Deux anomalies non couvertes par la méthodologie BIP ont ainsi été révélées : une caractéristique non-documentée et donc omise du modèle BIP correspondant ; une erreur dans la génération du code exécutable.

Conclusions et orientations futures

La construction des systèmes sûrs a toujours été un défi pour les ingénieurs. Avec la demande croissante en systèmes autonomes, il devient de plus en plus important qu'ils soient construits avec une sûreté de fonctionnement démontrable, particulièrement vis-à-vis de propriétés de sûreté interdisant des comportements contradictoires ou dangereux. En effet, la violation de propriétés de sûreté par un système autonome critique peut être catastrophique en termes humains ou économiques. En conséquence, il est indispensable de mettre en œuvre des mécanismes efficaces de mise en vigueur de ces propriétés et de fournir des preuves convaincantes de leur implémentation correcte. Le travail présenté dans cette thèse est une contribution dans ce sens.

Dans cette thèse, nous avons résumé les notions principales de la sûreté de fonctionnement informatique et des systèmes autonomes, et présenté un état de l'art des travaux récents sur le test de robustesse. Nous avons identifié deux types de test de robustesse : le *test*

de robustesse vis-à-vis des entrées et le *test de robustesse vis-à-vis de la charge*. Nous avons également classifié les techniques de test de robustesse vis-à-vis des entrées en trois catégories: des approches basées sur un modèle du domaine d'entrée, des approches basées sur un modèle de comportement et des approches hybrides. La contribution principale de cette thèse est la définition d'une approche hybride de test de robustesse qui est une combinaison de l'injection de fautes aléatoires et du test passif. Nous avons défini un cadre conceptuel pour évaluer la robustesse des mécanismes de protection intégrés dans un système par rapport à des entrées asynchrones inopportunes. À notre connaissance, ce problème n'a pas été abordé auparavant.

Nous avons proposé une méthode et une plate-forme pour tester la robustesse des mécanismes de mise en vigueur des propriétés de sûreté implémentés dans la couche fonctionnelle d'un système autonome avec une architecture hiérarchique. L'application au robot Dala montre plusieurs avantages de notre méthode. L'adoption d'une approche de test boîte noire nous a permis d'effectuer la campagne de test sans disposer d'une spécification formelle du comportement interne du système sous test (SUT). Avec peu ou pas d'informations sur les activités ou les états internes du SUT, nous avons pu comparer l'efficacité des mécanismes de protection de trois implémentations différentes de la couche fonctionnelle en s'appuyant sur la catégorisation des comportements possibles vis-à-vis des propriétés de sûreté. Nous pensons que cette approche est appropriée pour le test de composants "sur étagère", pour lesquels une spécification formelle de comportement n'est pas toujours disponible.

La technique de test passif nous a permis d'évaluer la robustesse du SUT en se basant sur un traitement *hors ligne* des traces d'exécution de test. Dans notre cas d'étude, le processus de test se compose de deux phases : une phase d'activation du SUT avec les 293 cas de test (qui prend environ 25 heures) et une phase d'analyse des résultats d'exécution avec l'oracle (qui prend environ 30 minutes). La technique de test passif appliquée aux traces d'exécution sépare l'observation de système du processus d'activation de système, et nous évite donc de re-exécuter entièrement le jeu de test (25 heures) chaque fois nous voulions raffiner l'oracle de test, par exemple, pour ajouter une nouvelle propriété.

Nous avons essayé de définir des propriétés de sûreté les plus génériques possibles. À cette fin, nous avons défini cinq propriétés de base, avec leurs procédures de mise en vigueur, qui peuvent être instanciées en tant qu'exigences de robustesse temporelle de la couche fonctionnelle d'un système autonome: *pré-condition*, *démarrage exclue*, *exécution exclue*, *exclusion (asymétrique)*, et *exclusion mutuelle*. Nous pensons qu'elles sont suffisamment générales pour être appliquées à d'autres systèmes. Pour chaque propriété de base, nous avons également défini l'oracle de test de robustesse correspondant.

Nous avons développé et présenté un environnement de test qui nous a permis d'évaluer la robustesse de la couche fonctionnelle du robot Dala en injectant des entrées invalides dans le domaine temporel. À partir d'une *charge de travail* (une mission typique d'un explorateur planétaire) décrite par un script, nous avons perturbé le SUT en créant des scripts mutés contenant des entrées soumises au "mauvais moment". La simulation du matériel physique du robot et de son environnement facilite notre procédure de test intensif et garantie que les fautes injectées ne peuvent donner lieu qu'à des dommages "virtuels". Cependant, la simulation ne peut pas totalement remplacer le test sur une vraie plate-forme, qui peut révéler des phénomènes qui sont difficiles à simuler correctement (par exemple, des aspects temps réel ou des imprécisions des moyens de perception et d'actuation).

L'implémentation de l'oracle comme un ensemble de requêtes de SQL s'est montrée d'une grande souplesse et facile à maintenir. L'environnement de test a montré son efficacité en comparant et en évaluant différents systèmes. En effet, grâce aux possibilités offertes pour explorer la réaction du SUT vis-à-vis à des entrées invalides dans le domaine temporel, notre

approche de test de robustesse permet à la fois l'élimination des faute (en étudiant les conséquences de l'injection de faute), et la prévision des fautes (évaluation) au moyen de mesures statistiques sur le comportement de système vis-à-vis de l'occurrence de fautes.

Cependant, notre approche présente certaines limitations.

Le test sans une spécification formelle de comportement du SUT peut conduire à la définition d'un oracle inexact, et les testeurs doivent donc l'améliorer de façon progressive et manuelle. La question est comment ? Dans l'approche hybride de test de robustesse proposée par *Cavalli et al.* [CMM08], les auteurs ont vérifié l'exactitude de leurs invariants en les validant par rapport à un modèle formel du comportement de système avant de les déployer dans l'oracle de test de robustesse. Cette approche ne pouvait pas être utilisée dans notre contexte puisque nous n'avions aucune spécification digne de foi des implémentations comparées. Nous avons dû analyser manuellement les résultats produits par l'oracle pour identifier des singularités (par exemple, un trop grand nombre de *faux positifs* ou de *faux négatifs*), et puis examiner les traces d'exécution pour diagnostiquer l'origine des singularités (une inexactitude de l'oracle ou un réel mauvais comportement de SUT). En effet, l'oracle et le SUT sont testés "dos-à-dos" et itérativement corrigés. De ce point de vue, notre travail a été facilité par le fait que nous disposions de plusieurs implémentations distinctes, dont une ($G^{en}oM$) était une implémentation mature (du moins, par rapport à un sous-ensemble des propriétés de sûreté requises).

Une autre limitation est la possibilité de verdicts incorrects en raison de fausses observations, qui sont inévitables vu que nous ne pouvons pas contrôler le temps de propagation des événements dans le SUT. Dans le cas d'étude de Dala, nous avons conclu que tous les *faux négatifs* observés sur l'implémentation *BIP-B* correspondaient en fait à de tels verdicts incorrects. Dans chaque cas, il y avait une explication plausible de comment un comportement correct du SUT pourrait être incorrectement interprété en tant que comportement incorrect à cause des retards de propagation.

L'inverse est tout aussi possible, c'est-à-dire, une mauvaise interprétation d'un comportement incorrect comme étant un comportement correct. Malheureusement, de telles mauvaises interprétations ne peuvent pas, par principe, être identifiées car le comportement observé ne présente pas de singularités (c'est le comportement attendu) et il n'y a donc aucune raison de douter. Cela est particulièrement problématique pour la mauvaise interprétation d'un faux négatif en tant que vrai négatif, car cela serait optimiste du point de vue de la sécurité-innocuité. Ainsi, un niveau plus fin d'observation du SUT sera probablement nécessaire pour tester la robustesse de systèmes extrêmement critiques.

Le travail a présenté dans cette thèse ouvrent plusieurs directions de recherche futures.

Une direction permettant d'améliorer l'approche serait de réduire le nombre de verdicts incorrects de test induits par le test boîte noire. Au moins deux axes de recherche complémentaires peuvent être considérés :

1. Prendre en compte explicitement le temps réel dans les oracles de propriété pour signaler des verdicts "suspicieux". Dans notre cas, seul l'implémentation *BIP-B* présentait un nombre suffisamment faible d'anomalies pour envisager une analyse manuelle et fastidieuse des traces d'exécution.

2. Étudier les modifications ou les extensions possibles au protocole d'interface du SUT pour faciliter le test de robustesse (par exemple, en exigeant que la prise en compte des requêtes soit systématiquement acquittée).

Une autre direction d'amélioration peut être envisagée au niveau de la génération des

cas de test. En particulier, une automatisation plus poussée de la génération de cas de test serait souhaitable. Par exemple, il pourrait être possible d'adapter un outil automatique de mutation de programmes, tel que SESAME [CWLP06], pour injecter automatiquement des fautes dans le script d'or. Autrement, une génération plus déterministe des scripts de test pourrait être envisagée, en visant, par exemple, une activation systématique des mécanismes de mise en vigueur des propriétés.

Il serait intéressant aussi (et relativement simple) d'étendre notre approche de test afin d'inclure des propriétés classiques portant sur le domaine des valeurs en entrée.

Plus généralement, une direction de recherche intéressante serait de fournir un *guide* pour conseiller les concepteurs de systèmes sur le choix et la définition des propriétés de sûreté à assurer. Un aspect d'un tel guide serait une méthode de définition des propriétés à partir d'une analyse de risque de l'application considérée. Un autre aspect pourrait être la définition de propriétés "composables", c'est-à-dire, des propriétés qui peuvent être établies à partir de combinaisons d'autres propriétés. La décomposition d'une propriété complexe en plusieurs propriétés plus simples faciliterait les processus de définition des propriétés et d'implémentation d'oracles de test correspondants.

Introduction

Autonomous systems cover a broad spectrum of applications, from robot pets and vacuum cleaners, to museum tour guides, planetary exploration rovers, deep space probes and, in the not-too-distant future, domestic service robots. As autonomous systems are deployed for increasingly critical and complex tasks, there is a need to demonstrate that they are sufficiently reliable and will operate safely in all situations that they may encounter.

By reliable, we mean that the autonomous system is able to fulfill its assigned goals or tasks with high probability, despite unfavorable endogenous and exogenous conditions, such as internal faults and adverse environmental situations. Reliability in the face of adverse environmental situations (sometimes referred to as (system-level) robustness) a particularly important requirement for autonomous systems, which are intended to be capable of operating in partially unknown, unpredictable and possibly dynamic environments.

By safe, we mean that the autonomous system should neither cause harm to other agents (especially humans) in its environment nor should it cause irreversible damage to its own critical resources. Protection of its own integrity is in fact a pre-requisite for an autonomous system to be reliable: if critical resources are no longer usable, then no amount of automated reasoning will be able to find a course of actions enabling the system to fulfill its goals.

This thesis addresses the assessment of a particular type of safety mechanism for an autonomous robot, implemented within its control software. The safety mechanism aims to enforce a set of *safety constraints* that specify inconsistent or dangerous behaviors that must be avoided. Examples of safety constraints are, for instance, that a mobile manipulator robot should not move at high speed if its arm is deployed, or that a robot planet observation satellite should not fire its thrusters unless its camera lens is protected.

The safety constraint enforcement mechanisms are implemented within the lowest layer of robot control software (called here the *functional layer*), which interfaces directly with the robot hardware. Typically, such a software layer contains built-in system functions that control the robot hardware and provides a programming interface to the next upper layer (which, for the time being, we will call the *application layer*). Specifically, clients of the functional layer (situated at the application layer) can issue requests to initialize modules, update their internal data structures, or start and stop various primitive behaviors or *activities*, such as: rotate the robot wheels at a given speed, move the robot to given coordinates whilst avoiding obstacles, etc.

Application-layer clients can build more complex behaviors by issuing asynchronous requests to start and stop activities at the functional layer[2]. We consider that the high-level safety constraints are expressed in terms of *safety properties* that place restrictions on when given functional layer activities can be executed. For example, a property might require

[2]It is useful to be able to issue requests asynchronously so that application-layer clients can use abstract representations of robot behavior and operate in different timeframes to that of the functional layer.

mutual exclusion between activities x and y. Thus, if an application-layer client issues a request for x while y is executing, enforcement of the property would require, for example, the request for x to be rejected.

In this thesis, we define a method for assessing the effectiveness of such property enforcement mechanisms. We address the problem from the perspective of *robustness testing*, where robustness if defined as the "degree to which a system or a component can function correctly in the presence of invalid inputs or stressful environmental conditions" [IEE90]. From our perspective, an invalid input is an application-layer request that, if executed in the current state of the functional layer, would cause a safety property to be violated. We specifically address invalidity in the time domain (i.e., requests issued at the "wrong" moment), although our approach could easily be extended to embrace the more classic notion of invalidity in the value domain (i.e., incorrect request parameters).

We adopt a random testing approach based on fault injection, through which a large number of test cases are generated automatically by mutating a sequence of valid inputs. Our test approach thus allows robustness evaluation in the sense that we can provide descriptive statistics of the robustness behavior of the system under test (in our case, a functional layer implementation) with respect to the population of test cases. Moreover, we follow a black-box testing approach, which does not consider internal details of the system under test. Thus, a set of test cases generated using our approach can be applied as a robustness benchmark to compare different functional layer implementations.

The dissertation is structured as follows.

Chapter 1 presents some definitions and general notions about autonomous systems and dependability. We present a general discussion on the notion of robustness in autonomous systems, and then focus on robustness at the level of the functional layer of a hierarchical autonomous system architecture. Finally, we introduce some methods and techniques that address safety property enforcement in such a functional layer and which constitute the framework for the theory, methods and tools developed in this thesis.

Chapter 2 presents a state of the art of robustness testing. We distinguish two main types of robustness testing: load robustness testing and input robustness testing. We then survey different approaches for input robustness testing, which we classify in two broad categories: approaches based on input-domain models, where robustness testing is performed with respect to invalid inputs that are generated based on a formal specification of system inputs, and approaches based on behavior models, which use a formal model of the system under test to conduct the test. A third hybrid category is then considered, in which test cases are generated using an input-domain model and robustness is adjudicated by means of robustness oracle formalized as a set of invariants on system interface behavior.

Chapter 3 introduces the framework that we propose for assessing the robustness of the functional layer of a hierarchically-structured autonomous system. As we consider robustness as safety properties that the functional layer should ensure, we present in this chapter a set of safety property classes that are implementable in the functional layer and study possible property enforcement policies. The description of safety enforcement policies allows us to define an oracle to characterize the behavior of the system under test by observing behavior at its service interface (only). Our approach thus considers the system under test as a black box, which allows comparison of different implementations offering the same functionality.

Chapter 4 presents an application of our framework to a case study: the Dala planetary exploration rover robot. Our framework exercises the property-enforcing mechanisms of the Dala functional layer by executing mutated exploration mission scripts containing temporally

invalid test inputs that may endanger the safety properties. We use our approach to compare the robustness of several implementations of the Dala functional layer.

Finally, a conclusion summarizes the essential points of this thesis, and presents several directions for future research.

Chapter 1

Dependability and Robustness in Autonomous Systems

In this introductory chapter, we first present some general notions about autonomous systems and dependability, and then discuss the dependability issues in hierarchical autonomous system architectures. Finally, we present the lowest "functional" layer of such a hierarchical architecture, which will be the focus of the theory, methods and tools developed in this dissertation.

1.1 Autonomous systems

We first discuss the notion of autonomy and then describe current architectural approaches for building autonomous systems.

1.1.1 Autonomy

The Oxford English Dictionary defines autonomy as "the condition of being controlled only by its own laws, and not subject to any higher one". It thus essentially refers to the right or ability of an entity to act upon its own discretion, i.e., self-rule or independence. In the robotics domain, autonomy, however, means more than just the independent operation of a system.

There are several works that have tried to define and characterise the degree of autonomy of a system. The McGraw-Hill Science and Technology Dictionary defines an autonomous robot as one that not only can maintain its own stability as it moves, but also can plan its movements. *Fogel* [MAM+00] described autonomy of a system as "the ability to generate one's own purposes without any instruction from outside", which indicates that systems are able to carry out intelligent actions of their own volition. *Clough* in [Clo02] also stresses the characteristic of an autonomous system "having free will". This paper also proposed four essential abilities of an autonomous system (perception, situational awareness, decision-making and cooperation), along with a scale, based on these abilities, for classifying the level of autonomy of a system. It should be noted that "autonomous" is different than "automatic". The fact that a system is automatic means that it will do exactly as programmed, whereas an autonomous system seeks to accomplish goal-oriented tasks whose implementation details are not defined in advance and without instructions from outside.

Huang [Hua07] proposed the following definition of an autonomous system, which specifically characterizes an autonomous robotic system interacting with humans:

> Autonomy: An unmanned system's own ability of sensing, perceiving, analyzing, communicating, planning, decision-making, and acting, to achieve its goals as assigned by its human operator(s) through desired Human Robot Interaction. Autonomy is characterized into levels by factors including mission complexity, environmental difficulty, and level of Human-Robot Interaction (HRI) to accomplish the missions.

This definition of "autonomy" underlines two important factors in autonomous systems. The first factor is the decision-making capacity, obtained through the application of various techniques from artificial intelligence decision theory. The second is the capacity of dealing with the difficulty, the uncertainty and the evolution of the system's environment. This aspect is related to the notion of *robustness*.

The construction of systems that operate autonomously is a major challenge for roboticists. In the following section, we review the main software architectures that have been proposed for structuring autonomous systems.

1.1.2 Autonomous system architecture

There are three main accepted software architectures for structuring autonomous systems: the subsumption architecture [Bro86], the hierarchical architecture [Gat97] and the multi-agent architecture [MDF+02]. Most practical systems currently use the hierarchical approach: the CIRCA architecture [MDS93] developed by Honeywell Research Center, the CLARAty architecture [VNE+01] of NASA's JPL (Jet Propulsion Laboratory), and the LAAS architecture [ACF+98, ICA01] developed by LAAS-CNRS.

The hierarchical architecture divides the autonomous system software into layers corresponding to different levels of abstraction. Each layer has different temporal constraints and manipulates different data presentations. The architecture is typically composed of three layers [Gat97]: a decisional layer, an executive layer, and a functional layer. We present here the LAAS architecture as a typical example of a hierarchical architecture for autonomous systems (Figure 1.1).

1. The *decisional layer*: At the top of the hierarchy, this layer carries out the decision-making process of the system, including the capacities of producing task plans and supervising their execution. It takes charge of generating task plans from goals given by operator, which are then processed by the executive layer, and deals with the informations (reports, errors, etc.) sent from the lower layers.

   In the LAAS 3-layer architecture, this layer uses the IxTeT planner [GL94] to produce the task plan. IxTeT is a temporal constraint planner, combining high level actions to build plans, and capable of carrying out temporal execution control, plan repair and re-planning.

2. The *executive layer*: This layer carries out the task plans (made by the decisional level) by choosing the elementary functions that the functional layer must execute. It also reacts to errors or failed tasks, referring the problem to the decisional layer when unable to solve it itself.

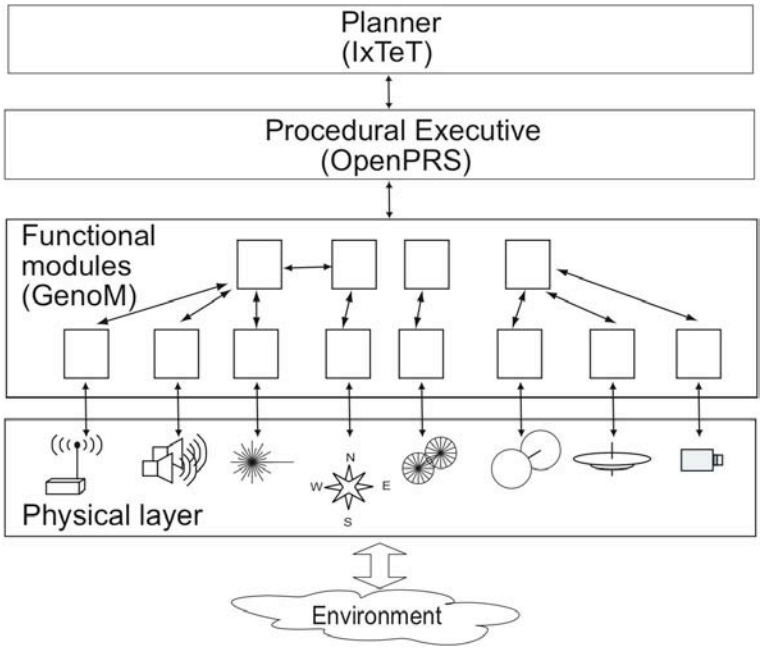


Figure 1.1: The LAAS architecture

In the LAAS architecture, OpenPRS (Open Procedural Reasoning System) [ICAR96] is used to execute the task plan sent from the decisional layer, while being at the same time reactive to events from the functional levels. The procedural executive OpenPRS is in charge of decomposing and refining plan actions into lower-level actions executable by functional components, and executing them. This component links the decisional component (IxTeT) and the functional level. During execution, OpenPRS reports any action failures to the planner, in order to re-plan or repair the plan.

3. The *functional layer*: This is the layer that controls the basic hardware and provides a functional interface for the higher-level components. It includes all the basic built-in system functions and perception capabilities (obstacle avoidance, trajectory calculation, communication, etc.).

   In the LAAS architecture, these functions are encapsulated into $G^{en}oM$ modules [FHC97]. Each module can be in charge of controlling a hardware component (e.g., a camera, a laser sensor, etc.) or accomplishing a particular functionality (e.g., navigation). Modules provide services that can be activated by requests from the executive layer and export data for use by other modules or higher layers. The algorithms in modules are decomposed into code elements called *codels*.

1.2 Dependability

In this section, we give a brief presentation of the basic concepts and techniques of dependability.

1.2.1 Key concepts

The original definition of *dependability* of a system is the ability to deliver services that can justifiably be trusted [LAB+96, ALRL04]. This definition stresses the need for justification of trust. The concepts of dependability can be organized as a tree containing three branches (Figure 1.2): the *attributes* of dependability, the *threats* that can endanger the dependability of a system, and the *means* (methods and techniques) to achieve dependability.

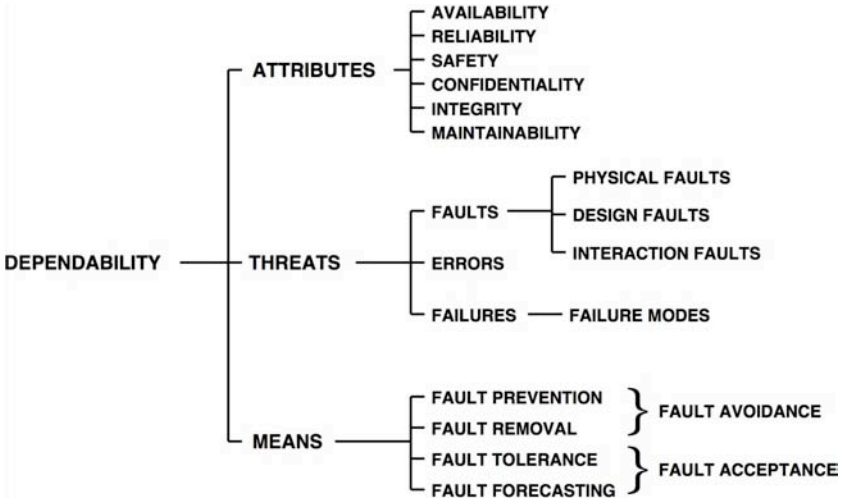


Figure 1.2: Dependability tree (from [ALRL04])

Attributes The dependability concept includes different properties depending on the system and the environment that the system works in, but there are six general attributes that must be considered: *availability*, *reliability*, *safety*, *integrity*, *maintainability*, and, in the security domain, *confidentiality* [LAB+96, ALRL04].

Threats There are three categories of threats that can endanger the dependability of a system: *faults*, *errors* and *failures*. A *failure* is defined as an event that occurs when the delivered service deviates from correct service. An *error* is defined as that part of the system's total state that may lead to a failure. A *fault* is the cause of an *error*.

An activated fault produces an error that can propagate in a component or spread from one component to another to provoke a failure. The failure of a component causes a permanent or transient fault in the system that contains the component, and that component may deliver an incorrect service to other components (or to an external system if the considered component is an interface component). This "chain of threats" is illustrated in Figure 1.3

Means Many methods can be used in synergy to make systems dependable. These methods can be classified into four main categories:

- Fault prevention: development and specification methods used to prevent the occurrence or introduction of faults.
- Fault tolerance: the use of redundancy to avoid service failures in the presence of faults; fault tolerance is carried out via error detection and system recovery

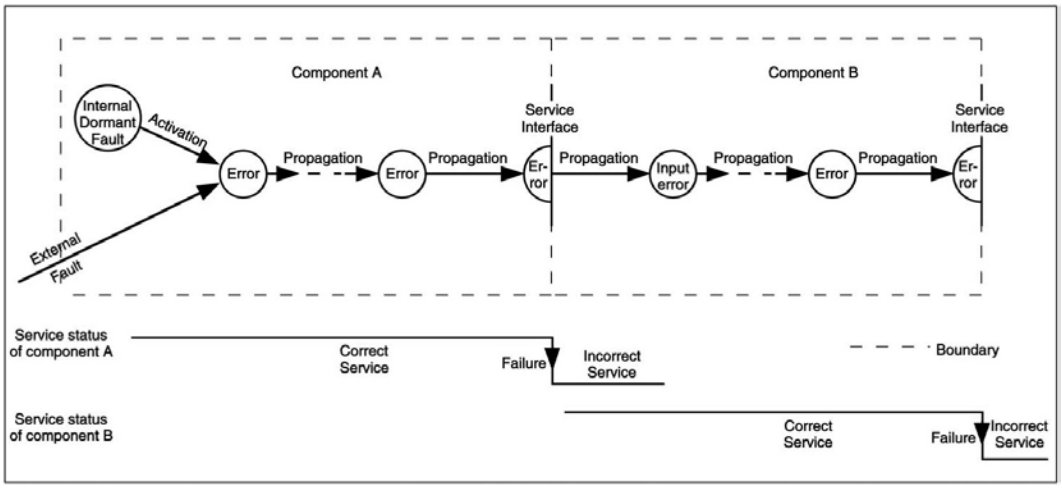


Figure 1.3: Error propagation (from [ALRL04]

- Fault removal: methods aimed at reducing the number and severity of faults; during the development phase, fault removal is carried out by verification, diagnosis and correction; during operation, fault removal corresponds to preventive and curative maintenance.
- Fault forecasting: how to estimate the presence, the future incidence, and the likely consequences of faults; it entails an evaluation of the system's behaviour with respect to fault occurrence or activation.

Fault prevention and fault removal can be grouped into the notion of *fault avoidance*: how to create a system with as few faults as possible. Fault tolerance and fault forecasting can be grouped into the notion of *fault acceptance*: how to live with a system despite faults.

1.2.2 Techniques for dependability

We briefly review each of the four categories of dependability means in the specific context of autonomous systems.

1.2.2.1 Fault prevention

The main techniques for preventing faults during the development of autonomous systems are the use of software decomposition techniques and software design tools.

Using *software decomposition* allows a complex system to be built out of smaller software components or modules, which facilitates system development and maintenance. Furthermore, properly-designed modules can be re-used in another system if they fit that system's requirements, reducing system development to new modules for new functionality, and integration of new and pre-existing modules. From the dependability point of view, this decomposition simplifies software development and testing, and thus reduces the risk of fault occurrence.

Modular decomposition can be found in all general architectures for autonomous systems (e.g., the hierarchies of the LAAS and CLARAty architectures, or the multi-agent architecture in IDEA), as well as within layers of these architectures (e.g., the functional and the decisional level of the LAAS architecture are composed of various modules and components).

The use of *software design tools* facilitates system development and helps to prevent the introduction of faults thanks to automatic code and document generation. Various tools have been used for the development of autonomous systems, for example the $G^{en}oM$ toolkit [FHC97] in the LAAS architecture, the ControlShell [SCPCW98], ORCCAD [BCME+98] and SIGNAL environment [MRMC98], the YARP robot platform[1].

1.2.2.2 Fault removal

The two main techniques of fault removal are formal verification and testing. *Formal verification* is used during the development phase of a system life-cycle. Formal verification allows developers to check whether the system, or system components, satisfies certain properties. If the properties are not satisfied, developers can diagnose the design to identify the faults that caused the verification to fail, and then perform the necessary correction. These three steps are repeated to check that fault removal had no undesired consequences. We will briefly present such a formal verification approach in Section 1.4.2.

Verifying a system through exercising it (including via simulation) constitutes dynamic verification, which is usually simple termed *testing*. Testing is an essential process in autonomous system development. Testing can target a particular component or function, like a decisional mechanism or a functional layer module (unit testing), or the whole system. Observing the test outputs and deciding whether or not they satisfy the verification conditions is known as the *oracle problem* [Gau95]. This problem is one of the main challenges for testers.

This thesis addresses a particular form of testing – *input robustness testing* – which is concerned with verifying the mechanisms that protect a system from invalid inputs. A state of the art specifically addressing robustness testing will be presented in Chapter 2.

1.2.2.3 Fault tolerance

Fault tolerance, which is aimed at failure avoidance, is carried out via error detection and system recovery. There exist two classes of error detection techniques:

- *concurrent error detection*: which takes place during service delivery;
- *preemptive error detection*: which take places while normal service delivery is suspended or as a background activity, and checks the system for latent errors and dormant faults.

In autonomous systems, concurrent error detection [LLC+05, DGDLC10] is mainly implemented through:

- timing checks (watchdogs), to check that critical functions have not been halted, e.g., as in the RoboX autonomous system [TTP+03];
- reasonableness checks, e.g., to verify that critical variables remain in specified intervals;
- acceptance or "safety-bag" checks, to verify before execution, that commands respect certain properties or assertions, e.g., as in the R2C component [Py04] developed for the LAAS architecture.

[1] http://eris.liralab.it

Preemptive error detection in autonomous systems often takes the form of model-based fault diagnosis, and is widely used to reveal dormant hardware faults [GDRS00, HKSW07].

System recovery transforms a system state containing errors into a state that can be activated again without detected errors and faults. Error handling and fault handling are two techniques in recovery. Error handling tries to eliminate errors from the system state. Three forms of error handling are:

- roll-back, where the system is brought back to a saved sate that existed prior to error occurrence.
- roll-forward, where the system is transformed to a new state without detected errors
- compensation, where the erroneous state contains enough redundancy to enable error elimination

Fault handling, a process to prevent faults from being activated again, consists in four steps: diagnosis, fault isolation, system reconfiguration and reinitialization. Usually, after fault handling, a corrective maintenance is carried out to removing faults that were isolated by fault handling.

In autonomous systems, the most common faults are physical faults affecting sensors or effectors [CM05] so system recovery is either implemented through fault-tolerant control techniques (see, e.g., [DCY05]), which can be viewed as form of error-handling though compensation, or limited to fault handling, through which functionally redundant hardware is exploited to find a system configuration that can allow continued operation. Examples of the latter approach include the Mode Identification and Reconfiguration (MIR) component of the RAX architecture [MNPW98] and the Global Recovery Module (GRM) of the CO-TAMA architecture [DGDLC10]. Tolerance of software design faults is rarely considered. One notable exception is the fault-tolerant planning capability experimented in the LAAS architecture, that implements roll-back recovery to tolerate residual design faults in planner domain models [LGG+07a, LGG+07b].

1.2.2.4 Fault forecasting

Fault forecasting is conducted by performing an evaluation of the system behaviour with respect to fault occurrence. Evaluation has two aspects: qualitative evaluation, which aims to classify the failure modes and the events that lead to system failures; and quantitative, or probabilistic evaluation, which aims to measure in terms of probabilities the extent to which some of the attributes are satisfied. Example of fault forecasting applied to autonomous systems can be found in [CM03, CM05, TTP+03]. The results vary quite considerably. For example, [CM03, CM05] describe results in which software design faults are the causes of 30% to 54% of system failures, whereas [TTP+03] reports software faults to be the cause of 90% of the critical failures.

This thesis also addresses a particular form of fault forecasting – *input robustness evaluation* – which is concerned with quantifying the effectiveness of the mechanisms that protect a system from invalid inputs.

1.3 Robustness in autonomous systems

The common definition of *robust* is something "strong and healthy" and the property of something being robust, i.e., *robustness*, is frequently used in this sense in various scientific fields (e.g., economy, biology, statistics and automatic control). The same is true in the field of robotics, where the term is often used as a sort of synonym for dependability (see, e.g., [IG03, YHY09, Gra10]). In this section, we first review some more focussed definitions of this concept and give illustrations of robustness in the context of autonomous systems.

1.3.1 Robustness definitions

In dependable computing, Avizienis *et al.* [ALRL04] refer to robustness as a specialized secondary attribute of dependability, which they define as "*dependability with respect to* ***external faults****, which characterizes a system reaction to a specific class of faults*".

In the context of autonomous systems, Lussier [Lus07] defines robustness as "*the delivery of a correct service despite* ***adverse situations*** *arising due to an uncertain system environment (such as an unexpected obstacle)*". Lussier's definition of robustness broadens the dependable computing definition beyond "external faults" to include other phenomena "outside" the considered system that can negatively impact its operation, but that cannot really be considered as "faults" (e.g., the roughness of the terrain faced by a mobile robot). The purpose of Lussier's definition is to distinguish the set of "*robustness*" techniques for ensuring the dependability of an autonomous system with respect to external phenomena from "*fault-tolerance*" techniques for ensuring dependability of an autonomous system with respect to faults affecting its own (internal) resources (e.g., actuator faults, sensor faults, processor faults, or software design faults).

