Strengthening fonctional validation of critical system by using Model Checking : application to Instrumentation and control systems in nuclear power plants

Yanjun Sun

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2017-ENST-0047

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Yanjun SUN

le 9 octobre 2017

Strengthening fonctional validation of critical system by using Model Checking : Application to Instrumentation and Control systems in nuclear power plants

Directeur de thèse : **Gérard MEMMI**
Co-encadrement de la thèse : **Sylvie VIGNES**

Jury

Mme Lydie DU BOUSQUET, Professeur, Université Grenoble-Alpes, LIG	Rapporteur
M. Alessandro FANTECHI, Professeur, Université de Florence, Italie	Rapporteur
M. Luc COYETTE, Esterel technologies/ ANSYS	Examinateur
M. Frédéric DAUMAS, Expert Sûreté de Fonctionnement CC, EDF R&D	Examinateur
M. Elie NAJM, Professeur, Télécom ParisTech	Président du jury
M. Gérard MEMMI, Professeur, Télécom ParisTech	Directeur de thèse
Mme Sylvie VIGNES, Maître de conférence, Télécom ParisTech	Directeur de thèse

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

T H È S E

Acknowledgment

I would like to thank my thesis supervisors Gérard Memmi and Sylvie Vignes for their precious guidance and support during the past four years. I truly could not have imagined having better mentors than them.

I would like to thank the members of my committee for their insightful comments and encouragement: Lydie du Bousquet, Alessandro Fantechi and Elie Najm.

My thanks also go to "CONNEXION" colleagues: Catherine Devic, Frédéric Daumas, Maxime Neyret, Gaëtan Robin from EDF; Luc Coyette and François-Xavier Dormoy from ESTEREL Technologies; Benjamin Blanc, Jean-Yves Pierron from CEA; François Chastrette from ALL4Tech; Olivier Bruneau from CORYS. This thesis is partially funded by the "CONNEXION" project. A special gratitude to Adrien Champion for his valuable help and advice.

I am grateful to other colleagues at École Télécom ParisTech for their advice and friendship. I am also grateful to school staff for their unfailing assistance.

Finally I would like to thank my parents, my parents-in-law, my husband and my son, for their love and support.

Abstract

The verification and validation of safety-critical real-time system are subject to stringent standards and certifications. Recent progress in model-based system engineering should be applied to such systems since it allows early detection of defects and formal verification techniques. The French project "CONNEXION" proposes an innovative process introducing multi-level model-based functional validation supporting the nuclear I&C system development life cycle. Various modeling and verification tools provided by project partners automate this process as much as possible. "CONNEXION" also proposes a verification platform integrating various models and tools, which allows closed-loop simulation of the control/command system and its physical environment.

As part of the "CONNEXION" project, this thesis proposes a model-based testing methodology dedicated to functional validation of safety-critical real-time systems. The methodology focuses on an iterative use of a model checker to generate coverage-based open-loop test sequence. We also propose a refinement technique of progressively adding environment constraints during test generation. The refinement is expected to support the passage from coverage-based open-loop test sequence to functional requirements-based closed-loop test case. Our methodology also considers the state explosion problem of a model checker and proposes a heuristic called hybrid verification which combines model checking and simulation. Finally we design an information system of traceability to support the co-simulation verification platform.

Part of our methodology has been tested on a "CONNEXION" case study, with support of "CONNEXION" tools. To implement the complete methodology, in particular hybrid verification, we have tested several academic model checking tools for the synchronous data-flow language Lustre (choice of "CONNEXION"). We review state of the art of Lustre regarding version evolution, related model checking tools, translator of Lustre to other modeling languages, etc. For Lustre-based models, implementation of the complete methodology may require an integration of several model checking tools.

Résumé français

Introduction

Contrôle de processus industriels, contrôle de véhicules (automobiles, trains, avions. . .), systèmes militaires de contrôle-commande..., voici quelques exemples de systèmes réactifs temps réel. Le terme "réactif" indique que le système interagit en permanence avec son (éventuellement physique) environnement; nous réservons le terme "temps réel" pour les systèmes réactifs soumis à des contraintes temporelles externes, comme suggéré par Benveniste et Berry[26]. Les systèmes hautement critiques en sûreté de fonctionnement sont en général des systèmes réactifs temps réel, tels que le système de contrôle-commande dans une centrale nucléaire ou le système de contrôle de vol de l'avion. Ce domaine d'application nécessite une conception très soignée et une vérification ainsi qu'une validation très rigoureuses. La sûreté de fonctionnement est une propriété cruciale ici car un simple défaut peut produire des conséquences extrêmes et catastrophiques.

Historiquement, la conception des systèmes réactifs temps réel a longtemps été la préoccupation des automaticiens. La théorie du contrôle automatique a été appliquée à la conception des systèmes dont le comportement est attendu, d'une manière dite "bottom-up". Au niveau de Vérification et Validation (V&V), la tâche ne peut être approchée que tardivement dans le cycle de développement. Prototypes physiques des systèmes (ou sous-systèmes) sont connectés à la plateforme de simulation "Hardware-in-the-Loop (HIL)", pour étudier le comportement au niveau du système. Pour les système unitaires avec un nombre limité des variables d'entrée et de sortie, il est possible de les tester de manière exhaustive. Mais le défi reste à effectuer une validation complète et systématique, étant donné le grand nombre de situations pertinentes pour les systèmes complexes [160].

Au cours des décennies précédentes, des contributions majeures ont été faites pour établir des approches dites "basées sur modèles" (model-based en anglais) pour la description, la conception et l'analyse de systèmes [71]. Divers langages de modélisation graphique ont été développés, afin de rendre les modèles plus visuels

et de faciliter la communication avec les parties prenantes. En 2007, l'INCOSE (International Council on Systems Engineering) a lancé l'initiative "INCOSE MBSE"[1], où MBSE indique **l'Ingénierie dirigée par les modèles** (Model-Based Systems Engineering). Depuis les deux dernières décennies, divers méthodologies et outils MBSE ont été développés. Un résumé peut être trouvé dans l'enquête INCOSE portant sur les méthodologies MBSE [79].

Definition 1. *Ingénierie des systèmes dirigée par les modèles (MBSE) est l'application formalisée de la modélisation pour soutenir les exigences, la conception, l'analyse, la vérification et validation du système, à partir de la phase de conception et tout au long des phases de développement et de cycle de vie ultérieur [7].*

En bref, MBSE donne aux modèles un rôle central dans le processus de l'ingénierie de la spécification à la conception, l'intégration et la validation d'un système [79]. Les approches MBSE comprennent l'analyse comportementale, l'architecture du système, la traçabilité des exigences, l'analyse de la performance, la simulation, les tests, etc. Cela résulte en une transition de l'ingénierie système traditionnellement "documents-centriques" vers une approche "modèles-centriques" préconisée par l'INCOSE[7]. De plus MBSE permet aux ingénieurs logiciels et systèmes de mieux comprendre l'impact de changement de conception, de communiquer l'intention de conception et d'analyser la conception d'un système avant le développement.

L'application des progrès récents en MBSE aux systèmes industriels réactifs temps-réel et critique constitue le contexte de cette thèse. La vérification et la validation de ces systèmes sont soumises à des normes strictes [9]. Des méthodes formelles sont parfois nécessaires, au moins en ce qui concerne les propriétés hautement critiques. La thèse s'appuie sur le cluster de projets R&D français "CONNEXION" dans le domaine nucléaire. À partir d'un cas d'étude industriel et d'un ensemble préconisé d'outils, nous avons eu l'occasion d'examiner les difficultés d'adopter les techniques modernes MBSE dans un contexte industriel critique. De plus, MBSE permet de réaliser la V&V tôt dans les phases de conception et donc de détecter des défauts au plus tôt possible. Cette approche est très rentable pour les systèmes réactifs temps-réel et critiques puisque le coût des défauts trouvés plus tard dans le système peut être extrêmement élevé.

[1] `http://www.incose.org/ChaptersGroups/WorkingGroups/transformational/mbse-initiative`

Problématique

Depuis 2012, les principaux partenaires industriels de nucléaire français et académiques ont initié un ambitieux programme de R&D appelé "CONNEXION". Regroupant plusieurs projets, "CONNEXION" [71] vise à améliorer le processus de développement du système d'instrumentation et contrôle (I&C) dans des centrales nucléaires. "CONNEXION" s'appuie sur les expertises existantes des opérateurs dans l'industrie nucléaire française: EDF, ALSTOM, AREVA et RRCN; ainsi que sur divers outils fournis par des partenaires: CEA, CORYS, ESTEREL Technologies et ALL4TEC.

Definition 2. *Validation Fonctionnelle [9] est en fait la vérification des modèles du système construits en amont de la conception par rapport aux exigences fonctionnelles du système.*

La conception du système I&C se décline en plusieurs phases. Les modèles de plus en plus détaillés décrivant le comportement du système sont construits dans les phases amont. La vérification des ces modèles par rapport aux exigences fonctionnelles du système, définie comme Validation Fonctionnelle, est actuellement réalisée d'une manière manuelle. Un objectif de "CONNEXION" est d'automatiser autant que possible la validation fonctionnelle, en s'appuyant sur l'outillage fournie par des partenaires.

Contribution de thèse

Dans le cadre du projet "CONNEXION", notre travail de recherche s'appuie sur différents outils fournis par des partenaires et traite un cas d'étude industriel du système I&C. Nous proposons une méthodologie de test dirigée par les modèles (**Model-Based Testing ou MBT**), pour renforcer la validation fonctionnelle des modèles du système I&C. La méthodologie proposée n'est pas limitée au secteur nucléaire et peut être généralisée et appliquée à d'autres systèmes réactifs temps-réel et critiques en sûreté de fonctionnement.

Notre méthodologie repose principalement sur la technique dite "*model checking*" [19]. Cette technique a été proposée initialement comme une méthode formelle pour vérifier l'exactitude d'un modèle par rapport à un ensemble de propriétés. L'outil qui automatise la technique *model checking* est un *model checker*. Dans notre méthodologie, un *model checker* est utilisé de façon itérative, comme un outil de génération de test. En effet la validation fonctionnelle du système I&C doit être réalisée non seulement par rapport aux exigences fonctionnelles, mais également par rapport aux critères de couverture structurelle comme MC/DC. Les objectifs

de test basés sur la couverture sont spécifiés comme des "propriétés de piège" [91] pour forcer le *model checker* à générer des contre-exemples. Ces contre-exemples sont ensuite élaborés en cas de test permettant améliorer la couverture structurelle.

Le système I&C est composé de deux sous systèmes: un système contrôle-commande et un environnement physique. Le contrôle-commande est en réaction permanente et en temps-réel avec l'environnement physique. En raison de leurs différentes caractéristiques, l'environnement et le contrôle-commande sont en général spécifiés par deux différents langages de modélisation. Une solution est de reconstruire l'environnement avec le même langage de spécification du contrôle-commande, tel que présenté dans [43]. Mais ce n'est pas le choix de "CONNEXION". Par conséquent, nous avons décidé d'utiliser des hypothèses sur l'environnement pour raffiner la génération de test par *model checking*. Nous proposons une technique qui ajoute progressivement ces hypothèses pour obtenir un test de plus en plus réaliste. Cela aide les experts du système à concevoir un test en boucle fermée à partir d'un test en boucle ouverte. Nous donnons ci-dessous les définitions de "boucle fermée" et "boucle ouverte":

Definition 3. *Boucle fermée: un test en boucle fermée est une co-exécution d'un système réactif temps-réel avec son environnement physique.*

Definition 4. *Boucle ouverte: un test en boucle ouverte exécute seulement un système réactif temps-réel, sans prendre en compte les réactions de son environnement physique.*

Enfin, dans notre méthodologie, nous considérons le "Time Out" d'un *model checker* suite au problème d'explosion des états. Nous proposons une heuristique dite **vérification hybride** qui combine simulation et *model checking*. *Model checking* explore l'ensemble de l'espace d'état tandis que la vérification hybride n'explore qu'un sous-ensemble de tous les états possibles. Nous proposons également les éléments pour mettre en pratique la vérification hybride. Ces éléments comprennent:

- Chercher un *model checker* puissant et complet pour effectuer la vérification hybride sur un modèle en Lustre. Nous passons en revue les différentes versions du langage Lustre et nous avons testé plusieurs outils académiques sur le cas d'étude "CONNEXION".

- Nous avons également considéré la traduction d'un modèle en Lustre vers un autre langage de modélisation, SMV par exemple, pour bénéficier d'autres outils de *model checking*.

Nous arrivons à la conclusion qu'un seul outil n'est pas suffisant pour appliquer la vérification hybride à un modèle en Lustre de taille industrielle. La vérification hybride combine plusieurs techniques: exploration exhaustive de l'espace d'état, mémorisation des traces d'exploration, génération avant et arrière de traces, simulation pas-à-pas, etc. Une intégration de plusieurs outils est une solution.

Model checking et son application

Model checking

La vérification formelle est une technique qui applique des méthodes mathématiquement formelles à la vérification de systèmes. L'objectif est de prouver l'exactitude (ou l'inexactitude) du système sous test d'une manière rigoureuse. La recherche sur la vérification formelle au cours des deux dernières décennies a abouti à des techniques prometteuses. Ces techniques sont aussi soutenues par de puissants outils logiciels qui peuvent être utilisés pour automatiser divers étapes de vérification et ainsi réduire le coût.

Au début des années 1980, le *model checking* est proprosé comme une technique de vérification formelle. Il provient du travail indépendant de deux équipes: Clarke et Emmerson [61]; Sifakis et Queille [145]. Un *model checker* est l'outil qui automatise *model checking*. À une spécification formelle du système (c'est-à-dire un modèle du système), un *model checker* explore tous les états possibles du système. L'objectif est de prouver avec une rigueur mathématique l'exactitude du modèle de système par rapport à une propriété. Des *model checkers* d'avant garde peuvent gérer des espaces d'état d'environ 10^8 à 10^9 états avec des algorithmes explicites d'exploration. De plus grands espaces d'état, de 10^{20} jusqu'à 10^{476} états, peuvent être traités pour des problèmes spécifiques, en utilisant des algorithmes intelligents et dees structures de données sur mesure [19]. Cependant, le problème d'explosion d'état reste fondamental pour le *model checking*. En raison de ce problème, les *model checkers* peuvent s'arrêter sur un "time out" (TO), c'est à dire terminer le calcul sans donner une réponse au problème de vérification. Au cours des dernières années, les techniques de *model checking* ont suscité l'intérêt de nombreuses industries ayant des besoins en sûreté de fonctionnement, telles que la signalisation ferroviaire [42], l'avionique [43, 132, 133] et le nucléaire [172, 118, 171, 116].

Le premier prérequis pour utiliser le *model checking* est de disposer d'un modèle du système considéré. En informatique, les systèmes de transition sont souvent utilisés comme modèles pour décrire le comportement des systèmes. Ils sont illustrés par des graphes orientés où les nœuds représentent les états et les arêtes

représentent les transitions entre états. Un état décrit quelques informations sur le comportement du système à un moment donné. Les transitions précisent comment le système peut évoluer d'un état à l'autre. Le système ne peut se trouver que dans un état à la fois. L'état dans lequel il se trouve à un moment donné est appelé l'état actuel.

Le modèle du système et les propriétés à vérifier doivent être décrits de manière précise et non ambiguë. La logique temporelle est un formalisme mathématique adapté aux déclarations et au raisonnement où le temps est impliqué. Elle offre des opérateurs spécifiques pour le temps, proches du langage naturel (les adverbes comme "toujours", "jusqu'à" par exemple). Elle vient aussi avec une sémantique formelle, faisant de la logique temporelle un outil indispensable pour formaliser les propriétés concernant les comportements dynamiques d'un système. Les logiques temporelles les plus courantes sont les LTL (Linear Time Logic) [142] et CTL (Computation Tree Logic) [62].

Étant donné un modèle M et une propriété ϕ du système considéré, l'algorithme de *model checking* répond à la question "M satisfait-il la propriété ϕ ?". Si une violation de propriété est détectée, un *model checker* est capable de retourner un contre-exemple, illustrant comment la violation se produit. Dans la littérature, plusieurs algorithmes de *model checking* ont été proposés: le *model checking* explicite [124, 63, 145, 167] étant la première génération; le *model checking* symbolique [128] la deuxième génération et le *model checking* borné [37] la dernière génération.

Un problème de *model checking* est généralement un problème de vérification de propriété. D'autre part, les cas de test sont liés à certains objectifs de test. Si les objectifs du test peuvent être spécifiés dans la logique temporelle, puis utilisés comme propriétés pour forcer le *model checker* à générer des contre-exemples, le problème de génération de test est en fait transformé en un problème classique de *model checking*. L'idée principale de tester avec les *model checkers* est de forcer les *model checkers* à générer des contre-exemples, et puis interpréter ces contre-exemples comme des cas de test, comme montré par la Fig. 2.5. C'est important de noter que la spécification de propriété devrait être la négation d'un objectif de test de sorte qu'un contre-exemple violant la propriété, en fait, satisfait le test objectif.

Une approche répandue consiste à créer des propriétés en fonction des critères de couverture. En effet, de tels critères sont des objectifs de test très couramment utilisés. Les propriétés basées sur les critères de couverture sont initialement ap-

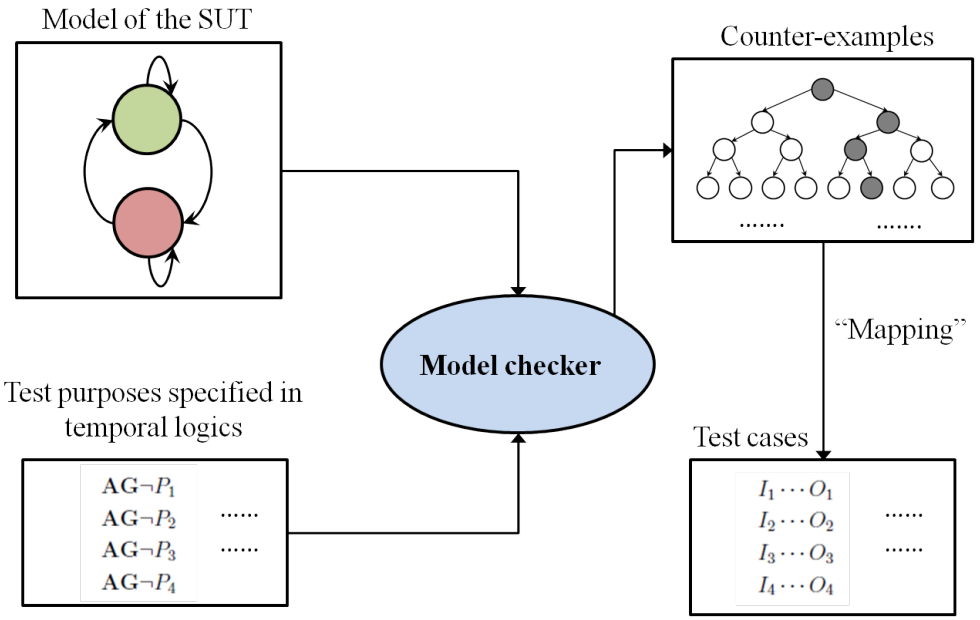


Figure 1: Testing with model checkers

pelées "trap properties" [91]. Une telle propriété déclare que certains éléments ne pourraient jamais être couverts par une exécution. Dans ce cas-là, le *model checking* de cette propriété va générer un contre-exemple illustrant comment l'élément peut être couvert si la propriété est violée. La majorité de générations de tests basées sur la couverture utilisent des critères de couverture structurelle. Il est parfois souhaitable de créer les cas de test basés sur d'autres objectifs de test. Différentes techniques comprennent, par exemple, l'approche basée sur les exigences [51, 170] et l'approche basée sur la mutation [15].

Approche synchrone pour les systèmes réactifs et temps réel

Autour des années 90s, trois langages de programmation synchrones ont été proposés par des groupes académiques français: Esterel [36, 32], Signal [122, 28, 29] et Lustre [52, 101]. Ces langages de programmation partagent les mêmes principes: hypothèse synchrone et concurrence déterministe. En pratique, l'hypothèse synchrone suppose que le programme est capable de réagir à un événement externe avant qu'un autre événement se produise. Ces langages synchrones diffèrent les uns des autres dans le style, ce qui correspond à leur usage dans différents domaines d'application. On s'intéresse dans ce document à Lustre puisque c'est le choix du projet "CONNEXION" ainsi que la thèse.

Lustre est un langage déclaratif synchrone, basé sur les flots de données (data-flow). Lustre est basé sur deux notions fondamentales: (1) flot: chaque variable et expression Lustre est considérée comme un flot, c'est-à-dire, une séquence de valeurs d'un type donné; (2)horloge: une horloge représente une séquence de temps. Chaque flot est implicitement associé à une horloge: le flot prend la $n^ième$ valeur de sa séquence au $n^ième$ instant de son horloge. Si le comportement d'un système peut être décrit de manière cyclique, alors ce cycle qui définit une séquence de temps est appelée l'horloge globale (ou l'horloge de base). Tout flot dont l'horloge est l'horloge globale prend la $n^ième$ valeur au $n^ième$ cycle d'exécution. D'autres horloges différentes peuvent être définies par rapport à l'horloge globale en utilisant un flot de valeur booléenne: par exemple, l'horloge est une séquence de temps où le flot booléen prend la valeur "vrai".

Un programme Lustre décrit les relations entre ses entrées et ses sorties par les variables et équations. Chaque variable est un flot, donc les variables X et Y sont respectivement $(x_1, x_2, ..., x_n)$ et $(y_1, y_2, ..., y_n)$. L'équation $X = Y$ dénote $x_i = y_i$ où $0 \leq i \leq n$. Lustre propose des types de données de base et des opérateurs habituels (arithmétiques, booléens, relationnels, contrôle). Lustre offre également deux opérateurs temporels: (1) l'opérateur `pre` fait référence à la valeur de son opérande au cycle précédent; (2) l'opérateur `->` est utilisé pour attribuer à son opérande la valeur au cycle initial.

Lustre a eu du succès dans deux projets industriels (l'un nucléaire et l'autre avionique) dans les années 1980. Par conséquent, Lustre a été commercialisé et donne lieu à SCADE (Safety-Critical Application Development Environment). SCADE propose un environnement synchrone dirigé par les modèles pour la conception, la validation et la mise en œuvre de logiciels embarqués. Combinant diagrammes de blocs et machines à états hiérarchiques, SCADE permet aux utilisateurs de créer les modèles graphiques de haut niveau avec des spécifications formelles rigoureuses. SCADE supporte également la simulation visuelle et l'analyse de couverture de test ainsi que des techniques de vérification formelle. Le kit complet SCADE est maintenant largement utilisé dans les industries critiques telles que l'avionique, l'automobile, la ferroviaire, etc. SCADE est également le choix de "CONNEXION" pour spécifier des modèles du système I&C.

Côté académique Lustre continue à évoluer depuis plus de vingt ans et a abouti à deux versions couramment utilisées: Lustre V4 et Lustre V6. Le formalisme de la version actuelle de SCADE est Scade 6 [65]. Le formalisme Scade 6 est constitué de diagrammes de blocs (block diagrams) et de machines à états (safety

state machines). Lustre V6 est le langage textuel sous-jacent des diagrammes de blocs. Les machines à états de sécurité ont évolué à partir du langage Esterel et du modèle SyncCharts des statecharts synchrones. Il a été démontré que les machines à états sont adaptables aux grands systèmes de contrôle [33]. La Fig. 2.10 illustre la relation entre différentes versions de Lustre et Scade 6.

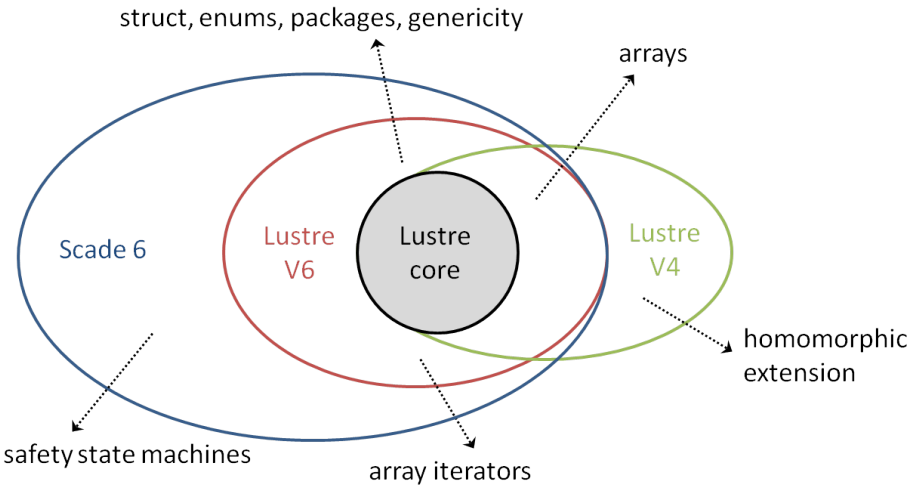


Figure 2: Relationship between academic Lustre versions and Scade 6

Le langage Lustre bénéficie déjà de divers outils de vérification. Ils comprennent des *model checkers* open-source: Lesar [99, 100], Kind 2 [57], PKind [114], jKind [89], Zustre [117]). Ces outils sont orientés vers *model checking* traditionnel: vérification de propriété. Il y a également des outils plus orientés pour la génération de test: Lutess [44, 72, 140], Lurette [149, 111, 109, 110] et GATeL [126, 127]. GATeL a été choisi par le projet "CONNEXION". Dans cette thèse, on étudie Lesar, Kind 2 et GATeL.

Projet "CONNEXION": vers un environnement complet de V&V

Les objectifs de "CONNEXION" concernent les Systèmes de Contrôle Nucléaire et les Technologies Opérationnelles pour maintenir un haut niveau de sûreté de fonctionnement, pour offrir de nouveaux services améliorant l'efficacité des activités opérationnelles. Avec l'approche actuelle de l'ingénierie, des modèles de

plus en plus détaillés décrivant le comportement du système I&C ont été construits dans les phases amont de conception. Ces modèles sont spécifiés avec un langage métier qui est formel mais pas directement exécutable. Par conséquent, la validation fonctionnelle de ces modèles est actuellement réalisée d'une manière manuelle. Les tests automatisés ne sont possible qu'au niveau de l'implémentation.

Un objectif du projet est donc d'automatiser autant que possible la validation fonctionnelle. "CONNEXION" rassemble un ensemble unique et complet d'outils de modélisation et de vérification. Intégrant les progrès récents de MBSE, "CONNEXION" propose de renforcer le cycle en V de développement en introduisant deux sous-cycles de validation fonctionnelle. Les deux sous-cycles effectuent la vérification de la conception en modèles par rapport aux exigences fonctionnelles, résultant en un cycle innovant en triple V, schématisé par la Fig. 3.1. Le processus de développement est aligné avec la norme CEI [5], en encourageant les outils d'automatisation des tests. Le cycle de vie actuel repose sur une approche document-centrique; "CONNEXION" permet la transition vers un pratique centrée sur les modèles préconisée par l'INCOSE [7].

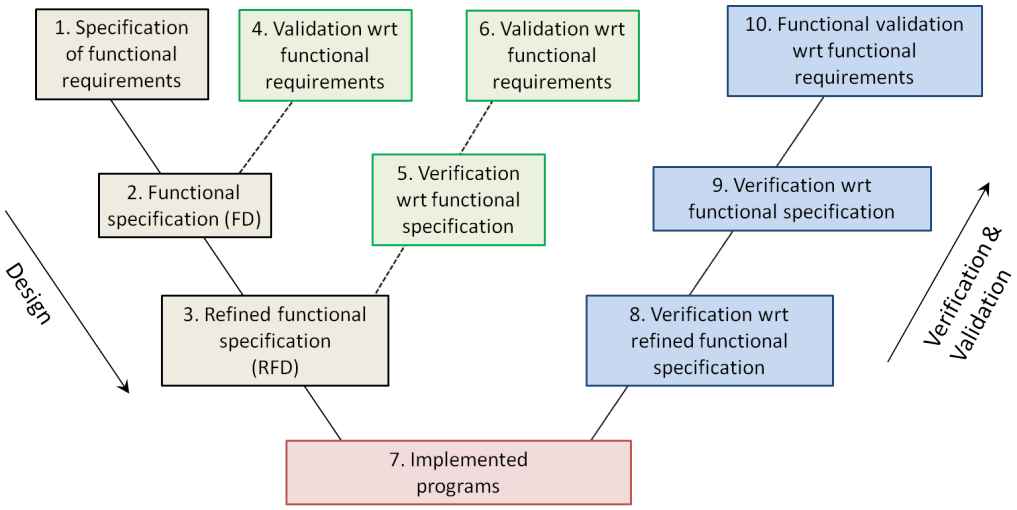


Figure 3: Triple V system life cycle of the I&C system

Le système I&C d'une centrale nucléaire est composé de plusieurs centaines de systèmes élémentaires (SE), contrôlant avec un très haut niveau de sécurité des milliers d'actionneurs commandés à distance: environ 8000 signaux binaires et 4000 signaux analogiques envoyés à la salle de contrôle, concernant plus de 10 000 sous fonctions I&C et plus de 300 armoires I&C. Chaque SE est un ensemble de circuits et de composants, remplissant une fonction essentielle au fonctionnement

de la centrale nucléaire. Chaque SE est documenté par un dossier de système élémentaire (DSE), contenant des documents détaillant le SE dans différents aspects: fonctionnement, contrôle-commande, équipement, etc.

Un système élémentaire est composé de deux sous-systèmes:

- le **procédé** représente l'infrastructure physique et l'équipement, par exemple, échangeurs de chaleur, vannes, tuyaux, etc.
- le **contrôle-commande (CC)** est un système réactif temps-réel réagissant en permanence avec le procédé. Il est responsable de la protection, du contrôle et de la supervision du fonctionnement du procédé.

Le procédé et le contrôle-commande représentent différents aspects du SE et donc correspondent à différents documents dans le DSE. Le contrôle-commande, décrivant l'aspect fonctionnel du SE, est spécifié par **diagramme fonctionnel (DF)**. Le DF est un langage formel dédié au secteur nucléaire basé sur blocs prédéfinis.

Le cycle en V , correspondant à l'approche actuelle, contient les phases 1, 2, 3, 7, 8, 9 et 10 (voir Fig. 3.1). Le côté descendant du modèle V (1, 2, 3, 7) représente la conception et l'implémentation du système et le côté montant (8, 9, 10) représente la vérification et la validation. Il est important de noter que le côté descendant du modèle V concerne un seul système élémentaire, mais les activités de V&V sont réalisées en intégrant ce SE à une abstraction de son environnement [71]. De plus, la granularité du modèle de procédé utilisé en co-simulation est adapté aux objectifs de validation spécifiques à chacune de ces trois dernières phases (8, 9 et 10).

La conception commence en phase 1 avec une modélisation globale de la spécification fonctionnelle du SE à partir d'un schéma simplifié du procédé et de ses divers configurations opérationnelles. Le schéma simplifié du procédé est une spécification fonctionnelle pour l'environnement physique et son contrôle-commande [71]. En phase 2, une spécification fonctionnelle dédiée au contrôle-commande est développée sous la forme d'un diagramme fonctionnel par des ingénieurs de système. Ce DF représente une spécification explicite du comportement de contrôle-commande cohérent avec le diagramme du procédé et la description des exigences fonctionnelles produits dans la phase 1. Cette spécification de conception est ensuite progressivement détaillée dans un diagramme fonctionnel raffiné (DFR) en phase 3, qui est prêt à être transformé en programmes implémentés en phase 7. La V&V du système commence dans la phase 8 en vérifiant l'implémentation (en sortie de la

phase 7) par rapport à sa spécification de conception (phase 3). En phase 9, le contrôle-commande du SE est intégré aux autres systèmes élémentaires déjà validés. A la fin du cycle de vie (phase 10), les techniques Hardware-In-the-Loop (HIL) [30] sont appliquées à tous les SEs interconnectés et validés en phase 9.

Le projet "CONNEXION" cherche à améliorer la validation fonctionnelle en introduisant un premier sous-cycle en V (phases 1, 2 et 4) et un second sous-cycle en V (phases 1, 2, 3, 5 et 6). Les deux sous-cycles introduisent des modèles exécutables de contrôle-commande: en phase 2 et 3, les spécifications exécutables correspondant à la DF et DFR sont développées. Phase 1 présente un modèle de haut niveau décrivant le système élémentaire. Le modèle exécutable du DF produit en phase 2 est vérifié par rapport aux exigences fonctionnelles (phase 4). Le modèle exécutable du DFR produit en phase 3 sera (i) vérifié par rapport au modèle de spécification fonctionnelle développé en phase 2 (phase 5) et (ii) vérifié par rapport aux exigences fonctionnelles (phase 6). Les deux sous-cycles V se traduisent par un cycle de vie innovant du système en triple V.

Un autre objectif de "CONNEXION" est de développer une plateforme complète de V&V soutenant les méthodologies de validation fonctionnelle présentées ci-dessus. La plateforme accompagnerait toutes les activités de V&V nécessaires au développement du système I&C tout au long de son cycle de vie. La Fig. 3.2 illustre le principe d'une telle plate-forme au niveau d'un système élémentaire. Grâce à la co-simulation du procédé et du contrôle-commande, cette plateforme permet aux ingénieurs de vérifier divers aspects du système pendant ses phases de développement. Le modèle des propriétés attendues du système et le modèle d'environnement, tel que perçu par le contrôle-commande, est également nécessaire pour décrire en particulier les exigences que le SE doit respecter et les contraintes sur sa sollicitation. Nous spécifions que le modèle d'environnement introduit les contraintes qui proviennent d'autres systèmes élémentaires interconnectés au SE sous test. Il est tout à fait possible d'effectuer une vérification sans contexte du SE sous test, qui dans ce cas ne nécessite aucun modèle d'environnement particulier. Enfin, nous pensons que la plateforme devrait être supportée par un système d'information de traçabilité (SIT). Le SIT accompagne les activités de V&V et enregistre non seulement les données mais aussi l'historique de leur relations.