In software engineering, robustness has been defined as "*the degree to which a system or component can function correctly in the presence of* ***invalid inputs*** *or* ***stressful environmental conditions***" (IEEE Std. 610-12, 1990 [IEE90]). This definition is quite general and covers the two earlier definitions: "external faults" can affect the system either as invalid functional inputs (e.g., incorrect data or unexpected input events) or as non-functional (and therefore "invalid") inputs (e.g., electromagnetic interference); "adverse situations" affecting the system can be viewed as stressful environmental conditions. We will return to this definition when defining the notion of robustness testing in Chapter 2.

1.3.2 System-level robustness

The main reason for the popularity of the term "robustness" in the field of autonomous systems is certainly due to the emphasis that is placed on such systems being able to deal with partially unknown, unpredictable and possibly dynamic system environments (such as rough terrain and unexpected obstacles). Autonomous systems can be made robust to such situations if they are able to sense the current environment, assess the situation with which they are faced, plan a course of actions to reach their goals, and carry out that plan, while simultaneously updating the plan as new information becomes available (including changes to the environment). Moreover, least commitment planning techniques allow plans to be flexible and accommodate new situations as they unfold [Wel94] . Also, plan repair and re-planning allow autonomous systems to be robust to new situations which invalidate their current plan [LC04, FGLS06].

However, freedom to act autonomously and flexibly in the face of the unexpected can lead

to chaos unless due care is taken. The uncertainty of the environment can lead to unforeseen events that may endanger the system or its environment (e.g., a steep slope can cause the robot to slide down with great speed). There is thus a need to equip autonomous systems for critical applications with appropriate safety mechanisms, such as safety monitors in charge of detecting and reacting to dangerous situations (see for example, [RRAA04, GPBB08, OKWK09]), or safety interlocks or property enforcers, in charge of preventing the execution of actions that could violate safety constraints (see, for example, [ACMR03]).

The latter are particularly important in hierarchical autonomous systems since, to remain tractable, high-level domain models (e.g., for planning) must remain at a high level of abstraction, without introducing low-level details such as the state of hardware controllers and other functional layer modules. As a consequence, it may be the case that a client of the functional layer (e.g., a plan execution controller) issues requests for actions that are in conflict with the current low-level state and could, if executed, cause the functional layer to reach an inconsistent or dangerous state, with potentially undesired or even catastrophic consequences. It may also be the case that inconsistent requests are issued due to bugs in hand-written client code. Ideally, therefore, the functional layer should protect itself against requests that are "invalid" given its current state, i.e., the functional layer should itself be *robust*.

1.3.3 Robustness of the functional layer

The functional layer of a hierarchical autonomous system (cf. the hierarchical LAAS architecture of Figure 1.1) typically consists of a set of modules, possibly controlling different hardware devices. Safety constraints may be imposed between modules to prohibit inconsistent or dangerous behaviors. For example:

- A mobile manipulator robot should not move at high speed if its arm is deployed.
- A planet observation robot satellite should not fire its thrusters unless its camera lenses are protected.
- An industrial cleaning robot should only rotate its brush when it has been lowered to be in contact with the floor.
- A planetary exploration rover should not move while taking a high definition science photo.
- A personal assistant robot should not open its gripper while holding a bottle.

In this thesis, we consider the enforcement of such safety constraints as a robustness problem. A request to the functional layer is deemed to be invalid if its execution would violate a safety constraint. The functional layer is then said to be robust if, despite such invalid requests, it correctly enforces the specified set of safety constraints.

Altisen *et al.* [ACMR03] tackled the enforcement of safety constraints such as those listed above by means of a property-enforcing layer, built using controller synthesis techniques, and placed immediately above the functional layer. Py *et al.* [PI04] implemented a similar approach for the LAAS architecture by means a component, called "Request and Resource Controller" (R2C), that captures all the incoming service requests and checks that they do not lead to a prohibited state. If so, alternate actions are proposed (e.g.,request rejection, process suspension, etc.) to keep the system safe and consistent. Figure 1.4 presents the architecture of the R2C controller:

- *Input Buffer* captures the incoming events from the system. Possible events are: requests from the upper (decisional) layer for functional layer services, reports regarding the execution of those services, and data changes.

- *System State Database* maintains a representation of the system state. If the state has changed, it activates the State Checker.

- *State Checker* verifies whether incoming events lead to a bad state. If so, it deduces actions to keep the system in a consistent state with properties. If not, incoming events are forwarded as is.

- *Output Buffer* launches actions deduced by R2C (including forwarded events) and reports to the service clients at the decisional layer.

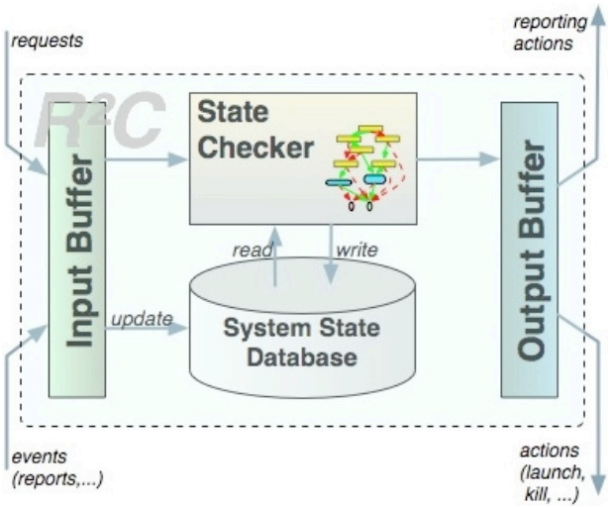


Figure 1.4: The Request and Resource Checker (R2C) (from [PI04])

In the next section, we present a new property enforcement approach that has been recently been applied to the functional layer of the LAAS architecture.

1.4 Software design tools for the LAAS architecture functional layer

In this final section, we first briefly present the $G^{en}oM$ development environment [FHC97], which is used to design and implement the modular functional layer of the LAAS architecture. We then outline the BIP framework developed at Vérimag [BBS06], which supports a model-based compositional design methodology for specifying, verifying and implementing heterogenous real-time programs. Finally, we summarize the application of BIP to $G^{en}oM$ in the framework of the MARAE project[2], which underpins the work described in the remainder of this thesis.

[2]MARAE signifies "*Méthode et Architecture Robustes pour l'Autonomie dans l'Espace*" (Robust Method and Architecture for Autonomy in Space). The project was partially supported by the *Fondation Nationale de Recherche en Aéronautique et l'Espace* (FNRAE) and carried out by a consortium consisting of LAAS-CNRS, Vérimag and EADS Astrium

1.4.1 The *GenoM* environment

The *GenoM* (Generator of Modules) environment was developed to facilitate the design of re-usable real-time modules and their integration into the LAAS hierarchical architecture for autonomous systems [FHC97]. Each module of the functional layer of the LAAS architecture is responsible for particular functionalities of the robot. The modules are instances of a common template, called the *generic module*, that can support synchronous and asynchronous processes and offers a standard communication interface. A module is responsible for a physical resource (sensor or actuator) or a logical resource (data). It embeds the necessary algorithms and functions along with basic error- and fault-handling mechanisms (e.g., error detection, procedure interruption, etc.) to control the resource and ensure its integrity. Complex modalities (such as navigation) are obtained by a cooperation between modules. For example, we can design a module to control a camera (taking photos) or a proximity sensor, etc., or to process data acquired from other modules, as would be the case, say, for a path planning module.

Each module provides a service interface composed of several requests which can be called by users or higher levels to control (i.e., parameterize, start or stop) services of module. Every service has one or more associated requests, which may contain input parameters. There are two types of requests:

- Execution request: a request that starts an actual service, called an *activity*.
- Control request: a request that only modifies internal data of a module. It is mainly used to set parameters so it has a very short execution time. It does not carry out any resource allocation or activity creation.

The client is informed of the execution of an activity by a reply. There are two types of replies:

- Intermediate reply: a reply that indicates the effective start of an activity.
- Final reply: returned when a service is terminated. When a service is terminated, a reply is returned to the client. This reply includes an execution report, which qualifies the result of the service execution (e.g., `OK` - service has terminated normally, `E_BLOCK` - error message with domain-specific semantics such as 'something prevents the movement of robot'), and possibly data results. The execution report allows the client to react appropriately to the service execution (e.g., re-plan the trajectory, stop a service, etc.).

Figure 1.5 illustrates the general structure of a *GenoM* module. A module contains two internal databases (called Internal Data Structures): the functional IDS (*fIDS*) and the control IDS (*cIDS*) dedicated to the internal routines. The *fIDS* includes all the data related to a module: request parameters, request replies, etc. It also provides a means to exchange data between tasks inside a module. The *cIDS* contains parameters relating to the execution of a module (e.g., states of the activities, period, etc.). A module contains two types of tasks (threads under Unix) that execute code: the control task and one or more execution tasks.

- Control task: receives the requests for the module, checks the validity of the requests' parameters and checks for the absence of conflicts, and then executes the request (case

of a control request) or allocates the corresponding execution task (case of an execution request).

- Execution tasks: contain one or more activities relating to user code that implement particular functions. Execution tasks are cyclic tasks (threads in most implementations) that execute the activities corresponding to the active services.

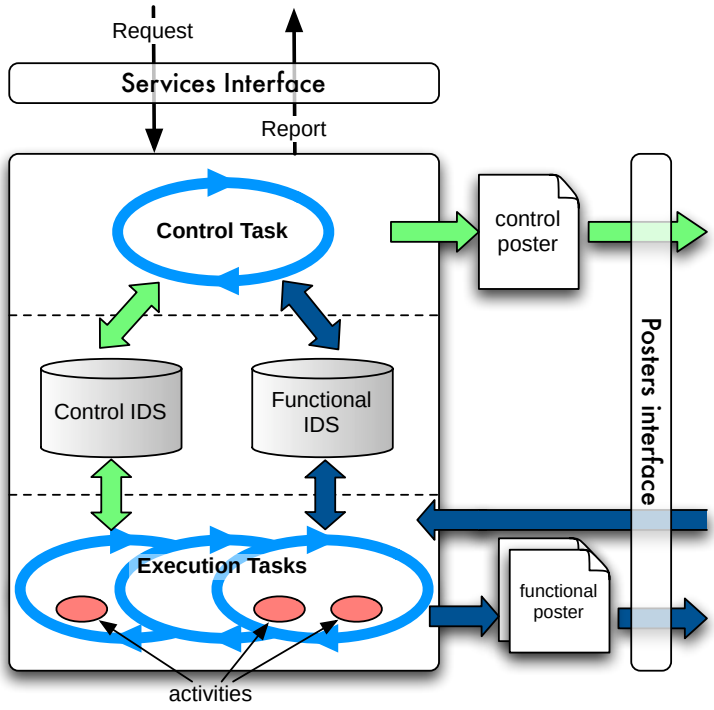


Figure 1.5: General structure of a module

During the execution, activities may have to use data produced by other modules (e.g. a navigation module uses data from an obstacle map builder module to plan a trajectory without hitting the obstacles). These exchanges are done via *posters*. A *poster* of an activity is a structured shared memory that is may be read by other activities in the module but can only be updated by its owner.

Each module is an instance of this generic module. To create new modules with the generic structure, the $G^{en}oM$ environment allows developers to describe a module in the $G^{en}oM$ declarative language and automatically generates the new module (in C language). A module description contains five parts: module declaration, data structures and fIDS declaration, requests definition, posters definition, and execution tasks declaration. Developers then fill in the generated module skeleton with their own code elements (called *codels*) to implement the algorithms of the module's functions.

Figure 1.6 illustrates the automaton of an activity as executed by all launched services. Initially in the ETHER state, indicating no activity of service, the automata goes to the START state when an activity is created, and then to the EXEC state when the activity is running. Depending on the termination of the activity, the automaton can reach the END state (normal termination), the FAIL state (termination with error), or the INTER state (termination by interruption).

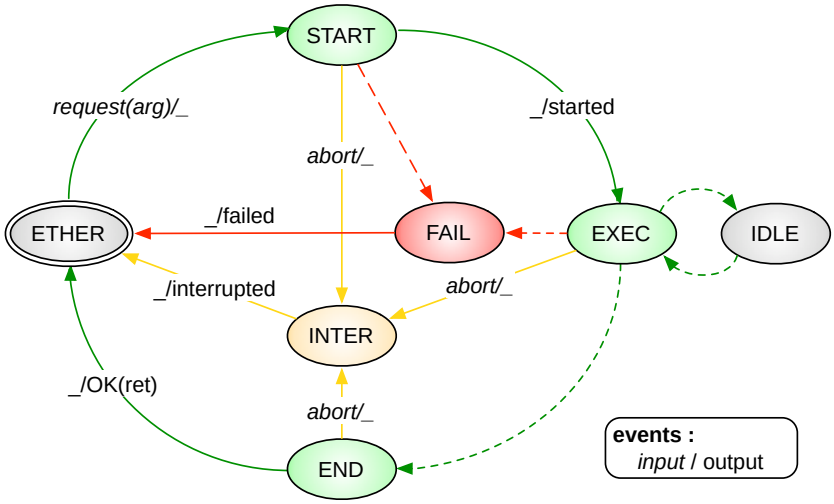


Figure 1.6: Control graph of an activity

1.4.2 The *BIP* framework

BIP [BBS06] is a framework for modeling heterogeneous real-time programs. The name BIP is derived from **B**ehavior, **I**nteraction and **P**riority, the three main foundations of the framework. Paraphrasing from [BBS06], the main characteristics of BIP are the following:

- BIP supports a model-based design methodology where parallel programs are obtained as the superposition of three layers. The lowest layer describes behavior. The intermediate layer includes a set of connectors describing the interactions between transitions of the behavior. The upper layer is a set of priority rules describing scheduling policies for interactions. Layering implies a clear separation between behavior and structure (connectors and priority rules).

- BIP uses a parameterized composition operator on components. The product of two components consists in composing their corresponding layers separately. Parameters are used to define new interactions as well as new priority rules between the parallel programs. Such a composition operator allows incremental construction, i.e., obtaining a parallel program by successive composition of other programs.

- BIP provides a mechanism for structuring interactions involving strong synchronization (rendezvous) or weak synchronization (broadcast). Synchronous execution is characterized as a combination of properties of the three layers.

BIP allows hierarchical construction of compound components from atomic ones by using connectors and priorities:

- *Atomic components* are a class of components with behavior specified as a set of transitions and having empty interaction and priority layers. Triggers of transitions include ports, which are action names used for synchronization.

- *Connectors* are used to specify possible interaction patterns between ports of atomic components.

- *Priority relations* are used to select amongst possible interactions according to conditions depending on the state of the integrated atomic components.

An atomic components consists of: *(i)* a set of ports $P = \{p_1 \dots p_n\}$ used for synchronisation with other components; *(ii)* a set of control states/locations $S = \{s_1 \dots s_k\}$, which denote locations at which the components await synchronisation; *(iii)* a set of variables V used to store local data; and *(iv)* a set of transitions modeling atomic computation steps. A transition is a tuple of the form (s_1, p, g_p, f_p, s_2), representing a step from control state s_1 to s_2. A transition is allowed to be executed if the guard g_p (boolean condition on V) is true and some interaction including port p is offered.

Figure 1.7 shows a simple atomic component with two ports *in*, *out*, variables x, y, and control states *empty*, *full*. At control state *empty*, the transition labeled *in* is possible if the guard $[0 < x]$ is true. When an interaction through *in* takes place, the variable x is modified and a new value for y is computed. From control state *full*, the transition labeled *out* can occur.

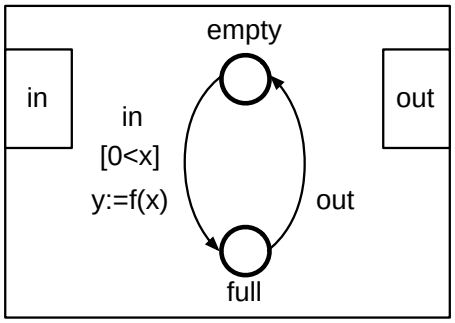


Figure 1.7: An atomic component (from [BBS06])

Components are built from a set of atomic components with disjoint sets of names for ports, control locations, variables and transitions. The notation for sets of ports is simplified by writing $p_1|p_2|p_3|p_4$ for the set $\{p_1, p_2, p_3, p_4\}$. A connector γ is a set of ports of atomic components which can be involved in an interaction. It is assumed that connectors contain at most one port from each atomic component. An interaction of γ is any non empty subset of this set. For example, if p_1, p_2, p_3 are ports of distinct atomic components, then the connector $\gamma = p_1|p_2|p_3$ admits seven feasible interactions: $p_1, p_2, p_3, p_1|p_2, p_1|p_3, p_2|p_3, p_1|p_2|p_3$.

Each interaction with more than one port represents a synchronization between transitions labeled with its ports. Given a connector γ, there are two basic modes of synchronization: *(i)* strong synchronization or *rendezvous*, when the only feasible interaction of γ is the maximal one, i.e., it contains all the ports of γ; and *(ii) weak* synchronization or *broadcast*, when feasible interactions are all those containing a particular port which initiates the broadcast.

Given a system of interacting components, many interactions can be enabled at the same time, which can lead to nondeterminism. Priorities can be used to restricted nondeterminism in the system behaviour. Priorities are a set of rules used to specify which of the interactions should be preferred among enabled ones. Finally, the model of a system is represented as a compound component, which defines new components from existing sub-components (atoms or compounds) by creating their instances, specifying the connectors between them and the priorities.

The *D-Finder* [BBSN08] tool implements a compositional method for checking safety properties of component-based system described in BIP. To cope with state explosion, D-Finder applies a divide-and-conquer approach: verify properties of individual components and infer global properties of systems from properties of their constituent components and constraints on their interactions. Verifying components separately limits state explosion. The method (illustrated on Figure 1.8) is based on the use of two types of invariants: *component invariants* and *interaction invariants*. Component invariants are over-approximations of the set of the reachable states of atomic components and are generated by simple forward propagation techniques. Interaction invariants, obtained by computing traps[3] of a Petri net representation of the interactions between components, express global synchronization constraints between atomic components. D-Finder is used for deadlock verification of systems described in BIP. From a BIP model, D-Finder applies proof strategies to eliminate potential deadlocks by computing increasingly stronger invariants.

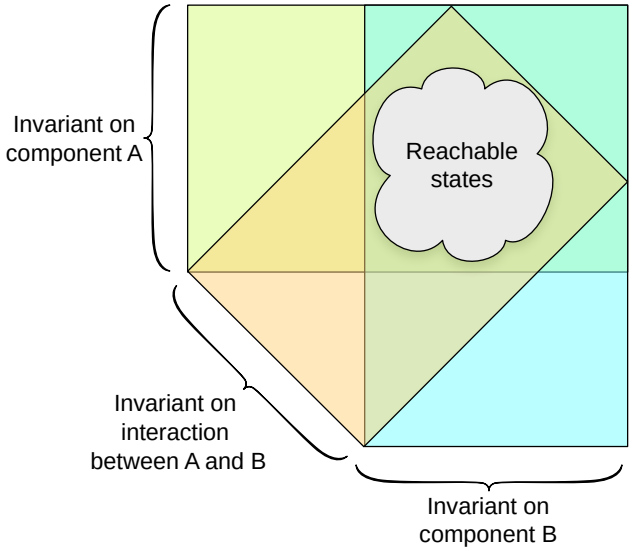


Figure 1.8: D-Finder compositional verification principle based on an over-approximation of the set of reachable states (from [BBSN08]

1.4.3 The MARAE context: *BIP* applied to $G^{en}oM$

The goal of the MARAE project was to design and apply the *BIP* framework to build a robust architecture for autonomous systems [BdSG+10]. The *BIP* framework was applied to two case studies: (i) the Dala autonomous rover robot; (ii) a Proba-like autonomous earth observation satellite (simulated). In Chapter 4, we will describe an application of the method developed in this thesis in the context of the Dala case study.

In both case studies, *BIP* was used to rebuild, with integrated safety enforcement, a functional layer previously developed with $G^{en}oM$. We present here the steps involved in this rebuild process.

[3] A nonempty subset of places Q in a Petri net is called a *trap* if every transition having an input place in Q has an output place in Q [Mur89].

A guiding principle was to retain the modular and hierarchical organisation of the LAAS architecture and map it into in the *BIP* framework. First, the $G^{en}oM$ generic module and its components are modelled in the *BIP* language. Then, each specific module is instantiated with this generic module. Finally, the existing codels (written in C) are connected to the resulting component and a *BIP* model of all the $G^{en}oM$ modules is obtained. Then, a *BIP* model of the interactions between the modules is added. The properties to be enforced by the functional layer are described within this interaction model. The resulting global *BIP* model is used to synthesize a controller for the overall execution of all the functional modules and to enforce, by construction, the constraints and the rules defined both within and between the various functional modules.

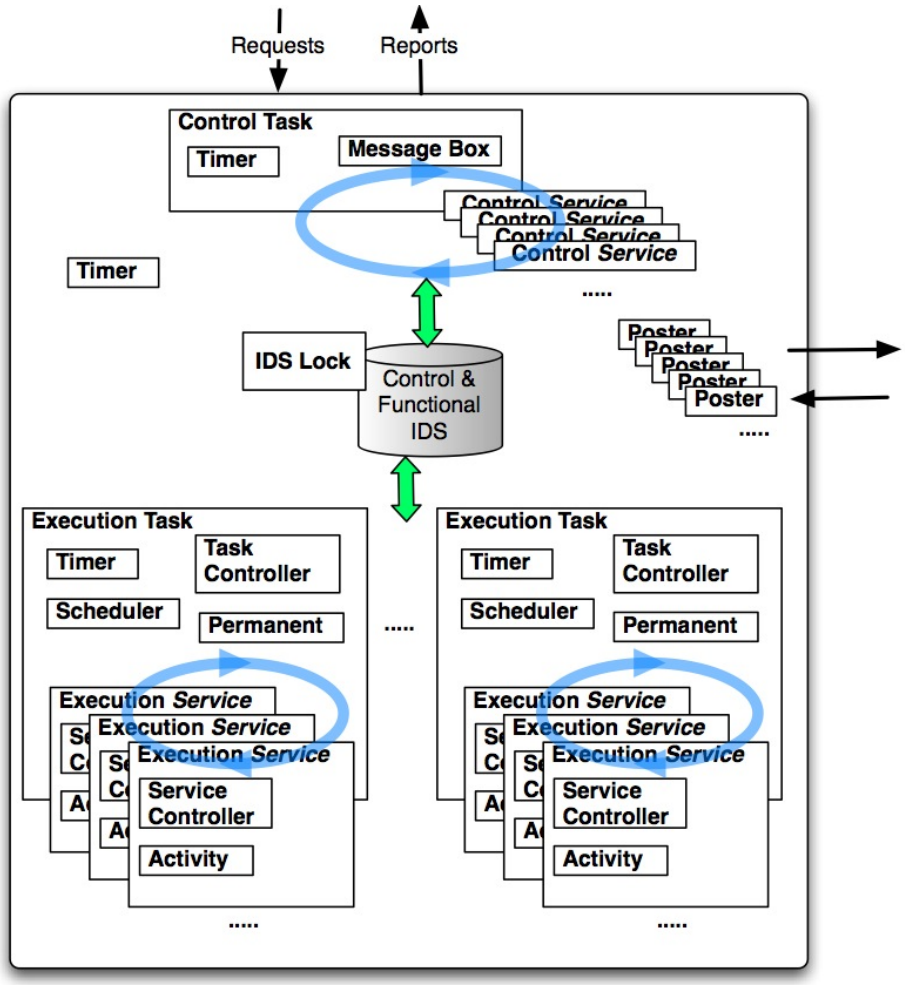


Figure 1.9: The $G^{en}oM$ generic module componentisation

The $G^{en}oM$ generic module structure (Figure 1.5) is modelled as a set of *BIP* components, as shown in Figure 1.9. In the componentization, an *Execution Task* is a compound component consisting of: a *Scheduler* (atomic) component, to control the execution of the associated *Activity* component of some *Service* component; a *Task Controller* (atomic) component to stop the Scheduler if none of the associated *Execution Service* components are running; a *Timer* component to control the execution period of the *Execution Task*; and a *Permanent* component. Data associated with the module is stored in the *Poster* components that are accessible by read and write operations. The *IDS Lock* component ensures mutual exclusion between different *Execution Task* components and *Execution Service* components

when manipulating *Poster* components. *Poster* components use the *Timer* component (in the overall *Module*) to determine how much time has elapsed, in terms of "ticks", since the last modification to their data.

To form compound components from several atomic components (such as *Execution Service* components), the necessary connectors are added between the latter. In turn, these compound components are combined using connectors to form the even more compound component *Module*, corresponding to a $G^{en}oM$ module. By combining components "bottom-up" in this way, if constituent components are proven to be correct with respect to some properties, we obtain also a resulting compound component that is also correct with respect to these properties.

The *BIP* framework allows developers to integrate safety property enforcement inside a module (intramodule) or between modules (intermodule) in the functional level, instead of through a separate layer or component such as R2C (cf. Section 1.3.3). To do this, one has to identify in the safety constraints the components involved, then add connectors between component ports and the appropriate guards (if needed). The *BIP* runtime execution engine will then automatically synchronize the components in order to guarantee these properties.

As an example, the following connectors check the pre-condition of an activity A of module M: A may only be started if there is at least one successfully completed B service and at least one successfully completed C service.

```
connector allow_A_if_B_and_C_is_set(M.do_A, M.status_B, M.status_C)
define [M.do_A, M.status_B, M.status_C]
on M.do_A, M.status_B, M.status_C
provided M.status_B.done ∧ M.status_C.done
do {}

connector reject_A_if_B_and_C_is_not_set(M.do_A, M.status_B, M.status_C)
define [M.do_A, M.status_B, M.status_C]
on M.do_A, M.status_B, M.status_C
provided ¬M.status_B.done ∨ ¬M.status_C.done
do { M.do_A.rep ← B-OR-C-NOT-SET }
```

1.5 Conclusion

We presented in this chapter some basic notions of the two domains at the intersection of which our work is positioned: autonomous systems and dependability, along with the hierarchical approach for architecting autonomous systems. We also reviewed some general definitions of the robustness concept and its interpretation in the context of autonomous systems. In the scope of robustness in autonomous systems, we retain the following points:

- Autonomous systems must be made robust to deal with a dynamic system environment. Nevertheless, the uncertainty of the environment can lead to unforeseen events that may lead the robot into dangerous situations. Therefore, there is a need to construct safety mechanisms in autonomous systems to protect them and their environment. However, there has been little work to date responding to this need.
- The robustness of autonomous systems relies on the robustness of the functional layer. Indeed, the functional layer provides the basic services to control the system's physical

resources. If we can ensure the robustness in the functional layer, we can reduce the risk of catastrophic consequences to the system as a whole. Thus, robustness testing of the functional layer is an important confidence-building process that should be carried out during the construction of critical autonomous systems.

- As we consider the enforcement of safety constraints as a robustness problem, robustness testing must take into account the functionalities of the enforcement mechanisms integrated in the system.

The work in this thesis focusses on robustness testing in autonomous systems. In the next chapter, we present a state of the art in robustness testing to investigate techniques that might be applicable in the context of autonomous systems.

Chapter 2

Robustness Testing: State Of The Art

Robustness is defined as "the degree to which a system or component can function *correctly* in the presence of *invalid inputs* or *stressful environmental conditions*" (IEEE Std. 610-12, 1990 [IEE90]). Therefore, robustness testing aims to verify and evaluate this capacity of a system. It concerns the design and execution of test cases (invalid inputs, demanding workloads, etc.) that may lead to an incorrect operation of the system under test (SUT) (e.g., a crash or a deadlock), and processing of the test results in order to obtain a final verdict or assessment about the robustness of the SUT.

According to the definition of robustness by the IEEE, there are two important aspects to be considered in robustness testing: *invalid inputs* and *stressful environmental conditions*. *Invalid inputs* relates to external faults (from the system boundary viewpoint [ALRL04]) which can be injected either directly at the interface of the SUT, or into other systems that interact with the SUT in order to create an erroneous execution environment for the SUT. The *environment* is the entire set of conditions under which the system operates, be it the physical environment, the physical process with which it is interacting, or its workload. Software behaviour is significantly dependent on the environment in which it operates.

There are two objectives that may be addressed by robustness testing: *verification* of a system's protection mechanisms and *evaluation* of their efficacy when faced with adverse conditions. Robustness verification aims to determine if a system's behaviour conforms to its specification (including exceptional behaviour) and to *identify deficiencies* in its protection against external adverse situations, whereas robustness evaluation aims to *measure* the degree of protection that is provided.

Several approaches for robustness testing have been proposed in the literature, which can be classified in two broad categories: approaches based on input-domain models and approaches based on behaviour models. Methods based on input domain models perform robustness testing with respect to *invalid inputs* that are generated based on an informal or formal specification of system inputs, or by mutating supposedly valid inputs. Input-domain model-based approaches are usually based on some form of random fault injection and focus on robustness *evaluation*. Behaviour model-based approaches use a formal model of the SUT to derive test cases and to define an oracle to adjudicate whether or not the SUT is robust. These robustness testing methods focus on robustness *verification*. A third approach to robustness testing can be defined, called hybrid robustness testing, which uses both input-domain models (for test case generation using random fault-injection) and behaviour models (to define a model-based oracle for robustness adjudication).

This chapter is structured as follows. We first briefly survey different types of robust-

ness testing and then successively consider input-domain model-based robustness testing, behaviour model-based robustness testing and hybrid robustness testing.

2.1 Types of robustness testing

Broadly speaking, robustness testing may be concerned with either *invalid inputs* or *stressful environmental conditions*. Testing a system in the presence of invalid inputs leads to the notion of *input robustness testing*. Testing a system with respect to "stressful environmental conditions" can cover a whole set of techniques, depending on how this term is interpreted. Here, we only consider the influence of the environment on the system's *functional inputs*, i.e., we do not consider robustness due to physical environmental factors such as temperature, pressure, radiation, power fluctuations, etc. In the context of a system's functional inputs, we thus interpret "stress" from the environment in terms of the load submitted to the system, leading to the notion of of *load robustness testing*. In this section, we first consider the latter type of robustness testing (load robustness) and then focus on the former (input robustness).

2.1.1 Load robustness testing

In the testing literature, the term *load testing* is defined as the process of determining the SUT's behaviour under both normal and anticipated peak load conditions [Wika]. In software engineering, *load testing* is considered as a part of performance testing, which is used not so much to find bugs, but to determine the effectiveness of a computer, network, software program or device under a particular workload. Examples of load testing include, for instance: testing a printer by sending it a very large job, testing a word processor by editing a large document, or testing an e-commerce web site by emulating the activity of a large number of users. In the context of robustness testing, we define the term *load robustness testing* as the set of techniques that can be used to to assess the behaviour of a SUT beyond the anticipated peak load conditions. This set of techniques includes:

- *Stress testing*: Stress testing exercises software at the maximum design load, as well as beyond it [Soc04]. Testers subject the SUT to a stressful environment by making it process an unusually high workload in order to evaluate its ability to remain effective under stressful conditions. For example, an e-commerce web site may be tested by various denial of service tools to observe its behaviour. This type of testing is especially important for systems with large numbers of users.
- *Endurance testing* (also known as Soak testing [Wikc]): This type of testing aims to explore the system's behaviour under sustained use. The focus is on creating long-term stressful conditions, where testers run the SUT at high levels of load during a prolonged period of time. Indeed, after a long period of execution, it is possible for a system to be affected by several problems that can lead to poor performance or system failure, such as memory leaks. This type of testing aims to identify such defects.

2.1.2 Input robustness testing

Input robustness testing focuses on testing a system with respect to invalid inputs. Testers usually define "invalid" in the sense of inputs that are not described in the specification of the SUT. One classic invalid input selection technique is to choose values outside an input

parameter's domain. In addition to the input value domain, we can also define the term "invalid" in the time domain, in which case the validity of an input is considered not only in its value but also in the instant at which it is processed by the SUT. A valid command with valid parameters which arrives at a wrong time can cause the SUT to fail. For example, in an OS, a request to write into a file while it is being written by another thread can cause failure if the OS does not possess a correct implementation of mutual exclusion. Thus, the arrival of a correctly-formed input in an inappropriate state (as in the previous example) can still be considered as an invalid input.

In [SKRC07], *Saad-Khorchef et al.* proposed a similar classification of *inputs* for testing communication software. They reserve the term "*invalid input*" to designate either an unspecified input or a specified input that contains errors. They then use the term "*inopportune input*" to designate a syntactically correct input (existing in the alphabet of the specification), but not expected in the given state. Testing with respect to such inopportune inputs is similar in spirit to the work described in [WTF99], which explores the robustness of re-usable concurrent objects with respect to multiple inter-leavings and timings of method requests.

Here, we prefer to consider an input as "valid" if it is correct in both the value *and* time domains, so we identify two kinds of "invalid input" as well as two corresponding types of input robustness testing:

- *Invalid input*:
  - *Invalid value input*: Any unspecified input value. Some examples of this kind of input are wrong syntax input, out-of-range input value, in-range but incorrect value.
  - *Invalid time input*: Input event (valid in value domain) that is not expected in the current state.
- *Input robustness testing*:
  - *Value domain robustness testing*: Test cases that aim to fail the SUT by submitting values that do not satisfy the input domain specification.
  - *Time domain robustness testing*: Test cases that aim to fail the SUT by sending inputs with valid values, but that are not expected in its current state.