Les outils préconisés par les partenaires de "CONNEXION" peuvent être divisés en deux catégories: outils de modélisation et outils de vérification. Les outils de modélisation contiennent: (1) Papyrus [90] de CEA (basé sur SysML) est utilisé pour créer un model de haut niveau d'un système élémentaire; (2) Dymola (outil commercial basé sur Modelica) pour modéliser le procédé; (3) SCADE Suite pour

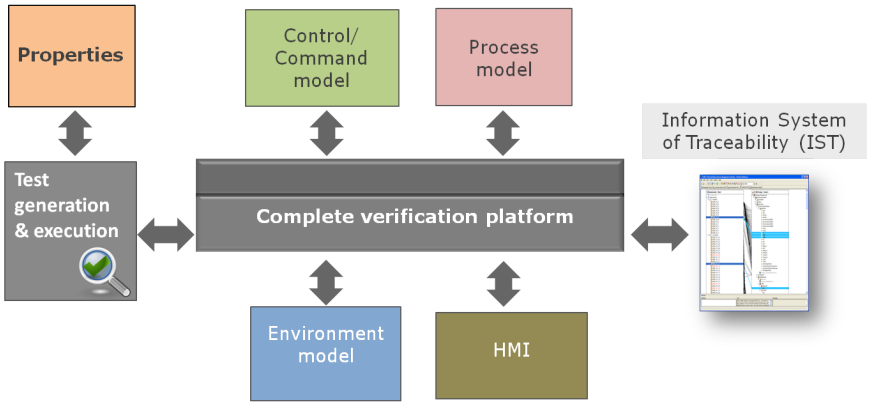


Figure 4: A complete verification platform

spécifier le contrôle-commande à différents niveaux d'abstraction, correspondant à DF et DFR. Les outils de vérification sont résumés dans le tableau 1 ci-dessous.

MaTeLo	outil de génération de test dirigé par les modèles
GATeL	Model checker pour modèles en Lustre
SCADE QTE	analyse de couverture pour modèles Scade
ALICES	plateforme de co-simulation
ARTiMon	observateur temps-réel

Table 1: Aperçu des outils V&V dans "CONNEXION"

- MaTeLo [58] de ALL4TECH est un outil de génération de test dirigé par les modèles pour test d'usage statistique [123]. La génération de test est basée sur un modèle d'usage (chaîne de Markov), créées manuellement à partir de la spécification des exigences fonctionnelles. Un cas de test correspond à un chemin choisi au hasard dans le modèle d'utilisation. Dans "CONNEXION", le modèle d'utilisation est construit en fonction des exigences et les cas de test générés sont destinés aux simulations en boucle fermée.

- Puisque le modèle procédé et le modèle contrôle-commande sont spécifiés dans différents languages, la simulation en boucle fermée est réalisée sur une plateforme: ALICES [136] de CORYS. ALICES est responsable de l'échange et de la synchronisation de données entre les modèles via le standard open source FMI/FMU.

- Pendant la co-simulation, ARTiMon [90] du CEA fournit une observation temps-réel des résultats d'exécution de sorte que toute violation des propriétés sera enregistrée. ARTiMon est également intégré sur ALICES.

- SCADE QTE (Qualified Testing Environment) est utilisé pour mesurer le taux de couverture structurelle à partir des résultats d'exécution.
- GATeL du CEA est un *model checker* pour les modèles en Lustre. Avec un objectif de test exprimé dans une version étendue de Lustre, GATeL génère des données de test qui conduisent le système à un état satisfaisant l'objectif du test. Les modèles Scade peuvent être automatiquement transformés en modèles Lustre par l'outil s2d [2].

MBT pour Validation Fonctionnelle: vers vérification hybride

Dans cette thèse, une méthodologie est définie comme une collection de processus, méthodes, et outils. Une méthodologie de MBT (Model-Based Testing) est donc la collection de processus, méthodes et outils, utilisée pour effectuer la V&V d'un système dans un contexte dirigé par les modèles. Nous proposons une nouvelle méthodologie de MBT pour des systèmes réactifs temps-réel et critique. Dans "CONNEXION", cette méthodologie est appliquée à la validation fonctionnelle des modèles du système I&C développés dans les phases amont de conception.

Terminologie

Definition 5. *Unité structurelle (structural unit ou SU en anglais) Une SU est la mesure de l'unité de couverture sur le modèle, indépendamment des critères de couverture choisis.*

Definition 6. *Vérification de l'accessibilité. Pour une unité structurelle donnée, un model checker vérifie formellement si cette unité structurelle peut être exécutée par n'importe quel test. Si oui, cette unité structurelle est dite accessible.*

Definition 7. *Test en boucle ouverte. Dans un test en boucle ouverte, seul le système réactif temps réel lui-même est exécuté. Comportement de son environnement physique n'est pas pris en compte.*

Definition 8. *Test en boucle fermée. Dans un test en boucle fermée, le système réactif temps réel et son environnement sont exécutés ensemble, également appelé co-exécution. Le comportement du système réactif temps réel est donc influencé par son environnement.*

[2]The *s2d* tool is developed and provided by *Laboratoire Sûreté des Logiciels, CEA/DRT/DT-SI/SOL, 91191 Gif sur Yvette, France*

Workflow et outils

Notre méthodologie est composée de trois phases principales:

- **Phase 1**: Génération de cas de test basés sur des objectifs de test dérivés de exigences fonctionnelles de haut niveau. Ces cas de test fonctionnels sont ensuite exécutés en boucle fermée. Un outil MBT est utilisé pour la génération de test (MaTeLo dans "CONNEXION"). Un simulateur soutenant l'exécution de tests en boucle fermée est également requis. (la plate-forme ALICES dans "CONNEXION").

- **Phase 2**: Après l'exécution, la couverture structurelle de ces cas de test est mesurée (couverture MC/DC dans "CONNEXION"). Il est important de noter que la couverture est mesurée uniquement sur le système réactif temps réel. Les unités structurelles non couvertes sont collectées. Un outil de l'analyse de la couverture est requis dans cette phase (SCADE QTE dans "CONNEXION").

- **Phase 3**: Pour chaque unité non couverte, un *model checker* est utilisé pour générer les séquences de test exécutant l'unité considérée. Le *model checker* travaille sur le modèle du système réactif temps réel et donc génère des séquences de test en boucle ouverte. Avec l'aide des experts du système, les cas de test en boucle fermée peuvent être développés à partir de ces tests en boucle ouverte. Les nouveaux cas de test ne devraient pas seulement améliorer la couverture structurelle, mais également être liés aux exigences fonctionnelles. Ce processus est itéré sur chaque unité non couverte jusqu'à ce que les critères de couverture soient satisfaits. L'outil requis dans cette phase comprend un *model checker* (GaTeL in "CONNEXION"), pour la génération de tests basés sur la couverture. L'outil utilisé en phase 1 est aussi nécessaire pour en construire un test fonctionnellement réaliste couvrant l'unité structurelle considérée.

Méthodologie MBT: Partie 1 sur 2

La première partie de la méthodologie est schématisée par la Fig. 4.2. Un outil de génération de tests fonctionnels est d'abord utilisé pour dériver une suite de tests fonctionnels (TS) selon les exigences fonctionnelles (étape 1). Cette suite de tests est ensuite exécutée en co-simulation sur les modèles du système (étape 2). La couverture structurelle (SC) de la suite de tests est mesurée par un outil de l'analyse de couverture (étape 3). Nous définissons SU^U comme l'ensemble de toutes les unités structurelles non couvertes après l'exécution de TS (étape 4). SU^A est l'ensemble de toutes unités réellement inaccessibles et SU^P l'ensemble

de toutes les unités potentiellement inaccessibles, pour lesquelles la méthode n'a pas réussi à répondre à la question d'accessibilité. Initialement ces trois ensembles sont tous vides. Prenez une unité structurelle non couverte *su* de SU^U (étape 6) et appliquez un *model checker* pour vérifier si *su* est accessible (étape 7):

- Si *su* n'est pas accessible, envoyez un message d'alerte à l'utilisateur (étape 11) et enregistrez *su* comme une unité structurelle réellement inaccessible (étape 12): $SU^A_j = SU^A_{j-1} \cup su$. Passez à l'étape 5 et continuez les étapes suivantes.

- Si *su* est accessible, le *model checker* doit avoir produit des séquences de test en boucle ouverte qui forcent le système à atteindre *su*. Ces données seront utilisées pour construire un cas de test fonctionnel en boucle fermée (noté *ntc*) qui couvre cette unité structurelle particulière et probablement d'autres unités (éventuellement dans SU^P qui devait être calculé à nouveau) (étape 9). À ce stade, un retour au niveau des exigences fonctionnelles est nécessaire pour assurer la réalité fonctionnelle de *ntc*. Complétez l'ancienne suite de tests TS avec ce nouveau cas de test (étape 10): $TS_i = TS_{i-1} \cup ntc$. Passez à l'étape 2 et continuez les étapes suivantes.

Méthodologie MBT: Partie 2 sur 2

La Fig. 4.3 représente la deuxième partie de notre méthodologie. La troisième possibilité de la vérification d'accessibilité d'une unité (étape 7) est TO: le *model checker* arrête son exécution sans donner une réponse. Notre solution est d'abord d'augmenter TO puis d'appliquer la vérification hybride (une combinaison de *model checking* et simulation) de manière similaire à [131] pour vérifier l'accessibilité de cette unité (étape 13). Si *su* est accessible alors allez à étape 8 et continuez le processus suivant. La vérification hybride peut aussi produire un TO, ce qui conduit à envoyer un message "abandon" à l'utilisateur (étape 15) puis enregistrer *su* comme potentiellement inaccessible (étape 16): $SU^P_k = SU^P_{k-1} \cup su$. Passer ensuite à l'étape 5 et poursuivre le processus suivant. Notez qu'à l'étape 5, nous avons $SU^U_i = SU^U_i - SU^A_j - SU^P_k$.

Ce processus converge lorsque le critère de couverture structurelle est satisfait ou s'il n'y a plus d'unités structurelles non couvertes inexplorées, c'est-à-dire $SU^U_i = \emptyset$. Comme $TS_i \supset TS_{i-1}$, ceci entraîne $SU^U_i \subset SU^U_{i-1}$ et $SC_i > SC_{i-1}$ car au moins une unité structurelle en plus est couverte.

Il est possible que le processus se termine immédiatement après l'exécution de la suite de tests fonctionnels initiaux TS_0, si le SC_0 correspondant est déjà

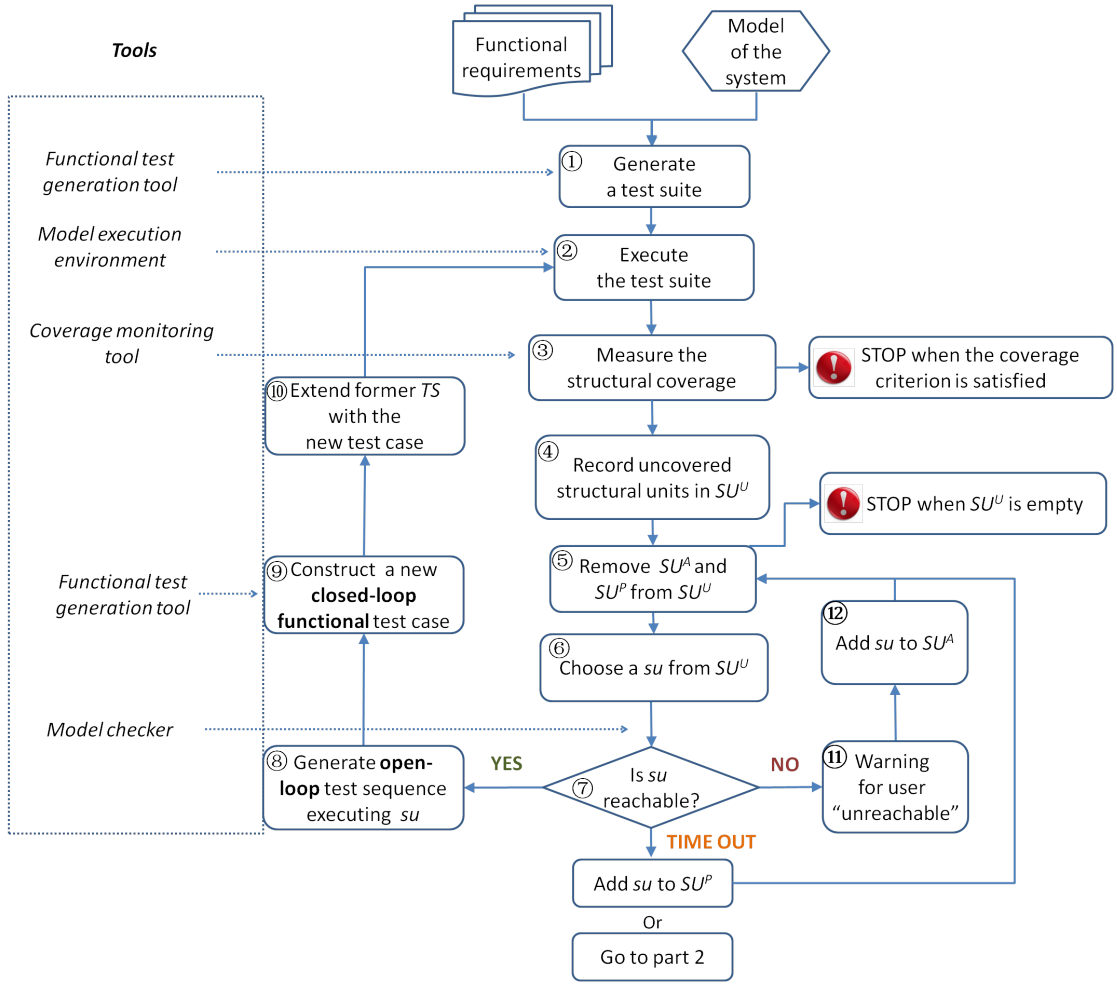


Figure 5: Model-based testing methodology: part 1 of 2

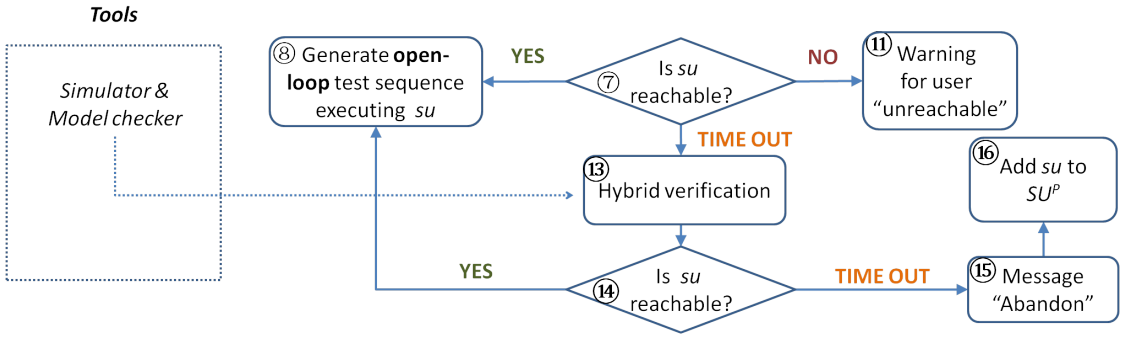


Figure 6: Model-based testing methodology: part 2 of 2

satisfaisant. Sinon, à la fin du processus, si la boucle à gauche est exécutée au moins une fois, nous avons une couverture de test améliorée; si la boucle à droite est exécutée au moins une fois, c'est-à-dire $SU^A \cup SU^P \neq \emptyset$, une analyse plus approfondie avec les auteurs de la spécification est requise car au moins une unité structurelle est suspectée d'être code mort ou même un bug.

Une heuristique: vérification hybride

La vérification hybride [158, 157, 55] est une technique combinant *model checking* et simulation. Le *model checking* explorer tous les états possibles tandis que la simulation explore partiellement l'ensemble de l'espace d'état. La Fig. 4.4 représente le principe de vérification hybride. La vérification hybride nécessite diverses techniques telles que l'exploration de l'espace d'état, la mémorisation des traces d'exploration, la génération avant/arrière, la simulation étape par étape, etc.

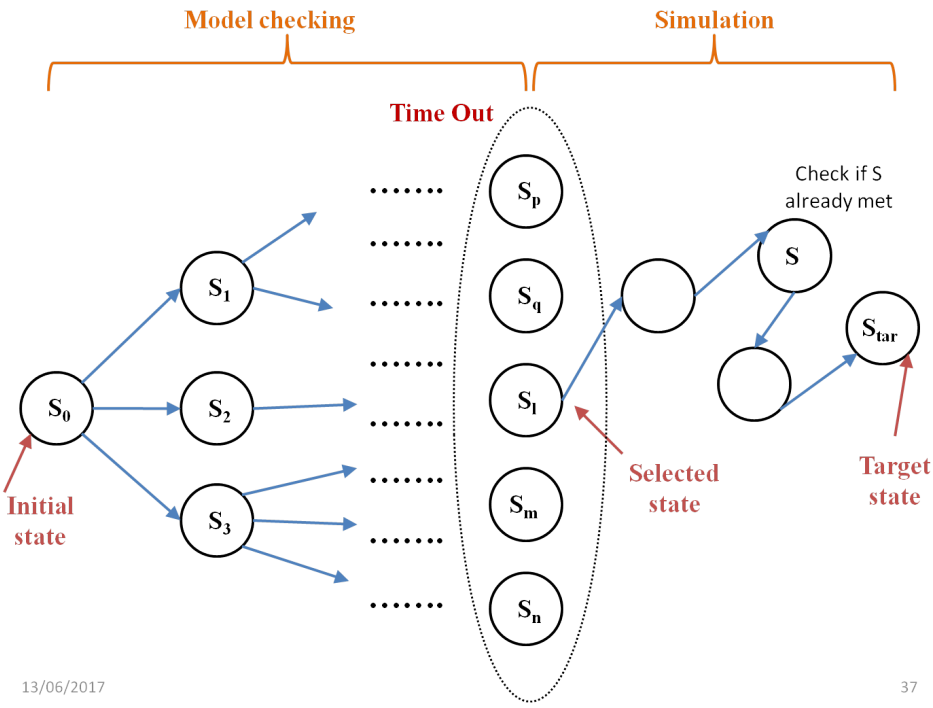


Figure 7: Principles of hybrid verification

Supposons que nous ayons un modèle du système sous test et que le problème à vérifier est le suivant: à partir d'un état initial S_0, pouvons-nous trouver un cas de test qui arrive à l'état cible S_{tar}. Dans le cadre de notre méthodologie, une unité

structurelle non couverte peut être considérée comme l'état cible. Le *model checker* effectue le calcul jusqu'à épuisement du temps ou de la mémoire (Time Out). Nous supposons que le *model checker* est capable de mémoriser les traces d'exploration des états déjà explorées juste avant TO. Par exemple, le *model checker* s'est arrêté à un cycle de calcul où les cinq états $S_p, S_q, S_l, S_m et S_n$ ont été explorés. Nous sélectionnons parmi les cinq états un candidat, disons S_l, comme l'état candidat pour démarrer une simulation étape par étape. La sélection de l'état candidat est basée sur une distance informelle définie par l'utilisateur entre chaque état candidat potentiel et l'état cible. La définition d'une telle sélection reste la partie la plus délicate et la plus difficile de la technique. Certaines heuristiques peuvent être facilement trouvées, cependant, nous doutons qu'une distance formelle peut être établie. Si tel était le cas, cela pourrait être intégré au *model checker*.

Raffinement par ajout progressif des contraintes

Cette section présente une technique pour raffiner la génération de test en boucle ouverte présentée dans la méthodologie ci-dessus. La technique est illustrée avec le *model checker* GATeL. GATeL prend le modèle Lustre du système de contrôle-commande comme une entrée. Il permet également deux autres entrées: un objectif de test et une description de l'environnement. Notre objectif de test est de vérifier si une unité *su* donnée peut être couverte par n'importe quel test, c'est-à-dire l'accessibilité de *su*. Au cas où *su* est atteignable, GATeL génère des séquences de test atteignant *su* désirée au dernier cycle de calcul. La description de l'environnement est composée d'expressions booléennes destinées à sélectionner parmi toutes les valeurs possibles de variables celles correspondant aux réactions réalistes de l'environnement physique. Chaque expression de sélection est indiquée comme une directive "assert" qui doit être vraie à chaque cycle de séquences générées. Ces expressions sont utilisées par GATeL pour dériver des contraintes définissant des relations entrées/sorties.

Les contraintes sont divisées en trois catégories: (1) Contraintes physiques: filtrer les valeurs qui ne pouvaient pas apparaître dans le système réel physique où le modèle est appliqué. Ces contraintes concernent seulement des variables d'entrée. (2) Contraintes d'initialisation: définir les valeurs de variables au cycle initial. Ces contraintes concernent seulement des variables d'entrée. (3) Contraintes d'exigences: dérivées des exigences fonctionnelles du système, définir les relations entre des variables d'entrée et de sortie. L'ordre pour ajouter progressivement les contraintes est en premier ajout des contraintes physiques, puis ajout des contraintes d'initialisation et finalement ajout des contraintes d'exigences. Notez que toutes les exigences fonctionnelles ne peuvent être traduites en contraintes invariantes. Dans cette thèse nous ne traitons que les exigences invariantes.

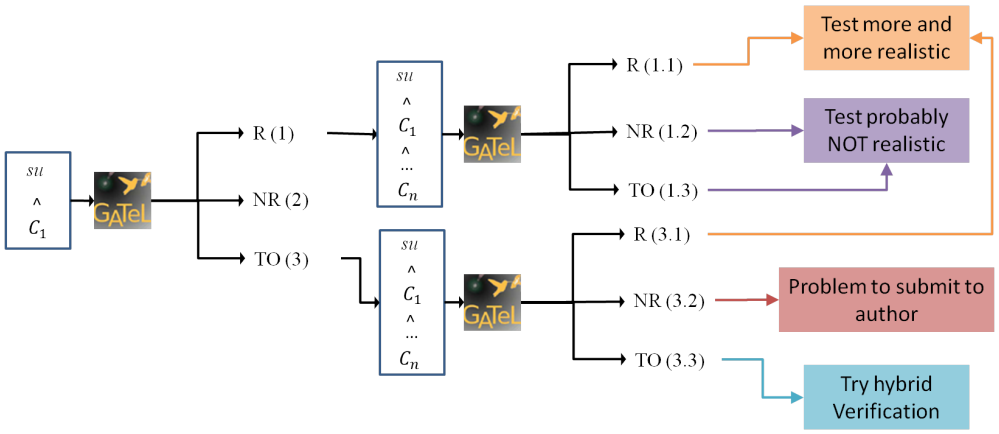


Figure 8: Refining test generation by adding constraints to the model checker

La Fig. 4.14 illustre le principe de la technique de raffinement. Étant donné une unité structurelle *su* non-couverte, GATeL vérifie si *su* est accessible tout en respectant les contraintes définies dans la description de l'environnement. D'abord cette description contient seulement une contrainte C_1 et la vérification d'accessibilité peut produire trois résultats:

- (1) *su* est accessible (R);
- (2) *su* est trouvé non-accessible (NR). À ce stade, il n'est pas nécessaire d'ajouter plus de contraintes. *su* sera enregistrée comme une unité non accessible.
- (3) GATeL a un TO.

Pour le résultat (1) ou (3), de nouvelles contraintes sont ensuite ajoutées progressivement à la description de l'environnement, une à la fois. Après chaque ajout d'une nouvelle contrainte, GATeL vérifie à nouveau l'accessibilité de *su*. Avec plus de contraintes, les résultats possibles sont:

- (1.1) *su* est toujours accessible. Dans ce cas, GATeL génère des séquences de test où *su* est couverte au dernier cycle. Les séquences construisent un nouveau cas de test en boucle ouverte satisfaisant les contraintes $C_1 \wedge C_2 \wedge ...C_n$. Cela aide à construire un cas de test en boucle fermée couvrant *su*.
- (1.2) *su* devient NR avec la nouvelle contrainte, ce qui signifie que le résultat précédent R (1) n'était pas réaliste. Cela pourrait suggérer que *su* est un unité structurelle inaccessible qui nécessite une analyse plus approfondie.

Cela pourrait aussi indiquer un bug quelque part: la contrainte n'est peut être pas correctement formalisée ou bien il peut y avoir une violation de l'exigence décrite par la contrainte.

- (1.3) GATeL a un TO. En général, ajouter des contraintes réduit l'espace d'état que GATeL doit explorer. Une explication possible est que les séquences de test précédemment générées dans (1) sont éliminées par la contrainte nouvellement ajoutée et GATeL, exactement comme dans (1.2), ne peut pas trouver une autre nouvelle trace couvrant *su* dans une période de temps restreinte.
- (3.1) Avec plus de contraintes ajoutées à la description de l'environnement, GATeL génère des séquences de test de plus en plus réalistes couvrant *su*.
- (3.2) Nous avons une hypothèse plus forte que "*su* est inaccessible".
- (3.3) Un TO. D'après Fig. 4.3 la vérification hybride est la dernière solution à envisager.

Cas d'étude "CONNEXION": SRI

L'étude de cas proposée dans "CONNEXION" est SRI: un système élémentaire présent dans le système I&C de toutes les centrales nucléaires françaises. La première partie de notre méthodologie a été testée sur SRI et les résultats seront présentés ci-dessous. Bien que la deuxième partie de la méthodologie n'ait pas été testée avec succès, nous avons essayé trois *model checkers* (GATeL, Lesar, Kind2) sur SRI et les avons comparés par rapport aux techniques requises pour la vérification hybride.

La fonction principale de SRI est d'assurer la réfrigération de plusieurs autres systèmes élémentaires en interface, dits les clients de SRI. SRI s'interface également avec une source de refroidissement SEN à travers des échangeurs de chaleur. Le procédé de SRI comprend deux échangeurs de chaleur en parallèle, où l'eau froide de SEN et l'eau chaude de ses clients se mélangent. La température de l'eau à la sortie des échangeurs de chaleur est régulée par trois vannes parallèles, variant par leur ouverture le débit et donc l'efficacité des échangeurs. Un réservoir d'eau est utilisé pour compenser la fuite éventuelle du circuit. Trois pompes en parallèle assurent la circulation de l'eau dans le système (La troisième pompe n'est utilisée qu'un remplacement). La Fig.5.1 illustre un schéma simplifié du procédé de SRI. Le contrôle-commande de SRI est modélisé en diagramme de blocs dans SCADE Suite.

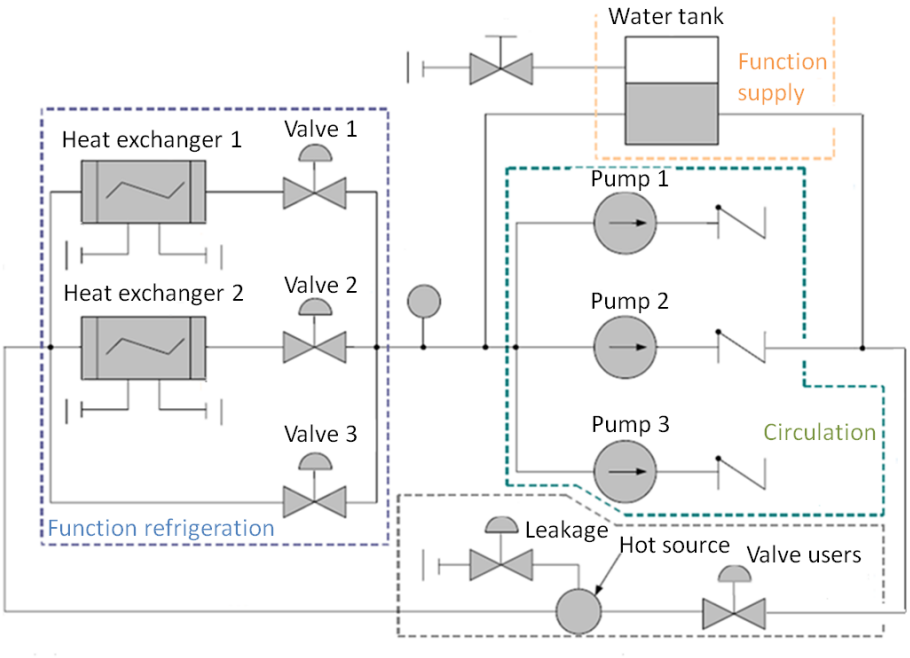


Figure 9: A simplified schema of SRI

Les résultats d'expérimentations sont organisés en trois phases, de la même manière que la méthodologie est présentée précédemment.

Phase 1

Les activités réalisées pendant cette phase comprennent la génération de tests fonctionnels avec l'outil MaTeLo et l' exécution de ces tests en boucle fermée sur la plateforme ALICES. À partir des exigences fonctionnelles, AREVA, un partenaire du projet, a créé un modèle d'usage de SRI dans MaTeLo et a généré une suite de test contenant 10 cas de test. Ensuite, nous exécutons ces 10 cas de test par simulation en boucle fermée sur ALICES. Nous obtenons les scripts enregistrant les données d'échange entre le procédé et le contrôle-commande pendant la simulation.

Phase 2

Les scripts obtenus en phase 1 sont maintenant utilisés pour mesurer la couverture MC/DC sur chaque opérateur composant le modèle de contrôle-commande, comme indiqué dans la Fig. 5.3. Plus de détails concernant les métriques de couverture des modèles Scade peuvent être trouvés dans [80]. Le modèle du système de contrôle-commande en Scade est structuré comme une hiérarchie d'opérateurs. Au niveau

de l'opération racine (l'opérateur du plus haut niveau), le taux de couverture MC/DC mesuré d'environ 50%.

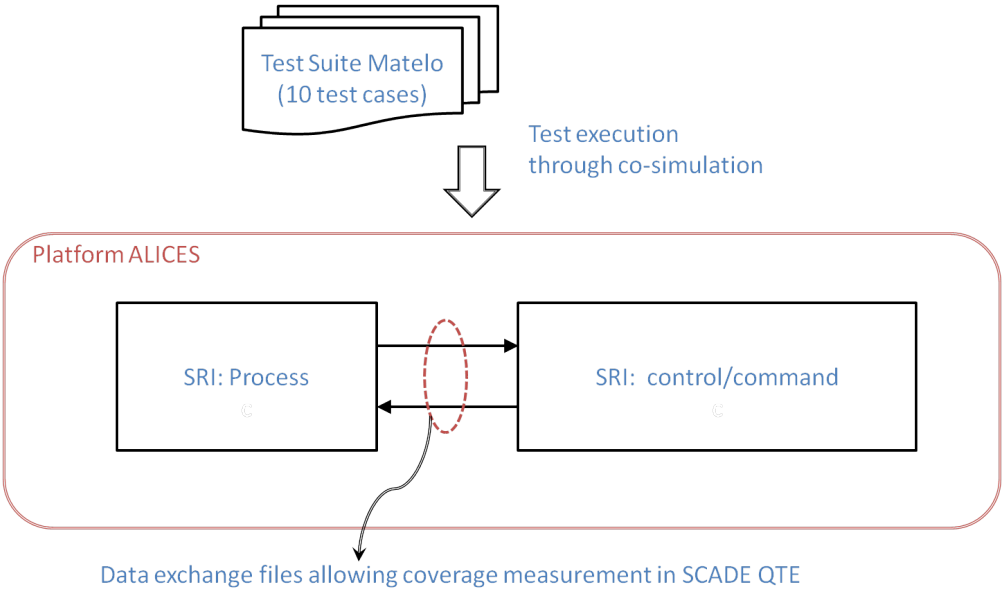


Figure 10: Coverage measurement

Phase 3

Nous devons choisir une unité structurelle non couverte pour effectuer la vérification d'accessibilité. Une suggestion est de commencer avec un opérateur au plus "profond" possible, qui ne contient pas d'autres opérateurs. L'avantage est qu'un test exécutant cette unité au bas niveau a une grande chance de couvrir également d'autres SU au niveau supérieur qui n'ont pas été couvertes. L'unité structurelle que nous avons choisie est à trois niveaux plus bas par rapport à l'opérateur racine: les deux entrées booléennes d'un opérateur "and" logique n'ont jamais pris la valeur VRAI tous les deux à la fois.

GATeL est ensuite utilisé pour vérifier l'accessibilité de cette unité structurelle selon la technique de raffinement: ajouter progressivement des hypothèses sur le comportement de l'environnement. Ces hypothèses sont traduites en contraintes invariantes pour GATeL. Ces contraintes peuvent être trouvées dans l'annexe A. Au début il y avait une seule contrainte physique $C1$. Cette contrainte suppose trois situations: (1) la température de l'eau est comprise entre 0°C et 100°C; (2) le niveau d'eau dans le réservoir est compris entre 0 et le niveau maximum; (3) les variables booléennes représentant l'état de la pompe et le défaut de la pompe

ne peuvent pas prendre la valeur VRAI en même temps (si la variable d'état est vraie, cela signifie que la pompe n'a pas de défaut).

Sous la contrainte $C1$, cette unité structurelle a été trouvée accessible et une séquence de test contenant 3 cycles a été généré. Nous avons mesuré à nouveau la couverture MC/DC, y compris ce nouveau test généré. Le nouveau test a non seulement couvert l'unité considéré mais aussi il a augmenté le taux de couverture de 50% à 80% au niveau de l'opération racine. Cela signifie que d'autres unités structurelles non couvertes précédemment ont également été exécutées par le nouveau test.