Various testing techniques to be found in the software engineering literature can be classified as *input robustness testing* techniques:

- *Negative testing* [Bei95] (also known as Dirty testing): A test whose primary purpose is falsification; that is, tests designed to break the software. Another definition can be found in [Wikb]: negative testing is testing technique where testers try to fail the SUT by sending improper inputs to determine the behaviour of the system outside of what is defined. Negative testing is thus synonymous to input robustness testing.
- *Fuzz testing*: Fuzz testing is a software testing technique that submits invalid, unexpected, or random data inputs to a program. An example of this technique is the CRASHME [Car96] tool. Fuzz testing thus blindly tests both value and time domain robustness.
- *Parameter corruption*: In this technique, testers try to replace the correct parameter values by invalid values (e.g., out-of-range values) or incorrect but valid values. This technique requires a thorough study of the input parameter specification in order to

determine appropriate selective parameter substitutions. Parameter corruption is evidently a form of value domain robustness testing.

- *Boundary value analysis* [Jor95, Soc04]: Test cases are chosen on and near the boundaries of the input domain of variables, with the underlying rationale that many faults tend to concentrate near the extreme values of inputs. Boundary value analysis is thus also a form of value domain robustness testing.

- *Syntax testing*: This is a black box testing technique in which test cases are designed based upon the syntax of the input domain to test if the program can execute properly formed syntax and reject bad syntax. Syntax testing is a form of value domain robustness testing that is specifically tailored to command-driven systems.

2.2 Robustness testing methods based on input domain models

Robustness testing based on input domain models is closely related to fault injection, which simulates faults and errors in order to see what impact they have and how the system behaves [Voa97]. In this section, we first present some background material on fault injection methodology and then some notable works in applying this method for robustness testing.

2.2.1 Fault injection methodology

Fault injection aims to exercise the error detection and fault tolerance mechanisms of a system by deliberately injecting faults into the system. It allows us to understand and discover system failures, and to evaluate and validate a system's dependability mechanisms. For realistic system testing, fault injection must be carried out in the presence of an appropriate *workload*. Indeed, previous research has shown that workload can heavily influence system behaviour regarding faults and errors [IRH86, MW88] and can have a significant impact on dependability measures obtained from fault injection experiments [CS92]. The notion of workload can vary according to the system under test. A workload could be a specific working environment, a given program or set of tasks that the system under test must process, or a particular set of system calls. The overall input space for a fault injection experiment thus consists of the product of the submitted workload and the set of injected faults (or "*faultload*").

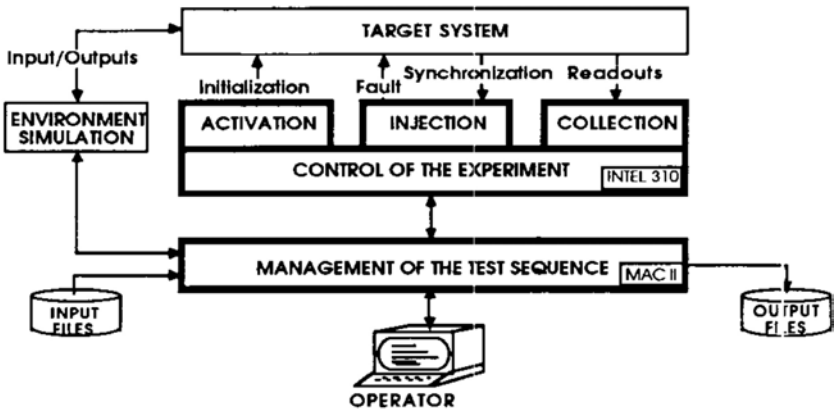


Figure 2.1: General architecture of MESSALINE (from [AAA+90])

In early work on fault injection for dependability validation, *Arlat et al.* [AAA+90] proposed the **FARM** fault injection framework, which consists of 4 elements: the inputs correspond to a fault set $\boldsymbol{F}$ and an activation set $\boldsymbol{A}$ to functionally exercise the system; the outputs correspond to a set of readouts $\boldsymbol{R}$ and a set of measures $\boldsymbol{M}$ derived from the readouts. The activation set $\boldsymbol{A}$ characterizes the workload of the SUT. Each experiment is executed with a fault f selected from $\boldsymbol{F}$ and an activation a selected from $\boldsymbol{A}$. The behaviour of the system is observed and saved into a readout r which characterizes the experiment result. An experiment is thus characterized by a triple $< f, a, r >$. The readout set $\boldsymbol{R}$, which is composed of the readout r from each experiment, is processed to derive one or more members of a set of measures $\boldsymbol{M}$ that characterize the dependability of the considered system. Figure 2.1 presents an implementation of this framework in the MESSALINE fault injection tool.

MESSALINE is a tool for physical level fault injection that is capable of adapting easily to various target systems and to different measures. The tool consists of four modules:

- A fault injection module to inject one element of the set F. This module includes 32 hardware fault injection elements connected to IC pins by means of a probe.
- A target system activation module to ensure the initialization of the target system according to the elements of the A set.
- A readout collection module to log the R set.
- A test sequence management module to compute the set of measures M, such as the asymptotic coverage of the hardware and software diagnostics.

MESSALINE has been used, for example, to test a computerized interlocking system (called PAI) for railway control applications designed by SNCF (French National Railway). In this study, the workload (A) corresponds to the execution of a test program to test the functions of the CPU (input, output, processor, RAM, etc.). The readout set (R) included four error detection signals that ensure hardware diagnosis: DA (double addressing), DTACK (lack of acknowledgement from memory and peripheral circuits), RAM (RAM addressing check), PROM (PROM addressing check). Another readout consisted of the messages delivered by the software diagnostic test program.

The testing campaign consisted of more than 6000 experiments on 144 IC's. As an example of the results obtained, Figure 2.2 presents the efficiencies (or "coverages") of the hardware diagnosis error detection mechanisms and, for comparison, an assessment of the efficiency of software diagnosis. The messages delivered by software diagnosis are decomposed into four classes: i) *complete diagnosis*: the message properly identifies the test module that has detected the error, ii) *muteness*: no message is delivered, iii) *garbage*: the message consists of a sequence of random symbols, and iv) *no detection*: the diagnosis message is identical to that obtained in the absence of injected faults.

2.2.2 FUZZ

Fuzz testing, or "fuzzing", is a software testing technique that submits invalid, unexpected, or random data inputs to a program. The name of this technique came from *Barton et al.* in [MKP+95, FM00] where they proposed *FUZZ*, a tool used to verify the robustness of several operating systems (Windows NT and various versions of Unix) and their applications.

Location	IC	Number of Experiments	Hardware Diagnosis					Software Diagnosis			
			Global Coverage	Error Detection Signals				Total	Form of the Message		
				DA	DTACK	RAM	PROM		Complete	Muteness	Garbage
Random Access Memory											
ZE7	TC5565	77	23.38%	0.00%	23.38%	20.78%	0.00%	98.70%	74.03%	23.38%	1.30%
ZE8	TC5565	77	24.68%	0.00%	24.68%	20.78%	0.00%	97.40%	72.73%	23.38%	1.30%
ZF7	TC5565	77	58.44%	0.00%	58.44%	57.14%	0.00%	94.81%	28.57%	55.84%	10.39%
ZF8	TC5565	77	59.74%	0.00%	58.44%	54.55%	3.90%	96.10%	22.08%	61.04%	12.99%
Read Only Memory											
ZL6	2716	66	42.42%	0.00%	42.42%	42.42%	0.00%	68.18%	19.70%	45.45%	3.03%
ZN6	2716	66	46.97%	0.00%	43.94%	42.42%	7.58%	78.79%	28.79%	43.94%	6.06%
ZJ7	27128	76	92.11%	0.00%	90.79%	85.53%	5.26%	97.37%	5.26%	86.84%	5.26%
ZJ8	27128	76	92.11%	0.00%	92.11%	82.89%	1.32%	97.37%	3.95%	90.79%	2.63%
Global Statistics		592	55.24%	0.00%	54.56%	51.01%	2.20%	91.72%	32.26%	54.05%	5.41%

Figure 2.2: Results obtained for the memory circuits (from [AAA+90])

This tool was developed based on the idea of delivering random events to the applications to stress them. The *FUZZ* program is basically a generator of random characters. It produces a continuous string of characters (printable or control characters) and then uses them as inputs for testing OS utilities and applications.

FUZZ was first used to test the reliability of 88 utilities on 7 versions of UNIX. Later, *Forrestor et al.* [FM00] used this tool in a study of the reliability of various application programs running on Windows NT. In this work, *FUZZ* was used to produce a faulty workload composed of random events and random Win32 messages. Random system events simulate actual keystroke or mouse events to be sent to the target application. Random Win32 messages consist of valid message types with completely random parameters. These messages are sent to the target application by using the Win32 functions $PostMessage$ and $SendMessage$. Figure 2.3 shows where these injections occur in the Windows NT architecture.

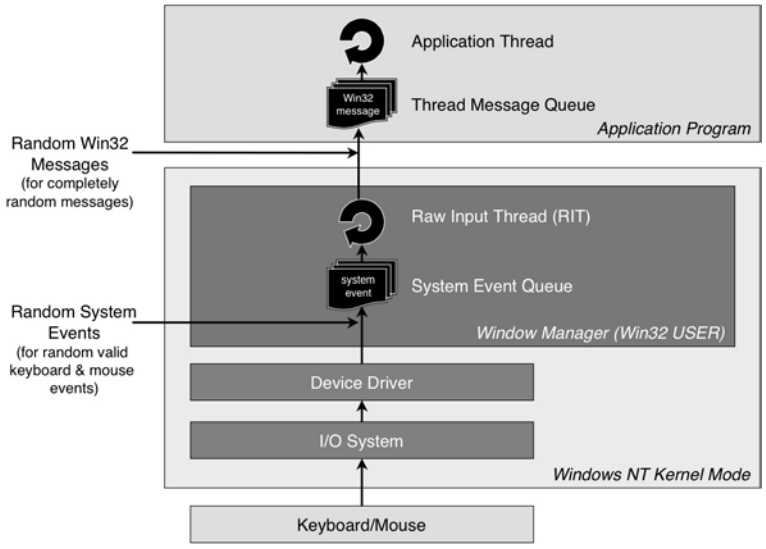


Figure 2.3: Injection of random inputs (from [FM00])

The robustness testing campaign described in [FM00] used a very large faultload, consisting of 1,000,000 random Win32 messages and 25,000 random events sent to 47 applications on Windows NT. Test outcomes are only observed in terms of the consequences at the application level. The outcome of each test is classified in one of three categories:

- *crash*: the application crashed completely
- *hang*: the application hung (stopped responding)
- *normal*: the application processed the input and was then closed via normal application mechanisms (testers pressed Alt-F4 or clicked the button "Close Window")

The lack of a more detailed inspection (e.g., event logs, error message logs) at a lower level of the system precludes a more profound observation and classification of the test outcomes.

2.2.3 BALLISTA

Ballista is a tool that was developed by *Koopman et al.* [KSD+97, KD99, KKS98] at Carnegie Mellon University to carry out robustness testing of off-the-shelf operating systems and middleware. It combines the ideas of both software testing and fault injection. The robustness testing methodology supported by *Ballista* is based on using combinations of valid and invalid parameter values for system calls and functions of a software component (see Figure 2.4.a). Each test case consists of a single system call execution with a particular set of parameter values. Each parameter is associated with a number of predefined values, which can be obtained from the system specification and from parameter analysis. These values are stored in a database of valid and invalid values based on the data type of each argument of the system call. The advantage of this approach is that these valid and invalid parameter values depend only on the parameter type, and thus allow the automatic generation of test cases from the specification regardless of the functionality of the system call (see Figure 2.4.b).

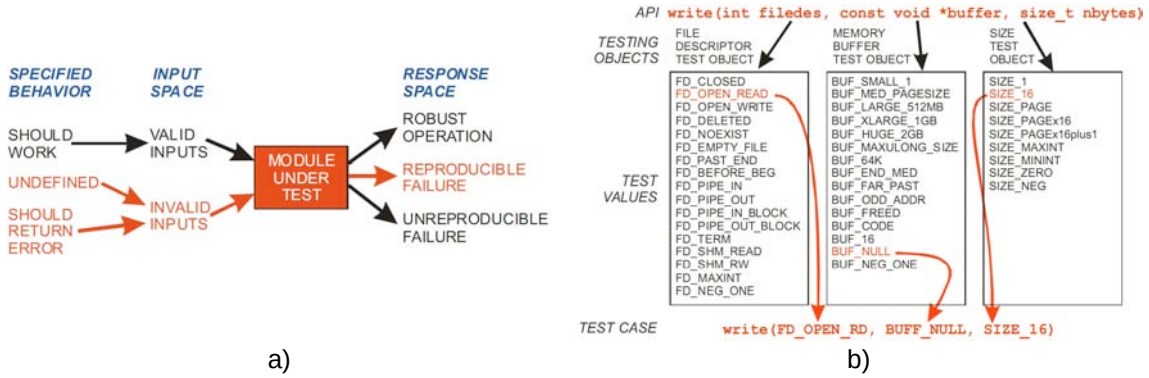


Figure 2.4: *Ballista* approach and test case generation (from [KD00])

To characterize the robustness of the target system, *Koopman et al.* defined the observation scale "*CRASH*" based on the severity of the observed failure:

- **C** - *Catastrophic* : the entire OS crashes or reboots
- **R** - *Restart* : a system call is hung and must be terminated by force
- **A** - *Abort* : interruption or abnormal termination of a task
- **S** - *Silent* : no error indication is returned
- **H** - *Hindering* : an incorrect error indication is returned

The *Ballista* approach has been applied in several robustness evaluation studies: robustness testing of 233 system calls on 15 operating systems supporting the POSIX (Portable Operating Systems Interface) interface [KSD+97, KD99, KD00], robustness testing of the Windows family (2000, NT, 95, 98, 98SE and CE), robustness comparison between Linux and Windows [SKD00], and robustness characterization of several CORBA ORB implementations [PKS+01].

As an example of the type of results obtained, Figure 2.5 presents the results of using Ballista to test 15 OS implementations and 233 selected POSIX functions [KD00]. Overall "failure rates" (proportions of tests leading to failure) considering both Abort and Restart failures range from a low of 9.99 % (AIX) to a high of 22.69 % (QNX 4.24). Abort failures were common because it is easy to provoke a core dump from an instruction within a function or system call. In Figure 2.5, we can observe that robustness was not necessarily improved even after a version upgrade of system. Failure rates were reduced in some cases (from Irix 5.3 to Irix 6.2, from OSF 3.2 to OSF 4.0, from SunOS 4 to SunOS 5). However, from HP-UX 9 to HP-UX 10, and from QNX 4.22 to QNX 4.24, the failure rates actually increased.

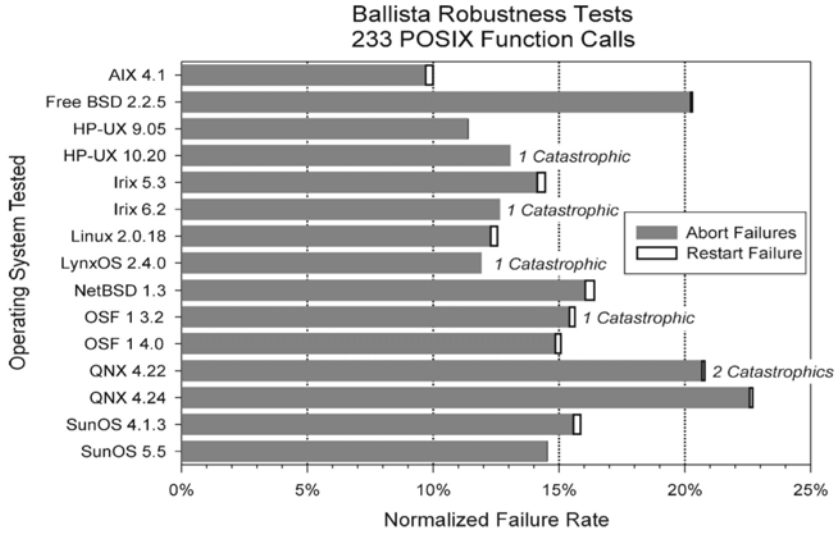


Figure 2.5: Normalized failure rates for POSIX calls on 15 OS tested by *Ballista* (from [KD00])

2.2.4 MAFALDA

MAFALDA (Microkernel Assessment by Fault injection Analysis and Design Aid), developed at LAAS-CNRS, is a software fault injection tool that is aimed at providing quantitative information on the behaviour of a microkemel in the presence of faults, and assisting its integration into a safety-critical system. The goal is to improve the dependability properties of the microkemel by i) characterizing its failure modes, and ii) hardening it with error confinement wrappers [RSFA99, AFRS02, FRAS00, SRFA99]. The analysis of the microkernel behaviour in the presence of faults is intended to evaluate the dependability properties of the microkemel in terms of i) its interface robustness and ii) its internal error detection mechanisms. Therefore, two fault models are defined in *MAFALDA*:

- Faults on parameters of services accessible via the microkernel API that are invoked during a kernel call, simulating the effect of external faults.

- Faults on the code segments and data of the microkernel, simulating the effect of physical faults.

Figure 2.6 illustrates the general architecture of *MAFALDA*. The *Interception* and *Injection* software modules control the injection on kernel call parameters and on memory locations storing the code and data of the functional components of the target microkernel. The relevant events (kernel calls though the kernel API (application programming interface) and internal service calls among the kernel functional components) are trapped by the *Interception* module, corrupted by the *Injection* module and then forwarded for execution in the microkernel. Event corruption is carried out by random bitflipping of event parameters. The parameters defining the fault injection campaigns are stored in two input files on the host machine: the Campaign Descriptor and the Workload Descriptor. The experiment results are stored in the Log files and Campaign Results files, which can be used later for more elaborate custom processing and analysis.

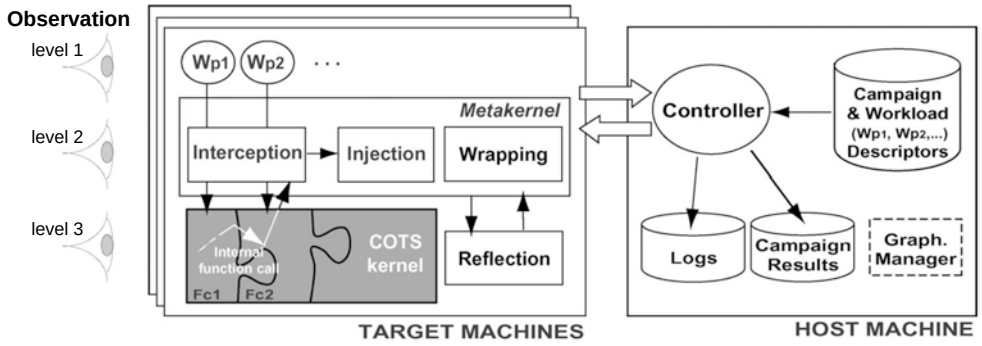


Figure 2.6: *MAFALDA* architecture (from [AFRS02])

The results provided by MAFALDA are a synthesis of the events observed during the fault injection campaign, including: i) error distribution; ii) error propagation; iii) error notification (exceptions, error status signals); and iv) the corresponding latencies (e.g., for the exceptions raised). More precisely, the target machine is observed at three distinct levels in order to group the different failure modes of the candidate microkernel:

- *Application level*: observation of the symptoms of a propagated error: an erroneous behaviour (*Application failure*) or a service stop without erroneous output (*Application hang*)
- *Interface level*: observation of the notification of a detected error: an error status is returned or an exception is raised.
- *Kernel level*: this level concerns microkernel crashes, including diagnosis if the kernel has hung or if control has been passed to the microkernel's debugger, which means that the microkernel has detected a critical error and deliberately put the system into a safe state by freezing the microkernel.

MAFALDA has been used to test an early version of the Chorus ClassiX r3 microkernel and LynxOS. These microkernels are composed of various components following the categories given hereafter:

- *Core (CORE)*: Management of threads and actors (Chorus multithread processes) and hardware related basic functions (e.g., interrupts, timers, traps, MMU, etc.), some of which do not belong to the microkernel API.
- *Synchronization (SYN)*: Management of semaphores, mutexes, event flags, etc.
- *Memory Management (MEM)*: Management of memory segments including functions for flat memory, protected address spaces management policies, address space sharing, etc.
- *Scheduling (SCH)*: Various schedulers, including priority-based FIFO preemptive scheduling or with fixed quanta, Unix Time Sharing, etc. SCH handles running threads and actors and is thus indirectly activated by the creation of threads and actors.
- *Communication (COM)*: Local and remote communication.

As an example of the results obtained, Figure 2.7 shows different behaviours observed when injecting faults at the Chorus and LynxOS kernel interfaces. The observed proportion of application failures is extremely high in Chorus when faults are injected into the SYN component (see Figure 2.7.a). This result reveals that Chorus has no input parameter checks implemented in the basic synchronization facilities. In fact, this was a design choice made by the Chorus developers in favour of system performance, which leaves error-handling to the system integrator. On the other hand, the large percentage of application hangs observed in Figure 2.7.b reveals some vulnerability of the memory management component (MEM) in LynxOS.

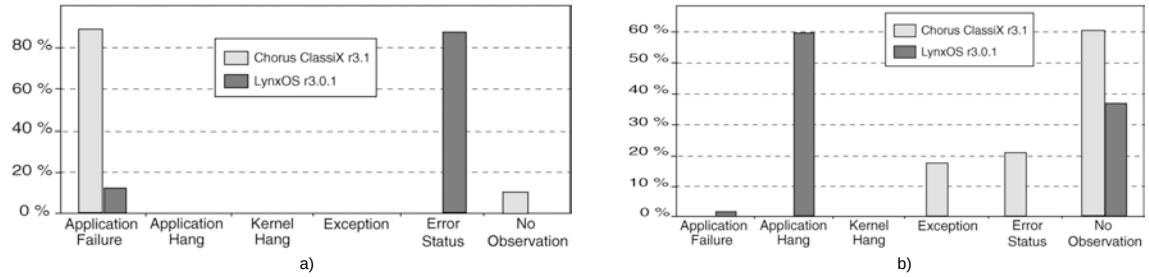


Figure 2.7: Chorus and LynxOS kernel robustness experiments: (a) injection into SYN component call parameters; (b) injection into MEM component call parameters (from [AFRS02])

2.2.5 DBench

Notable work on the evaluation of computer system robustness was carried out in the DBench (Dependability Benchmarking) project (`http://www.laas.fr/DBench`). The goal of dependability benchmarking is to a provide generic and reproducible way of characterizing the behaviour of a computer system in the presence of faults. The objective of the DBench project was to provide a framework and guidelines for defining dependability benchmarks for computer systems, and a means for implementing them. DBench developed a framework for defining dependability benchmarks for computer systems, with emphasis on COTS components and systems, via experimentation and modeling. DBench identified the main dimensions that are decisive for defining dependability benchmarks and the way experimentation can be carried out in practice: i) the target system and the benchmarking context,

ii) the benchmark measures and measurements to be performed on the system for obtaining them, iii) the benchmark execution profile to be used, and iv) the set-up and related implementation issues required for running a benchmark.

As a part of the DBench project, *Kanoun et al.* [KCK+05] define a dependability benchmark for general-purpose operating systems and apply it to six versions of Windows OS and four versions of Linux OS. This study aimed to measure the robustness of Windows and Linux in the presence of faults by injecting erroneous inputs to the OS via its API. The benchmark execution profile consisted of a PostMark workload together with fault injection into parameters of system calls. PostMark creates a large pool of continually changing files and measures the transaction rates (a workload approximating a large Internet electronic mail server). Fault injection was carried out using the parameter corruption technique relying on analysis of system call parameters to define substitution values to be applied to these parameters. The value of data can be replaced by an out-of-range value or by an incorrect but valid (not out-of-range) value. During runtime, the workload system calls are intercepted, corrupted and re-inserted. Figure 2.8 reviews the substitution values associated with the basic data type classes.

Data type class	Substitution values					
Pvoid	NULL	0xFFFFFFFF	1	0xFFFF	-1	Random
Integer	0	1	MAX INT	MIN INT	0.5	
Unsigned integer	0	1	0xFFFFFFFF	-1	0.5	
Boolean	0	0xFF (Max)	1	-1	0.5	
String	Empty	Large (> 200)	Far (+ 1000)			

Figure 2.8: Parameter substitution values (from [KCK+05])

Test results are classified and measured according to the system state observation:

- **SEr**: An error code is returned.
- **SXp**: An exception is raised.
- **SPc**: The system is in panic state. The OS is not servicing the application.
- **SHg**: Hang state. It is necessary to hard reboot the OS.
- **SNS**: No-signaling state.

The robustness measure is presented in Figure 2.9. Concerning the robustness measure, all OS's of the same family are roughly equivalent. None of the catastrophic outcomes (*Panic* or *Hang* OS states) were observed for any Windows and Linux OS. Linux OS notified more error codes than Windows, while more exceptions were raised with Windows than Linux. A more detailled analysis [KCK+05] also shows that the Windows reaction time to system calls activated by the workload is shorter than the Linux reaction time.

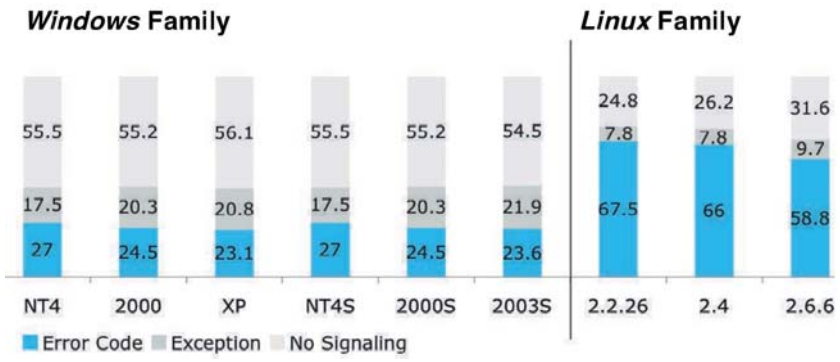


Figure 2.9: Robustness measure in DBench (from [KCK+05])

2.3 Robustness testing methods based on behaviour models

When a behaviour model of the SUT exists that specifies robust behaviour in the presence of invalid inputs, robustness testing can be expressed in the same terms as conformance testing. Several works have been devoted to transforming a nominal behaviour specification into a robust behaviour specification, and then deriving test cases to verify the conformance of the SUT with this robust behaviour specification, in which case the system is declared to be robust [TRF05, SKRC07, FMP05]. In this section, we present two notable methods based on behaviour models.

2.3.1 Rollet-Fouchal approach

In [Rol03, FRT05, TRF05], the authors propose a method for robustness testing of embedded systems with time constraints. They use the timed input-output automata (TIOA) formalism.

Timed Input-Output Automata A timed input-output automaton A is defined as a tuple $(\Sigma_A, S_A, s_A^0, C_A, T_A)$, where Σ_A is a finite set of actions, S_A is a finite set of states, s_A^0 is the initial state, C_A is a set of clocks, and T_A is a set of transitions.

Σ_A is partitioned into 2 sets: A_I is the set of input actions (written $?i$), and A_O is the set of output actions (written $!o$).

T_A is a transition set having the form $Tr_1.Tr_2...Tr_n$. $Tr_i =< s; a; d; EC; C_s >$, where s and d are the start and destination states. 'a' is the action of the transition. EC is an enabling condition to trigger the transition, equal to the result of the formula $x \sim y$ where $\sim \in \{<, >, \leq, \geq, =\}$. C_s is a set of clocks to be reset at the execution of a transition. After the execution of Tr_i, all clocks in C_s are reset.

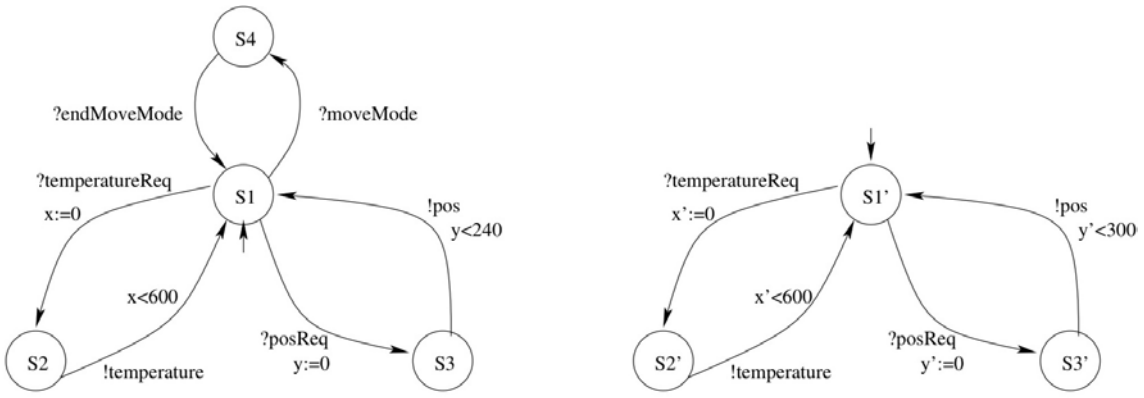
Using this formalism, a system is described by two specifications:

- a *nominal* specification $NS = (\Sigma, S, s_0, C, T)$ which describes the behaviour of the system in the absence of errors,
- a so-called *degraded* specification $DS = (\Sigma_{degr}, S_{degr}, s_{0degr}, C_{degr}, T_{degr})$ which describes admissible behaviour of the system in the presence of errors, i.e., the vital functionalities that must be ensured when the system is in an abnormal environment

For example, we could require in the degraded specification that a robot has to send its position at least every 10 seconds whereas it sends its position every 1 second in the nominal specification. Figure 2.10 shows an example of a nominal specification M with initial state s_1 and its degraded version M'. An arrow between two states represents a transition and is labelled by (a, EC, C_s).

Based on these two specifications, the authors' robustness testing approach (summarized in Figure 2.11) consists in:

- generating a test sequence set from the nominal specification based on any classic conformance testing method;
- seeding generated test sequences with errors (called "hazards" on Figure 2.11);
- exercising the SUT with the mutated test sequences;
- verifying the conformance of the obtained execution trace against the degraded specification DS, in which case the SUT is declared to be robust.



(a) Nominal spec. (b) Degraded spec.

Figure 2.10: Example of Nominal specification (a) and Degraded specification (b) as TIOA models

Test sequences are generated from the nominal specification using the test purpose technique[1] [FPS01] and then mutated by insertion of anomalous inputs. The mutation is done manually, based on the experience of the tester, by modifying the testing sequence. Several mutations are considered: replacing an input action by another, changing the instant of an input action occurrence, exchanging two input actions, adding an unexpected transition, removing a transition. After the execution of each test sequence, the results are recorded as timed execution traces, which are later tested against the degraded specification. The verdict is simple: if all execution traces are accepted, then the component implemented is considered as "robust enough".

[1] A test purpose is an abstract specification of some property, e.g., whenever event a occurs, it must be followed by event b within τ seconds. The derived test sequences must satisfy the given test purpose.

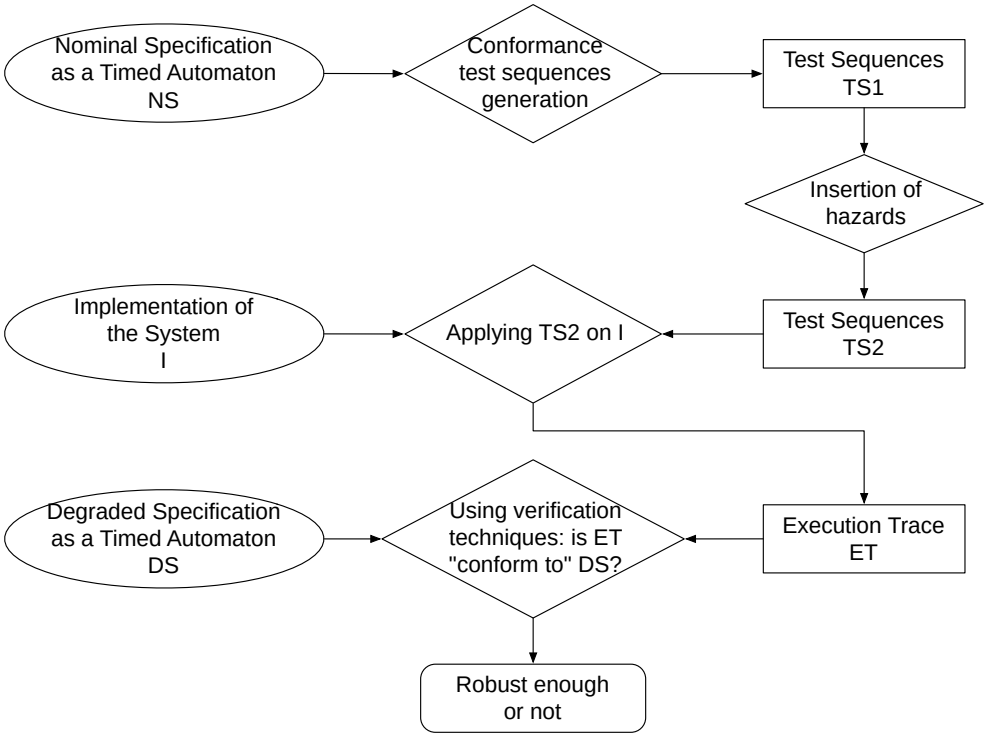


Figure 2.11: Rollet-Fouchal robustness testing approach (from [FRT05])

2.3.2 Saad-Khorchef approach

Another behaviour-model based approach for robustness testing in communication protocol is proposed by *Saad-Khorchef et al.* [SKRC07, Saa06]. This is one of a few works that take into account both value domain and time domain invalidity in robustness testing. This approach also uses two specifications of the system: a nominal specification, which describes the normal behaviour of the system; and a so-called "augmented specification", which describes the acceptable behaviours of the system in the presence of invalid inputs. Testers generate test cases from the augmented specification and then use them to exercise the SUT. The test results are then evaluated with respect to this specification to give out the robustness verdict.

The nominal specification, denoted S, is described by an IOLTS model (Input Output Labelled Transition System) that allows the system to be defined as a set of states, an alphabet of transition actions and a set of transition relations. Note that the nominal specification describes the expected behaviour in nominal conditions, i.e., in the absence of invalid or inopportune inputs.

IOLTS [Tre96]: An IOLTS is a quadruplet $S = (Q, \Sigma, \rightarrow, q_0)$ such that: Q is a nonempty finite set of states, Σ is the alphabet of actions, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, and q_0 is the initial state. The alphabet Σ is partitioned into an input alphabat Σ_I and an output alphabet Σ_O.

The framework consists of two phases (Figure 2.12): i) Construction of an *augmented specification* by integrating "hazards" into the nominal specification; ii) Generation of test cases from the augmented specification.

The term "hazard" denotes any event not expected in the nominal specification of the

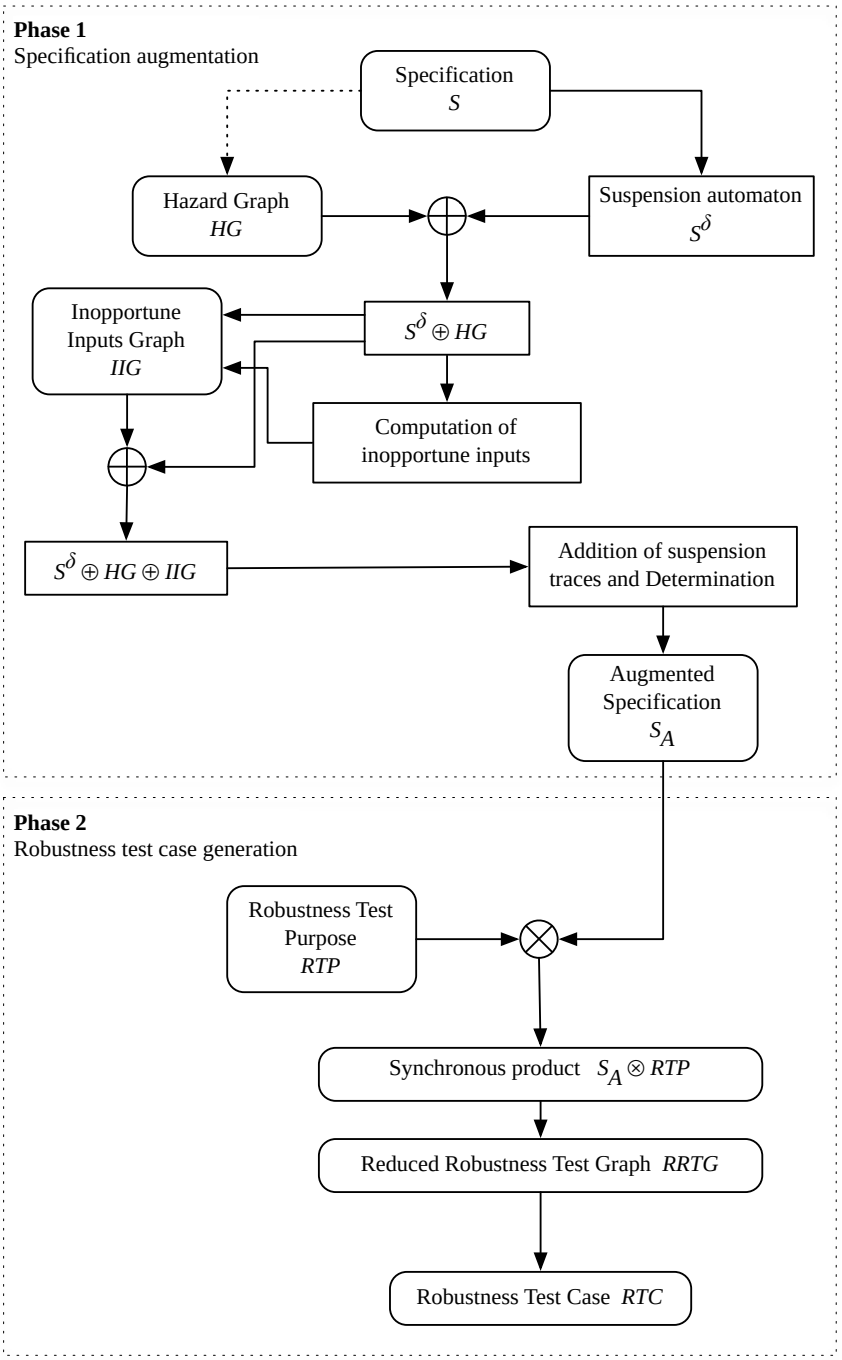


Figure 2.12: Saad-Khorchef approach (adapted from [Saa06]

system. Two types of input hazard are considered:

- Invalid inputs: Any unspecified input, i.e., the set $\{a' | a' \notin \Sigma_I\}$
- Inopportune inputs: Actions existing in the alphabet of the specification, but not expected in the given state, i.e., for state q, the set $\{\Sigma_I \backslash In(q)\}$, where $In(q) = \{a \in \Sigma_I$ such that q has an output transition labelled by $a\}$.

The aim of creating the *augmented specification* in the first phase is to formally describe the acceptable behaviours in presence of hazards, which will act as a reference for the generation of robustness test cases in the second phase.

The main steps of the framework are the following:

- **Phase 1** Creation of the augmented specification
  - Analysis of blockages specified in S and transformation of S into a *Suspension Automaton* S^δ that explicitly identifies timeouts on states in which a conforming implementation may block (i.e., blockages that exist in the specification S).
  - Integration of a *Hazard Graph* (HG), which represents invalid inputs and specifies behaviour with respect to these inputs. HG is an extension to the specification; it is supplied by the system designer when the robustness tester asks "what happens in this state when this invalid input is received?"
  - Integration of an *Inopportune Input Graph* (IIG), which represents inopportune inputs and specifies behaviour with respect to these inputs. IIG is a a second extension to the specification; it is supplied by the system designer when the robustness tester asks "what happens in this state when this inopportune input is received?"
  - Analysis of new blockages specified in the composite specification $S^\delta \oplus HG \oplus IIG$, and corresponding modification of this specification (additional timeouts), resulting in the desired augmented specification S_A.
- **Phase 2**: Generation of robustness test cases
  - For each robustness behaviour to be tested, a *Robustness Test Purpose* (RTP) is modelled using the IOLTS formalism. An RTP is an abstract specification of the robustness behaviour, e.g., "if this invalid input is presented to the system then this output should be produced".
  - The augmented specification S_A and the given RTP are combined by taking their synchronous product $S_A \bigotimes RTP$. The result is an IOLTS that models all behaviours of S_A that fulfill the test purpose modelled by RTP.
  - The synchronous product $S_A \bigotimes RTP$ is then inverted (outputs replaced by inputs and vice versa) and simplified to obtain a *Reduced Robustness Test Graph* $RRTG$, which describes all the test sequences, relevant to the test purpose RTP, that a tester can apply to an implementation to check that it conforms to the augmented specification S_A.
  - Elementary *Robustness Test Cases* (RTC) are then generated randomly from $RRTG$ by an algorithm that only retains test sequences that contain hazards, i.e., that are not contained within the nominal specification.

2.4 Hybrid robustness testing

Cavalli et al. proposed in [CMM08] a robustness testing approach that combines aspects of both fault injection and *passive testing* [LCH+02], a process of detecting erroneous behaviour of a system under test by passively observing its input/output behaviour without interrupting its normal operation.

This approach aims to test the robustness of a communication protocol implementation in the face of faults affecting the communication channel. The faults injected are based on a model of communication failures, which are grouped in three classes: omission, arbitrary and (late) timing failures. *Omission failures* can be emulated by intercepting messages sent and received by a host. *Arbitrary failures* can be emulated by corrupting messages received by the SUT. *Timing failures* can be emulated by delaying message delivery longer than specified.

Robust behaviour is expressed as a set of *invariant* properties, which specify the allowable input and output sequences that a system can produce. The proposed approach consists of the following main steps: (i) build a formal model of the system behaviour; (ii) define the invariant properties and check them against the formal model; (iii) define the fault model and the faults to inject; (iv) instrument the SUT for fault injection and monitoring purposes; (v) execute the tests by activating the SUT while injecting the faults and monitoring the SUT interactions; (vi) analyze the results based on the defined invariants and produce a robustness verdict.

The test architecture is presented in Figure 2.13. The SUT (a protocol entity, labelled here IUT, for *Implementation Under Test*) is instrumented to provide points of observation (PO) at its upper and lower interfaces, in order to log input and output events and obtain an execution trace. The *TESTINV* modules process the execution traces to transform them into a suitable format for the analysis, then parse them according to the properties expressed as invariants and emit a verdict *Pass, Fail*, or *Inconclusive*. *FIRMAMENT* is an IP-level fault injector.

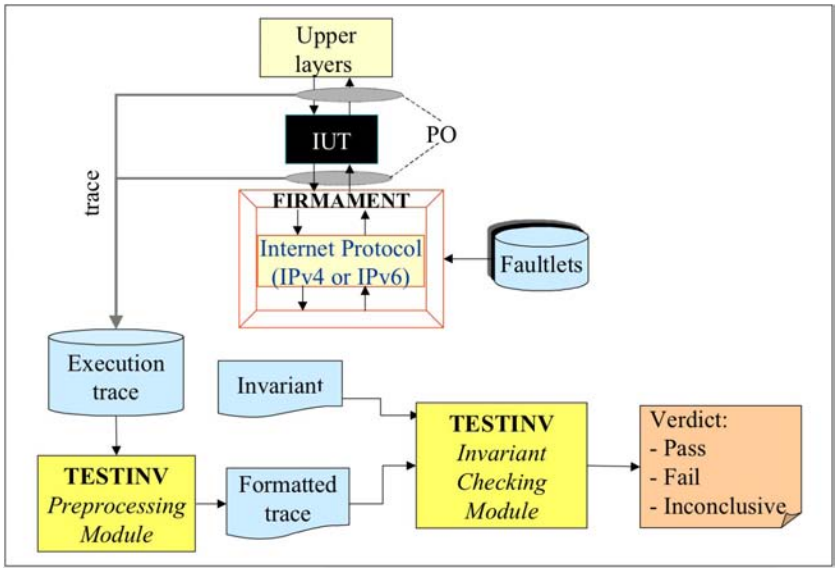


Figure 2.13: Test architecture proposed by *Cavalli et al.*

The formal model of the system behaviour, in the form of a finite state machine, is extracted from the system specification.

Finite State Machine A Finite State Machine (FSM), M, is a tuple $M = (E, \mathcal{I}, \mathcal{O}, T, s_0)$, where $E, \mathcal{I}, \mathcal{O}, T$ are finite sets of states, input symbols, output symbols and transitions, respectively, and s_0 is the initial state. Each transition $t \in T$ is a tuple $t = (s, s', i, o)$, where $s, s' \in E$ are the current and next states of the transition, respectively, $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output symbols.

Trace A trace, tr, is a sequence of pairs, $tr = i_1/o_1, ..., i_n/o_n$, with $n \geq 1$, such that $\forall k, 1 \leq k \leq n : i_k \in \mathcal{I}$ and $o_k \in \mathcal{O}$

The invariant properties are presented as a sequence of input/output pairs, defined using Extended Backus Naur form (EBNF), a family of metasyntax notations used for expressing context-free grammars. Below is an example of an invariant, which has the meaning: if the output event *page_ sent* is observed in the trace, then a page must have been requested before and the server has acknowledged the reception of the request.

$$O_1 = request_page/req_ack, *, ?/\overline{\{page_sent\}}$$

The "?" replaces any symbol. The "$*$" can replace any sequence of symbols except the input symbols which are right next to it.

A very preliminary experimental campaign was carried out to test the robustness of a WAP (Wireless Application Protocol) implementation (Kannel, version 1.4.1). Three functional invariants were defined that expressed properties required from a protocol implementation in both the absence and the presence of communication faults. In one experiment, one of these invariants was not satisfied. Since some experiments resulted in the implementation blocking, the authors added two non-functional inputs ($HANG$ and $PANIC$) and corresponding invariants to take into account such non-robust behaviours.

2.5 Conclusion

We presented in this chapter a state of the art in robustness testing. We identified two robustness testing types, *input robustness testing* and *load robustness testing*, respectively based on two important aspects of the IEEE robustness definition, *invalid inputs* and *stressful environmental conditions*. We also presented various notable works on input robustness testing, the key features of which are summarized in Table 2.1.

Column two of the table shows the key features of the MESSALINE hardware-fault injection tool, for which the FARM framework was first defined (cf. Section 2.2.1). This is not a robustness testing technique, but is given as a baseline for comparison purposes. The (input) robustness testing techniques are grouped into three categories, according to the system model on which they are based: input-domain, behaviour, or hybrid. For each technique, we have identified the key features according to the FARM framework.

- The input-domain model-based approaches focus mainly on interface faults with value domain invalidity. They aim to verify the functional aspects of the SUT by attacking extensively its interface, regardless of the design of the system's fault-tolerance or protection mechanisms. These techniques are primarily aimed at evaluating the achieved robustness (fault forecasting), but can also be viewed as random testing techniques that identify robustness deficiencies as a sort of serendipitous side-effect.
- Behaviour-model based approaches focus more on verification of the system's fault-tolerance or protection mechanisms, i.e., their prime purpose is debugging. The testing

campaign is thus more directed thanks to the test case generation method based on a formal system behaviour specification. However, when testing off-the-shelf systems, such a formal specification is not always available.

- The hybrid approach is interesting in that it decouples robustness adjudication from test case generation as in the input-domain model-based approaches. The passive testing notion of a "robustness oracle" defined as a set of invariant properties on observable traces allows a more detailed specification and analysis of robust behaviours than the coarse-grain scales used, for example, in the BALLISTA and DBench work.

In this thesis, we propose a hybrid robustness testing approach that, like the work described in Section 2.4, is a combination of random fault-injection and passive testing. Our focus is on injecting system inputs that are invalid in the time domain in order to assess the effectiveness of a SUT's built-in capabilities to defend itself against asynchronous inputs that cannot be processed in its current state. We follow a passive testing approach in that we define a property-based robustness oracle that allows the analysis of system traces obtained in the presence of invalidly-timed inputs.

The method is described in detail in the following chapters.

Table 2.1: Summary of surveyed techniques

	Comparison baseline	Input robustness testing techniques						
	MESSALINE	FUZZ	BALLISTA	MAFALDA	DBENCH	Rollet - Fouchal	Khorchef et al.	Cavalli et al.
System model	Input model					Behaviour model		Hybrid model
	Hardware architecture + IC pin-out specification (black box)	Message / event syntax specification (black box)	API specification (black box)	API specification (black box)	API specification (black box)	TIOA model (grey box)	IOLTS model (grey box)	FSM model + trace invariants (grey box)
Typical target	Self-checking or fault tolerant hardware	OS, COTS software	OS, COTS software	Microkernel-based COTS software	Microkernel-based COTS software	Real-time component based system	Communication protocol implementation	Communication protocol implementation
Fault Forecasting (Evaluation)	×	×	×	×	×			possible
Fault Removal (Debugging)	Spin-off benefit					×	×	×
F: faults *(faultload)*	Pin level logic faults	Generation of system messages / events	Test case generation with API parameter mutation by substitution	API call interception & Parameters mutation by bit-flipping	API call interception & Parameter mutation by substitution	Mutation of conformance test sequences generated from nominal model	Test sequences generated from robust behaviour (augmented) model (includes invalid and inopportune inputs)	IP message corruption
A: activity *(workload)*	Any application	N/A	N/A	Application processes exercising kernel modules	PostMark file exchange Benchmark	N/A	N/A	Mobile Web Browser simulator
R: read-outs	Hardware / software error detection signals, latency	System failures (3-category outcome classification)	System failures & Interface errors (5-category outcome classification)	System failures & Interface errors (6-category outcome classification)	System failures & Interface errors (5-category outcome classification)	Verdict wrt robustness requirement behaviour model (degraded model)	Verdict wrt robust behaviour (augmented) model	Verdicts wrt invariants of behaviour model
M: measures	Descriptive or inferential statistics	Descriptive statistics	Descriptive statistics	Descriptive statistics	Descriptive statistics	N/A	N/A	N/A

Chapter 3

Functional Layer Robustness Testing Framework

We present here our framework for assessing the robustness of the functional layer of a hierarchically-structured autonomous system. As presented in Section 1.1.2, the functional layer includes all the basic built-in robot action and perception capacities. We develop our approach in the context of the LAAS architecture, which is a typical hierarchical architecture for autonomous systems. However, our framework is general enough to be applicable to other systems with a hierarchical architecture.

Our framework is intended to allow *comparison* of different implementations of the same functionality, with the same Application Programming Interface (API), in a similar spirit to the Ballista, Mafalda and DBench work on benchmarking operating system robustness (cf. Sections 2.2.3 to 2.2.5). As for those approaches, we seek to randomly generate a large number of test cases so that, in addition to detecting robustness deficiencies that should be corrected, we can generate statistics to evaluate measures of the robustness of different implementations.

To compare different implementations objectively, the same test cases must be applicable to every implementation, irrespectively of its internal structure. The generation of test cases thus cannot rely on a detailed behaviour model of the implementation of the sort considered, for example, in the formal robustness testing approaches described in the previous chapter (cf. Section 2.3). Instead, a *black-box* testing approach is necessary. in which only an interface specification is provided to the tester.

Nevertheless, a robustness verdict must be delivered for every test case.

One possibility that was envisaged [CAI+09, CAK+09] was to define a classification of robustness test outcomes, like in the surveyed work on operating system robustness benchmarks. However, the functional layer of an autonomous robot is very different to an operating system, so not all concepts are directly transferable. One transferable concept is the coarse-grain notion of system blockage (like the crashes and hangs of the Ballista *CRASH* scale and the *SPc* and *SHg* categories of the DBench test result classification): a functional layer (or, for that matter, any system) that blocks as a result of an invalid input is evidently non-robust. Conversely, anomaly reports such as the error signals and exceptions considered in OS robustness testing, need a finer analysis.

Indeed, the functional layer of an autonomous robot interacts in real-time with its physical environment, so anomaly reports sent to upper layers (such as the non-nominal final replies

of $G^{en}oM$ modules, cf. Section 1.4.1) may pertain not only to inappropriate behaviour on the behalf of functional layer clients but also to unsuccessful interactions with the physical environment. Thus, it is not possible to consider the number of anomaly reports as a measure of non-robustness of a functional layer with respect to its client layer. A finer-grain interpretation is necessary.

The approach followed here is to assess robustness at the level of the *safety properties* that the functional layer should respect. A system under test (SUT) that respects a specified set of safety properties in the presence of invalid inputs will be considered as robust. Since we adopt a black-box testing approach, the judgement as to whether or not the SUT respects the safety properties can only be based on the observations of requests and replies crossing the API. In this respect, the framework defined in this chapter resembles the hybrid robustness testing approach developed by *Cavalli et al.* [CMM08] for communication protocols. Here, however, the robustness test cases will correspond to inappropriate behaviour on the behalf of functional layer clients rather than communication faults considered by *Cavalli et al.*.

The chapter is structured as follows. In Section 3.1, we define our system model. In Section 3.2, we describe a set of safety property classes that are implementable by a property-enforcing layer placed above the functional layer, and study possible property enforcement policies. For each property class, Section 3.3 analyzes the possible system behaviours that can be deduced from observation the trace of requests and replies to and from the system, and defines a corresponding robustness test oracle.

3.1 System model

We assume a layered architecture with modules at the lower functional layer based on code produced by the $G^{en}oM$ tool (cf. Section 1.4.1) and module clients at upper layers abstracted into an "application" layer (Figure 3.1). Clients can issue requests to functional layer modules to initialize them, to control them (e.g., make changes to their internal data structures), and to start and stop *activities* (see Figure 3.2).

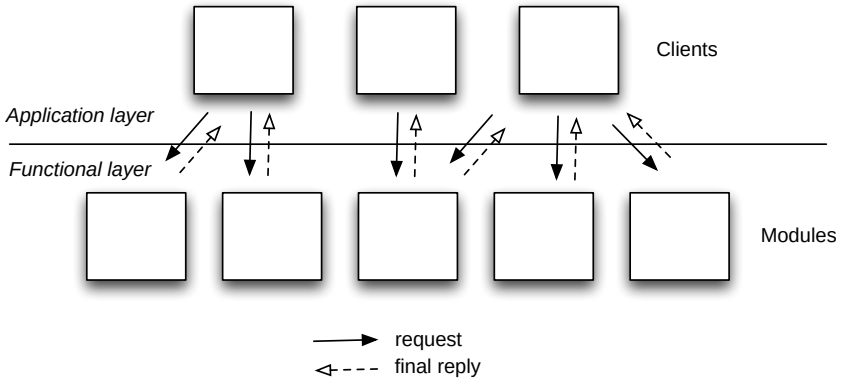


Figure 3.1: System architecture

3.1.1 Interface protocol

Clients issue non-blocking execution *requests*. For each request, the called module will return a *final reply* that indicates the outcome of the activity whose execution was requested. Three

cases can be distinguished (Figure 3.2):

1. The request is accepted by the called module, and the final reply indicates the outcome of the requested activity, as specified by the module code. Nominally, the final reply returns *ok*, but the programmer can specify other, application-specific reasons for termination.

2. The request is rejected by the called module, and the final reply indicates the cause of the rejection (e.g., incorrect call parameters)

3. The request is accepted by the called module, but a client (the same one or a different one) issues another execution request to the same module. The module code may specify in this case that the earlier activity be interrupted[1] in order to start a new activity. The final reply to the initial request indicates the reason for the interruption.

Modules may also return *intermediate replies* containing an activity identifier to keep the client up-to-date with the state of the activity (e.g., the activity is accepted but not yet executed, the activity has been executed for 10 seconds, etc.). However, in the current $G^{en}oM$ implementation, intermediate replies are a feature offered to the module programmer but are by no means obligatory. Intermediate replies are thus not sent systematically, so although an intermediate reply indicates that the request has been taken into account, the absence of an intermediate reply does not imply the contrary. Intermediate replies are therefore not very useful from a testing viewpoint.

3.1.2 Notation

We define the following notation for requests and responses to the functional layer:

- I_{module}: set of initialization requests to *module*;[2]
- C_{module}: set of control requests to *module*;
- E_{module}: set of execution requests to *module*;
- $i(i) \in I$: an initialization request with request identifier i;
- $c(i) \in C$: a control request with request identifier i;
- $e(i) \in E$: an execution request with request identifier i;
- $x(i) \in \{I, C, E\}$: a request, of any basic type[3], with request identifier i.
- F_x: set of final reply values defined for request x
  with $F_x = \{R_x, Z_x, T_x\}$, such that:
  - R_x: set of final reply values indicating request rejection;
  - Z_x: set of final reply values indicating activity interruption;

[1] It would be more appropriate to say "abandoned" since an interrupted activity is not automatically resumed. However, we retain the terminology commonly accepted by roboticians.

[2] When there is no ambiguity, the suffix $_{module}$ can be omitted.

[3] We use the term "*basic* type" (of a request) to denote the classification of the request as an initialization, control or execution request. We reserve the term "type" (of a request) to denote an abstract categorization of requests when defining properties.

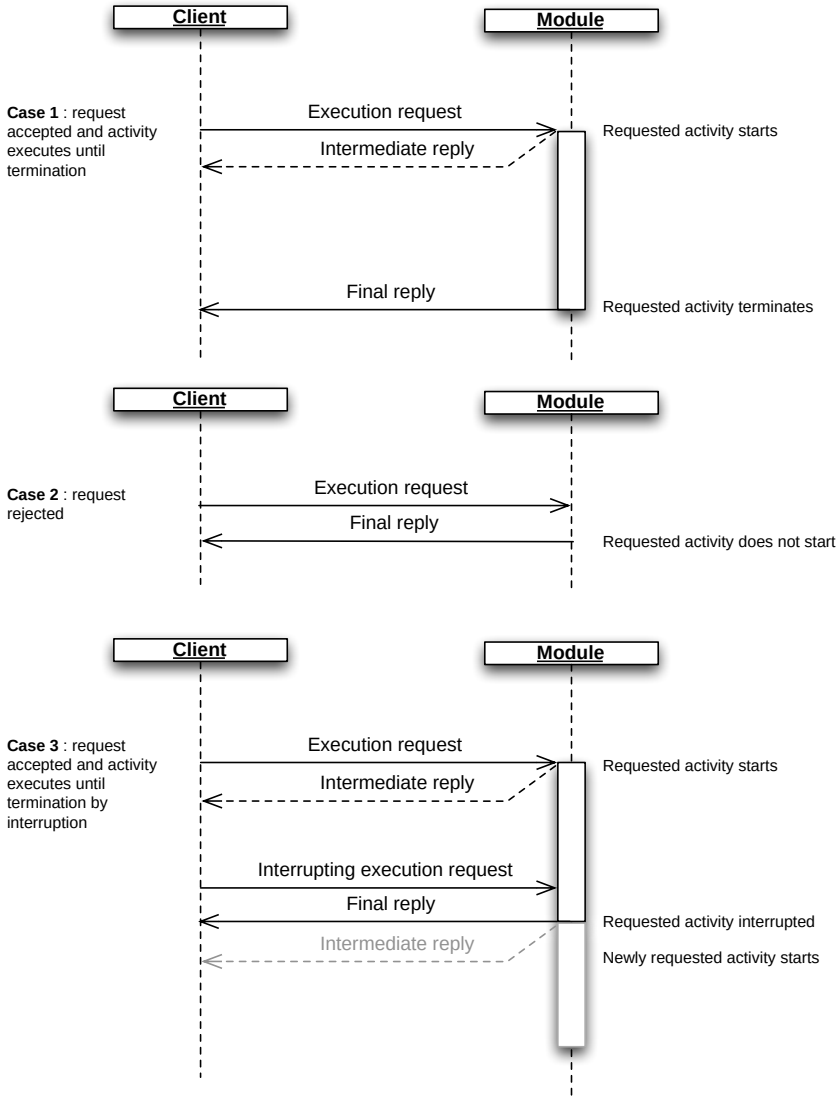


Figure 3.2: Execution request protocol

  - T_x: set of final reply values indicating activity termination other than by interruption (in particular, T_x contains the final reply value *ok*, which indicates correct termination of the activity);

- $f_x(i) \in F_x$: the final reply to $x(i)$;
- $r_P \in R_x$: value of $f_x(i)$ signaling rejection of request $x(i)$ to enforce property P;
- $z_P \in Z_x$: value of $f_x(i)$ signaling interruption[4] of request $x(i)$ to enforce property P.

3.2 Property types and enforcement policies

We define robustness of the functional layer in terms of a set of safety *properties* that the functional layer should ensure in the face of asynchronous requests issued by its clients. To protect itself against requests issued at the "wrong" moment, the functional layer can either reject them, queue them until some better time, or change its internal state (e.g., by interrupting one of its current activities) in order to serve them.

We identify the following types of properties that can be enforced in such a way by a property-enforcing layer placed between the application layer and the functional layer.

Precondition property $PC[x, C_{PRE}]$: an activity of (abstract) type x may only be started if a specified precondition C_{PRE} is true at the instant that x is requested.[5]

Property $PC[x, C_{PRE}]$ can be enforced:

- by rejecting or queuing requests for x while C_{PRE} is false, or
- by forcing C_{PRE} to be true (if that is possible) in order to serve requests for x.

Here, we only consider enforcement by rejection. We note this $PC[x, C_{PRE}]$ property enforcement policy: $x \leftarrow^R \overline{C_{PRE}}$, i.e., a *false* value of C_{PRE} causes *rejection* of requests for x.

Excluded start property $ES[x, y]$: an activity of type x may only be started if there is no ongoing activity of type y at the instant that x is requested.

Property $ES[x, y]$ can be enforced by rejecting or queuing requests for an activity of type x while there is an ongoing activity of type y. Here, we only consider enforcement by rejection of requests for x. We note this $ES[x, y]$ property enforcement policy: $x \leftarrow^R y$, i.e., the presence of an ongoing activity y causes *rejection* of requests for x.

Excluded execution property $EE[x, y]$: an activity of type x may only execute in the absence of requests for activities of type y.

Property $EE[x, y]$ can (only) be enforced by interrupting any ongoing activity of type x to serve a request for an activity of type y. We note this $EE[x, y]$ property enforcement policy: $x \leftarrow^I y$, i.e., a request for activity y causes *interruption* of activity x.

Exclusion property $EX[x, y]$: an activity of type x is excluded by an activity of type y.

Property $EX[x, y]$ can be enforced by rejecting or queuing requests for an activity of type x while there is an ongoing activity of type y, *and* interrupting any ongoing

[4] We use "interruption of request $x(i)$" as a shorthand for "interruption of activity corresponding to request x(i)"

[5] In the following, we will often refer simply to "x" rather than saying "activity of type x"

activity of type x in the event of a request for an activity of type y. Here, we only consider enforcement by *rejection* of requests for, or *interruption*, of x. We note this $EX[x, y]$ property enforcement policy: $x \leftleftarrows^{I}_{R} y$.

We note that $EX[x, y] \equiv ES[x, y] \wedge EE[x, y]$, and that $\left(x \leftleftarrows^{I}_{R} y\right) \equiv \left(x \leftarrow^{I} y\right) \wedge \left(x \leftarrow^{R} y\right)$. We also observe that $EX[x, y]$ defines mutual exclusion between activities x and y, giving priority to y.

Mutual exclusion property $MX[x, y]$: activities of types x and y cannot be executed at the same time.

Property $MX[x, y]$ can be enforced by rejecting or queuing requests for an activity of type x (respectively y) while there is an ongoing activity of type y (respectively x), *or* by interrupting execution of an activity of type x (respectively y) in the event of a request for an activity of type y (respectively x).

We consider two enforcement policies for $MX[x, y]$:

- *mutual rejection*, noted $x \,{}^{R}\!\rightleftarrows_{R}\, y$, which favours the currently executing activity (we use the shorthand notation $MX_R[x, y]$ to denote this property/enforcement pair);
- *mutual interruption*, noted $x \,{}_{I}\!\leftrightarrows^{I}\, y$, which favours the latest request (we use the shorthand notation $MX_I[x, y]$ to denote this property/enforcement pair).

3.3 System observation

To assess the robustness of the system under test, we adopt a passive testing approach similar to that employed by *Cavalli et al.* [CMM08] in the context of communication protocols (cf. Section 2.4). Testing is passive in that the system under test is observed and assessed in a way that is totally independent of system activation and test generation, at the level of the trace of requests and replies crossing its API. Given a trace of requests and responses, our aim is to define an oracle that can classify the behaviour of the SUT with respect to a set of safety properties. In this section, we describe our oracle to characterize the behaviour of the SUT for each request concerned by a given property P, where P is any of the properties defined in Section 3.2.

3.3.1 Behaviour categories

For a given property P, we can classify the system behaviour for each relevant request according to the following basic outcomes:

true negative (TN): execution of the request is authorized since it does not endanger the property P; no invocation of property-enforcement (correct behaviour);

true positive (TP): execution of the request is forbidden since it endangers the property P; property-enforcement is invoked (correct behaviour);

false negative (FN): execution of the request is forbidden since it endangers the property P; however, there is no invocation of property-enforcement (incorrect behaviour);

false positive (FP): execution of the request is authorized since it does not endanger the property P; however, property-enforcement is invoked (incorrect behaviour);

In addition, we consider the following specific outcomes (for reasons that will be explained subsequently):

other positive (op): execution of the request is forbidden since it endangers another property $P' \neq P$, leading to invocation of property-enforcement for P' rather than that for P;

not applicable (na): P is not applicable to the considered request;

truncated trace (ω): the end of the trace is reached before any conclusion can be made;

To ensure exhaustivity of test oracles based on this behaviour categorization, an observation error is declared if the observed system behaviour for a given relevant request cannot be classified in any of the previous categories.

3.3.2 Property oracles

We successively examine each of the five property types defined in Section 3.2. For each property, we present an illustration of the chronology of events corresponding to property enforcement, an exhaustive analysis of the possible normal and abnormal behaviours of the system under test, and finally the corresponding property test oracle.

3.3.2.1 Property $PC[x, C_{PRE}]$: precondition

Figure 3.3 illustrates enforcement of the precondition property by rejection $x \leftarrow^R \overline{C_{PRE}}$.