Ensuite, une contrainte d'initialisation $C2$ a été ajoutée à la description de l'environnement. $C2$ suppose que dans l'état initial du système, tous les deux échangeurs de chaleur fonctionnent normalement. Nous avons rencontré TO au départ: GATeL a arrêté l'exécution sans donner de résultat. En fait, la configuration dans GATeL concernant le nombre maximum de cycles générés avait été fixé à 20, ce qui n'est pas suffisant sous les hypothèses $C_1 \wedge C_2$. Nous avons donc augmenté cette configuration à 30 et nous avons obtenu une séquence de test de 23 cycles. Le temps de calcul est 6391 secondes et la mémoire utilisée 272309 kilo-octets.

Ce résultat indique que la séquence de test à trois cycles obtenue la première fois n'est pas réaliste. Cela prouve que le raffinement en ajoutant progressivement des contraintes peut aider à améliorer la réalité fonctionnelle de test généré. Nous admettons qu'à partir d'un test en boucle ouverte de 23 cycles, construire un test en boucle fermée ne sera pas facile. Mais la profondeur du test suggère aussi que la conception manuelle d'un test couvrant l'unité structurelle peut être très difficile.

Vers la vérification hybride

Nous avons testé trois *model checkers* pour le langage Lustre: GATeL, Lesar et Kind 2, pour mettre en œuvre la méthodologie complète, y compris la vérification hybride en particulier. Lesar prend en entrée un model en Lustre V4; Kind2 accepte Lustre V4 étendu avec une partie de V6; alors que GATeL travaille sur une extension spécifique de Lustre V4. La première étape de cette expérimentation consiste à construire un modèle d'entrée correct pour chaque *model checker*. Nous avons commencé avec le modèle du contrôle-commande de SRI en Scade textuel, généré automatiquement à partir du modèle graphique par SCADE KCG (générateur de code). Ce modèle est ensuite traduit par l'outil s2d en une extension de Luster V4 directement acceptable par GATeL.

L'unité structurelle non couverte choisie pour l'expérience est à trois niveaux plus bas par rapport à l'opérateur racine. Par conséquent, les entrées et sorties de cette unité sont considérées comme variables internes ou locales. Cependant, pour Kind 2 et Lesar, il est nécessaire de spécifier une propriété concernant uniquement les entrées et les sorties de l'opérateur racine. Donc nous décidons de modifier le modèle graphique en Scade de manière à faire ressortir ces variables internes pertinentes aux sorties du niveau racine. Ensuite, en suivant la procédure ci-dessus, nous obtenons un modèle modifié du contrôle-commande de SRI pour GATeL.

Le modèle d'entrée pour GATeL n'est pas directement lisible pour Kind 2 et Lesar. Heureusement, le modèle pour GATeL ne contient pas de syntaxe non incluse ni dans Lustre V4 ni dans V6. Cependant, il doit encore être soumis à une modification de syntaxe incluant:

- Pour la définition de données complexes, par exemple, `type T_NS = {T1: real, T2: real, T` toutes les virgules `,` doivent être remplacées par des points-virgules `;`.
- GATeL permet d'écrire `assume expression_1` pour indiquer que `expression_1` est toujours vrai. Dans Lustre, la directive `assume` doit être remplacée par `assert` avec la même sémantique.
- Scade et GATeL offrent les ID utilisateur `#1`, `#2`, `#3` pour identifier différentes instances du même opérateur. Cette définition n'existe pas dans Lustre V4 ou V6.
- Dans Scade et GATeL, l'évaluation d'une variable structurée peut être effectuée à travers un opérateur `make` comme `make T = (v1, v2, v3)` où `T` est une variable de type `T_NS`. Dans Lustre, cela devrait écrire comme `T.T1=v1`; `T.T2=v2`; `T.T3=v3`.
- Scade et GATeL offrent un opérateur `caseof` comme une déclaration "switch". Dans Lustre,`caseof` doit être remplacé par plusieurs `if then else`.

Après ces modifications, le modèle GATeL a été traduit en modèles d'entrée pour Kind2 et Lesar. La figure 5.9 donne un aperçu de la relation entre les différents modèles.

Les trois *model checkers* répondent assez différemment au même problème:

- Lesar est à court de mémoire, c'est-à-dire Time Out. En fait, Lesar est un *model checker* qui gère exclusivement les valeurs booléennes. Il est donc clairement non adapté pour le modèle de contrôle-commande de SRI. Lesar n'enregistre pas les résultats temporaires pendant l'exploration de l'espace

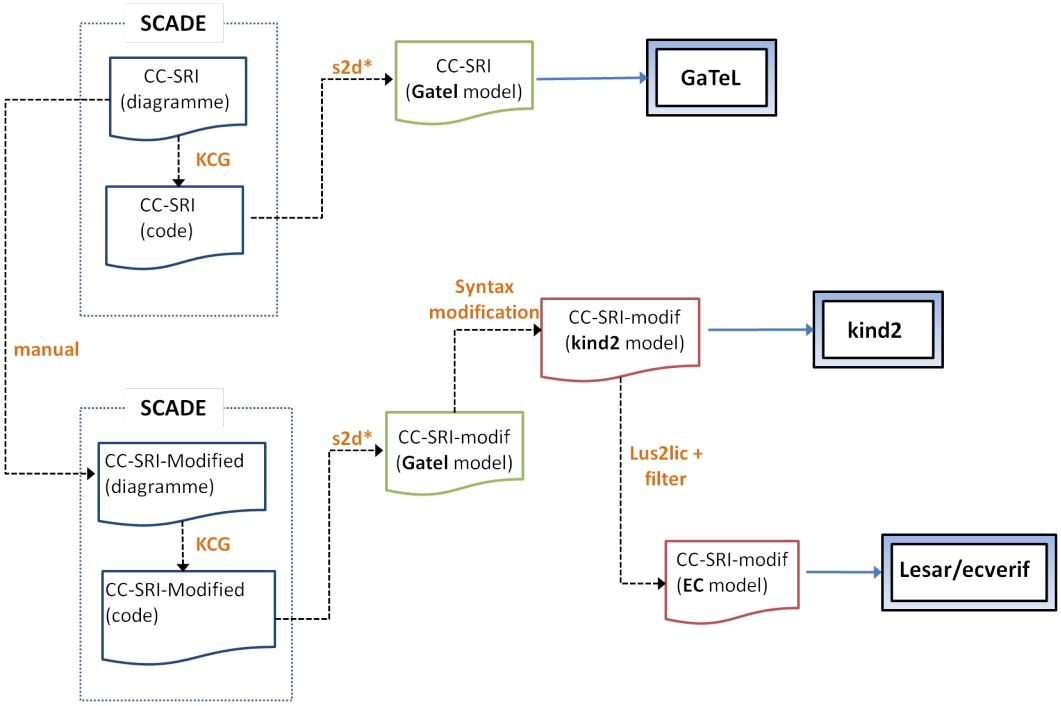


Figure 11: Applying different model checkers to the SRI CC

d'état. En conséquence, il est pas possible d'effectuer la vérification hybride, ce qui nécessite une mémorisation de traces d'exploration.

- GATeL réussit à générer des séquences de test qui améliorent la couverture structurelle, comme discuté dans la section précédente. Cependant, GATeL n'offre que la génération de test en arrière (backward) et donc il n'est pas adapté pour la vérification hybride.

- Kind 2 repose sur les solveurs SMT disponibles sur le marché. Les solveurs SMT ne sont pas adaptés pour les expressions non-linéaires telles que la multiplication/division des variables (présentes dans le modèle du contrôle-commande de SRI). Une solution consiste à remplacer ces expressions non linéaires par des abstractions définies dans des contrats de type "assume-grantee". Le modèle ainsi obtenu peut convenir à la vérification des propriétés, s'il est prouvé équivalent au modèle original. Mais notre objectif est la génération de test.

Nous avons également cherché des traducteurs de Lustre vers d'autres langages de modélisation afin de profiter d'autres outils puissants tels que NuSMV. Il existe un framework du traducteur Lustre qui est propriétaire de Rockwell Collins

et l'Université du Minnesota. Mais le projet n'est plus financé. Un autre traducteur académique de Lustre à SMV n'est plus maintenu. Concernant SCADE Design Verifier (basé sur le *model checker* Prover), sa licence n'était pas incluse dans "CONNEXION". Par conséquent nous sommes arrivés à la conclusion que la mise en œuvre de notre méthodologie complète nécessite une intégration de plusieurs outils. Une solution alternative est de développer un nouveau *model checker* avec les caractéristiques suivantes: génération en avant et arrière; vider les états inexplorés après TO; reprendre l'exploration après simulation.

Conclusion

La vérification et validation du système réactif temps-réel et critique sont soumises aux normes et certifications rigoureuses. Intégration des progrès récents dans l'ingénierie des systèmes dirigée par les modèles (MBSE) dans le cycle de vie de cette catégorie de systèmes constitue le contexte de cette thèse. Approches MBSE, telles que "Model-Based Design" (MBD) et "Model Based Testing" (MBT), encouragent la vérification dans les étapes amont de conception, permettant une détection des défauts le plus tôt possible. C'est très rentable pour de tels systèmes puisque le coût des défauts découverts dans le système réel peut être extrêmement élevé.

Nous proposons une méthodologie MBT dirigée par la couverture structurelle et exigences fonctionnelles. La méthodologie repose sur une utilisation répétitive de *model checker* pour générer des séquences de test en boucle ouverte basées sur la couverture. Le passage de boucle ouverte à boucle fermée nécessite une expertise des ingénieurs du système et n'est pas dans le champ de notre recherche. Cependant, nous proposons un raffinement de génération de test en ajoutant progressivement des contraintes de l'environnement. Ces contraintes sont dérivées des conditions physiques, des conditions d'initialisation et des exigences fonctionnelles. Sous ces contraintes, les séquences de test générées doivent respecter les comportements réalistes par rapport au système complet. Notre méthodologie considère également qu'un *model checker* peut TO en raison de problème d'explosion d'état et propose une heuristique dite vérification hybride qui combine le *model checking* et la simulation. Le principe est de collecter les états explorés au cycle de calcul juste avant le TO, sélectionner un état candidat qui devrait être le plus proche de l'état cible, et de commencer une simulation étape par étape à partir de cet état candidat en essayant d'atteindre l'état cible.

La méthodologie proposée dans cette thèse est très coûteuse et donc adaptée exclusivement pour les systèmes hautement critiques en sûreté de fonctionnemen-

t. Les activités de V&V de ce type de systèmes sont soumises à des normes rigoureuses. De tels systèmes nécessitent un niveau de couverture très élevé au-delà des exigences fonctionnelles. Notre méthodologie offre une solution pour automatiser autant que possible la génération de test basée sur la couverture, qui est un processus purement manuel dans l'approche d'ingénierie actuelle. Deuxièmement, la méthodologie inclut un raffinement de génération de test en boucle ouverte. Le passage de la boucle ouverte à la boucle fermée nécessite une collaboration des experts du système. Notre raffinement peut aider les experts du système à intervenir efficacement dans la préparation de cas de test fonctionnels en boucle fermée. Notre méthodologie est donc rentable car il permet aux experts du système de gagner du temps. Enfin nous avons examiné divers *model checkers* pour le langage Lustre pour mettre en œuvre la méthodologie. Nous arrivons à la conclusion qu'une intégration de plusieurs outils est nécessaire. À notre avis, il est également avantageux de revisiter les versions académiques de Lustre et les outils associés. Cependant, ces recommandations argumentées dépassent le travail de cette thèse.

Les travaux futurs concernent (1) la définition et éventuellement le développement d'un nouvel outil pour effectuer la vérification hybride; (2) la proposition de stratégies de sélection utilisées en vérification hybride, c'est-à-dire, comment définir une distance entre deux états dans l'espace d'état d'un système de transition; (3)un framework généralisé de raffinement de la génération de test en boucle ouverte; (4) l'exploration de la génération de test par Kind 2 en utilisant les contrats "assume-garantee" par rapport à la vérification hybride.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Our daily life relies more and more on all kinds of software and hardware systems. The complexity of these systems is also increasing rapidly. No longer standalone, these systems are typically embedded in a larger environment, connecting and interacting with other components. **Embedded systems**, first appeared in large industrial applications: factories, power plants, transportation systems, avionics and automotive. Nowadays they also concern a wide variety of everyday smart objects: cell phones, digital watches, microwave ovens, etc. Embedded system can be considered as a combination of software, hardware, and sometimes additional mechanical, electronic or other parts. These components are interconnected, designed to perform a dedicated function. This distinguishes an embedded system from a general-purpose computer for multiple tasks. An embedded computer is just one functional element of a system (another brick in the wall), rather than a stand-alone versatile computing machine. Embedded systems are vulnerable to errors since the number of defects grows exponentially with the number of interacting system components.

Most embedded systems are **reactive** systems and **real-time** systems. We call "reactive" systems that are permanently interacting with its (possibly physical) environment; and we reserve the term "real-time" for reactive systems that are subject to externally defined timing constraints, as suggested by Benveniste and Berry [26]. In other words, we distinguish two kinds of time constraints:

1. *Synchronous hypothesis* assumes that the system is able to react to an external event from its environment before any further event occurs. Synchronous hypothesis concerns the *input frequencies*, meaning between two successive inputs, the environment is considered unchanged.

2. *Real-time constraint* requires that the system is not only able to react to its environment but also to guarantee response within specified time constraints.

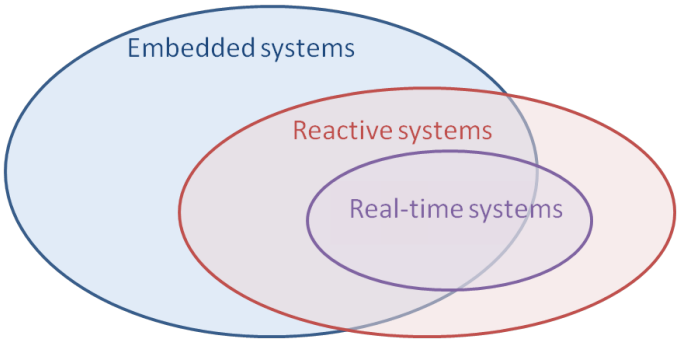


Figure 1.1: Embedded systems, reactive systems and real-time systems

Real-time constraint is imposed on the *input-output response time*.

Reactive systems are systems that satisfy the synchronous hypothesis. Real-time systems are reactive systems subject to the real-time constraint. The broad class of reactive systems contains therefore real-time applications as well as non-real-time applications, e.g., communication protocols, man-machine interfaces, etc. The relationship between different systems is illustrated in Fig. 1.1.

The notion of reactive systems was first introduced in the eighties [103, 143] to distinguish them from transformational systems and interactive systems. A reactive system receives inputs from the environment (sensors, human operators, etc) and responds with calculated outputs to command the environment (Fig. 1.2). The definition and characteristics of reactive systems have then been pointed out many times [35, 31, 97].

An **embedded system** is a computer system with a dedicated function, embedded as a part of a complete device system that includes hardware, such as electrical and mechanical components.

Reactive system maintains permanent interaction with its environment, meaning it receives inputs from the environment and responds with calculated outputs to command its environment.

Real-time system (RTS) is a *reactive system* interacting with an environment that **cannot wait**. The input/output calculation time must respect the time constraint imposed by its environment.

Highly **safety-critical** systems are usually real-time systems, such as control/-command system in a nuclear power plant or a commercial aircraft flight control system. This domain of application requires very careful design and very stringent verification and validation. Safety is crucial concerning this area, since a simple bug can produce extreme and catastrophic consequences. A real-time system is subject to at least two kinds of requirements:

- *Logical correctness*: the classical respect of the input/output specification is essential for all kinds of applications
- *Temporal correctness*: additional requirement for RTS. A logically correct RTS can fail to adequately control its environment if the outputs are not produced on time, i.e. the input/output computation time does not respect the time constraint imposed by its environment.

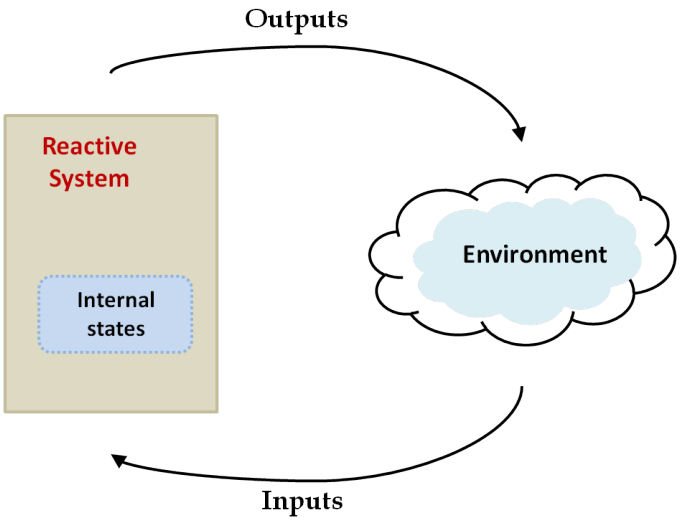


Figure 1.2: Reactive systems

Historically, the design of reactive and real-time systems have long been the occupation of control engineers. Automatic control theory has been applied to design systems with desired behavior, in a so-called "bottom-up" manner. Reactive and real-time applications evolved mostly from the use of analog machines and relay circuits to the use of microprocessors and computers. Various application fields are treated by practitioners having their own methods and vocabulary, although strongly technically related, little relation has been established between them.