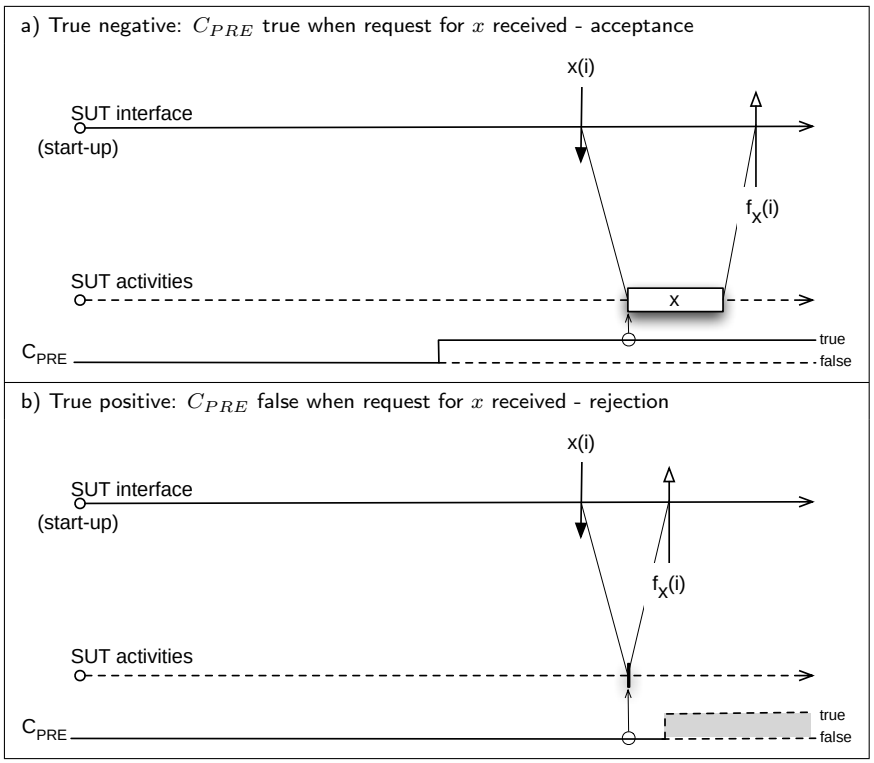


Figure 3.3: Property $PC[x, C_{PRE}]$ enforced by rejection ($x \leftarrow^R \overline{C_{PRE}}$)

To describe the two situations depicted on Figure 3.3, we introduce the following notation:

- $t(event)$ is the time of occurrence of *event* on the internal "SUT activities" time-line;
- $C_{PRE}(x(i))$ is the truth value of the precondition at time $t(x(i))$.

The two situations on Figure 3.3 differ according to whether the pre-condition C_{PRE} is true or false at the instant that a request $x(i)$ is received on the internal time-line. In the first case (Figure 3.3a, $C_{PRE}(x(i)) = true$), the request $x(i)$ is accepted and the corresponding activity is executed for some time until the emission of a final reply $f_x(i)$. In the second case (Figure 3.3b, $C_{PRE}(x(i)) = false$), the request $x(i)$ is rejected and a final reply $f_x(i)$ is returned immediately. Note that when $C_{PRE}(x(i)) = true$, the property $PC[x, C_{PRE}]$ is respected for $x(i)$ without any need for explicit enforcement. Enforcement (in this case, by rejection) is only needed when $C_{PRE}(x(i)) = false$. Formally, we can express this as follows:

$$PC[x, C_{PRE}] \models behaviour_{SUT}\,|_{x(i)} \Leftrightarrow\ C_{PRE}(x(i)) \vee reject_{x(i)}$$

which reads: the behaviour of the SUT with respect to request $x(i)$ satisfies the property $PC[x, C_{PRE}]$ if and only if either $C_{PRE}(x(i))$ is true or the request $x(i)$ is rejected.

Of course, the property $PC[x, C_{PRE}]$ is trivially satisfied (for any $x(i)$) if all requests of type x are systematically rejected. From a robustness evaluation viewpoint, it is therefore interesting to distinguish whether or not $x(i)$ is rejected for the right reason. This can be determined by examining the value of the final reply $f_x(i)$. A request $x(i)$ is judged to be:

- *accepted*, if its final reply $f_x(i)$ indicates either interruption ($f_x(i) \in Z_x$) or termination ($f_x(i) \in T_x$) since an activity can only be interrupted or terminated (be it normally or abnormally) if it has started to execute, i.e., the corresponding request has not been rejected;
- *rejected* by the property enforcement mechanism, if its final reply $f_x(i)$ is equal to r_P, the specific rejection message defined for the considered property (here, we have $r_P = r_{PC[x,C_{PRE}]}$);
- *rejected for some other reason*, if its final reply $f_x(i)$ is in the set $R_x \backslash r_P$ (i.e., an "other positive" outcome, due to the enforcement of a property other than the considered property).

If no final reply is received before the end of the analyzed trace then no conclusion can be drawn as to whether $x(i)$ was accepted or rejected. We refer to this case as a *truncated trace*.

Considering now both the correct behaviours of the SUT (as depicted on Figure 3.3) and possible incorrect behaviours (incorrect acceptance and incorrect rejection), Table 3.1 enumerates all the possible observable behaviours of the SUT for a request $x(i)$ with respect to the precondition property $PC[x, C_{PRE}]$ enforced by rejection $x \leftarrow^R \overline{C_{PRE}}$. Table 3.2 summarizes the corresponding test verdicts, including the "truncated trace" case (no final reply $f_x(i)$ is ever received) for which the test verdict is classified as ω (cf. Section 3.3.1).

Table 3.1: Analysis of possible behaviours for $PC[x, C_{PRE}]$ ($x \leftarrow^R \overline{C_{PRE}}$)

$C_{PRE}(x(i))$	$f_x(i)$	Analysis	Verdict
$true$	$\in \{Z_x, T_x\}$	correct acceptance of x, see Figure 3.3.a	true negative
	r_P	incorrect rejection of x	false positive
	$\in R_x \backslash r_P$	x rejected for some other reason	other positive
	$\varnothing$	no response	truncated trace
$false$	$\in \{Z_x, T_x\}$	incorrect acceptance of x	false negative
	r_P	correct rejection of x, see Figure 3.3.b	true positive
	$\in R_x \backslash r_P$	x rejected for some other reason	other positive
	$\varnothing$	no response	truncated trace

Table 3.2: Test verdicts for $PC[x, C_{PRE}]$ ($x \leftarrow^R \overline{C_{PRE}}$)

	$f_x(i)$			
$C_{PRE}(x(i))$	$\in \{Z_x, T_x\}$	r_P	$\in R_x \backslash r_P$	$\varnothing$
$true$	TN	FP	op	ω
$false$	FN	TP	op	ω

The pre-condition C_{PRE} can be defined in numerous ways, leading to multiple different instantiations of this property type. Here, we specifically consider preconditions defined in terms of sets of activities that must be successfully completed (in any order) before the considered request $x(i)$ is accepted (see Figure 3.4). Let A_{PRE} denote the set of activities corresponding to the precondition C_{PRE}. The condition $C_{PRE}(x(i))$ is then defined as follows:

$$C_{PRE}(x(i)) = \forall a \in A_{PRE}, \exists k, (f_a(k) = ok) \wedge (t(f_a(k)) < t(x(i))) \tag{3.1}$$

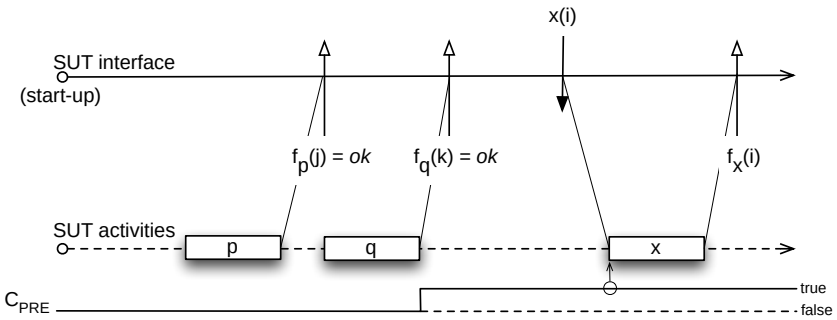


Figure 3.4: True negative for precondition property with respect to completion of all activities in set $A_{PRE} = \{p, q\}$

3.3.2.2 Property $ES[x, y]$: exclusive start

Figure 3.5 illustrates enforcement of the exclusive start property by rejection $x \leftarrow^R y$.

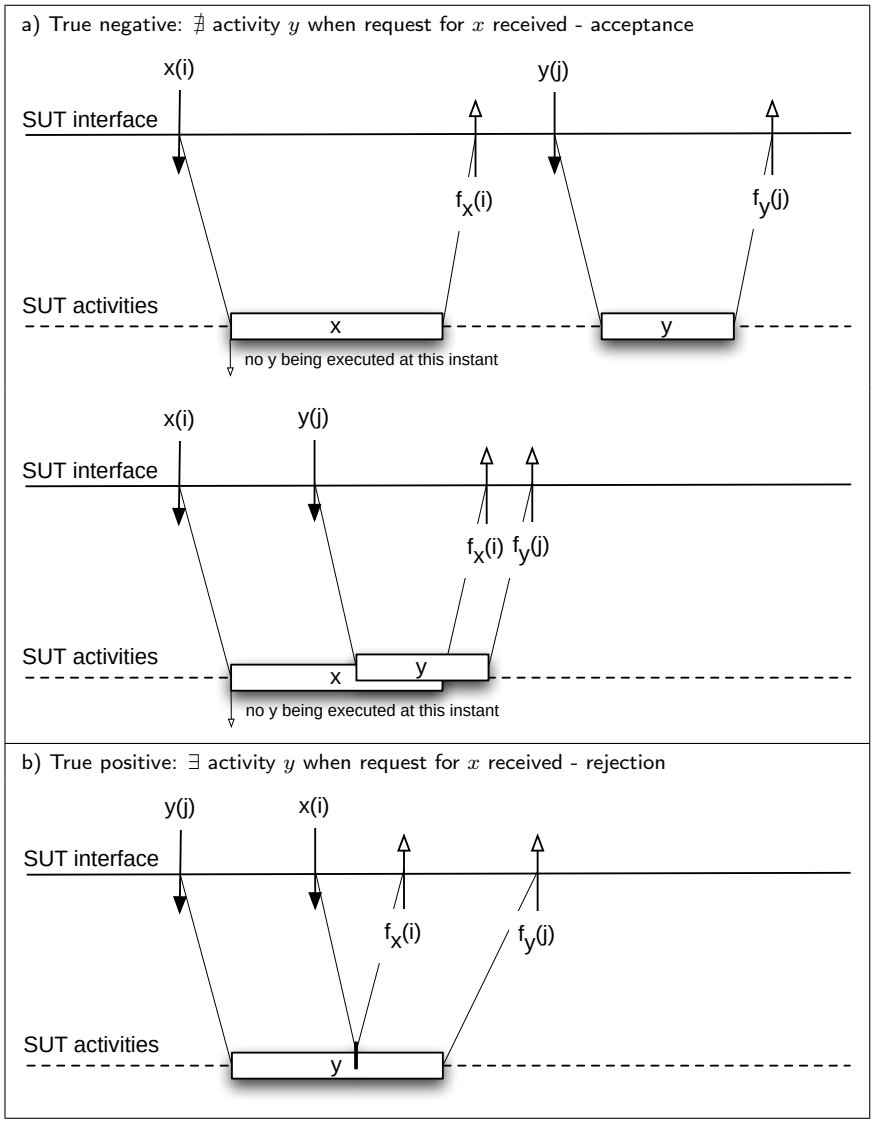


Figure 3.5: Property $ES[x, y]$ enforced by rejection ($x \leftarrow^R y$)

Using the same notation as before, a request $x(i)$ is rejected to enforce the property $ES[x, y]$ if there is an ongoing activity corresponding to a request y, i.e., $\exists y(j), t(x(i)) \in [t(y(j)), t(f_y(j))[$[6] (see Figure 3.5b). The property $ES[x, y]$ can be viewed as the property $PC[x, C_{PRE}]$ with a precondition $C_{ES}(x(i))$ defined as the negation of this condition:

$$C_{ES}(x(i)) = \nexists y(j), t(x(i)) \in [t(y(j)), t(f_y(j))[\tag{3.2}$$

Note that when $C_{ES}(x(i)) = true$, the property $ES[x, y]$ is respected for $x(i)$ without any need for explicit enforcement. Enforcement is only needed when $C_{ES}(x(i)) = false$.

[6]The right endpoint is excluded from the interval since emission of the final reply f_y implies that the execution of activity y has terminated.

Table 3.3 enumerates the possible behaviours of the SUT for a request $x(i)$ with respect to the exclusive start property $ES[x, y]$ enforced by rejection $x \leftarrow^R y$. Although the table is related to request $x(i)$, it must also now include a column corresponding to the observed final replies for the considered conflicting request $y(j)$, i.e., when the condition $C_{ES}(x(i))$ evaluates to false. In particular, it is necessary to consider the situation in which the conflicting request is in fact itself rejected (due to enforcement of some other property). In that case, no activity $y(j)$ can have started to execute, so there is no need to reject request $x(i)$. Thus, a new outcome, "not applicable", is introduced.

Since the tester must also observe the final reply for the conflicting request $y(j)$, the possibility of that final reply never being observed (case ω corresponding to a truncated trace), must also be taken into account.

Table 3.4 summarizes the corresponding test verdicts. The terms $\forall' y$ and $\exists' y$ in the $\{f_y(j)\}$ column of tables 3.3 and 3.4 refer respectively to universal and existential quantification over the set of requests $\{y(j)\}$ that negate expression (3.2), i.e., $\forall' y$ is a shorthand for $\forall y(j), \overline{C_{ES}(x(i))}$, and $\exists' y$ is shorthand for $\exists y(j), \overline{C_{ES}(x(i))}$.

Table 3.3: Analysis of possible behaviours for $ES[x, y]$ $(x \leftarrow^R y)$

$C_{ES}(x(i))$	$f_x(i)$	$\{f_y(j)\}$	Analysis	Verdict
$true$	$\in \{Z_x, T_x\}$		correct acceptance of x (see Figure 3.5.a)	true negative
	r_P		incorrect rejection of x	false positive
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to x	truncated trace
$false$	$\in \{Z_x, T_x\}$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, incorrect acceptance of x	false negative
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $ES[x, y]$ no longer applicable to x	not applicable
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	r_P	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, correct rejection of x (see Figure 3.5.b)	true positive
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $ES[x, y]$ no longer applicable to x and rejection of x is incorrect	false positive
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to x	truncated trace

Table 3.4: Test verdicts for $ES[x, y]$, $x \leftarrow^{R} y$

		$f_x(i)$			
$C_{ES}(x(i))$	$\{f_y(j)\}$	$\in \{Z_x, T_x\}$	r_P	$\in R_x \backslash r_P$	τ
$true$		TN	FP	op	ω
$false$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	FN	TP	op	ω
	$\forall' y, f_y(j) \in R_y$	na	FP	op	ω
	$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	ω	ω	op	ω

3.3.2.3 Property $EE[x, y]$: exclusive execution

Figure 3.6 illustrates enforcement of the exclusive execution property by interruption $x \leftarrow^I y$.

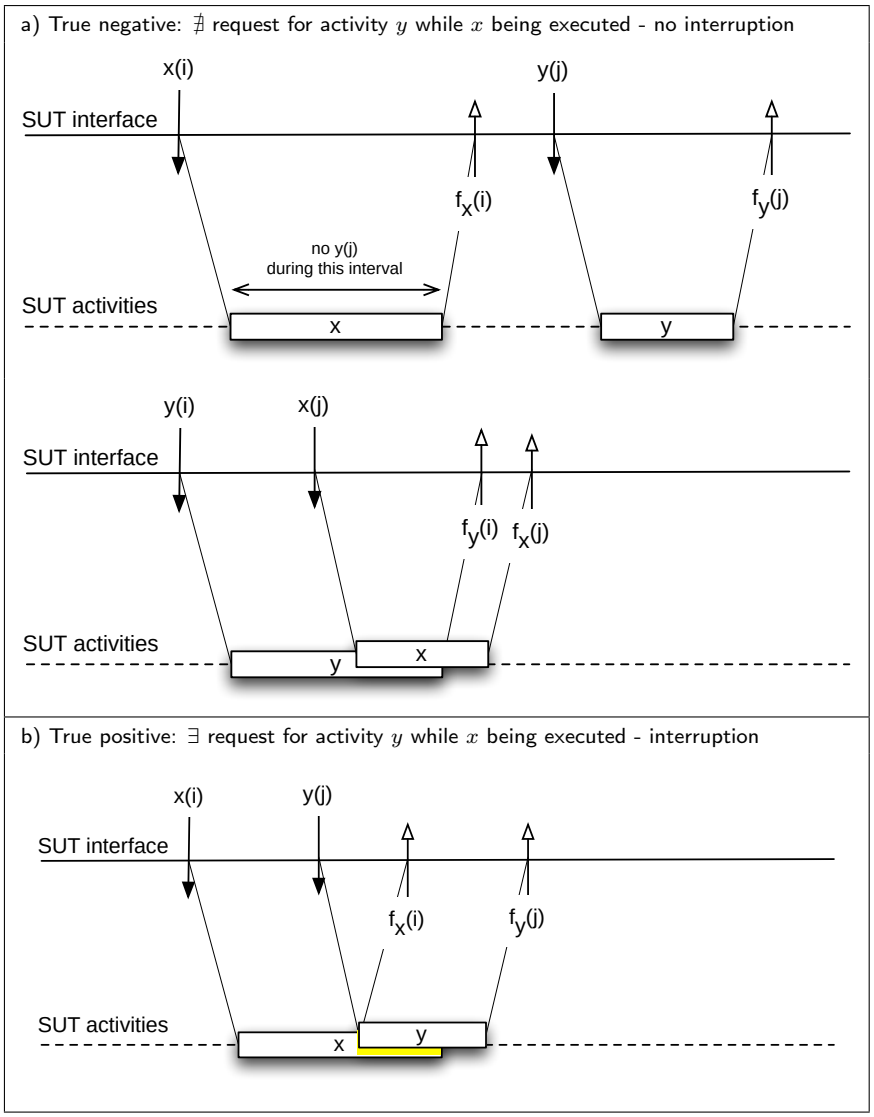


Figure 3.6: Property $EE[x, y]$ enforced by interruption $(x \leftarrow^I y)$

In this case, the activity corresponding to the request $x(i)$ is interrupted to enforce the property $EE[x, y]$ if, during its execution, a request y is received, i.e., $\exists y(j), t(y(j)) \in [t(x(i)), t(f_x(i))[$ (cf. Figure 3.6b). We define the negation of this condition as:

$$C_{EE}(x(i)) = \nexists y(j), t(y(j)) \in [t(x(i)), t(f_x(i))[\tag{3.3}$$

Note that when $C_{EE}(x(i)) = true$, the property $EE[x, y]$ is respected for $x(i)$ without any need for explicit enforcement. Enforcement is only needed when $C_{EE}(x(i)) = false$.

Table 3.5 defines the possible behaviours of the SUT for a request $x(i)$ with respect to the exclusive execution property $EE[x, y]$ enforced by interruption $x \leftarrow^I y$. Table 3.6 summarizes the corresponding test verdicts. The terms $\forall' y$ and $\exists' y$ in the $\{f_y(j)\}$ column of tables 3.5 and 3.6 refer respectively to universal and existential quantification over the set of requests

$\{y(j)\}$ that negate expression (3.3), i.e., $\forall' y$ is a shorthand for $\forall y(j), \overline{C_{EE}(x(i))}$, and $\exists' y$ is shorthand for $\exists y(j), \overline{C_{EE}(x(i))}$).

Note that Table 3.5 also contains the "not applicable" case, for both truth values of $C_{EE}(x(i))$. Indeed, there can be no activity to interrupt if the request $x(i)$ has been rejected for some other reason. Similarly, when $C_{EE}(x(i)) = false$, it is necessary to take account of the outcome of request $y(j)$ to decide whether or not $x(i)$ needs to be interrupted. Indeed, if $y(j)$ is rejected (e.g., due to the enforcement of some other property), $x(i)$ need not be interrupted.

Table 3.5: Analysis of possible behaviours for $EE[x, y]$ ($x \leftarrow^I y$)

$C_{EE}(x(i))$	$f_x(i)$	$\{f_y(j)\}$	Analysis	Verdict
$true$	$\in \{Z_x \backslash z_P, T_x\}$		correct continuation of x (see Figure 3.6.a)	true negative
	z_P		incorrect interruption of x	false positive
	$\in R_x$		no activity x to interrupt	not applicable
	τ		no response to x	truncated trace
$false$	$\in \{Z_x \backslash z_P, T_x\}$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, incorrect continuation of x	false negative
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $EE[x, y]$ no longer applicable to x	not applicable
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	z_P	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, correct interruption of x (see Figure 3.6.b)[7]	true positive
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $EE[x, y]$ no longer applicable to x and interruption of x is incorrect[8]	false positive
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	$\in R_x$		no activity x to interrupt	not applicable
	τ		no response to x	truncated trace

[7] With the considered level of observation, it cannot be decided whether the interruption of x is due to the first or a subsequent conflicting request y. Thus, the correct implementation of $EE[x, y]$ for $y = y_1$ can mask the incorrect implementation of $EE[x, y]$ for $y = y_2$ in a test sequence in which y_2 precedes y_1.

[8] Judging interruption of x to be incorrect takes the pessimistic assumption that, when a conflicting request y occurs, rejection of y takes precedence over interruption of x.

Table 3.6: Test verdicts for $EE[x, y]$ ($x \leftarrow^I y$)

		$f_x(i)$			
$C_{EE}(x(i))$	$\{f_y(j)\}$	$\in \{Z_x \backslash z_P, T_x\}$	z_P	$\in R_x$	τ
$true$		TN	FP	na	ω
$false$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	FN	TP	na	ω
	$\forall' y, f_y(j) \in R_y$	na	FP	na	ω
	$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	ω	ω	na	ω

3.3.2.4 Property $EX[x, y]$: exclusion

Figure 3.7 illustrates enforcement of the exclusion property by rejection/interruption $x \leftleftarrows_R^I y$.

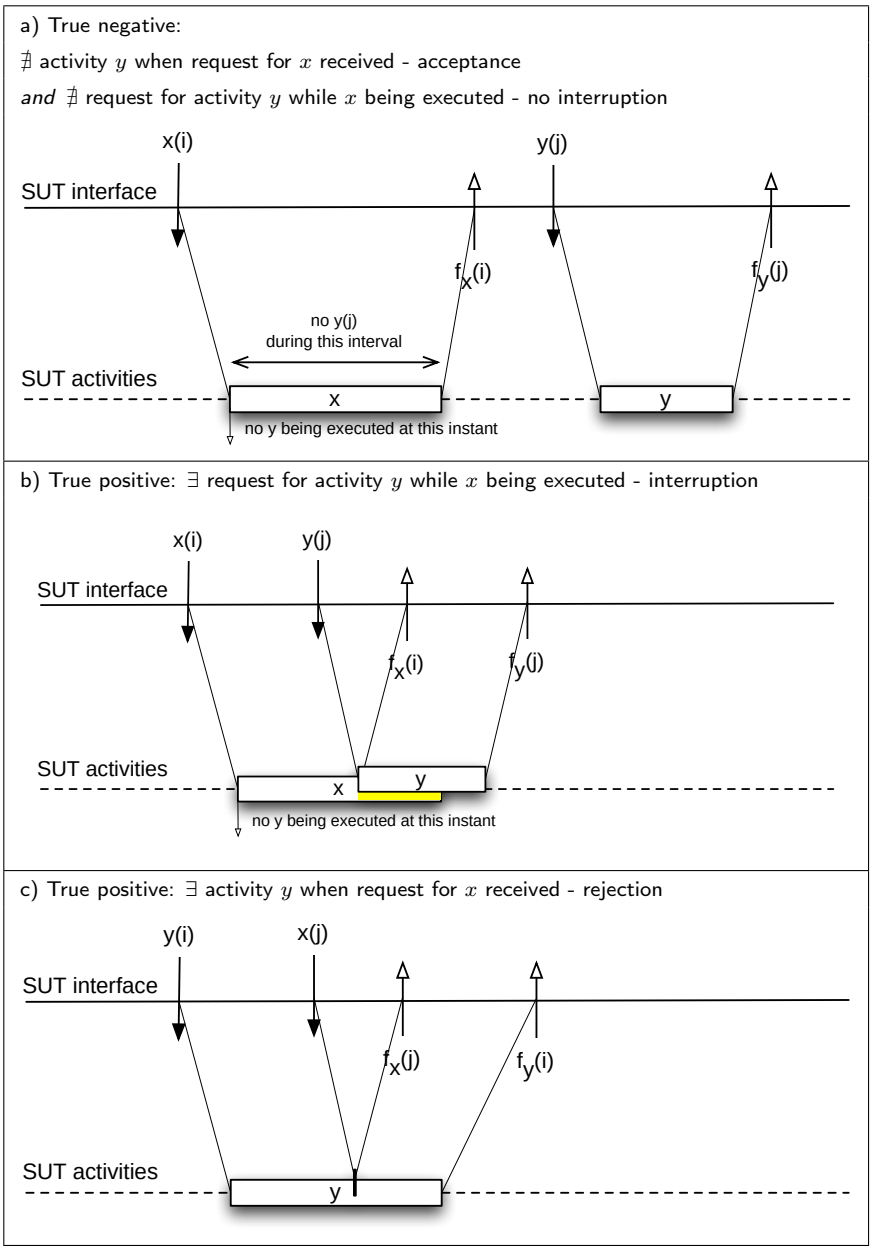


Figure 3.7: Property $EX[x, y]$ enforced by rejection/interruption ($x \leftleftarrows_R^I y$)

This property corresponds to the combination of properties $ES[x, y]$ and $EE[x, y]$:

- a request $x(i)$ must be rejected if $C_{ES}(x(i)) = false$, cf. expression (3.2);
- an activity $x(i)$ must be interrupted if $C_{EE}(x(i)) = false$, cf. expression (3.3).

We thus define the condition:

$$C_{EX}(x(i)) = C_{ES}(x(i)) \wedge C_{EE}(x(i)) \tag{3.4}$$

whence:

$$C_{EX}(x(i)) = \nexists y(j),\ \Big(t(x(i)) \in [t(y(j)), t(f_y(j))[\Big) \vee\ \Big(t(y(j)) \in [t(x(i)), t(f_x(i))[\ \Big) \quad (3.5)$$

Note that when $C_{EX}(x(i)) = true$, the property $EX[x, y]$ is respected for $x(i)$ without any need for explicit enforcement. Enforcement is only needed when $C_{EX}(x(i)) = false$.

Table 3.7 enumerates the possible behaviours of the SUT for a request $x(i)$ with respect to the exclusion property $EX[x, y]$ enforced by rejection/interruption $x \leftleftarrows^{I}_{R} y$. Table 3.8 summarizes the corresponding test verdicts.

The terms $\forall' y$ and $\exists' y$ in the $\{f_y(j)\}$ column of tables 3.7 and 3.8 refer respectively to universal and existential quantification over the set of requests $\{y(j)\}$ that negate expression (3.5), i.e., $\forall' y$ is a shorthand for $\forall y(j), \overline{C_{EX}(x(i))}$, and $\exists' y$ is shorthand for $\exists y(j), \overline{C_{EX}(x(i))}$).

Table 3.7: Analysis of possible behaviours for $EX[x, y]$, $x \leftleftarrows_R^I y$

$C_{EX}(x(i))$	$f_x(i)$	$\{f_y(j)\}$	Analysis	Verdict
true	$\in \{Z_x \backslash z_P, T_x\}$		correct inaction (see Figure 3.7.a)	true negative
	r_P		incorrect rejection of x	false positive
	z_P		incorrect interruption of x	false positive
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to x	truncated trace
false	$\in \{Z_x \backslash z_P, T_x\}$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, incorrect acceptance of x	false negative
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $EX[x, y]$ no longer applicable to x	not applicable
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge$ $\exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	r_P	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, correct rejection of x (see Figure 3.7.b)	true positive
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $EX[x, y]$ no longer applicable to x and rejection of x is incorrect	false positive
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge$ $\exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	z_P	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request y started execution, correct interruption of x (see Figure 3.7.c)[9]	true positive
		$\forall' y, f_y(j) \in R_y$	all conflicting requests y rejected for some other reason, so $EX[x, y]$ no longer applicable to x and interruption of x is incorrect[10]	false positive
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge$ $\exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to x	truncated trace

[9] With the considered level of observation, it cannot be decided whether the interruption of x is due to the first or a subsequent conflicting request y. Thus, the correct implementation of $EX[x, y]$ for $y = y_1$ can mask the incorrect implementation of $EX[x, y]$ for $y = y_2$ in a test sequence in which y_2 precedes y_1.

[10] Judging interruption of x to be incorrect takes the pessimistic assumption that, when a conflicting request y occurs, rejection of y takes precedence over interruption of x.

Table 3.8: Test verdicts for $EX[x, y]$ ($x \leftleftarrows^{I}_{R} y$)

		$f_x(i)$			
$C_{EX}(x(i))$	$\{f_y(j)\}$	$\in \{Z_x \backslash z_P, T_x\}$	$\in \{r_P, z_P\}$	$\in R_x \backslash r_P$	τ
$true$		TN	FP	op	ω
$false$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	FN	TP	op	ω
	$\forall' y, f_y(j) \in R_y$	na	FP	op	ω
	$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	ω	ω	op	ω

3.3.2.5 Property $MX[x, y]$: mutual exclusion

Figures 3.8 and 3.9 illustrate the enforcement of the mutual exclusion property by mutual rejection $x \overset{R}{\rightleftarrows}_R y$ and mutual interruption $x \overset{I}{\rightleftarrows}_I y$ respectively.

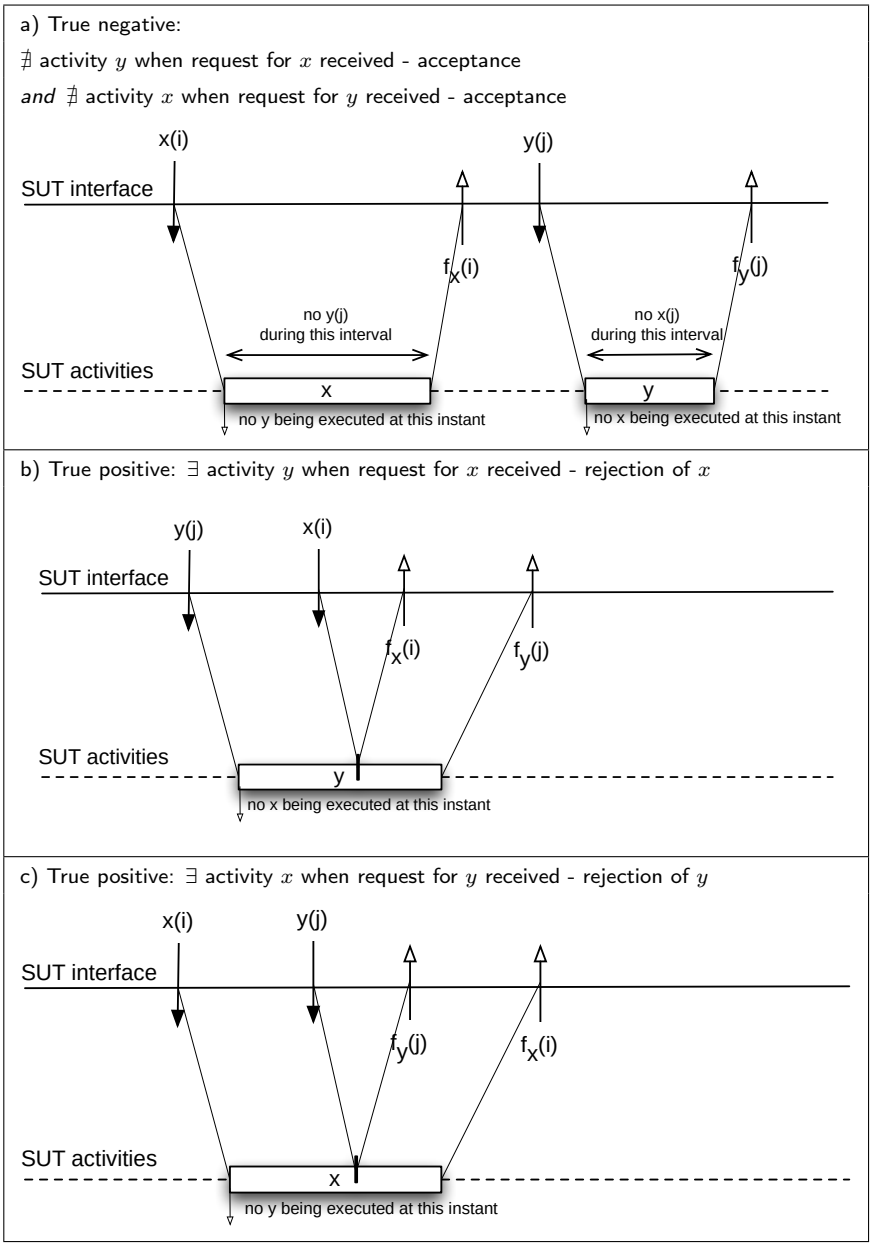


Figure 3.8: Property $MX[x, y]$ enforced by mutual rejection ($x \overset{R}{\rightleftarrows}_R y$)

Mutual exclusion between x and y means that no request for x (respectively y) must be accepted during the execution of y (respectively x). We thus define the exclusion condition as follows:

$$C_{MX}(x(i)) = \nexists y(j),\ \Big(t(x(i)) \in [t(y(j)), t(f_y(j))[\Big) \vee \Big(t(y(j)) \in [t(x(i)), t(f_x(i))[\Big) \quad (3.6)$$

Note that when $C_{MX}(x(i)) = true$ (cf. Figures 3.8a and 3.9a)., the property $MX[x, y]$ is respected for $x(i)$ and $y(j)$ without any need for explicit enforcement. Enforcement is only needed when $C_{MX}(x(i)) = false$.