When it comes to verification and validation, the task can be approached only late in the product development cycle. Physical prototypes of either complete systems or subsystems are connected to Hardware-in-the-Loop (HIL) simulation platforms, to investigate system-level behavior. For unit system with limited number of input and output variables, it is possible to test in an exhaustive manner. But the challenge remains to perform a systematic test and validation considering the huge number of situations that are relevant for complex embedded systems [160].

For a long time, engineering of reactive and real-time systems did not benefit from the recent progress in software and programming technology as much as did other fields. This situation must change rapidly, for the following reasons:

1) Modern applications will require strong interaction between different application fields, so specific terminologies and tools must be aligned for ontologies to keep large systems tractable.

2) Complex safety-critical systems usually have a long life cycle during which the systems are subject to partial changes and renovations "top-down" programming approach familiar to software engineers can be very helpful in improving the efficiency of validation and maintenance of this kind of systems. "Bottom-up" low-level programming techniques will remain acceptable during maintenance.

3) It will be necessary and sometimes even required to verify the correctness of programs using rigorous formal methods (Sec. 2.1), at least with respect to certain crucial safety properties. Application of formal methods requires mathematically rigorous concepts and programming tools, as well as automatic verification tools.

During the past half century, historical contributions have been made to establish model-based approaches for system description, design and analysis [71]. Based on a firm mathematical foundation, **model-based approaches** to system engineering support system design and analysis through machine readable models. Various graphical modeling languages have been developed by software and system engineers, in order to make models more visual and intuitive. The Unified Modeling Language (UML) [151] and the System Modeling Language (SysML) [10], for example, are two graphical machine-readable modeling languages adopted by the Object Management Group (OMG). In 2007, the International Council on Systems Engineering (INCOSE) launched the INCOSE MBSE Initiative[1], where **MBSE**

[1] `http://www.incose.org/ChaptersGroups/WorkingGroups/transformational/mbse-initiative`

denotes **Model-Based System Engineering**. Over the past two decades, various MBSE methodologies and supporting commercial modeling tools have been developed. A summary can be found in the INCOSE survey of MBSE methodologies [79].

> Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [7].

In a nutshell, MBSE is about elevating models in the engineering process to a central and governing role in the specification, design, integration and validation of a system [79]. In a model-based process, activities that support the engineering process are to be accomplished through development of increasingly detailed ("top-down") models. MBSE approaches include behavioral analysis, system architecture, requirement traceability, performance analysis, simulation, test, etc. This results in a transition from traditional document-centric system engineering approach to a model-centric practice as advocated by the INCOSE [7]. Shifting to MBSE enables software and system engineers to more readily understand design change impacts, communicate design intent and analyze a system design before it's built.

Various development lifecycle models have been created and applied to large-scale system and software development projects (e.g., the waterfall model [150], the spiral model [40], etc.) The V-model (or Vee-model) [84, 85] is considered as one of the most famous models of a system or software development lifecycle. Fig. 1.3 presents a simplified V-model describing system development lifecycle. The left side of "V" represents "top down" decomposition of requirements and creation of system specifications; whereas the right side of "V" represents integration of components/units, their verification and validation in a "bottom up" approach. In a nutshell, verification is always against the requirements (technical terms) and validation always against the real world or the user needs.

The V-model is a graphical presentation summarizing major steps in system development lifecycle. It represents one-to-one relationship between the documents on the left side and the test activities on the right side. However, to claim that verification and validation only occurs at the right side may not always be optimal. For example, requirements need to be validated first against the user needs and there is also some validation of system models that can partially be done at the left side. Based on the V-model, the W-model [156] shows how the tasks of

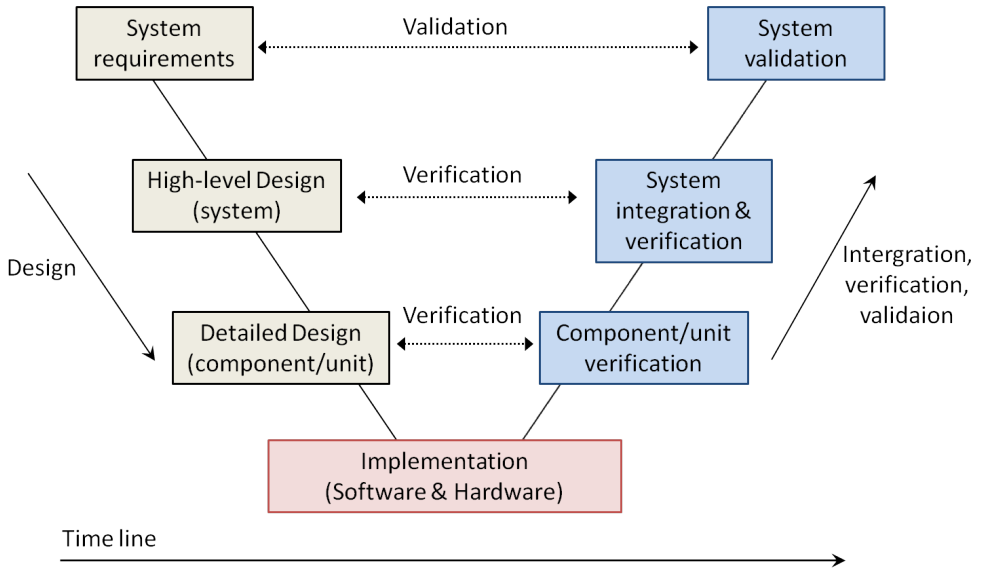


Figure 1.3: V-model: system development lifecycle

testing relate to the tasks in the development model. The purpose is to extend the V-model to support the broader view of testing as a continuously major activity throughout the system development lifecycle. The W-model (Fig. 1.4) clarifies the dependence between development and testing activities. The connection between various test stages and the basis for the test is clear with W-Model, which is not clear in the V-model. The W-model also points out that testing and debugging are different activities performed preferably by different persons.

The V-model and the W-model are clearly different; but they both show one thing: a system development lifecycle encompasses two kinds of activities: design and testing. Therefore integration of model-based approach to system development lifecycle naturally results in model-based design (MBD) [134] and model-based testing (MBT) [144, 166].

– Model-based design (MBD) refers to the use of domain specific modeling languages that can be executed and analyzed before the actual system is built. System engineers could create a model of the system with this kind of modeling language and then execute and analyze the model on their desktop rather than the target machine. With support of appropriate tools, the model can be used to automatically generate code as well as test cases.

– Model-based testing (MBT) is a method of testing that relies on abstract formal models describing the intended behavior of systems. Test cases can

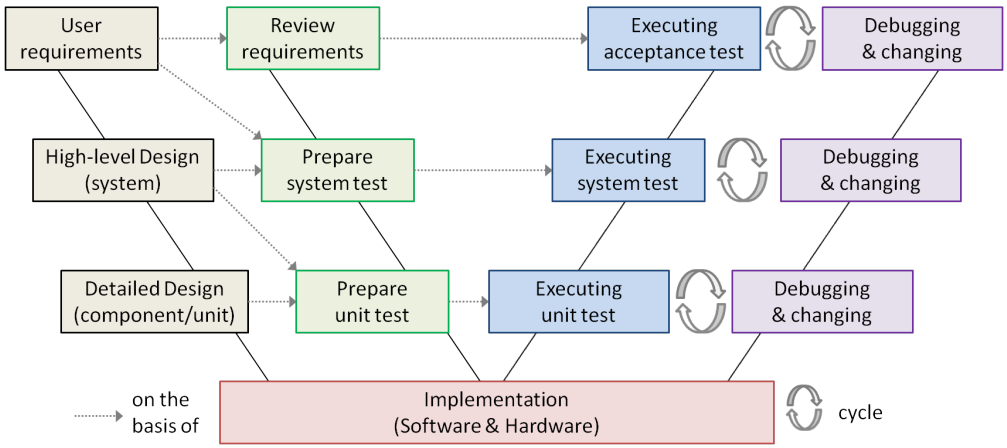


Figure 1.4: W-model: extension of V-model strengthening the bond between design and test

> be automatically generated from the model using specific tools, then adapted and executed on the system under test (SUT). The model used for test generation can be an abstraction of the SUT, or a specific model created in particular for this purpose, the so-called "environment model" [166].

The complexity of modern embedded systems keeps growing, same is the need to improve communication between development holders, to improve product quality as well as to reduce the verification and validation cost. All these new requirements result in the transition from document-centric approach to model-centric approach. Models should replace documents as the primary product or artefact of system engineering processes. The future of system engineering is, in our opinion, model-based.

Application of recent progress in MBSE and MBT in particular to industrial safety-critical real-time reactive system constitutes the background of this thesis. The verification and validation of these systems are subject to stringent standards and certifications [9]. Formal methods are sometimes necessary and even required, at least with respect to highly safety-critical properties. The thesis is part of the French methodology project "CONNEXION" in the nuclear domain. Profiting from a realistic industrial study case and a unique and complete set of modeling and verification tools, we had the opportunity to examine the difficulties of adopting modern MBSE techniques in a safety-critical industrial context. Moreover, MBSE encourages performing verification activities in the early design stages, allowing early detection of defects. This is very cost-effective for safety-critical RTS since the cost of defects found later in the actual system can be extremely high.

1.1 Problem statement

Since 2012, the main industrial and academic partners of the French nuclear industry initiated an ambitious R&D program called "CONNEXION". Regrouping a number of projects, "CONNEXION" [71] aims to improve the development process of the Instrumentation & Control (I&C) system of nuclear power plants (NPPs). "CONNEXION" is based on the existing expertises of major operators in French nuclear industry (EDF, ALSTOM, AREVA and RRCN) and various software tools provided by other partners (CEA, CORYS, ESTEREL Technologies and ALL4TEC).

The I&C system is the "central nervous system" of a nuclear power plant. As a safety-critical real-time system, the nuclear I&C system is required to be developed and validated under the strongest standards and certifications. With the current system engineering approach, increasingly detailed models describing the behavior of the I&C system have been built in early design phases. The models are specified in a formal language dedicated to the nuclear sector, using predefined blocks. The specification of this language complies with IEC (International Electrotechnical Commission) standard 61804-1 [4] and the implementation is defined by elementary blocks with reference to IEC standard 61131-3 [11].

During development phases, these models of the I&C system are subject to verification with respect to the functional requirements, which is defined as Functional Validation (FV) in IEC standard 61513 [9]. These models, specified in the previously mentioned language, are formally verifiable but not directly executable. As consequence, functional validation of these models are currently performed in a manual way. Automated testing is only performed at the implementation level. The actual approach, clearly fulfills the high quality requirements defined in the standards of nuclear sector. However the cost is very high considering the inherent evolution during the development process.

> Functional Validation (FV) [9], is actually an activity of verification: verification of model with respect to functional requirements.

Based on the conventional engineering approach, briefly presented above, "CONNEXION" gathers together a unique and complete set of modeling and verification tools to automate the functional validation as much as possible. In particular, the synchronous modeling language Lustre [52, 101] and its commercialized tool SCADE [33] have been chosen by "CONNEXION" to recreate equivalently the models of the I&C systems. Since SCADE models are machine readable and executable, they can be used to generate automatically test cases, with support of

appropriate tools. Another advantage is to apply formal methods and techniques (e.g. model-checking [64]) to these models, considering that there are quite a few formal verification tools for Lustre. "CONNEXION" has chosen GaTeL [126] from CEA List. Other tools for models specified in Lustre include Lesar [99, 100], Kind2 [57], jKind [89], just to name a few. Each of of these tools supports a slight different subset/extension of the Lustre language. The fact that Lustre is continuously being developed (different versions) on the academic side adds more complexity to the subject (see Sec. 2.4).

1.2 Thesis contributions

As part of the "CONNEXION" project, our research work benefits from the various tools provided by partners and an industrial case study of the nuclear I&C system. We propose a model-based testing (MBT) methodology, including related process, methods and tools, to enrich functional validation of the models of the I&C system. The proposed methodology is not restricted to the nuclear sector and it can be generalized and applied to other safety-critical real-time systems.

Our methodology relies mainly on model checking so let's briefly present this technique. Model checking [19] was originally proposed as a formal verification technique to evaluate the correctness of a model with respect to a given set of properties. Model checkers are software tools that automate model checking. A model checker considers every possible combination of input variables of the model, making the verification equivalent to exhaustive testing. If the property is found not true, the model checker returns a counterexample showing how the property is proven false.

During the last two decades, model checking has first proved effective in hardware verification [50, 83]; and more recently in software verification [113]. Model checking has also been applied to the verification and analysis of safety critical systems. In the avionic industry, model checking has been utilized in formal verification of important properties of flight control system [43, 132]. In the safety analysis [95] of an embedded control system of a tunnel tube, model checking has been used in proving functional correctness and assessing reliability of the system [138].

As for the nuclear industry, model checking has been applied to the Korean nuclear power plant automation systems [172, 118, 171]. Several critical logic errors have been identified in programmable logic controllers that had been specified

as function block diagrams. In the work of Lahtinen *et al.* [121], a systematic methodology for modeling function blocked based system designs in the nuclear domain has been proposed. Model checking has also been utilized to perform quantitative risk analysis [116].

As an enrichment and complement to these works, in this thesis model checking is applied to reinforce functional validation of early designs of nuclear control command system. Model checker is utilized in an iterative approach, as a test generation tool. Functional validation of the system is subject to requirements of functional correctness as well as of structural coverage criterion (MC/DC for example). Coverage-based test objectives are specified as "trap properties" for the model checker to force counterexample generation.

When use model checking on the specification of control/command system, the physical environment model permanently interacting with this specification must be taken into account. Due to their different characteristics, the physical environment and the control/command system are specified using different modeling languages. One solution would be rebuilding the environment with the same specification language of the control/command, as presented in [43]. But this was not the choice in the project "CONNEXION". As a result we decided to using assumptions about the environment to refine test generation by the model checker. We propose a technique to gradually adding these assumptions to get more and more realistic test. This helps system experts to pass from open-looped test (based only on structural coverage) to closed looped test (also related to functional requirements). Furthermore, our methodology also takes into account the "state explosion" problem of model checking and propose the "hybrid verification" heuristic. The heuristic combines both model checking and simulation (see 4.4). Finally, the project "CONNEXION" integrates multiple specifications at different abstraction levels as well as various model-based engineering tools. The project should be supported by an information system of traceability. We propose a design of such a system with UML and a prototype of implementation.

The methodology proposed in this thesis targets the functional validation of the models of a safety-critical RTS. The methodology consists of three major steps:

1. Prepare a set of test cases fulfilling the properties derived from the functional requirements of the system under test (SUT). The test generation can be automated with support of appropriate tools.

2. Execute these test cases on the model of the SUT and measuring the structure coverage of these test cases. Structure coverage and functional requirement coverage are not the same thing. A set of test cases covering 100% of

functional requirements does not necessarily execute every structural part (e.g. a line of code) of the model.

3. For all uncovered structural part of the model, "testing with model checker" approach is applied. A"trap property" [91] saying that this part can never be executed is used to force the model checker to generate a counterexample.

The step 3 can produce several outcomes: (a) the model checker does not generate a counterexample, i.e., there is no test that can execute the structural part under consideration. Further analysis is required since this is a sign of eventual defects. (b) the model checker produces a counterexample which can be adapted to a test case for the model. This test case enriches the original set of test by improving its structural coverage. The structural part under consideration, and possible other structural parts should be executed by the new test case. (c) the model checker has a "time out" (TO): it stops the calculation without giving an answer. "Time out" is usually due to the fundamental "state-explosion" problem of model checking. However, the "time out" situation is rarely explored in the literature on testing with model checkers.

In our methodology, we consider the TO of a model checker and propose a heuristic referred to as hybrid verification: to combine model checking and simulation. Model checking explores the entire state space while hybrid verification explores only a subset of all possible states. We also propose the elements for putting hybrid verification into practice. These elements include in particular:

- Looking for a powerful and comprehensive model checking tool for the Lustre language to perform hybrid verification. We review different versions of the Lustre language and test several academic model checkers on the "CONNEXION" case study.

- Considering translating a model specified in Lustre to another modeling language (e.g. SMV) to benefit from other model checking tools.

We come to the conclusion that to apply hybrid verification to an industrial-size Lustre model, a single model checking tool may not be enough. Hybrid verification combines several techniques: exhaustive state-space exploration, memorizing exploration traces, forward/backward trace generation, step-by-step simulation, etc. An integration of several tools is one solution.

The methodology presented in this thesis applies to safety-critical real-time systems. Let's recall the fact that real-time systems are at first reactive systems (Fig.1.2). They are interacting permanently to a physical environment. The RTS

and its environment are usually specified using different languages due to their different nature and characteristics, which is the case in "CONNEXION". This results in two kinds of test and simulation:

- **open-loop** simulation is execution of the RTS itself. Open-loop test is dedicated for this kind of execution.
- **closed-loop** simulation is co-execution of the RTS as well as its environment. Closed-loop test is dedicated for this kind of co-execution.

In "CONNEXION", models of the I&C system and its physical environment are specified using different languages. These models are integrated through a platform to perform closed-loop simulation. Various models exchange data and synchronize with each other via the Functional Mock-up Interface (FMI) [39].

It is important to precise that a model checker usually accepts one modeling language. In our case, the model checker takes as input the Lustre model of the I&C system. Counterexamples generated by the model checker can be interpreted as open-loop test, i.e., executable only on the I&C system. An open-loop test does not take into account how the environment reacts to the outputs produced by the I&C system. Our research work concentrates on using model checkers to produce open-loop counterexample. How to develop closed-loop test from open-loop data requires expertise from system engineers and is not in the scope of the thesis. In the practice, closed-loop test is more realistic and valuable than open-loop test and sometimes necessary or even required. Therefore our methodology proposes some techniques to help with the passage from open loop to closed loop. These techniques are based on formalizing properties describing the physical environment as invariant constraints for the model checker and using these properties to progressively refine the generation of counterexamples (Sec. 4.2).

Most safety-critical real-time systems have a long lifecycle, several decades in the case of nuclear I&C system. During this time, both the I&C system and its environment need to evolve, e.g., modification of a functional requirement of the I&C system, addition of a new sensor to its physical environment, etc. As a result, some partial renovations and limited modifications are to be made to the system and/or its environment. The "new" system needs to, without any doubt, be verified and validated. The techniques previously-mentioned allow to perform the new V&V more efficiently. Consider that a property concerning the environment has been changed. The modification can be taken into account through simply changing certain constraints for counterexample generation by model checking. Although the I&C system and its environment evolve independently, they are both considered for new test generation.

1.3 Thesis organization

This thesis is organized as follows:

- Chapter 2 is a state of art on model checking and its applications, as well as synchronous modeling languages, which ends up focusing on Lustre (language chosen in "CONNEXION").
  - Sec. 2.1 introduces model checking preliminaries including transition systems, temporal logic and properties and model checking algorithms.
  - Sec. 2.2 discusses the application of model checking in testing as a model-based testing approach. Traditional model checking problem (property verification) can be transformed to a test generation problem and model checkers can therefore be used to automatically generate complete and large sets of test cases.
  - Sec. 2.3 presents synchronous approaches since they are commonly-accepted specification for real-time systems. Three French academic synchronous languages (Esterel, Signal and Lustre) are briefly discussed and compared, with a special focus on the Lustre language (choice of "CONNEXION").
  - Sec. 2.4 is a short story of the Lustre language: different academic versions, model checking tools for Lustre models, translators of Lustre to other modeling languages, etc.
- Chapter 3 is dedicated to the project "CONNEXION".
  - Sec. 3.1 presents the triple V-model of the innovative system engineering lifecycle proposed in "CONNEXION". Different from the V-model (Fig. 1.3) and W-model (Fig. 1.4), the triple V-model introduces functional validation at multiple levels during development phases.
  - As a real-time system, the I&C system is in permanent interaction with a physical environment. Models of the I&C system and its environment are specified in different languages, which leads to the notions of open-loop and closed-loop. Sec. 3.3 discusses the additional challenges due to this characteristic.
  - Partners of "CONNEXION" provide a complete and unique set of modeling, verification and simulation tools. A brief presentation of these tools are given in Sec. 3.2.
- Chapter 4 presents the model-based methodology to enrich functional validation.

  - Our general methodology is presented in Sec. 4.1, including related processes, methods and tools. Application of the methodology to a simple example with support of "CONNEXION" tools is also discussed.
  - Sec. 4.2 explains the techniques to support the passage from open-loop counterexamples to closed-loop test cases.
  - Sec. 4.1.3 discusses the "hybrid verification" as an heuristic to address the time out of model checking.

- Chapter 5 is about the "CONNEXION" case study and related experiment results.
  - Sec. 5.1 gives a description of the case study SRI including its functions, requirements and the physical environment to which it permanently reacts.
  - Experiment results of applying our methodology to SRI using "CONNEXION" tools is discussed in Sec. 5.2.
  - To put hybrid verification into practice, three model checkers for Lustre models are tested on the SRI model. Sec. 5.3 covers this aspect.

- Chapter 6 concludes the thesis with a synthesis of the proposed methodology and experiment results as well as a discussion of future works.

Chapter 2

Model checking and its application

In the context of software and hardware systems, verification usually requires more time and effort than development. In a typical commercial development organization, the cost of debugging, testing, and verification activities can easily range from 50% to 75% of the total development cost [96].

System verification and validation are independent procedures used for checking that a system meets the requirements and specifications and that it fulfills its intended purposes. Although often confused, verification and validation are not the same thing.

> Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed **at the start of that phase** [2].

> Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [2].

As Boehm [41] points out:

- Verification: "Have we made what we were trying to make ?", i.e., does the product conform to the specifications?
- Validation: "Are we trying to make the right thing ?", i.e., is the product specified to the user's actual needs?

Formal methods are a particular kind of techniques to apply mathematically based methods for the specification, development and verification of systems. The aim is to prove the correctness (or incorrectness) of the system under consideration in a mathematically rigorous manner. The use of formal methods for software and hardware design is expected to contribute to the reliability and robustness of the design. As Dijkstra points out: "Testing shows presence not the absence of bugs". Formal methods aim at proving the absence of bugs.

Research in formal methods during the last two decades has resulted in the development of some promising verification techniques. These techniques offer a large potential in early detection of defects in the design process. They are also supported by powerful software tools that can be used to automate various verification steps and thus reduce verification cost. The great potential of formal methods has led to an increasing use by engineers for the verification of complex systems, especially in the case of safety related systems. According to standards of the IEC (International Electrotechnical Commission) and the ESA (European Space Agency), formal methods are one of the "highly recommended" verification techniques for safety-critical systems.

In the early 1980's, model checking is introduced as a formal verification technique. It originates from the independent work of two pairs: Clarke and Emmerson [61]; Sifakis and Queille [145]. The motivation was to encompass concurrent systems [63, 124] and to avoid the difficulties with manual deductive proofs [74]. Starting with a formal specification of the system (i.e. system model), model checking is a verification technique that explores all possible states of the system. The aim is to prove with mathematical rigor whether the system model truly satisfies a certain property. Model checker is the software tool that automates model checking. State-of-the art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space-exploration algorithms. Larger state spaces, 10^{20} up to even 10^{476} states, can be handled for specific problems, using clever algorithms and tailored data structures [19]. However, state explosion problem remains a big and fundamental problem of model checking. Due to state explosion problem, model checkers can "time out" (TO): end the calculation without giving an answer to the verification problem. In recent years, model checking techniques have raised the interest of many safety-critical industries such as railway signaling [42] and avionics [43, 133].

A model checker takes as input an automaton-based model of the system and a property formalized as temporal logic formula. The system model is usually automatically generated from a model description specified in some dialect of pro-

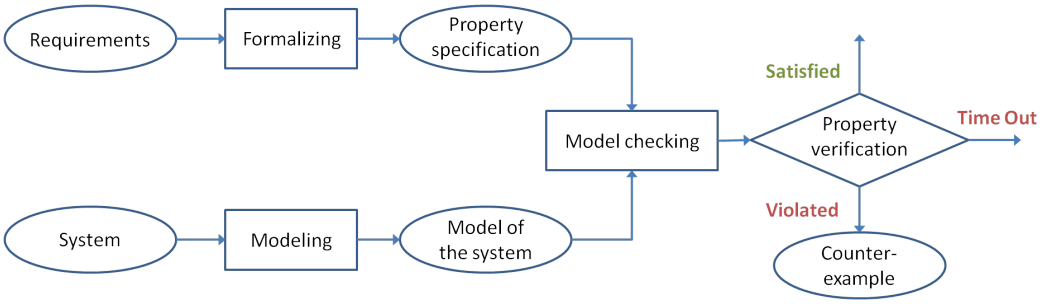


Figure 2.1: Model checking approach

gramming languages, such as Verilog or VHDL for hardware description and C or Java for software description. The property specification prescribes **what** the system should or should not do; whereas the model describes **how** the system behaves. The model checker then tries to explore the entire state space of the system model to determine whether the property is satisfied. If the state space exploration shows no property violations, then correctness with regard to the property is formally proved. Should a property violation be detected, the model checker is able to return a counterexample illustrating how the violation happens. With the help of a simulator, the violation scenarios can be played on the system model and useful debugging information can be obtained (see Fig. 2.1). The error may exist in the system model, or in the property specification or even in the model checker itself.

The complexity of a model checking problem depends on the complexity of the model. The third possible outcome of a model checking problem is "time out" (TO). The model checker terminates calculation without giving an answer to the property verification. TO is related to the fundamental state explosion problem of model checking. In fact, the state space grows exponentially with the number of variables of the model and quickly becomes too large for the model checker to explore. In the literature, the time out problem is naturally regarded as the result of state explosion and therefore rarely explored.

As a formal verification tool, model checkers are capable of providing counterexample illustrating property violations. Normally these counterexamples are meant to guide the analysis of discovering and locating defects in the system's specification. However they can also be very useful when interpreted as test cases, which has led to the idea of testing with model checkers. The main challenge is to force the model checker to automatically and systematically generate counterexamples; these counterexamples can then be used as test cases executable on the system model to fulfill specific testing purposes. In the late 1990', Callahan *et al.*

[51] and Engels *et al.* [77] initially proposed to use model checkers for the automated generation of test cases. Two decades of research on testing with model checkers has resulted in many different techniques of test generation, many problems have been solved, yet many issues remain.

This chapter is organized as follows: in Sec. 2.1 basic concepts regarding model checking technique are presented; Sec. 2.2 discusses several main stream approaches of automated test generation with model checkers; Sec. 2.3 presents and compares three synchronous modeling languages (Esterel, Signal, Lustre) for specifying real-time systems. We focus on the Lustre language as it has been chosen as the model description language in "CONNEXION"; Sec. 2.4 reviews the the story of Lustre including different academic versions, model checking tools, translator of Lustre to other modeling languages, etc.

2.1 Model checking preliminaries

2.1.1 Transition systems

The first prerequisite for model checking is a model of the system under consideration. In computer science, transition systems are often used as models to describe the behavior of hardware and software systems. They are illustrated by directed graphs where nodes represent *states* and edges state *transitions*. A state describes some information about a system at a certain moment of its behavior. Transitions specify how the system can evolve from one state to another. The system is only in one state at a time, the state it is in at any given time is called *current state*.

In the following we include *transition labels* for the transitions and *atomic propositions* for the states. Transition labels typically represent input expected, conditions to trigger the transition, or actions performed during the transition. Atomic propositions express known facts about the states of the system under consideration.

Definition 9. *Transition System.*

A transition system TS is a tuple (S, E, T, S_0, AP, L):

- *S is a set of states;*
- *E is a set of labels;*
- *$T \subseteq S \times E \times S$ is a set of labeled transitions.*

- $S_0 \subseteq S$ *is a set of initial states;*
- AP *is a set of atomic propositions;*
- $L : S \to 2^{AP}$ *is a labeling function mapping each state to a set of atomic propositions that holds in this state.*[1]

TS is called finite if S, E and AP are finite.

For convenience, we write $p \xrightarrow{e} q$ instead of $(p, e, q) \in T$, where $p, q \in S$ and $e \in E$.

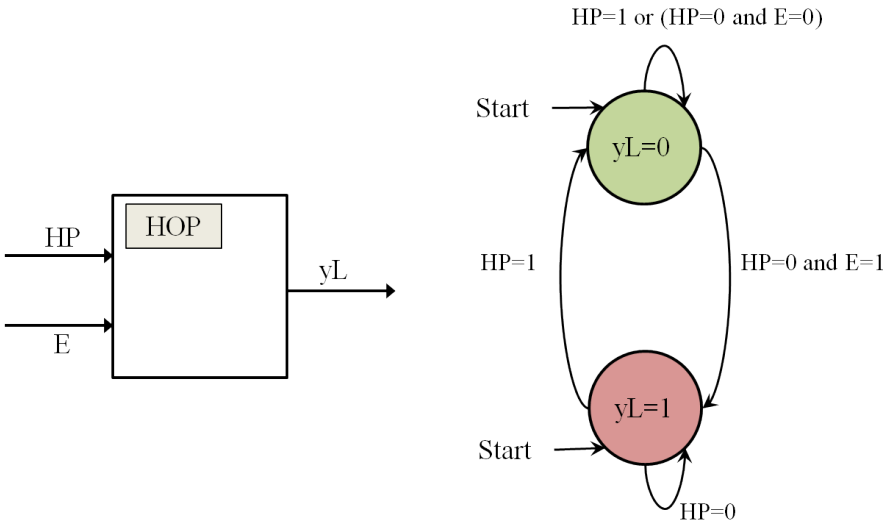
The labeling function L relates a set $L(s) \in 2^{AP}$ of atomic propositions to a state $s \in S$. $L(s)$ denotes those atomic propositions which are satisfied by state s. We know formally define the notion of "a formula satisfied in a given state". Given that Φ is a propositional logic formula, then s satisfies the formula Φ if the evaluation induced by $L(s)$ makes the the formula Φ true: $s \models \Phi$ iff $L(s) \models \Phi$.

The definition of transition system given above includes labels for both transitions and states. This definition encompasses Kripke structure [120], another formalism commonly used to describe model checking. A Kripke structure is basically a transition system where the labels of transitions are omitted and only the labels of states are kept.

Fig. 2.2 is a graphical representation of a very simple transition system, in a block diagram style at left and in an automaton style at right. This example is a function referred to as "HOP", extracted from the case study SRI discussed in Chapter 5. The system has three boolean variables, two inputs HP, E and one output yL. The system has two states based on the output yL. Note that there are two initial states as we do not make any assumption about the initial values of the three variables. For boolean variables, the value *true* is denoted by 1 and the value *false* denoted by 0. The transitions from the state $yL = 1$ are only related to the input variable HP: if $HP = 0$ then the system stays in this state, otherwise the systems "jumps" to another state $yL = 0$. The input variable E is irrelevant in this case. When the system is the state $yL = 0$, it will change state only when the next input vector is $(HP = 0, E = 1)$, otherwise it stays in the same state. The evaluation of input variables can also be used as transition labels.

In the case study SRI, the function HOP plays an important role in regulating water level in a tank. The input variable HP is the result of comparing measured

[1]Recall that 2^{AP} denotes the power set of AP.



a. Block diagram style b. Automaton style

Figure 2.2: Graphical representation of a finite transition system

water level in the tank with a predefined constant *max* (the high threshold). The input variable E is the result of comparing measured water level in the tank with a predefined constant *min* (the low threshold). $HP = 1$ and $E = 0$ denote that the water level in the tank already exceeds the high threshold, which results in the output $yL = 0$ (stop adding water into the tank). $HP = 0$ and $E = 1$ denote that the water level in the tank is lower than the low threshold, which results in $yL = 1$ (start adding water into the tank). All possible combinations of the input variables are summarized in Table 2.1.

According to the formal definition of transition system, we should define a set of atomic propositions and a labeling function mapping each state to those atomic propositions that hold and only hold in the state. The atomic propositions in the transition system depend on the properties under consideration. A simple choice is to let the state names act as atomic propositions, i.e., $L(s) = s$. However, if we want to verify the relation between the input variables HP, E and the output variable yL, the atomic proposition "the input variable E that the system has just received can never be 1" holds and only holds in the state $yL = 1$.

This example illustrates a certain arbitrariness concerning the choice of atomic propositions and transition labels. They will be casually dealt in the following.

HOP can be unwinded into an infinite execution tree (Fig. 2.3). Now we for-

Inputs	Meaning	Output
HP=1 and E=0	water level $>max$	yL=0
HP=0 and E=0	water level is normal	yL does not change value
HP=0 and E=1	water level $<min$	yL=1
HP=1 and E=1	practically infeasible	yL=0

Table 2.1: Application of function HOP in regulating water level

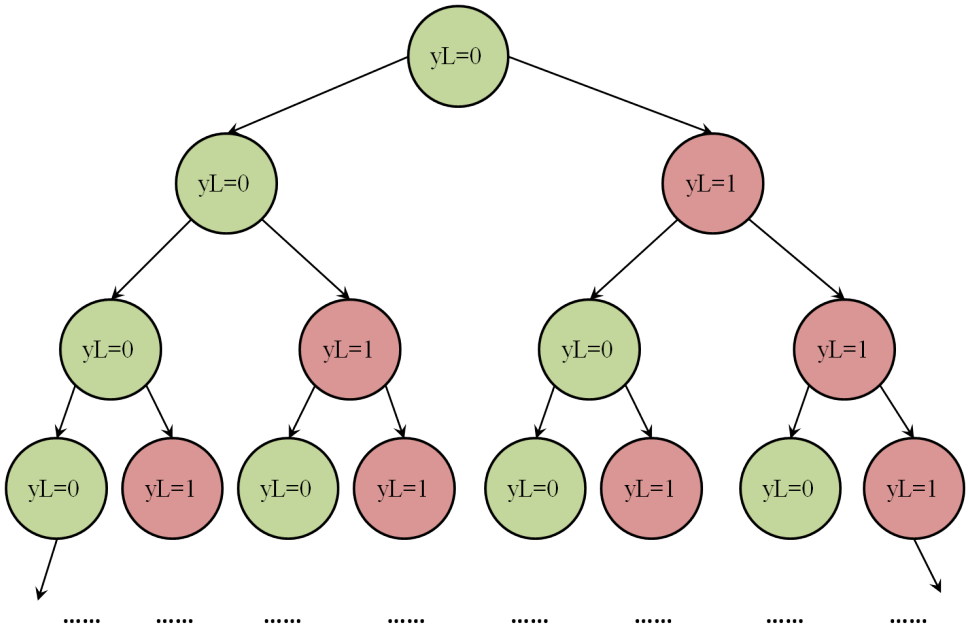


Figure 2.3: Execution tree of the transition system HOP

malize the behaviors of a transition system using the notion of *path* and *execution*.