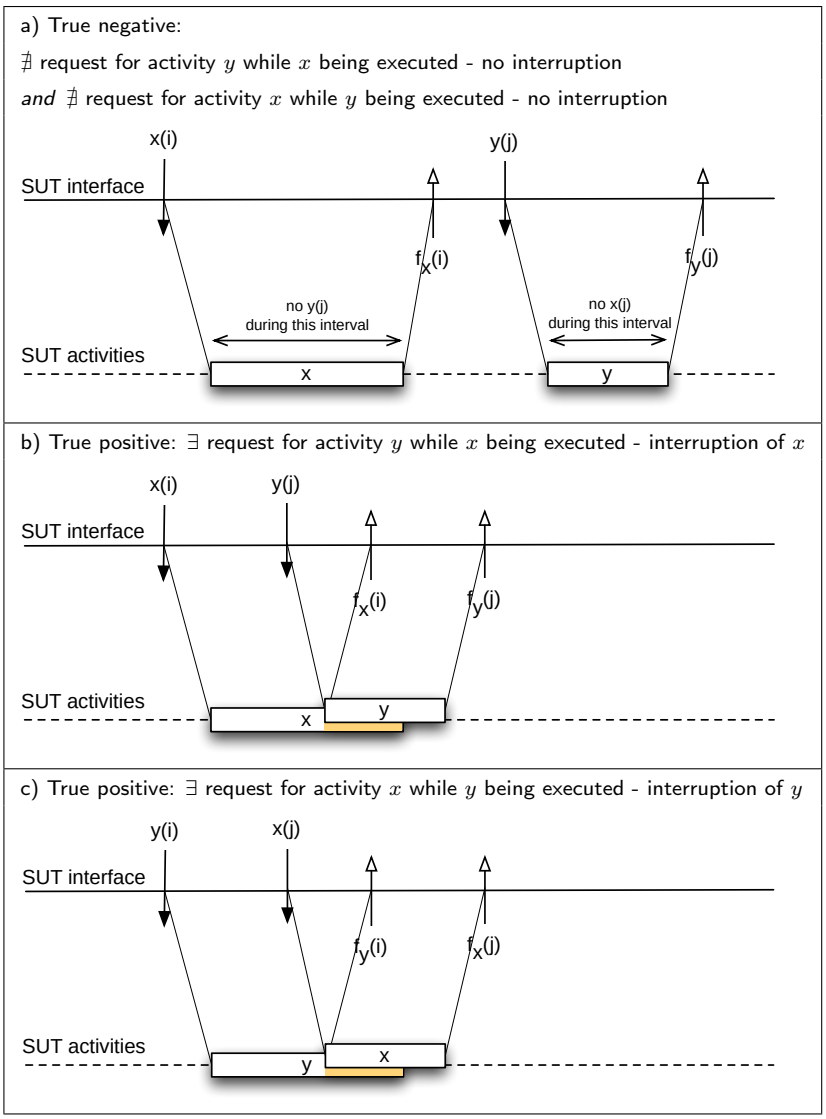


Figure 3.9: Property $MX[x, y]$ enforced by mutual interruption ($x \; {}_I\!\leftrightarrows^I \; y$)

It can also be noted that the expression 3.5 for $C_{EX}(x(i))$ is identical to expression 3.6 for $C_{MX}(x(i))$. Indeed, both define properties define exclusion between activities of types x and y. The difference between the two is that, in the former, exclusion is enforced asymmetrically whereas here, the exclusion is enforced symmetrically.

We successively consider the mutual exclusion property enforced by mutual rejection $x \overset{R}{\rightleftarrows}_R y$ and mutual interruption $x \overset{I}{\leftrightarrows}^I y$.

$MX_R[x, y]$: mutual exclusion enforced by rejection Table 3.9 defines the possible behaviours of the SUT for requests $x(i)$ and $y(j)$ with respect to the mutual exclusion property $MX[x, y]$ enforced by mutual rejection ($x \overset{R}{\rightleftarrows}_R y$). Table 3.11 summarizes the corresponding test verdicts.

The terms $\forall' y$ and $\exists' y$ in the $\{f_y(j)\}$ column of tables 3.9 and 3.11 refer respectively to universal and existential quantification over the set of requests $\{y(j)\}$ that negate expression (3.6), i.e., $\forall' y$ is a shorthand for $\forall y(j), \overline{C_{MX}(x(i))}$, and $\exists' y$ is shorthand for $\exists y(j), \overline{C_{MX}(x(i))}$).

$MX_I[x, y]$: mutual exclusion enforced by interruption Table 3.10 defines the possible behaviours of the SUT for requests $x(i)$ and $y(j)$ with respect to the mutual exclusion property $MX[x, y]$ enforced by mutual interruption ($x \overset{I}{\leftrightarrows}^I y$). Table 3.12 summarizes the corresponding test verdicts.

The terms $\forall' y$ and $\exists' y$ in the $\{f_y(j)\}$ column of tables 3.10 and 3.12 refer respectively to universal and existential quantification over the set of requests $\{y(j)\}$ that negate expression (3.6), i.e., $\forall' y$ is a shorthand for $\forall y(j), \overline{C_{MX}(x(i))}$, and $\exists' y$ is shorthand for $\exists y(j), \overline{C_{MX}(x(i))}$).

Table 3.9: Analysis of possible behaviours for $MX_R[x, y]$ ($x \overset{R}{\rightleftarrows}_R y$)

$C_{MX}(x(i))$	$f_x(i)$	$\{f_y(j)\}$	Analysis	Verdict
true	$\in \{Z_x, T_x\}$		correct acceptance of x, see Figure 3.8.a	true negative
	r_P		incorrect rejection of x	false positive
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to x	truncated trace
false	$\in \{Z_x, T_x\}$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request started execution, incorrect acceptance of x	false negative
		$\forall' y, f_y(j) \in \{r_P, R_y \backslash r_P\}$	all conflicting requests y rejected, either to ensure $MX_R[x, y]$ (see Figure 3.8.c) or for some other reason, so $MX_R[x, y]$ no longer applicable to x	not applicable
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	r_P	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	some conflicting request started execution, correct rejection of x (see Figure 3.8.b)	true positive
		$\forall' y, f_y(j) \in R_y \backslash r_P$	all conflicting requests y rejected for some other reason, so $MX_R[x, y]$ no longer applicable to x and rejection of x is incorrect	false positive
		$\forall' y, f_y(j) \in \{r_P, R_y \backslash r_P\} \wedge \exists' y, f_y(j) = r_P$	at least one conflicting request y rejected to ensure $MX_R[x, y]$, others either similarly rejected, or rejected for some other reason[11]	true positive
		$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	$\in R_x \backslash r_P$		x rejected for some other reason	other positive
	τ		no response to $x(i)$	truncated trace

[11] The situation where both x and one (or more) conflicting requests y has been rejected to ensure mutual exclusion corresponds to a race condition. In this situation, we consider that all rejections of requests involved in the race condition to be correct.

Table 3.10: Analysis of possible behaviours for $MX_I[x, y]$ ($x\ {}_I\!\leftrightarrows^I\ y$)

$C_{MX}(x(i))$	$f_x(i)$	$\{f_y(j)\}$	Analysis	Verdict
true	$\in \{Z_x \backslash z_P, T_x\}$		correction continuation of x, see Figure 3.9.a	true negative
	z_P		incorrect interruption of x	false positive
	$\in R_x$		no activity x to interrupt	not applicable
	τ		no response to x	truncated trace
false	$\in \{Z_x \backslash z_P, T_x\}$	$\exists' y, f_y(j) \in \{Z_y \backslash z_P, T_y\}$	some conflicting request y started execution, incorrect continuation of x	false negative
		$\forall' y, f_y(j) \in \{z_P, R_y\}$	all conflicting requests either interrupted to ensure $MX_I[x, y]$ (see Figure 3.9.c) or rejected for some other reason, so $MX_I[x, y]$ no longer applicable to x	not applicable
		$\forall' y, f_y(j) \notin \{Z_y \backslash z_P, T_y\} \wedge$ $\exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	z_P	$\exists' y, f_y(j) \in \{Z_y \backslash z_P, T_y\}$	some conflicting request y started execution, correction interruption of x, see Figure 3.9.b[12]	true positive
		$\forall' y, f_y(j) \in R_y$	all conflicting requests rejected for some other reason, so $MX_I[x, y]$ no longer applicable to x and interruption of x is incorrect[13]	false positive
		$\forall' y, f_y(j) \in \{z_P, R_y\} \wedge$ $\exists' y, f_y(j) = z_P$	at least one conflicting request y interrupted to ensure $MX_I[x, y]$, others either similarly interrupted, or rejected for some other reason[14]	true positive
		$\forall' y, f_y(j) \notin \{Z_y \backslash z_P, T_y\} \wedge$ $\exists' y, f_y(j) = \tau$	insufficient responses to conflicting requests y	truncated trace
	$\in R_x$		x rejected for some other reason	not applicable
	τ		no response to $x(i)$	truncated trace

[12] With the considered level of observation, it cannot be decided whether the interruption of x is due to the first or a subsequent conflicting request y. Thus, the correct implementation of $MX[x, y]$ for $y = y_1$ can mask the incorrect implementation of $MX[x, y]$ for $y = y_2$ in a test sequence in which y_2 precedes y_1.

[13] Judging interruption of x to be incorrect takes the pessimistic assumption that, when a conflicting request y occurs, rejection of y takes precedence over interruption of x.

[14] The situation where both x and one (or more) conflicting requests y has been interrupted to ensure mutual exclusion corresponds to a race condition. In this situation, we consider that all interruptions of requests involved in the race condition to be correct.

Table 3.11: Test verdicts for $MX_R[x, y]$ ($x \overset{R}{\rightleftarrows}_R y$)

		$f_x(i)$			
$C_{MX}(x(i))$	$\{f_y(j)\}$	$\in \{Z_x, T_x\}$	r_P	$\in R_x \backslash r_P$	τ
$true$		TN	FP	op	ω
$false$	$\exists' y, f_y(j) \in \{Z_y, T_y\}$	FN	TP	op	ω
	$\forall' y, f_y(j) \in R_y \backslash r_P$	na	FP	op	ω
	$\forall' y, f_y(j) \in R_y \wedge \exists' y, f_y(j) = r_P$	na	TP	op	ω
	$\forall' y, f_y(j) \notin \{Z_y, T_y\} \wedge \exists' y, f_y(j) = \tau$	ω	ω	op	ω

Table 3.12: Test verdicts for $MX_I[x, y]$ ($x \overset{I}{\leftrightarrows}^I y$)

		$f_x(i)$			
$C_{MX}(x(i))$	$\{f_y(j)\}$	$\in \{Z_x \backslash z_P, T_x\}$	z_P	$\in R_x$	τ
$true$		TN	FP	na	ω
$false$	$\exists' y, f_y(j) \in \{Z_y \backslash z_P, T_y\}$	FN	TP	na	ω
	$\forall' y, f_y(j) \in R_y$	na	FP	na	ω
	$\forall' y, f_y(j) \in \{z_P, R_y\} \wedge \exists' y, f_y(j) = z_P$	na	TP	na	ω
	$\forall' y, f_y(j) \notin \{Z_y \backslash z_P, T_y\} \wedge \exists' y, f_y(j) = \tau$	ω	ω	na	ω

3.3.3 Discussion

For each property $P \in \{PRE, ES, EE, EX, MX\}$, we have defined a condition $C_P(x(i))$ that, when evaluated for a particular request $x(i)$ (i.e., in expressions (3.1), (3.2), (3.3), (3.5), (3.6)), determines whether the property enforcement mechanism should be triggered ($C_P(x(i)) = false$) or not ($C_P(x(i)) = true$). We will henceforth generically refer to C_P as the *P-condition*.

For each property, the evaluation of the P-condition for a given request is expressed in terms of the sequence of events occurring on the internal activity line of the SUT (we will refer to the trace of such events as the *internal trace*). In practice, a real tester can only attempt to deduce the internal behaviour of the SUT from events observable at the SUT interface (which we will refer to as the *external trace*).

In general, the ordering of events on the external and internal traces may differ due to non-null event propagation times and possible re-ordering of request and reply events between the external and internal traces. It may thus be the case that the value of the P-condition evaluated on the external trace differs from the value of the P-condition evaluated on the (unobservable) internal trace. The potential for such differences raises the possibility of incorrect verdicts, which must be taken into account when evaluating test results for near-coincident events.

3.4 Conclusion

We proposed in this chapter our framework for testing the robustness of the functional layer of an autonomous system with respect to its required safety properties. The framework is a form of black-box testing where a formal behaviour specification of the SUT is not provided. We adopted a passive testing technique to evaluate the robustness of the SUT based on the observation of execution traces (requests and replies) of the system. Since we consider robustness as safety properties that the functional layer should ensure, we identified five basic safety property types along with corresponding safety enforcement policies.

The description of safety enforcement policies allows us to define an oracle to characterize the behaviour of the SUT based on execution trace analysis. For a given property, the oracle classifies the system behaviour to four main categories (*true negative*, *true positive*, *false negative*, *false positive*) and three additional categories (*other positive*, *not applicable*, *truncated trace*).

One disadvantage of black-box testing is its reduced level of system observability. This raises the possibility of drawing incorrect inferences about the system's actual internal state and the ordering of events.

In this thesis, we used the LAAS architecture as a baseline for defining our framework. However, as our framework is constructed regardless of any formal behaviour specification of the SUT, it should be applicable to other autonomous systems with a hierarchical architecture. In the next chapter, we present an application of our framework to a planetary exploration robot using the LAAS architecture. We use our approach to compare the robustness of several implementations of this robot's functional layer.

Chapter 4

Application to Dala Rover

We proposed in the previous chapter a framework for testing the robustness of the functional layer of an autonomous system. In this chapter, we present an application of our framework to a case study: the functional layer of the Dala Rover robot, which is currently used in LAAS for navigation experiments. This layer is required to respect a number of safety properties to protect the Dala Rover robot from combinations of activities that could lead to inconsistent or dangerous behaviour. Our robustness testing framework exercises the property-enforcing mechanisms of the Dala Rover functional layer by means of mutated exploration mission scripts containing potentially *temporally invalid* test inputs that may endanger the safety properties. The execution traces, consisting of the requests and replies intercepted at the functional layer interface, are then processed by a trace analyzer tool to assess the robustness of the functional layer.

We first describe the functional layer of Dala Rover robot (the "system under test" (SUT)), and the safety properties that must be enforced by the SUT. Then we present our robustness testing environment, which is an application of the FARM framework [AAA+90], and the characteristics of the test campaign that has been carried out. Finally, we present and analyze the results of the case study.

4.1 Dala application

The system under test is the functional layer of the Dala robot. The functional layer consists of a set of modules that may interact directly with the robot hardware (see Figure 4.1). The "application layer" which issues requests to modules, is here an executive layer implemented in Open-PRS [ICAR96]).

Modules may communicate directly with each other by means of *posters*. Posters are shared data structures that are each maintained by a specific module, but can be non-destructively and asynchronously read by other modules.

The considered Dala functional layer consists of five modules (cf. Figure 4.1):

- Rflex: odometry and wheel control;
- Sick: laser range finder;
- Aspect: 2D environment map;
- Ndd: navigation and obstacle avoidance;

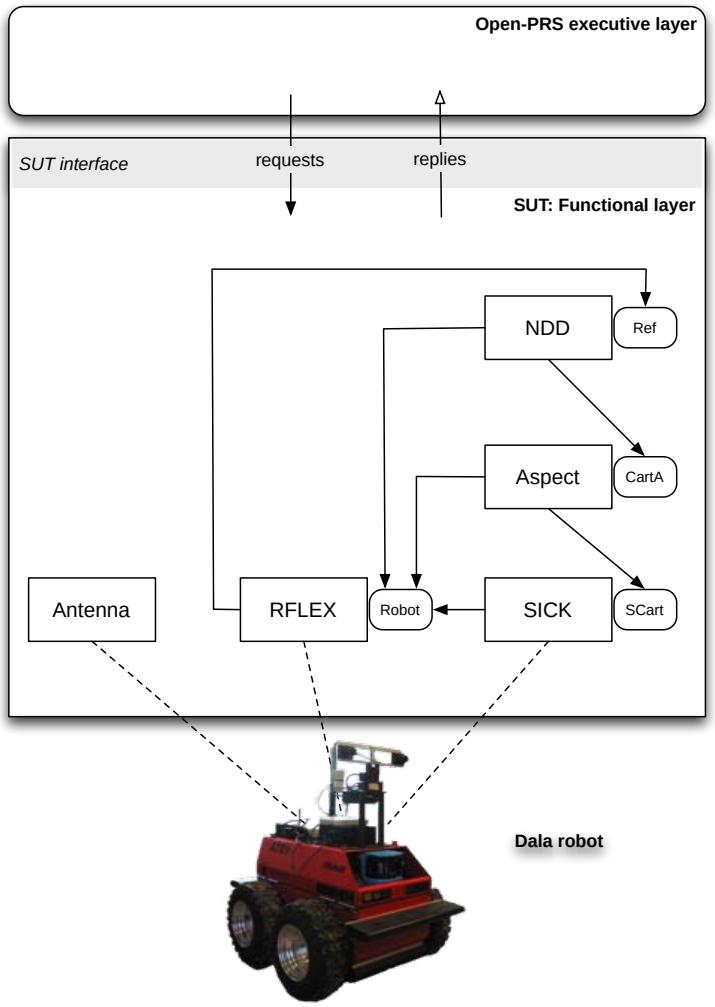


Figure 4.1: System under test: Dala robot functional layer

- Antenna: communication (to/from overhead orbiter) (simulated).

With the exception of Antenna, each module maintains a poster that is read by the other modules:

- Robot: current position of the robot as computed using the Rflex odometric function;
- Ref: reference velocity of the rover computed by Ndd;
- CartA: the 2D environment map computed by Aspect;
- SCart: the orientation and range of obstacles currently within view of the Sick obstacle detection laser.

Table 4.1 lists the requests that can be issued to each of the five considered modules, together with their associated types (init, control, exec) (c.f. 1.4.1).

Table 4.1: Module requests

module	**request name**	**basic type**
Antenna	init	init
-	comunicate[1]	exec
Sick	init	init
-	reset	exec
-	oneshoot	exec
-	continuousshot	exec
Aspect	setviewparameters	control
-	setdynamicsegssource	exec
-	aspectfromposterconfig	exec
Ndd	init	init
-	setparams	exec
-	setspeed	exec
-	goto	exec
Rflex	initclient	init
-	setmode	control
-	setwdogref	control
-	pom_tagging	control
-	trackspeedstart	exec
-	stop	exec

4.2 Specified safety properties

We define here the four families of safety properties that are required for the considered Dala robot application.

[1] Note, the spelling corresponds to that used in the real code...

4.2.1 PEX($module$): Precondition for EXec request

Definition: There must be at least one successfully completed initialization request for *module* before the latter processes any execution request.

Property type: Precondition $PC[x, C_{PRE}]$ (cf. §3.3.2.1), with:

- x = any *exec* request to *module* (request type denoted $exec(module)$);
- r_P = S_module_stdGenoM_WAIT_INIT_RQST ;
- A_{PRE} = set of initialization requests for *module*.

In the considered Dala application, there are four modules for which a PEX property is defined (cf. Table 4.1): Antenna, Ndd, Sick and Rflex. Note that Aspect is not concerned by this property since it does not have any "init" requests.

Table 4.2: Instances of PEX property

Property	$x = exec(module)$	A_{PRE}	r_P
PEX(antenna)	antenna.comunicate	antenna.init	S_antenna_stdGenoM_WAIT_INIT_RQST
PEX(ndd)	ndd.setparams; ndd.setspeed; ndd.goto	ndd.init	S_ndd_stdGenoM_WAIT_INIT_RQST
PEX(sick)	sick.reset; sick.oneshoot; sick.continuousshot	antenna.init	S_sick_stdGenoM_WAIT_INIT_RQST
PEX(rflex)	rflex.trackspeedstart: rflex.stop	antenna.initclient	S_rflex_stdGenoM_WAIT_INIT_RQST

4.2.2 AIB(x): Activity x Interrupted By

Definition: Activities of type x must either be inactive or be interrupted if any request of a type that dominates type x is issued.

Property type: Exclusive execution $EE[x, y]$ (cf. §3.3.2.3), with:

- x = every request appearing in a declaration "(y) incompatible_with x"[2]
- y = all requests that dominate type x (request type denoted $dom(x)$), i.e., that appear in a declaration "(y) incompatible_with x"
- z_P =S_module_stdGenoM_activity_interrupted

There are several AIB properties defined for each of the five modules in the considered Dala application, leading to a total 15 properties (Table 4.3).

[2]In the GenoM framework, such declarations are given in a module's .gen file.

Table 4.3: Instances of AIB property

Property	$y = dom(x)$	z_P
AIB(antenna.comunicate)	antenna.comunicate; antenna.init; antenna.stopcom	S_antenna_stdGenoM_ activity_interrupted
AIB(antenna.init)	antenna.comunicate; antenna.init	S_antenna_stdGenoM_activity_interrupted
AIB(aspect.aspectfromposterconfig)	aspect.aspectfromposterconfig	S_aspect _stdGenoM_activity_interrupted
AIB(aspect.setdynamicsegssource)	aspect.setdynamicsegssource	S_aspect _stdGenoM_activity_interrupted
AIB(ndd.goto)	ndd.exectraj; ndd.goto; ndd.gotorelative; ndd.stop	S_ndd_stdGenoM_activity_interrupted
AIB(ndd.init)	ndd.init; ndd.setparams	S_ndd_stdGenoM_activity_interrupted
AIB(ndd.setparams)	ndd.init; ndd.setparams; ndd.setspeed	S_ndd_stdGenoM _activity_interrupted
AIB(ndd.setspeed)	ndd.setparams; ndd.setspeed	S_ndd_stdGenoM_activity_interrupted
AIB(rflex.initclient)	rflex.initclient; rflex.endclient	S_rflex_stdGenoM_activity_interrupted
AIB(rflex.trackspeedstart)	rflex.stop; rflex.trackend; rflex.trackspeedstart; rflex.trackconfigstart; rflex.brakeon; rflex.brakeoff	S_rflex_stdGenoM_activity_interrupted
AIB(sick.continuousshot)	sick.init; sick.continuousshot	S_sick_stdGenoM_activity_interrupted
AIB(sick.oneshoot)	sick.init; sick.oneshoot; sick.continuousshot	S_sick_stdGenoM_activity_interrupted
AIB(sick.reset)	sick.init; sick.reset; sick.oneshoot; sick.continuousshot	S_sick_stdGenoM_activity_interrupted
AIB(sick.init)	sick.init	S_sick_stdGenoM_activity_interrupted

4.2.3 PRE(x): activity x PREceded by

Definition: Activities of type x cannot be executed until a specified set of activities have been successfully completed.

Property type: Precondition $PC[x, C_{PRE}]$ (cf. §3.3.2.1).

There are two PRE properties defined for the considered Dala application (Table 4.4).

Table 4.4: Instances of PRE property

Property	A_{PRE}	r_P
PRE(aspect.aspectfromposterconfig)	aspect.setviewparamter; aspect.setdynamicsource	aspect_init_sequence_error
PRE(ndd.goto)	ndd.setparams; ndd.setspeed	ndd_init_sequence_error

4.2.4 EXC(x, y): mutual EXClusion between activities x and y

Definition: Activities of types x and y cannot be executed simultaneously; priority to most recent request.

Property type: Exclusive execution $MX_I[x, y]$ (cf. §3.3.2.5).

A single EXC property is defined in the considered Dala application (Table 4.5).

Table 4.5: Instance of EXC property

Property	r_P
EXC(antenna.comunicate, rflex.trackspeedstart)	forbidden_action: cannot_move_while_communicating

4.3 Testing environment

4.3.1 Overview

Figure 4.2 illustrates our test environment, which is an application of the FARM framework [AAA+90] (cf. §2.2.1). We can find the four attributes *F, A, R, M* of the FARM terminology in our testing process, which is composed of the following main steps:

1. Create manually a *golden script* that defines a typical mission of a planetery explorer. (the **A**ctivity set)
2. Generate a database of *mutated scripts* by applying a *mutation* procedure to the golden script (the **F**ault set).
3. Submit the set of mutated scripts to OpenPRS for execution in order to exercise the robustness features implemented in the SUT.
4. Save the resulting execution traces, each consisting of the sequence of requests sent to and replies received from the SUT, in a Trace Database (the raw **R**eadout set).
5. Use the *Trace Analyzer* tool to parse and process the execution trace database to obtain SUT robustness verdicts for each property × request combination and for each trace (an extension of the **R**eadout set).
6. Evaluate the robustness (the **M**easure set).

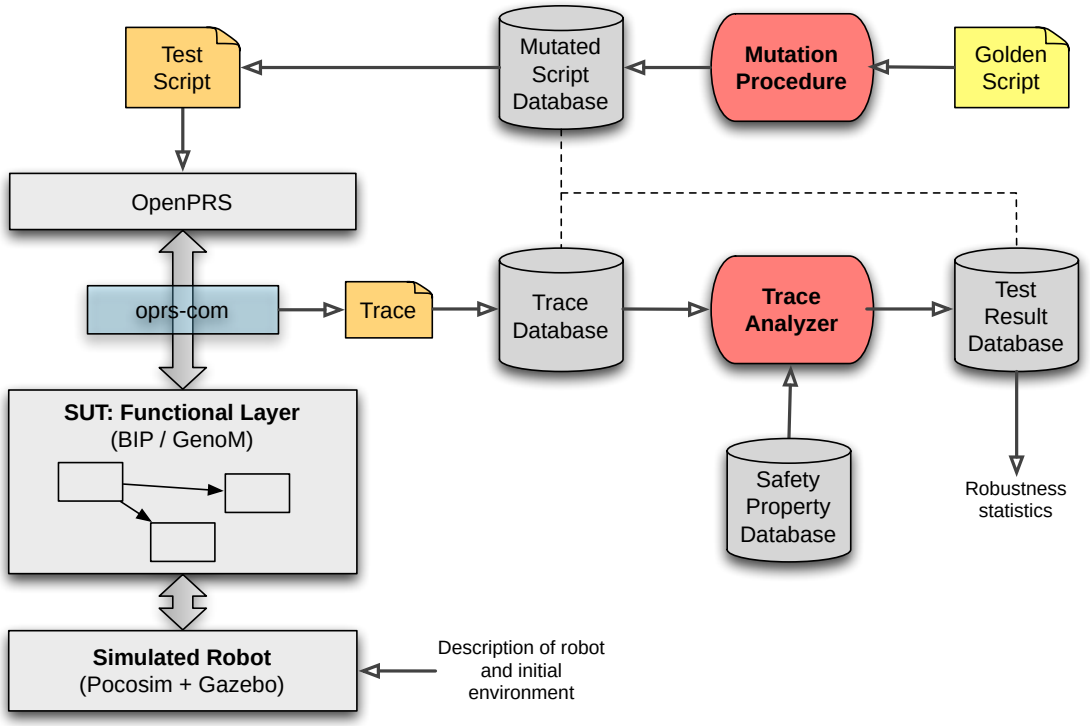


Figure 4.2: Robustness testing environment

Testing is carried out with a simulated robot instead of the real Dala robot since (a) testing needs to be automated in order to allow a large number of tests to be carried out, and (b) robustness testing implies the system under test be subjected to extreme test sequences, which could cause a real robot to behave very dangerously.

The simulation environment is composed of two modules: the Gazebo simulator and the Pocosim library.

The Gazebo simulator, which is available[3] as free open source under the *GNU General Public License*, simulates the physical parts of the robot, its environment, and their interactions. In particular, it allows simulation of the kinematics of rigid objects (acceleration, speed, collisions...), of typical robot sensors and actuators (cameras, laser range-finders, odometers, motorised wheels...). The simulator takes as input a description of the initial environment, including the mass and position of obstacles (and their speed in the case of dynamic objects), a description of the robot, its initial position, and a list of its sensors and actuators.

The Pocosim library [JALL05], developed during a previous project at LAAS, acts as a software bridge between the functional layer modules that control the hardware architecture of the real robot and the robot simulated by Gazebo. It transforms the hardware requests from the functional layer modules into actions or movements to be carried out by the simulated robot, and transmits data from sensors towards the modules.

The detailed implementation of this test process will be presented in the following sections.

[3]The Player/Stage project, http://playerstage.sourceforge.net

4.3.2 Workload

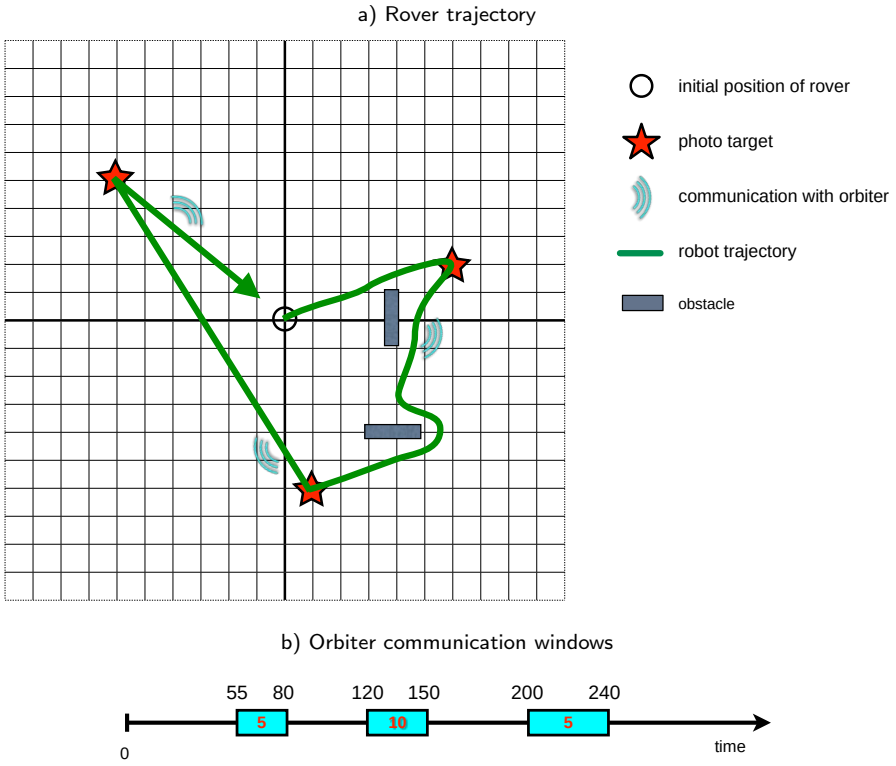


Figure 4.3: Golden script mission

The system under test is exercised by a script written in the interpretable language of OpenPRS. We defined a baseline script (called the golden script in Figure 4.2) that defines a typical mission of a planetary explorer:

- take photographs of a set of targets situated at different coordinates around the explorer's initial position;
- communicate data to a planetary orbiter within predefined periods of visibility;
- return to the initial position.

Our golden script defines three targets situated within a 20m x 20m grid, centered on the robot's initial position, and three visibility windows, as portrayed on Figure 4.3. This figure also shows two fixed rectangular objects that obstruct the robot's path and thus execise the robot's obstacle avoidance functionality (provided by the *Ndd* module). Figure 4.4 describes the golden script in our experimental setup. [4].

The golden script executed by OpenPRS and the initial environment description input to Gazebo (cf. Figure 4.2) together constitute the nominal *workload* of the system under test (or, in the FARM terminology of [AAA+90], the **A**ctivity set).

[4]The full golden script can be found at `http://homepages.laas.fr/hnchu/these/hnchu_testcase_2011.tar.gz`

```
1: procedure MainMission
2:     Initialize modules
3:     goalList ← Initialize list of target positions
4:     comWindows ← Initialize list of communication windows
5:     robotStatus ← NORMAL                          ▷ Robot is ready to move
6:     Execute in parallel MoveMission(goalList) and ComMission (comWindows)
7: end procedure

8: procedure MoveMission(goalList)                   ▷ Accomplish exploration mission
9:     for all goal_i ∈ goalList do
10:        attempt ← 0
11:        goalAchieved ← false
12:        repeat
13:            if robotStatus = NORMAL then          ▷ Check if robot is ready to move
14:                Setup modules to be ready for a move
15:                Set goal_i for robot
16:                Move the wheels
17:                Wait until Robot arrives at goal_i ∨ Error returned ∨ Time Out
18:                if Robot arrives at goal_i then
19:                    Remove goal_i from goalList
20:                    Stop the wheels
21:                    goalAchieved ← true
22:                else if Error returned ∨ Time Out then
23:                    attempt ← attempt + 1         ▷ Make another attempt
24:                end if
25:            end if
26:        until attempt = 3 ∨ goalAchieved = true   ▷ Robot can makes at most 3 attempts
27:    end for
28: end procedure

29: procedure ComMission(comWindows)                 ▷ Accomplish communication tasks
30:    for all comW_i ∈ comWindows do
31:        Wait until Robot is in comW_i
32:        Stop the wheels
33:        robotStatus ← COM                         ▷ Robot is in a communication
34:        Carry out communication for a duration t
35:        robotStatus ← NORMAL                      ▷ Robot is ready to move
36:    end for
37: end procedure
```

Figure 4.4: Golden script of Dala mission (in pseudo-code)

4.3.3 Faultload

To exercise the robustness features of the system under test, we need to subject it to *invalid* test inputs. Since our concern is with robustness with respect to inputs that are invalid in the time domain, we focus on inputs that are submitted at the "wrong time" rather than inputs with invalid syntax or invalid parameters. Our approach is based on fault injection by mutation. Starting from the golden script, we create a set of *mutated scripts*, each based on one of the following mutations (the **F**ault set in the FARM terminology of [AAA+90]):

- deletion of a module request;
- insertion of a module request;
- re-ordering of a pair of module requests.

These mutations are carried out randomly. The aim is to change the normal sequence of requests contained within the golden script and cause some requests to be invalid in the time domain. Of course, there is a high risk that the mutated script does not correspond to a meaningful mission at the application level, so it is not of much interest to observe whether the robot achieves its original goals or not. However, it is of concern to us whether the safety properties of the functional layer (the SUT) are respected.

Figure 4.5 presents an example of a re-ordering mutation. On the left is an excerpt of the golden script in charge of initializing the *Ndd* module: after retrieving the configuration parameters (lines 1-3), the initialization process is carried out by executing sequentially 3 requests: NDD-INIT , NDD-SETPARAMS , NDD-SETSPEED . By applying the re-ordering mutation on NDD-INIT and NDD-SETPARAMS , we create a new script that can endanger the property PEX(ndd) .

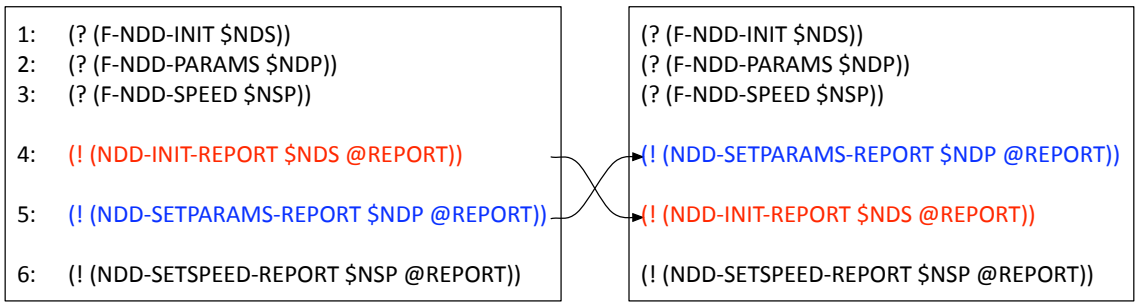


Figure 4.5: Mutation example (request re-ordering)

We generated 300 test scripts by an semi-automated random mutation process, which executes randomly one of the three mutation procedures shown as pseudo-code on Figure 4.6. Of these 300 scripts, 7 were refused by the OpenPRS interpreter. Our test script database thus consisted of 293 syntactically correct but semantically inconsistent OpenPRS scripts.

```
1: procedure REORDERING(goldenScript)                          ▷ Reordering mutation
2:     Choose randomly 2 golden script lines L1 and L2 that contain a SUT request
3:     Swap position of L1 and L2
4:     Save the mutated script
5: end procedure

6: procedure DELETION(goldenScript)                            ▷ Deletion mutation
7:     Choose randomly a golden script line L that contains a SUT request
8:     Delete line L
9:     Save the mutated script
10: end procedure

11: procedure INSERTION(goldenScript)                          ▷ Insertion mutation
12:    Choose randomly a module request R in the SUT request set
13:    Put R in a random position in the golden script
14:    Save the mutated script
15: end procedure
```

Figure 4.6: Mutation process (in pseudo-code)

4.4 Readouts

4.4.1 System logging

To retrieve the execution traces, we implemented an instrumented version of the oprs-com communication library, which allows communication between the functional layer and the OpenPRS procedural executive (cf. Figure 4.2). The instrumented version allows interception of requests sent to and replies received from the SUT by OpenPRS. The intercepted requests and replies are logged to an SQL *trace database* in both full (XML) and simplified (plain text) formats. Figure 4.7 presents an excerpt of trace in both formats. In the XML format execution trace (Figure 4.7a), a node represents an event (request/reply) sent to or received from the SUT. A node is composed of several tags that describe different information fields of an event:

- `<data type>`: the type of event (send: request; rcv: final reply; ir: immediate reply)
- `<id>`: the request identifier
- `<time>`: the system time in Unix time
- `<cmd>`: the command type (defined in OpenPRS)
- `<name>`: the request name
- `<module>`: the module that receives or sends the event
- `<params>`: the parameters of the request sent (only in a *send* event)
- `<report>`: the final reply of a request (only in a *rcv* event)

However, there are some information fields that are not significant for robustness testing, so we also implemented a simplified trace format (Figure 4.7b). This facilitates the post-processing of the data, leading to increased performance of the *Trace Analyzer* tool. This

```
<data type="send">
    <id>32</id>
    <time>1289392514.23</time>
    <cmd>14</cmd>
    <name>RFLEX_TRACKSPEEDSTART</name>
    <module>RFLEX</module>
    <params>
        <GENPOS_POSTER_NAME>
            <name>nddRef</name>
        </GENPOS_POSTER_NAME>
    </params>
</data>
```

a) Full trace (XML format)

```
1289392514.23   send   32   RFLEX_TRACKSPEEDSTART
1289392514.28   ir     32              RFLEX_TRACKSPEEDSTART
1289392516.32   rcv    31                        NDD_GOTO   OK
1289392516.43   send   33   RFLEX_STOP
1289392516.54   ir     33                        RFLEX_STOP
1289392516.64   rcv    32              RFLEX_TRACKSPEEDSTART   S_rflex_stdGenoM_ACTIVITY_INTERRUPTED
1289392516.64   rcv    33                        RFLEX_STOP   OK
```

b) Specified trace (plain text format)

Figure 4.7: Excerpts of trace in XML (a) and simplified format (b)

simplified format is also easier to read by humans, so it facilitates the manual debug and analysis process. The simplified version of the trace is composed of only some important fields of an event. Each line shows:

- the system time
- the type of event
- the request identifier
- the request name
- the final reply of a request (for *rcv* events).

This trace database contains the raw observation data obtained from each experiment (the **R**eadout set in the FARM terminology of [AAA+90]). This raw data is then processed by the *Trace Analyzer* tool to provide more synthetic readouts regarding the robustness behavior of the SUT. We use MySQL as the database management system, which is available as a free open source under the *GNU General Public License*.

4.4.2 *Trace Analyzer*: system observation tool

Our trace analyzer tool is based on the passive testing approach to robustness testing [CMM08] (cf. Section 2.4). However, instead of expressing the safety properties as regular expressions, we prefered the flexibility and expressiveness of an implementation based on SQL and Java. The tool is composed of two components (cf. Figure 4.8):

- A *Safety Property Database*: this database contains the definition of safety properties. Each property is defined as a set of SQL queries according to the set of behaviours $\{TN, TP, FN, FP, op, na, \omega\}$ with a database of relevant requests.
- An *Analyzer* module: this module, written in Java, orchestrates the query process.

The purpose of the queries is to analyze the robustness behavior of the system under test for every request event in the trace, and according to every property defined for that request. For example, *ndd.goto* requests are analyzed according to three properties:

- property PEX(ndd) , cf. Table 4.2;
- property AIB(ndd.goto) , cf. Table 4.3;
- property PRE(ndd.goto) , cf. Table 4.4.

Figure 4.9 summarizes the analysis procedure. For each property P, and for each relevant request x in the trace, the trace analyzer evaluates the test verdict according to the set of outcomes $\{TN, TP, FN, FP, op, na, \omega\}$ defined in Section 3.3.1.

The results of this analysis are available at three levels of granularity:

- for each individual request-property pair $< x, P >$;
- as an aggregate over all requests concerned by each property P;
- as an aggregate over all requests concerned by each property in a family of properties: PEX, AIB, PRE, EXC.

Additionally, the trace analyzer categorizes each *trace* as either *nominal* or *hang* according to whether the trace terminates normally or not. This categorization is based on an SQL query that searches for the responses to a particular pattern of requests that appears at the end, and only at the end, of each test script. In the current application, this particular pattern is a sequence of "abort" requests sent to each of the five modules of the considered Dala application.

4.5 Measures

Whereas the readouts indicate the results for each individual experiment (i.e., each trace in the database), the set of all readouts can be processed statistically to provide one or more measures of the overall robustness of the system under test (the **M**easure set in the FARM terminology of [AAA+90]).

There is no pretence to claim that the workload and faultload to which the system under test is submitted are in any way statistically representative of the real workload and faultload that would be experienced in a real deployment. The proposed measures are thus simply *descriptive statistics* of the behavior of the system under test with respect to the specific workload and faultload to which it was submitted. Nevertheless, since exactly the same workload and faultload is submitted to the several implementations of the system under test, they do provide a benchmark of their relative robustness.

We define the following descriptive statistics:

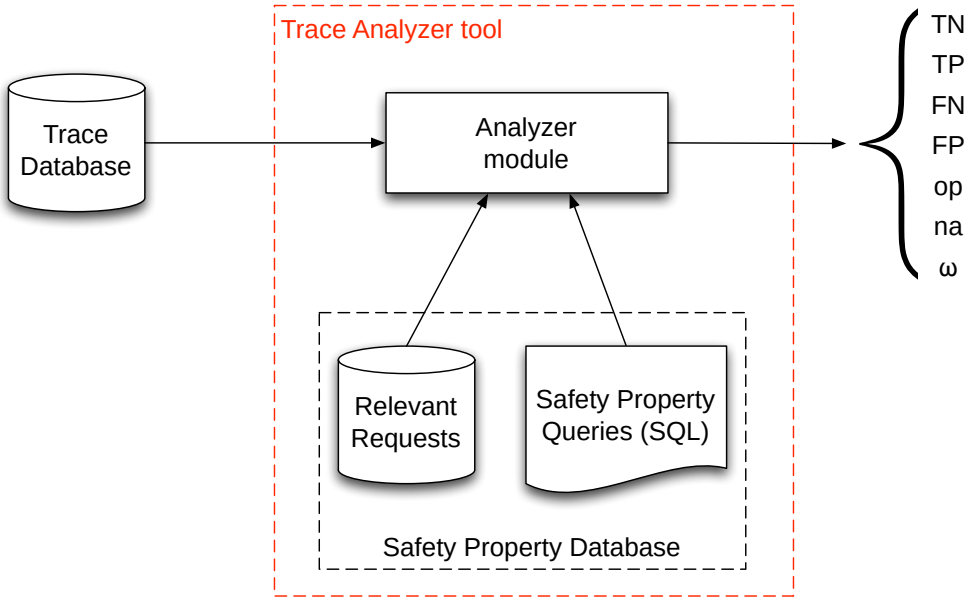


Figure 4.8: *Trace Analyzer* architecture

1: **for all** properties P **do**
2: **for all** behaviours $B_P \in \{TN, TP, FN, FP, na, \omega\}$ **do**
3: Get the SQL statement st_{BP} corresponding to B_P
4: **for all** requests rq_i relevant to P **do**
5: $result \leftarrow$Execute st_{BP} with rq_i as input
6: **end for**
7: **end for**
8: **end for**

Figure 4.9: Trace analysis procedure

Trace robustness (T_{ROB}): the proportion of experiments leading to traces exempt from false positives, false negatives and hangs. Ideally, the trace robustness should be 100%.

True positive rate (TPR): the proportion of correct (robust) reactions of the system under test when the *P-condition* (c.f. section 3.3.3) evaluates to $false$ (also known as the *sensitivity* or *coverage* of the mechanisms). For a given request population, we define TPR as:

$$TPR = \frac{N_{TP}}{N_{TP} + N_{FN}}$$

where N_{TP} (respectively N_{FN}) is the number of requests in the population for which the test verdict is *true positive* (respectively, *false negative*). Ideally, the true positive rate should be 100%.

False positive rate (FPR): the proportion of incorrect reactions of the system under test when the *P-condition* (c.f. section 3.3.3) evaluates to *true* (also known as the fall-out or false alarm rate). For a given request population, we define FPR as:

$$FPR = \frac{N_{FP}}{N_{FP} + N_{TN}}$$

where N_{FP} (respectively N_{TN}) is the number of requests in the population for which the test verdict is *false positive* (respectively, *true negative*). Ideally, the false positive rate should be 0%.

Here, we evaluate TPR and FPR for the set of requests in all traces concerned by each family of properties (PEX, AIB, PRE, EXC), and for *all* properties combined.

4.6 Results

The proposed robustness testing approach has been applied to three implementations of the simplified Dala application shown in Figure 4.1, which we designate as follows:

GenoM : a well-established implementation using the standard *GenoM* environment, which provides built-in protection to ensure properties of the families PEX and AIB only;

BIP-A : a preliminary implementation using the BIP framework, with a large proportion of BIP code generated automatically from the GenoM module descriptions, together with additional protection mechanisms generated from BIP inter-component connectors;

BIP-B : a more mature implementation using the BIP framework with, in particular, several corrections resulting from the experiments carried out on *BIP-A*.

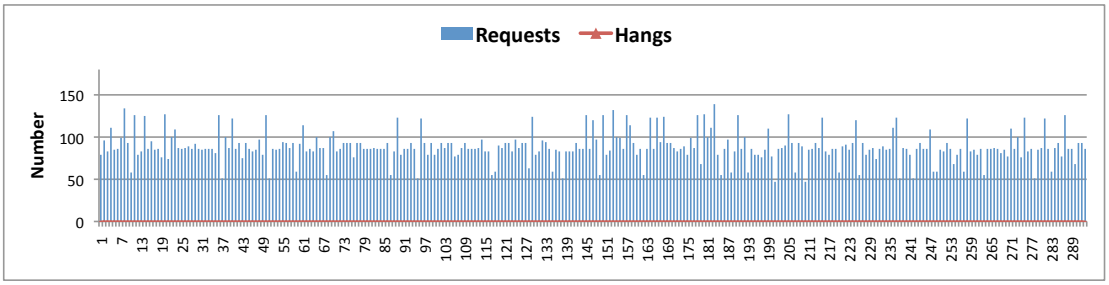
We describe hereafter the results obtained on each version.

4.6.1 Per trace results

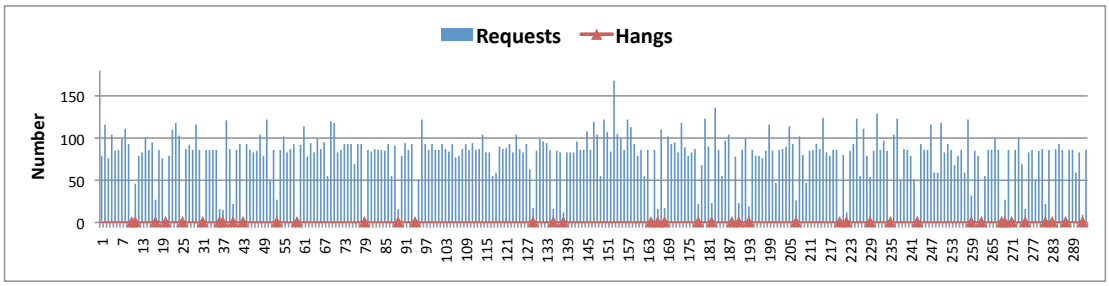
4.6.1.1 Requests and hangs

Figure 4.10 plots a histogram of the number of requests observed when executing each of the 293 mutated OpenPRS scripts on the three implementations, and indicates with a small triangle on the horizontal axis, those traces that hung before terminating.

a) *GenoM* implementation



b) *BIP-A* implementation



c) *BIP-B* implementation

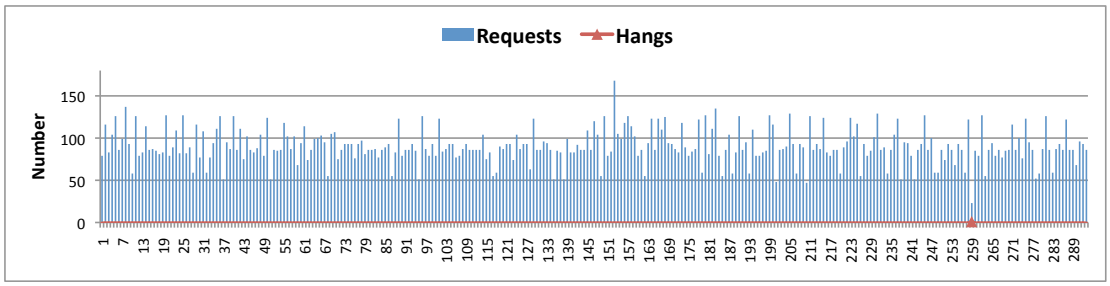


Figure 4.10: Requests and hangs per trace

We observe that the numbers of observed requests change from implementation to implementation even though the mutated scripts were identical in each case. This is a result of the *inherent non-determinism* of the system under test, which arises essentially due to non-deterministic scheduling of multiple threads of execution, amplified by the resulting divergences in interaction with the underlying (simulated) real-time system environment. This is even the case for repeated executions of a single script on a single implementation: two successive executions can differ, for example, regarding whether or not a particular robot activity needs to be interrupted to serve a higher priority request.

From the viewpoint of hung traces, we observe no hangs at all for the $G^{en}oM$ implementation, many for the *BIP-A* implementation, and a single hang for the *BIP-B* implementation, when executing script #265. When a trace hangs, there is naturally a correspondingly lower number of observed requests.

The relatively high number of hangs observed for implementation *BIP-A* underlines the fact this implementation was very immature, and considerably less robust than the original $G^{en}oM$ implementation. Details of these hung traces were fed back to the developers as pointers to necessary corrections to the implementation.

Even the mature implementation *BIP-B* is not totally free from hangs. The trace of script #265 is still being analyzed by the system developers in order to produce a future, corrected "*BIP-C*" implementation.

Table 4.6 summarizes the results obtained per trace for each of the considered implementations. In addition to the number of hung traces, the table reports the number of traces containing at least one fault negative or at least one false positive (for any property). The column "total bad traces" indicates the number of traces that either hung, or contained a false negative or positive. The final column gives the corresponding measure of *trace robustness* (T_{ROB}).

Table 4.6: Summary of per trace results

	Total traces	Hung traces	Traces with $\geqslant 1$ FN	Traces with $\geqslant 1$ FP	Total bad traces	Trace robustness (T_{ROB})
$G^{en}oM$	293		74	5	76	74.1%
BIP-A	293	42	40		80	72.7%
BIP-B	293	1	11		12	95.9%

We observe that the $G^{en}oM$ implementation gives rise to a higher number of traces with false negatives. This is hardly surprising, since the $G^{en}oM$ implementation contains no protection mechanisms to enforce properties PRE and EXC. More surprisingly, however, we note that the $G^{en}oM$ implementation displays some traces with false positives. We will detail these results in the next section.

Looking at the trace robustness figures, we observe that the $G^{en}oM$ and *BIP-A* implementations have a comparable (low) robustness, but for different reasons (false negatives/positives in the case of $G^{en}oM$, and hangs in the case of *BIP-A*). The *BIP-B* implementation appears to be a lot better from this respect, but not perfect.

4.6.1.2 Property robustness failures and analysis

We now analyze the results obtained for each of the implementations from the viewpoints of the property families PEX, AIB, PRE and EXC.

***GenoM* implementation** Figure 4.11 plots a stacked histogram of the numbers of false negatives (in red) and false positives (in blue) for each trace obtained on the $G^{en}oM$ implementation.

We observe that there are many false negatives for properties PRE and EXC. As already noted, this is to be expected, given that the $G^{en}oM$ implementation does not provide any protection mechanisms to enforce these properties.

Figure 4.11 also shows that the property responsible for the false positives observed for $G^{en}oM$ in Table 4.6 is AIB. This came as a real surprise. Section 4.2.2 defines the type of requests $dom(x)$ that should interrupt a given request type x as those that appear in a declaration "(y) incompatible_with x", in the considered module's .gen file. On closer inspection of the traces that gave rise to these false positives for AIB, we discovered that the "false" interruptions of execution requests were in fact due to requests of type init (initialization requests). After discussing with $G^{en}oM$ experts, we were told that this is a default feature of $G^{en}oM$, so there is no need for a "(y) incompatible_with x" declaration when y is of type init. Interestingly, we can observe from the $dom(x)$ column in Table 4.3 that, despite this default feature, many (but not all) requests of type init (cf. Table 4.1) were nevertheless included in .gen "(y) incompatible_with x" declarations (Table 4.3 was derived from the .gen files).

Another issue can also be seen on Figure 4.11: AIB also displays false negatives for a certain number of traces. Although expected for properties PRE and EXC, this was not the case for AIB. We will come back to this issue later.

***BIP-A* implementation** The false response histograms for the *BIP-A* implementation are given in Figure 4.12. Given that we have already noted earlier (cf. Section 4.6.1.1) that this implementation was premature, we are not too surprised to note that there are many false negatives to be seen, except for property EXC. Indeed, we observe that the protection enforcing EXC is 100% effective, since there are no false responses for this property. See, for comparison, the EXC outcomes for $G^{en}oM$ (Figure 4.11).

The same cannot be said for property PRE, for which there is a similar number of false negatives as for the $G^{en}oM$ implementation. Feeding this information back to the developers, it was discovered that the BIP connectors specifying the enforcement of PRE had been accidentally omitted from the BIP model from which implementation *BIP-A* was derived. This omission was corrected in the model underlying the revised implementation *BIP-B*.

***BIP-B* implementation** Figure 4.13 shows the false response histograms for the mature implementation *BIP-B*. As can be seen, there are much fewer false responses. Overall, there are exactly 11 of the 293 traces for which a total of 19 false negatives are reported (and no false positives). Manual inspection of these traces allowed us to determine that, in all cases, these false negative verdicts are in fact incorrect, i.e., the trace analyzer made a mistake in declaring them as such.

We take as an example, the false negative declared on property PRE for trace #137. An excerpt of the trace is shown on Figure 4.14.

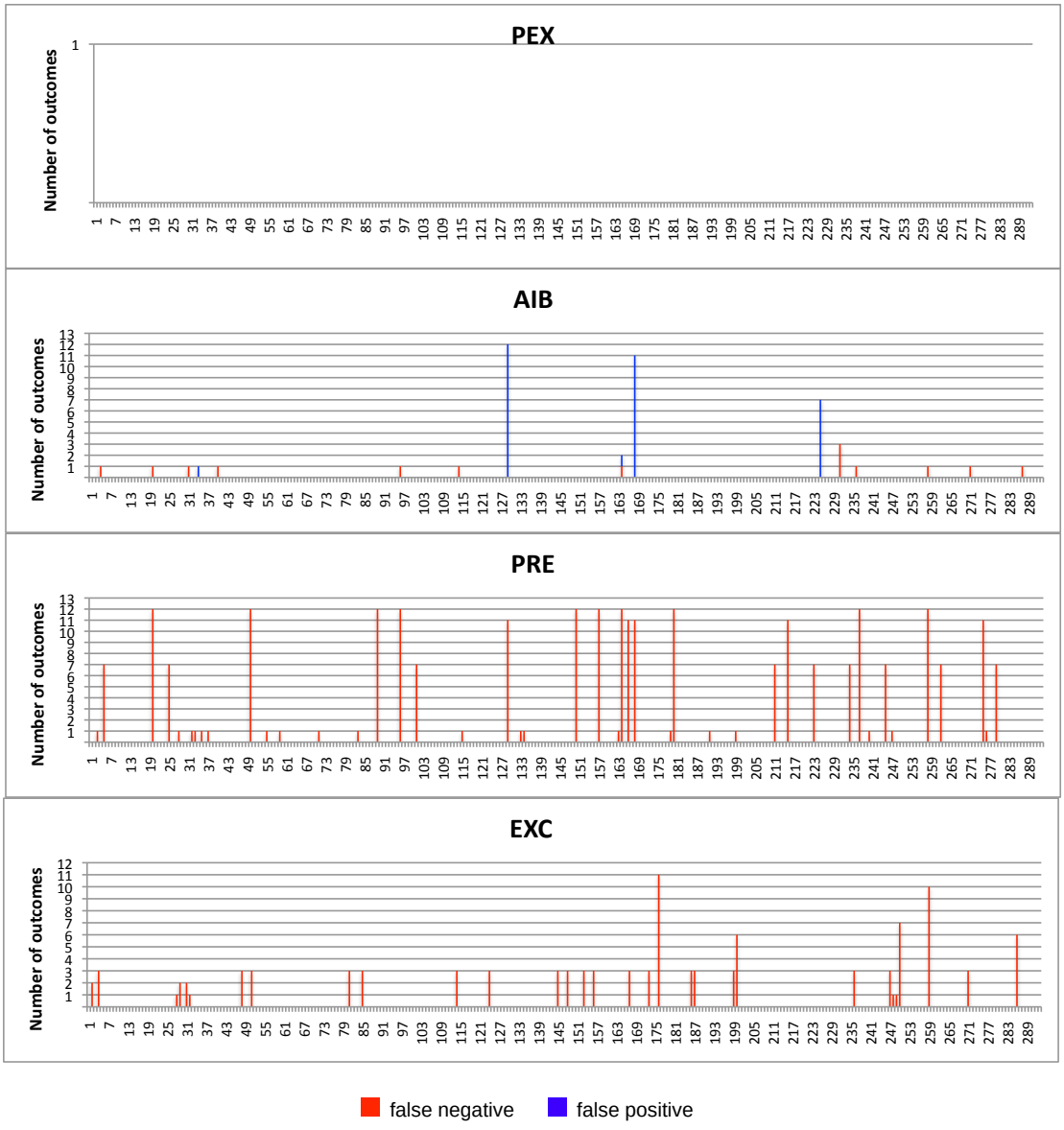


Figure 4.11: Robustness failures per trace for $G^{en}oM$ implementation

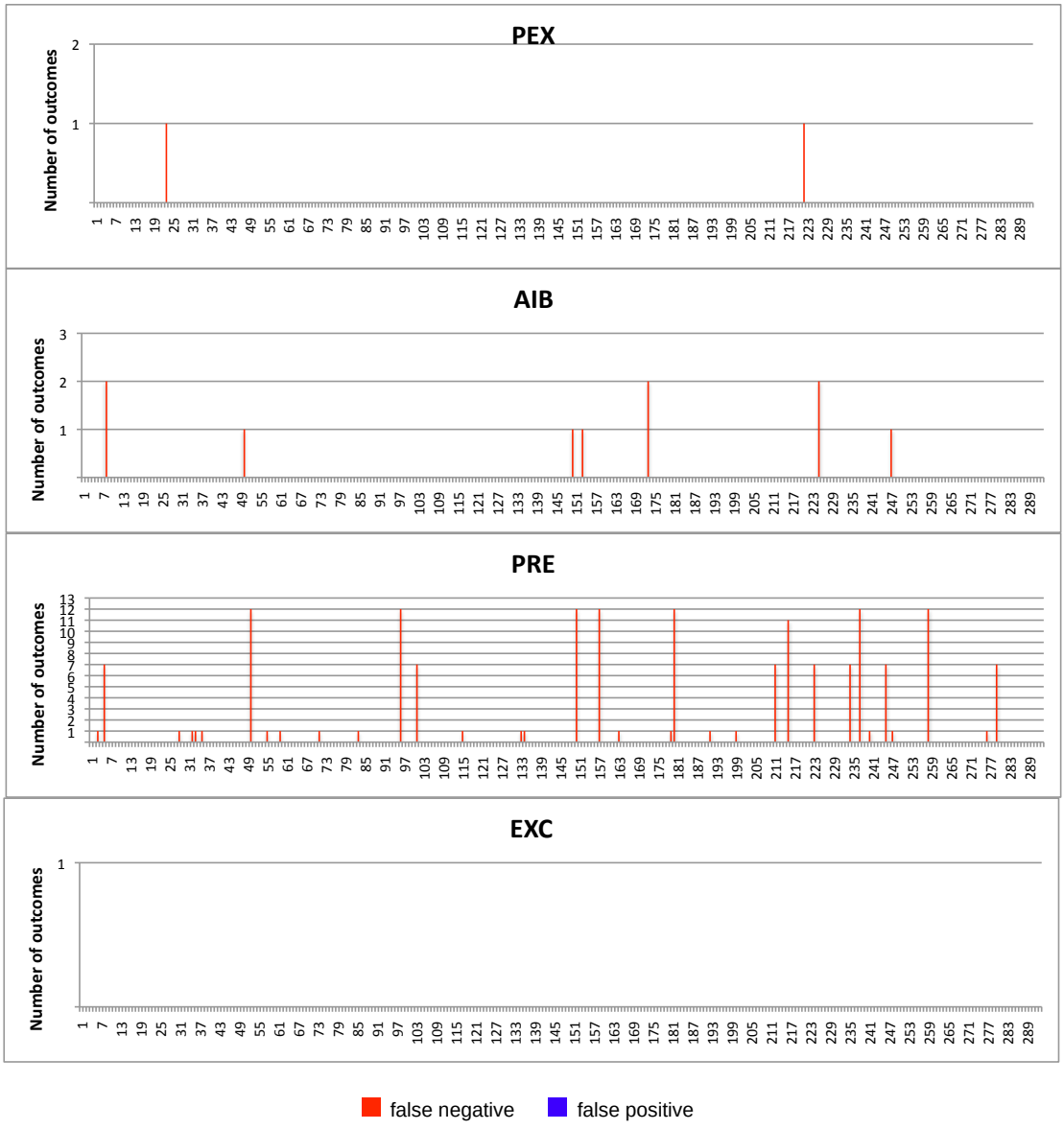


Figure 4.12: Robustness failures per trace for *BIP-A* implementation

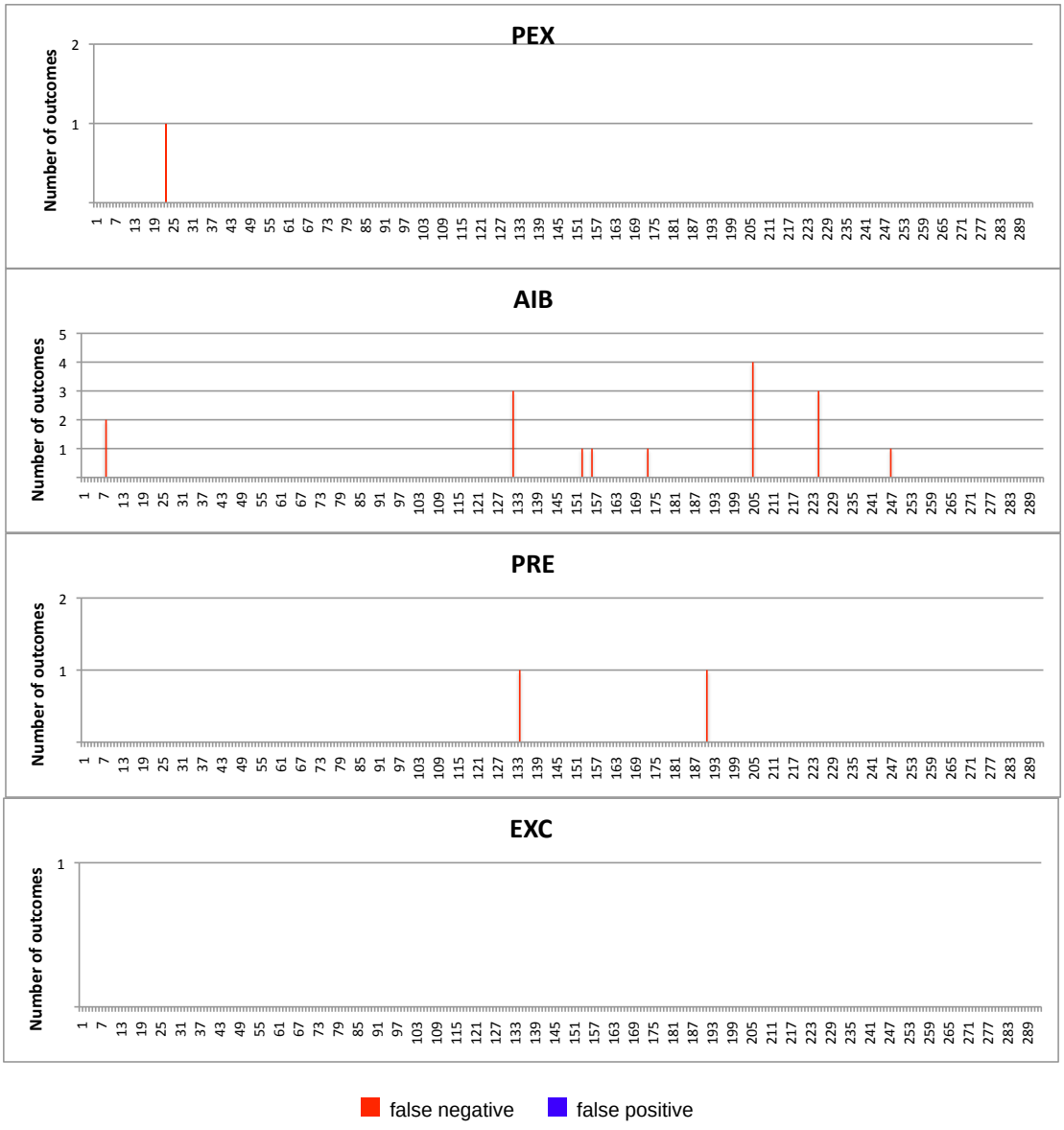


Figure 4.13: Robustness failures per trace for *BIP-B* implementation

```
1287189878.97 send   13    ASPECT_SETVIEWPARAMETERS
1287189879.07 rcv    13                ASPECT_SETVIEWPARAMETERS   OK
1287189879.17 send   14    SICK_CONTINUOUSSHOT
1287189879.27 ir     14                      SICK_CONTINUOUSSHOT
1287189879.88 rcv    14                      SICK_CONTINUOUSSHOT   OK
1287189879.98 send   15    NDD_INIT
1287189880.08 ir     15                                NDD_INIT
1287189880.18 rcv    15                                NDD_INIT   OK
1287189880.29 send   16    NDD_SETPARAMS
1287189880.39 ir     16                           NDD_SETPARAMS
1287189880.49 rcv    16                           NDD_SETPARAMS   OK
1287189880.59 send   17    NDD_SETSPEED
1287189880.69 ir     17                            NDD_SETSPEED
1287189880.79 rcv    17                            NDD_SETSPEED   OK
1287189884.10 send   18    ASPECT_SETDYNAMICSEGSSOURCE
1287189884.12 send   19    ASPECT_ASPECTFROMPOSTERCONFIG
1287189884.15 ir     18           ASPECT_SETDYNAMICSEGSSOURCE
1287189884.15 ir     19         ASPECT_ASPECTFROMPOSTERCONFIG
1287189884.21 send   20    NDD_GOTO
1287189884.24 send   21    RFLEX_TRACKSPEEDSTART
1287189884.25 rcv    18           ASPECT_SETDYNAMICSEGSSOURCE   OK
1287189884.25 ir     20                                NDD_GOTO
1287189884.25 ir     21                  RFLEX_TRACKSPEEDSTART
1287189909.09 send   22    NDD_STOP
1287189909.20 rcv    20                                NDD_GOTO   S_ndd_stdGenoM_ACTIVITY_INTERRUPTED
1287189909.20 rcv    22                                NDD_STOP   OK
1287189909.30 send   23    RFLEX_STOP
1287189909.30 send   24    RFLEX_STOP
1287189909.40 rcv    21                  RFLEX_TRACKSPEEDSTART   S_rflex_stdGenoM_ACTIVITY_INTERRUPTED
1287189909.40 rcv    23                            RFLEX_STOP   OK
1287189909.40 rcv    24                            RFLEX_STOP   OK
1287189909.52 send   25    ASPECT_STOPASPECT
1287189909.62 rcv    19         ASPECT_ASPECTFROMPOSTERCONFIG   S_aspect_stdGenoM_ACTIVITY_INTERRUPTED
1287189909.62 rcv    25                      ASPECT_STOPASPECT   OK
```

Figure 4.14: Excerpt of trace #137: incorrect false negative verdict for property EXC.

We observe that the request number 19 (ASPECT_ASPECTFROMPOSTERCONFIG) is sent at time 1287189884.12 , between request 18 (ASPECT_SETDYNAMICSEGSSOURCE) (at time 1287189884.10) and the corresponding positive final reply (at time 1287189884.25). According to the property oracle for PRE(aspect.aspectfromposterconfig) (see Table 4.4, applied to expression 3.1), we have that the *P-condition* $C_{PRE}(x(19))$ evaluates to $false$ (the *ok* appears after $x(19)$). Thus, from Table 3.2, given that $f_x(19)$ (ASPECT_ASPECTFROMPOSTERCONFIG) = S_aspect_stdGenoM_ACTIVITY_INTERRUPTED (i.e., $f_x(19) \in Z_x$) instead of the expected ASPECT_INIT_SEQUENCE_ERROR (i.e., r_P), the oracle returns a verdict of FN. The oracle gives the correct conclusion with respect to the observed trace excerpt, but is this the correct verdict?

Figure 4.15 shows that, due to the unknown delays between events on the SUT internal timeline and those observable on the SUT interface timeline (and *vice versa*), it is possible for the oracle to reach an incorrect conclusion as to the real behavior of the SUT due to event re-ordering (cf. Section 3.3.3). In this case, we can see that requests 18 and 19 were sent within 20 ms of each other, and that it must have been the case that request 19 "overtook" request 18 and was therefore processed *before* request 18. In fact, this particular trace gives further evidence to this hypothesis since it shows the immediate replies to requests 18 and 19 observed at exactly the same instant on the interface timeline[5]. We therefore conclude that the system under test did in fact react correctly (i.e., a true negative), and that the verdict

[5]As pointed out previously, the oracle cannot rely on information derived from intermediate replies since they are not sent systematically, as evidenced, e.g., for request 13 on Figure 4.14.

declared by the oracle is incorrect.

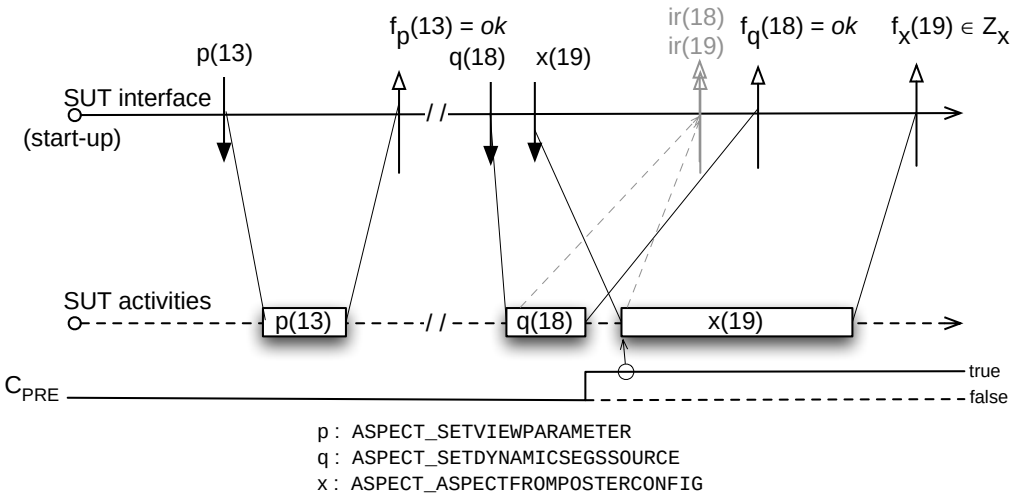


Figure 4.15: Incorrect false negative verdict on trace #137 due to event propagation delay

4.6.2 Per request results

Table 4.7 presents the sums of the verdicts obtained for the complete set of requests from all traces. The results are given for each family of properties, and overall. The tables include both the property-significant experiments (behaviors TN, TP, FN and FP) and those that are insignificant for the considered property: behaviors "other positive" (op), "non applicable" (na) and "truncated trace" (ω) (see page 49).

For the $G^{en}oM$ implementation, we can note again on Table 4.7a the high number of false negatives that are logically obtained for properties PRE and EXC since the this implementation does not provide any protection to support these properties. We also note the "false positive" anomaly for property AIB, already discussed in Section 4.6.1.2.

Similarly, for the premature *BIP-A* implementation, the absence of true positives for property PRE on Table 4.7b confirms the observation already made in Section 4.6.1.2 with respect to Figure 4.12, that the BIP connectors responsible for enforcing this had been omitted.

It is interesting to observe the figures given in columns op and na for the *BIP-B* implementation (Table 4.7c). Compared to the $G^{en}oM$ and *BIP-A* implementations, there are considerably higher numbers of "other positive" verdicts for property PEX and "non applicable" verdicts for property AIB. For property PEX, the number (292) of "other positives" (op) is matched by an approximately equal number (316) of "true positives" (TP) for property PRE, now correctly enforced in this implementation, i.e., request types that should be rejected to enforce property PEX are in fact already rejected to enforce property PRE. There is also an approximately equal additional number ($545-247=298$) of "non applicable" (na) results for property AIB in the *BIP-B* implementation (Table 4.7c) compared to that for the $G^{en}oM$ implementation (Table 4.7a).

The higher number (total of 249) of "truncated trace" (ω) verdicts for the *BIP-A* implementation (Table 4.7b), compared to 115 for $G^{en}oM$ and 127 for *BIP-B* can be explained by the considerably higher number of hung traces observed for this implementation (cf. Table

Table 4.7: Property test results

a) *GenoM* implementation

	TN	TP	FN	FP	Total	op	na	ω	Overall
PEX	12965	306			13271			17	13288
AIB	9549	4806	14	32	14401		247	50	14698
PRE	3928		258		4186	44		33	4263
EXC	2700		107		2807		76	15	2898
All	29142	5112	379	32	34665	44	323	115	35147

b) *BIP-A* implementation

	TN	TP	FN	FP	Total	op	na	ω	Overall
PEX	11629	44	2		11675			71	11746
AIB	8448	4364	10		12822	40	44	111	13017
PRE	3513		163		3676	6		34	3716
EXC	2446	87			2533		5	33	2571
All	26036	4495	175		30706	46	49	249	31050

c) *BIP-B* implementation

	TN	TP	FN	FP	Total	op	na	ω	Overall
PEX	12986	288	1		13275	292		25	13592
AIB	9399	5002	16		14417	88	545	52	15102
PRE	4039	316	2		4357	27		29	4413
EXC	2802	88			2890		75	21	2986
All	29226	5694	19		34939	407	620	127	36093

4.6).

Tables 4.8 and 4.9 show the true and false positive rates for the three implementations, calculated from the data shown in Table 4.7. We observe that the overall true positive rate grows, over the successive implementations, due to the additional property enforcement made possible by the BIP approach, and the correction carried out in the *BIP-B* implementation regarding the enforcement of property *PRE*. For this implementation, we would in fact have 100% true positive rates for all properties in the *BIP-B* implementation, if we were to manually correct the data presented in Table 4.7 to take account of the 19 false negative verdicts that were revealed to be incorrect by manual inspection of the traces (cf. Section 4.6.1.2).

The only non-null cell in Table 4.9 is that for the AIB property in implementation $G^{en}oM$. As previously reported (page 86), this is due to an undocumented feature of the GenoM implementation regarding the fact that all execution requests are interrupted by initialization requests, leading to the AIB property oracle to declare such interruptions as false positives. It is interesting to notre that there are no such false positives for the BIP implementations. This can be explained by the fact that the developers of the BIP implementations did not implement this undocumented "default" feature!

Table 4.8: True positive rates (%)

	$G^{en}oM$	*BIP-A*	*BIP-B*
PEX	100.0	95.7	99. 7
AIB	99.7	99.8	99.7
PRE	0	0	99.4
EXC	0	100.0	100.0
All	93.1	96.3	99.7

Table 4.9: False positive rates (%)

	$G^{en}oM$	*BIP-A*	*BIP-B*
PEX	0	0	0
AIB	0.3	0	0
PRE	0	0	0
EXC	0	0	0
All	0.1	0	0

4.7 Conclusion

We presented in this chapter an implementation of our robustness testing framework and its application to the Dala planetary rover case study. The realisation of this evaluation campaign is a complicated process, especially in the autonomous system context where the system behaviours, even in the absence of faults, can be non-deterministic.

The experiments that we carried out in this case study allowed us to test and evaluate the

robustness, in an objective way, of three implementations of the Dala functional layer : a mature $G^{en}oM$ implementation and two successive BIP implementations (*BIP-A* and *BIP-B*). The test results show that the BIP approach clearly improves the timing robustness of the Dala functional layer. The *BIP-B* implementation provides 100% property robustness (after manual correction of incorrect automatic test verdicts), as is to be expected for a "correct-by-construction" development approach. Nevertheless, testing proved to be a useful complement to the formal development approach. The robustness testing campaign to thoroughly explore and understand the different behaviours of the SUT. Indeed, our testing revealed an undocumented feature of $G^{en}oM$ implementation (see Section 4.6.1.2) and a remaining defect in the mature *BIP-B* implementation (giving rise to a hung trace).

From a practical viewpoint, implementation and exploitation of the robustness testing environment were severely hampered by the unreliability of the robot simulator, and in particular the GenoM-to-Gazebo software bridge Pocosim, a research prototype developed in a previous project. On a more positive front, implementation of the trace analyzer as a set of SQL queries on XML-structured trace data was a very astute design decision, which proved to be very flexible while iteratively developing and correcting the property oracles.

The data of the experiments (mutated scripts, execution traces, statistical results and the trace analyzer tool) can be found online at `http://www2.laas.fr/TSF/archives/2011/hnchu_thesis.gz`

Chapter 5

Conclusion and Future Work

Constructing dependable systems has always been a challenge for engineers. With the growing demand for autonomous systems, it is becoming increasingly important for them to be built with demonstrable dependability, especially with respect to safety constraints that prohibit inconsistent or dangerous behaviours. Indeed, the violation of safety constraints in critical autonomous systems may be catastrophic in human or economic terms. Therefore, it is essential to implement effective enforcement of safety constraints and to provide convincing evidence of their correct implementation. The work presented in this thesis is a contribution towards the satisfaction of this need.

In this thesis, we have summarized key notions from the fields of dependable computing and autonomous systems, and provided a state of the art of recent work in robustness testing. We identified two types of robustness testing: *input robustness testing* and *load robustness testing*. We also classified input robustness testing techniques into three categories: input-domain model-based approaches, behaviour model-based approaches and hybrid approaches. The main contribution of this thesis is the definition of a hybrid robustness testing approach that is a combination of random fault-injection and passive testing. We defined a framework for testing the robustness of the built-in defenses of a system with respect to untimely asynchronous inputs. To the best of our knowledge, this problem has not been previously addressed.

We have proposed a method and a platform for testing the robustness of the safety-property enforcing mechanisms of the functional layer of a hierarchically-structured autonomous system. The application to the Dala rover shows several advantages of our method. The adoption of a black-box testing approach allowed us to carry out the testing campaign without any formal behaviour specification of the system under test (SUT). With little or no information about the internal activities or states of the SUT, we were still able to compare the effectiveness of the safety enforcement mechanisms of two different implementations of the functional layer based on a behaviour categorisation with respect to safety properties. We think that this approach is appropriate for testing off-the-shelf components, for which a formal behaviour specification is usually not available.

Using the passive testing technique enabled us to evaluate the robustness of the SUT based on offline observation of logged test execution traces. In our case study, the whole testing process is composed of two phases: exercising the SUT with 293 test cases (which takes around 25 hours) and examining the execution results with the oracle (which takes about 30 minutes). The passive testing technique applied to logged execution traces separates the observation process from the system activation process, and thus avoids us having to re-run

the whole test set (25 hours) each time we want to refine the definition of the test oracle, for example, to add an additional property.

We have attempted to define properties that are as generic as possible. To this end, we defined five basic safety properties, along with their enforcement policies, that can be instantiated as timing robustness requirements of the functional layer of an autonomous system: pre-condition, excluded start, excluded execution, (asymmetric) exclusion, and mutual exclusion. We believe that they are sufficiently general to allow their application to other systems. For each property type, we have defined the corresponding input timing robustness testing oracle.

We have also developed and presented a testing environment that allows us to evaluate the timing robustness of the functional layer of the Dala planetary exploration rover by subjecting it to invalid inputs in the time domain. Starting from a workload (a typical mission of a planetary explorer) described in a script, we put stress on the SUT by creating mutated scripts containing inputs submitted at the "wrong time". Simulation of the physical hardware of the robot and its environment facilitates our intensive testing process and ensures that injected faults can only cause "virtual" damage. However, simulation cannot totally replace testing (albeit without injected faults) on the real platform, which can reveal phenomena that are hard to simulate faithfully (e.g., due to real-time issues, or hard-to-model sensor and actuator inaccuracies).

The implementation of the oracle as a set of SQL queries proved to be very flexible and easy to maintain. The evaluation environment showed its efficacy in comparing and evaluating different systems. Indeed, thanks to the ability to explore thoroughly the SUT's reaction to untimely inputs, our approach to robustness testing allows both fault removal (debugging) by studying the consequences of fault injection, and fault forecasting (evaluation) through statistical measures of system behaviour with respect to fault occurrence.

However, our approach does present certain limitations.

As we mentioned in Section 3.4, testing without the formal behaviour specification of the SUT may give rise to an inaccurate oracle, and testers thus have to progressively improve it manually. The question is how? In the hybrid robustness testing approach proposed by *Cavalli et al.* [CMM08], the authors verified the correctness of their invariant properties by checking them against a formal model of the system behaviour before using them as a robustness testing oracle. This approach could not be applied in our context since we did not have any document that could serve as an authoritative specification of the implementations being compared. We thus had to manually analyse the results produced by the oracle to identify singularities (e.g., too many *False Positives* or *False Negatives*), and then examine the execution trace to diagnose the source of the problem (oracle inaccuracy or SUT misbehaviour). In effect, the oracle and the SUT are tested back-to-back and iteratively corrected. We were aided in this respect by the fact that we had two distinct SUTs, one of which ($G^{en}oM$) was a mature implementation (at least with respect to a subset of the required safety properties).

Another limitation is the possibility of incorrect test verdicts due to false observations of *P-conditions*, which are inevitable due to the fact that we cannot control the propagation time of events in the SUT (c.f. Figure 4.15). In the Dala rover case study, our analysis concluded that all false negatives observed on the *BIP-C* implementation in fact corresponded to incorrect verdicts, i.e., true negative situations that were declared erroneously to be false negatives. In each case there was a plausible explanation of how correct behavior of the SUT (i.e., *true positive* or *true negative* behaviour) could be wrongly interpreted as incorrect behaviour due to uncertain propagation delays.

The inverse is also possible, i.e., misinterpretation of incorrect behaviour as correct behaviour. Unfortunately, such misinterpretations cannot, by essence, be identified since the observed behaviour presents no singularities (it is the expected behaviour), so there is no reason to bring it into doubt. This is especially problematic in the case of misinterpreting a false negative as a true negative, as that would be optimistic from a safety viewpoint. Thus, some finer degree of SUT observation is likely to be necessary for testing the robustness of extremely critical systems.

The work presented in this thesis opens up several directions for future research.

One area for improvement would be to reduce the number of incorrect test verdicts raised by black-box robustness testing, and the associated observability issues. At least two complementary directions can be considered:

1. Include explicit consideration of real-time in the property oracles to flag verdicts on closely-separated events as "suspicious". In our case study, only in the *BIP-B* implementation were there sufficiently few false negatives in order to justify a tedious manual inspection of execution traces.

2. Study possible modifications or extensions to the interface protocol to facilitate robustness testing (for example, by requiring "intermediate responses" to be sent systematically).

Another area for improvement is in test generation. In particular, the generation of test inputs could definitely be improved in order to make it more automatic. For example, it should be possible to adapt an automatic program mutation tool, such as SESAME [CWLP06], to automatically inject faults into a golden test script written in the Open-PRS format. Alternatively, a more deterministic generation of test scripts could be envisaged, focussing on the falsification of the considered P-conditions.

It would also be interesting (and relatively straightforward) to extend the proposed hybrid input timing robustness testing approach to include classic value domain robustness properties.

More generally, an interesting area for future research would be to provide guidelines to the system designer on the choice and definition of the properties to be enforced. One aspect of such guidelines could be on how to derive the required properties from a risk analysis of the considered application. Another aspect could be the definition of "composable" properties, i.e., properties that can be built from various combinations of other properties. The concept of dividing a complex property into various simpler ones will make properties flexible and extendable, as well as facilitate the processes of property definition and oracle implementation.

Bibliography

[AAA+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. on Software Engineering*, 16:166–182, 1990.

[ACF+98] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17:315–337, 1998.

[ACMR03] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Eric Rutten. Using controller-synthesis techniques to build property-enforcing layers. In Pierpaolo Degano, editor, *Programming Languages and Systems*, volume 2618 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-36575-3_13.

[AFRS02] Jean Arlat, Jean-Charles Fabre, Manuel Rodriguez, and Frédéric Salles. Dependability of COTS Microkernel-Based systems. *IEEE Transactions on Computers*, 51:138–163, 2002.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[BBS06] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *4th IEEE Int'l Conf. on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, ATVA '08, pages 64–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[BCME+98] Jean-Jacques Borrelly, Eve Coste-Manière, Bernard Espiau, Konstantinos Kapellos, Roger Pissard-Gibollet, Daniel Simon, and Nicolas Turro. The ORCCAD architecture. *The International Journal of Robotics Research*, 17(4):338–359, 1998.

[BdSG+10] S. Bensalem, L. de Silva, M. Gallien, F. Ingrand, and R. Yan. "Rock solid" software: A verifiable and correct-by-construction controller for rover and spacecraft functional level. In *i-SAIRAS 2010, The 10th International Symposium*

on Artificial Intelligence, Robotics and Automation in Space, Sapporo, Japan, 2010.

[Bei95] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[Bro86] R. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14 – 23, March 1986.

[CAI+09] H.-N. Chu, J. Arlat, F. Ingrand, M.-O. Killijian, B. Lussier, and D. Powell. Évaluation de la robustesse d'une architecture de contrôle de robot. *FNRAE projet MARAE, Rapport Technique RT5*, 2009.

[CAK+09] H.-N. Chu, J. Arlat, M.-O. Killijian, B. Lussier, and D. Powell. Robustness evaluation of robot controller software. In *European Workshop on Dependable Computing (EWDC-9)*, Toulouse, France, 2009. LAAS-CNRS.

[Car96] Geogre J. Carrette. CRASHME: random input testing, http://people.delphi.com/gjc/crashme.html, 1996.

[Clo02] Bruce T Clough. Metrics, schmetrics! how the heck do you determine a UAV's autonomy anyway. Technical report, http://www.dtic.mil/dtic/tr/fulltext/u2/a515926.pdf, August, 2002.

[CM03] J. Carlson and R. R Murphy. Reliability analysis of mobile robots. In *IEEE International Conference on Robotics and Automation, 2003. ICRA '03*, volume 1, pages 274 – 281 vol.1, September 2003.

[CM05] J. Carlson and R.R. Murphy. How UGVs physically fail in the field. *IEEE Transactions on Robotics*, 21(3):423 – 437, 2005.

[CMM08] Ana Cavalli, Eliane Martins, and Anderson Morais. Use of invariant properties to evaluate the results of fault-injection-based robustness testing of protocol implementations. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 21–30, Washington, DC, USA, 2008. IEEE Computer Society. ACM ID: 1440217.

[CS92] E.W. Czeck and D.P. Siewiorek. Observations on the effects of fault manifestation as a function of workload. *IEEE Transactions on Computers*, 41(5):559–566, 1992.

[CWLP06] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell. The SESAME experience: from assembly languages to declarative models. In *Second Workshop on Mutation Analysis, 2006.*, November 2006.

[DCY05] Z. Duan, Z. Cai, and J. Yu. Fault diagnosis and fault tolerant control for wheeled mobile robots under unknown environments: A survey. In *IEEE Int'l Conf. on Robotics and Automation*, pages 3428–3433, Barcelona, Spain, 2005.

[DGDLC10] B. Durand, K. Godary-Dejean, L. Lapierre, and D. Crestani. Global methodology in control architecture to improve mobile robot reliability. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2010*, pages 1018 –11023, 2010.

[FGLS06] M. Fox, A. Gerevini, D. Long, and I. Serina. Plan stability: Replanning versus plan repair. In D. Long, S. F. Smith, D. Borrajo, and L. McCluskey, editors, *International Conference on Automated Planning and Scheduling*, pages 212–221. AAAI, 2006.

[FHC97] S. Fleury, M. Herrb, and R. Chatila. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, 1997. IROS '97.*, volume 2, page 842–849 vol.2, 1997.

[FM00] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *4th conference on USENIX Windows Systems Symposium - Volume 4*, pages 59–68, Berkeley, CA, USA, 2000. USENIX Association.

[FMP05] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. A model-based approach for robustness testing. In Ferhat Khendek and Rachida Dssouli, editors, *Testing of Communicating Systems*, volume 3502 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin / Heidelberg, 2005. 10.1007/11430230_23.

[FPS01] Hacene Fouchal, Eric Petitjean, and Sébastien Salva. An user-oriented testing of real time systems. *In: Proceedings of the International Workshop on Real-time Embeded Systems RTES'01 (London), IEEE Computer Society*, 2001.

[FRAS00] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun. Building dependable COTS microkernel-based systems using MAFALDA. In *Pacific Rim International Symposium on Dependable Computing, 2000.*, pages 85–92, 2000.

[FRT05] Hacène Fouchal, Antoine Rollet, and Abbas Tarhini. Robustness of composed timed systems. In *SOFSEM 2005: Theory and Practice of Computer Science*, pages 157–166. 2005.

[Gat97] E. Gat. On three-layer architectures. In D. Kortenkamp, R.P. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. MIT/AAAI Press, 1997.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development 915*, pages 82–96. Springer Berlin / Heidelberg, 1995.

[GDRS00] P. Goel, G. Dedeoglu, S.I. Roumeliotis, and G.S. Sukhatme. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *IEEE International Conference on Robotics and Automation, 2000. ICRA '00.*, pages 2302 –2309 vol.3, 2000.

[GL94] M. Ghallab and H. Laruelle. Representation and control in IxTeT, a temporal planner. In *2nd Int. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*, pages 61–67, Chicago, IL, USA, 1994. AIAA Press.

[GPBB08] J. Guiochet, D. Powell, E. Baudin, and J.-P. Blanquart. Online safety monitoring using safety modes. In *6th IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, Pasadena, CA, USA, 2008.

[Gra10] A. Graeser. Ambient intelligence and rehabilitation robots - a necessary symbiosis for robust operation in unstructured environments. In *9th International Symposium on Electronics and Telecommunications (ISETC), 2010.*, pages 9 –16, nov. 2010.

[HKSW07] Michael Hofbaur, Johannes Köb, Gerald Steinbauer, and Franz Wotawa. Improving robustness of mobile robots using model-based reasoning. *Journal of Intelligent & Robotic Systems*, 48:37–54, 2007.

[Hua07] Hui-Min Huang. Autonomy levels for unmanned systems (ALFUS) framework: safety and application issues. *Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems*, page 48–53, 2007. ACM ID: 1660883.

[ICA01] F. Ingrand, R. Chatila, and R. Alami. An architecture for dependable autonomous robots. In *8th IEEE International Conference on Emerging Technologies and Factory Automation, 2001.*, volume 2, pages 657 –658, 2001.

[ICAR96] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *IEEE Int. Conf. on Robotics and Automation*, pages 43–49, Minneapolis, MN, USA, 1996.

[IEE90] IEEE. IEEE standard glossary of software engineering terminology, 1990.

[IG03] K. Ito and A. Gofuku. Emergence of adaptive behaviors by redundant robots - robustness to changes environment and failures. In *The Congress on Evolutionary Computation, 2003. CEC '03.*, volume 4, pages 2572 – 2579 Vol.4, dec. 2003.

[IRH86] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Transactions on Computer Systems*, 4(3):214–237, 1986.

[JALL05] Sylvain Joyeux, Rachid Alami, Simon Lacroix, and Alexandre Lampe. Simulation in the LAAS architecture. In *Principles and Practice of Software Development in Robotics (SDIR2005)*, Barcelona, Spain, 2005.

[Jor95] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1995.

[KCK+05] Karama Kanoun, Yves Crouzet, Ali Kalakech, Ana-Elena Rugina, and Philippe Rumeau. Benchmarking the dependability of Windows and Linux using PostMark™ workloads. In *16th IEEE Int. Symp. on Software Reliability Engineering*, pages 11–20, 2005.

[KD99] Philip Koopman and John DeVale. Comparing the robustness of posix operating systems. In *29 International Symposium on Fault-Tolerant Computing (FTCS - 29)*, FTCS '99, pages 30–38, Washington, DC, USA, 1999. IEEE Computer Society.

[KD00] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Transactions on Software Engineering*, 26:837–848, 2000.

[KKS98] N. P Kropp, P. J Koopman, and D. P Siewiorek. Automated robustness testing of off-the-shelf software components. *28th IEEE International Symposium on Fault-Tolerant Computing, 1998.*, pages 230–239, 1998.

[KSD+97] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz. Comparing operating systems using robustness benchmarks. *16th IEEE Symposium on Reliable Distributed Systems, 1997.*, pages 72–79, 1997.

[LAB+96] Jean-Claude Laprie, Jean Arlat, Jean-Paul Blanquart, A. Costes, Yves Crouzet, Yves Deswarte, Jean-Charles Fabre, H. Guillermain, Mohamed Kaâniche, Karama Kanoun, C. Mazet, David Powell, C. Rabéjac, and P. Thévenod. *Guide de la Sûreté de Fonctionnement.* Cépaduès edition, 1996.

[LC04] S. Lemai-Chenevier. *IxTeT-eXeC : planification, réparation de plan et contrôle d'exécution avec gestion du temps et des ressources.* PhD thesis, Institut National Polytechnique de Toulouse, France, 2004.

[LCH+02] David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu, and Xia Yin. A formal approach for passive testing of protocol data portions. In *IEEE International Conference on Network Protocols*, pages 122–132, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[LGG+07a] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell. Fault tolerant planning for critical robots. In *37th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2007)*, pages 144–153, Edinburgh, UK, 2007. IEEE CS Press.

[LGG+07b] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell. Planning with diversified models for fault-tolerant robots. In *17th. Int. Conf. on Automated Planning and Scheduling (ICAPS)*, pages 216–223, Providence, Rhode Island, USA, 2007. AAAI.

[LLC+05] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell. Fault tolerance in autonomous systems: How and how much? In *4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, Nagoya, Japan, 2005.

[Lus07] Benjamin Lussier. *Tolérance aux fautes dans les systèmes autonomes.* PhD thesis, Institut National Polytechnique de Toulouse, France, 131 pages, 2007.

[MAM+00] Alexander Meystel, J. Albus, E. Messina, J. Evans, and D. Fogel. Measuring performance of systems with autonomy: Metrics for intelligence of constructed systems. In *White Paper for the Workshop on Performance Metrics for Intelligent Systems. NIST*, Gaithesburg, Maryland, August 2000.

[MDF+02] Nicola Muscettola, Gregory A. Dorais, Chuck Fry, Richard Levinson, and Christian Plaunt. IDEA: planning at the core of autonomous reactive agents. In *in Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[MDS93] David J Musliner, Edmund H Durfee, and Kang G Shin. CIRCA: a cooperative intelligent Real-Time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, 23:1561–1574, 1993.

[MKP+95] Barton P. Miller, David Koski, Cjin Pheow, Lee Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin, USA, 1995.

[MNPW98] N. Muscettola, P.P. Nayak, B. Pell, and B.C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

[MRMC98] Eric Marchand, Eric Rutten, Hervé Marchand, and Francois Chaumette. Specifying and verifying active vision-based robotic systems with the SIGNAL environment. *The International Journal of Robotics Research*, 17(4):418–432, 1998.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, April 1989.

[MW88] J.F. Meyer and L. Wei. Analysis of workload influence on dependability. In *18th International Symposium on Fault-Tolerant Computing, 1988. FTCS-18*, pages 84–89, 1988.

[OKWK09] T. Ohkubo, K. Kobayashi, K. Watanabe, and Y. Kurihara. Development of safety system for electric wheelchair with thermography camera. In *ICCAS-SICE, 2009*, pages 1629–1632, 2009.

[PI04] F. Py and F. Ingrand. Dependable execution control for autonomous robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS2004)*, pages 1136–1141, Sendai, Japan, 2004. IEEE.

[PKS+01] Jiantao Pan, Philip Koopman, Daniel Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness testing and hardening of CORBA ORB implementations. pages 141–150, 2001.

[Py04] Frédéric Py. *Contrôle d'exécution dans une architecture hiérarchisée pour systèmes autonomes*. PhD thesis, Université Paul Sabatier de Toulouse, France, 145 pages, 2004.

[Rol03] Antoine Rollet. Testing robustness of real time embedded systems. In *Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of Formal Methods (FM) 2003 Symposium*, Pisa, Italy, September 2003.

[RRAA04] S. Roderick, B. Roberts, E. Atkins, and D. Akin. The ranger robotic satellite servicer and its autonomous software-based safety system. *IEEE Intelligent Systems*, 19(5):12–16, 2004.

[RSFA99] Manuel Rodríguez, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. Mafalda: Microkernel assessment by fault injection and design aid. In *Proceedings of the Third European Dependable Computing Conference*, EDCC-3, pages 143–160, London, UK, 1999. Springer-Verlag.

[Saa06] Fares Saad-Khorchef. *Cadre Formel pour le Test de Robustesse des Protocoles de Communication*. PhD thesis, Université Sciences et Technologies - Bordeaux I, 2006.

[SCPCW98] Stanley A Schneider, Vincent W Chen, Gerardo Pardo-Castellote, and Howard H Wang. ControlShell: a software architecture for complex electromechanical systems. *The International Journal of Robotics Research*, 17(4):360–380, 1998.

[SKD00] C.P. Shelton, P. Koopman, and K. DeVale. Robustness testing of the microsoft win32 API. In *30th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN 2000)*, pages 261–270, New York, NY, USA, 2000.

[SKRC07] Fares Saad-Khorchef, Antoine Rollet, and Richard Castanet. A framework and a tool for robustness testing of communicating software. In *2007 ACM symposium on Applied computing*, SAC '07, pages 1461–1466, New York, NY, USA, 2007. ACM. ACM ID: 1244315.

[Soc04] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): 2004 Version*. IEEE Computer Society Press, March 2004.

[SRFA99] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. MetaKernels and fault containment wrappers. In *29th International Symposium on Fault-Tolerant Computing, 1999.*, pages 22–29, 1999.

[Tre96] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[TRF05] A. Tarhini, A. Rollet, and H. Fouchal. A pragmatic approach for testing robustness on real-time component based systems. In *ACS/IEEE International Conference on Computer Systems and Applications*, page 143, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[TTP+03] N. Tomatis, G. Terrien, R. Piguet, D. Burnier, S. Bouabdallah, K. O Arras, and R. Siegwart. Designing a secure and robust mobile interacting robot for the long term. In *IEEE International Conference on Robotics and Automation, 2003. (ICRA '03)*, volume 3, pages 4246 – 4251, September 2003.

[VNE+01] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, pages 121–132, 2001.

[Voa97] Jeffrey Voas. Fault injection for the masses. *Computer*, 30:129–130, 1997.

[Wel94] D.S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Wika] Wikipedia. Load testing. http://en.wikipedia.org/wiki/Load_testing, accesed 29/8/2011.

[Wikb] Wikipedia. Negative test. http://en.wikipedia.org/wiki/Negative_test, accesed 29/8/2011.

[Wikc] Wikipedia. Soak testing. http://en.wikipedia.org/wiki/Soak_testing, accesed 29/8/2011.

[WTF99] H. Waeselynck and P. Thévenod-Fosse. A case study in statistical testing of reusable concurrent objects. In *3rd European Dependable Computing Conference (EDDC-3)*, volume LNCS no. 1667, pages 401–418, Prague, Czech Republic, 1999.

[YHY09] Jin Yaochu, Guo Hongliang, and Meng Yan. Robustness analysis and failure recovery of a bio-inspired self-organizing multi-robot system. *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 0:154–164, 2009.