Definition 10. *Path. A path of a transition system* $TS = (S, E, T, S_0, AP, L)$ *is a sequence* $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ *such that* $\forall i \geq 0, (s_i, e_i, s_{i+1}) \in T$. *The length of a path* σ, *denoted* $|\sigma|$ *can be finite or infinite.*

Definition 11. *Execution.*
A partial execution of a transition system $TS = (S, E, T, S_0, AP, L)$ *is a path starting from the initial state* s_0, *where* $s_0 \in S_0$. *The length of a partial execution can be finite or infinite. An execution is called maximal when it can not be extended. It is either infinite, or ends in a state out of which no transition is possible (we speak of a deadlock in this case). An execution can be seen as a "branch" in the execution tree.*

We conclude these definitions by introducing the notion of *reachable* state. A state is said to be reachable if there exists at least one execution in which it appears. A model called parallel program schema for the representation and study of programs containing parallel sequencing has been presented [115] including formal definition of reachability sets.

2.1.2 Temporal Logic and properties

Model checking consists in verifying with mathematical rigor the system model with respect to some properties of the system. Both the system model and the properties to verify need to be described in a precise and unambiguous manner. Temporal logic is a mathematical formalism tailored for statements and reasoning where time is involved. It offers special operators for time, fairly close to natural language statements (the adverbs "always", "until", etc). It also comes with a formal semantics, making temporal logic an indispensable tool for formalizing properties concerning dynamic behaviors of a system.

The most common temporal logics are the LTL (Linear Time Logic) [142] and the CTL (Computation Tree Logic) [62]. Time can either be interpreted to be linear (LTL) or branching (CTL). Both logics are subsets of CTL*, introduced by Emerson and Halpern [75]. Most model checkers support either LTL or CTL, or sometimes both. Other temporal logics used for model checking are HML (Hennessy-Milner Logic) [104], Modal μ-calculus [119], and different flavors of CTL such as timed [14], fair [76] or action [137] CTL.

The logic CTL* served to formally state properties concerned with the executions of a transition system. It consists of the following elements:

1. Atomic propositions $\phi \in AP$.
2. Boolean operators:
   - Constants $true$ and $false$.
   - Logical "not" $\neg$, logical "or" $\vee$, logical "and" $\wedge$.
   - Logical implication $\Rightarrow$ and double implication $\Leftrightarrow$. For ϕ, $\psi \in AP$, $\phi \Rightarrow \psi$ means if ϕ then ψ; $\Leftrightarrow$ states "if and only if".
   - Temporal operators **X**, **F**, **G** and **U**. $\mathbf{X}\phi$ states that ϕ holds for the next state; $\mathbf{F}\phi$ states that a future state satisfies ϕ without specifying which state; and $\mathbf{G}\phi$ that all the future states satisfy ϕ, i.e ϕ will always hold.

CTL* syntax		CTL	LTL
Atomic Propositions	$\phi \in AP$	✓	✓
Boolean operators	$true$, $false$, $\neg$, $\vee$, $\wedge$	✓	✓
Logical implication	$\Rightarrow$, $\Leftrightarrow$	✓	✓
Temporal operators	**X**, **F**, **G**, **U**	✓	✓
Path quantifiers	**A**, **E**	✓	✗

Table 2.2: CTL and LTL syntax comparison

U is the "until" operator. For $\phi, \psi \in AP$, $\phi \mathrm{U} \psi$ states that ϕ holds true until a state where ψ is satisfied.

- Path quantifiers **A** and **E**. The logic presented above deals with only one path of execution. There remains to express that several paths are possible starting from a given state. The path quantifiers allow one to quantify over the set of executions. $\mathbf{A}\phi$ states that all the executions out of the current state satisfy ϕ; whereas $\mathbf{E}\phi$ states that there exists an execution out of the current state where ϕ holds.

CTL adopts all the boolean and temporal operators as well as the path quantifiers listed just above. It is the subset of CTL* obtained by requiring that each temporal operator is immediately preceded by a path quantifier. This means that CTL formulas are state formulas, i.e. true in a specific state. In contrast LTL contains no path quantifiers and therefore LTL formulas are considered to be path formulas.

Definition 12. *CTL Syntax. The concepts presented above lead to the following formal grammar of CTL:*

$$\begin{aligned}\phi, \psi ::= & a \in AP | \neg\phi | \phi \vee \psi | \phi \wedge \psi | \\ & \boldsymbol{AX}\phi | \boldsymbol{AF}\phi | \boldsymbol{AG}\phi | \boldsymbol{A}(\phi \, \boldsymbol{U} \psi) | \\ & \boldsymbol{EX}\phi | \boldsymbol{EF}\phi | \boldsymbol{EG}\phi | \boldsymbol{E}(\phi \, \boldsymbol{U} \psi)\end{aligned}$$

The CTL semantics can be expressed by satisfaction relations for state formulas. $TS, s \models \phi$ denotes that a state formula ϕ is satisfied in state s of a transition system TS. $Paths(TS, s)$ denotes the set of paths starting from the state s of a transition system TS. For a path σ, σ_0 is its initial state and σ_i is the ith state with $i \in \mathbb{N}$.

Definition 13. *CTL semantics. Satisfaction of CTL formulas by a state $s \in S$ of a transition system $TS = (S, E, T, S_0, AP, L)$ is defined as follows, where $a \in AP$,*

$$
\begin{aligned}
TS, s \models a &\iff a \in L(s) \wedge s \in S \\
TS, s \models \neg\phi &\iff \neg(TS, s \models \phi) \\
TS, s \models \phi \vee \psi &\iff (TS, s \models \phi) \vee (TS, s \models \psi) \\
TS, s \models \phi \wedge \psi &\iff (TS, s \models \phi) \wedge (TS, s \models \psi) \\
TS, s \models \boldsymbol{AX}\phi &\iff \forall \sigma \in Paths(TS, s) : TS, \sigma_1 \models \phi \\
TS, s \models \boldsymbol{AF}\phi &\iff \forall \sigma \in Paths(TS, s) : \exists i \in \mathbb{N} : TS, \sigma_i \models \phi \\
TS, s \models \boldsymbol{AG}\phi &\iff \forall \sigma \in Paths(TS, s) : \forall i \in \mathbb{N} : TS, \sigma_i \models \phi \\
TS, s \models \boldsymbol{A}(\phi \boldsymbol{U} \psi) &\iff \forall \sigma \in Paths(TS, s) : \exists i \in \mathbb{N} : i \geq 0 \wedge \forall j < i : TS, \sigma_j \models \phi \wedge TS, \sigma_i \models \psi \\
TS, s \models \boldsymbol{EX}\phi &\iff \forall \sigma \in Paths(TS, s) : TS, \sigma_1 \models \phi \\
TS, s \models \boldsymbol{EF}\phi &\iff \forall \sigma \in Paths(TS, s) : \exists i \in \mathbb{N} : TS, \sigma_i \models \phi \\
TS, s \models \boldsymbol{EG}\phi &\iff \forall \sigma \in Paths(TS, s) : \forall i \in \mathbb{N} : TS, \sigma_i \models \phi \\
TS, s \models \boldsymbol{E}(\phi \boldsymbol{U} \psi) &\iff \forall \sigma \in Paths(TS, s) : \exists i \in \mathbb{N} : i \geq 0 \wedge \forall j < i : TS, \sigma_j \models \phi \wedge TS, \sigma_i \models \psi
\end{aligned}
$$

The difference between A and E is illustrated in Fig. 2.4.
The two most common categories of properties are:

- **Safety properties**: a safety property expresses that, under certain conditions, some event (usually something bad) should never occur.
- **Liveness properties**: a liveness property expresses that, under certain conditions, some event (usually something good) will eventually occur.

To formally express the two types of properties using temporal logic CTL, a safety property $\mathbf{AG}\neg\phi$ indicates that some "bad thing" ϕ will never happen; a liveness property can be written as $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)$, where ψ represents the "good thing" that ultimately happens whenever ϕ is true.

2.1.3 Model checking algorithms

Given a model M and a property ϕ of the system under consideration, the model checking algorithm answers the question "whether M satisfies the property ϕ ?" in a mathematically formal manner. Should a property violation detected, a model checker is capable to return a counterexample, illustrating how the violation occurs. Several different algorithms have been proposed, for property specification in different temporal logics, and using different data structures. As mentioned before, LTL formulas are considered to be path formulas as well as CTL formulas state formulas. Consequently, counterexamples for LTL formulas are also linear

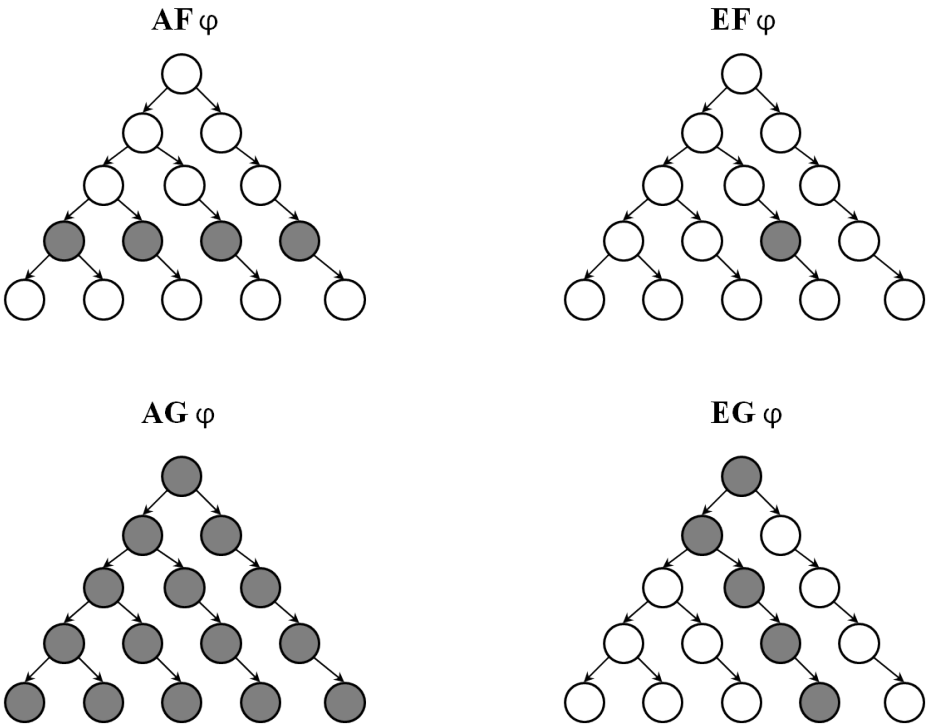


Figure 2.4: Comparison of path quantifiers **A** and **E**

sequences; whereas counterexamples for CTL formulas are a set of states and special algorithms [60] are then used to derive example paths.

Explicit (state) model checking is the first generation of successful model checking algorithms. Different approaches for LTL-based properties [124, 167] and CTL-based properties [63, 145] have been proposed. Explicit model checking explores explicitly the state space of a model and searches by forward exploration until a property violation is founded. In LTL model checking, the exploration algorithm can be depth-first or breadth-first: breadth-first approach finds the shortest possible counterexamples but demands significantly higher in memory use than depth-first approach. In CTL model checking, the set of all the states satisfying the given property are determined by recursively calculating the satisfied subformulas for each state. If all states are visited and no violation is detected, the model is considered to satisfy the property.

It is not surprising to see that explicit model checking approaches quickly encounters the state explosion problem. The second generation of model checking algorithms, *symbolic model checking* [129], uses a particular data structure called

Model checker	Input model language	Model checking technique
SPIN	Promela	Explicit
Kronos, Open-Kronos	Timed automata	Symbolic
SMV	SMV	
NuSMV	SMV	Symbolic and bounded (SMT based)
SAL	SAL	

Table 2.3: A summarize of some model checkers for different languages and techniques

BDD (Binary Decision Diagrams) [47, 48]. The application of BDD to model checking was suggested by three different groups [66, 50, 141]. BDD is used to represent more efficiently state sets and function relations on these states and thus allows the representation of significantly larger state space.

Bounded model checking [37] is the third generation of model checking algorithms. The model checking problem is reformulated to a constraint satisfaction problem (CSP), which allows using the propositional satisfiability (SAT) solvers to calculate counterexamples to a certain upper bound. Bounded model checking and traditional model checking techniques supplement each other: bounded model checking has been successfully applied to systems where traditional model checking fails; there are also many cases where bounded model checking fails while symbolic model checking is efficient.

Regarding the tools for different model checking techniques, SPIN (Simple Promela Interpreter) developed by Holzmann and Murray [105], is a commonly used explicit model checker. Kronos [45] and its successor Open-Kronos[164], SMV [128] and its deriviative NuSMV [59], as well as SAL (Symbolic Analysis Laboratory) [68], these model checkers perform symbolic model checking. NuSMV and SAL also support bounded model checking. All these tools are open source and available via Internet. Table 2.3 gives a summarize of model checkers based on different model checking techniques.

2.2 Testing with model checkers

The main idea of testing with model checkers is to force model checkers to systematically generate counterexamples and then interpret these counterexamples as test cases. Model checking is originally proposed as a formal verification technique and a model checking problem is normally a property verification problem. On the

other hand, test cases are related to certain test purposes, describing the desired characteristics of test cases. If the test purposes can be specified in temporal logic and then used as properties to force the model checker to generate counterexamples, the test generation problem is actually transformed to a "traditional" model checking problem (see Fig. 2.5).

Definition 14. *Test case.*

A test case t related to a transition system $TS = (S, E, T, S_0, AP, L)$ is a finite sequence $\langle s_0, s_1, \ldots, s_n \rangle$ where $\forall 0 \leq i < n, (s_i, e_i, s_{i+1}) \in T$ and $s_0 \in S_0$.

Definition 15. *Test suite.*

A test suite is a finite set of n test cases. The size of the test suite is n. The length of the test suite is the sum of the lengths of its n test cases.

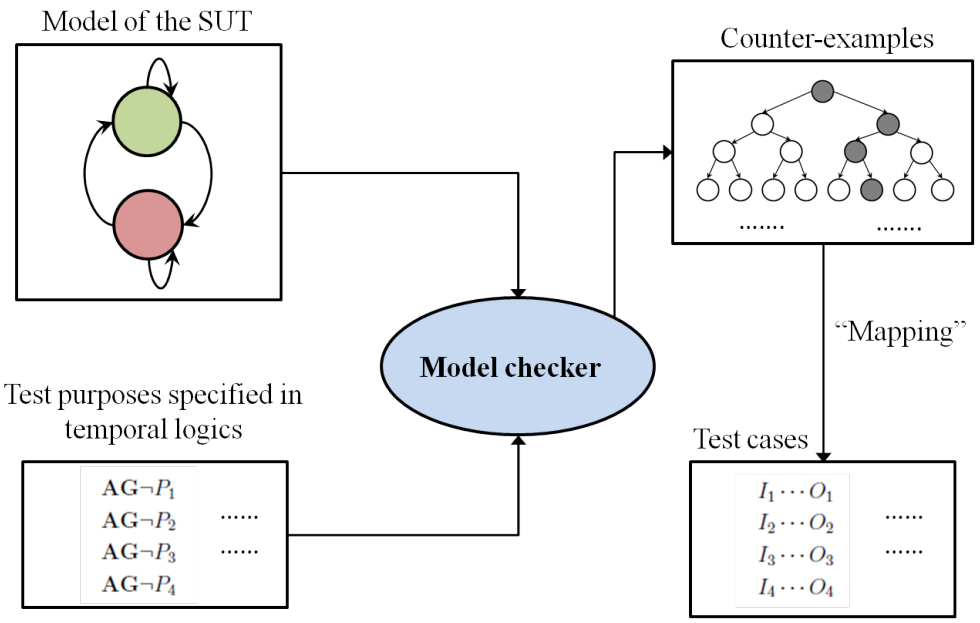


Figure 2.5: Testing with model checkers

The first challenge of using a model checker as a test generation tool is to specify test purposes as formal properties using temporal logic. It is important to notice that the property specification should be the **negation** of a test purpose so that a counterexample violating the property actually satisfies the test purpose. Early research work on testing with model checkers involved manually

specifying the negation of test purposes as "never-claims" [77]. Test purposes are specified as properties in temporal logic, then model checking a never-claim (negation of a property) on a model. The model checker returns a counterexample if the never-claim becomes false at some point. The counterexample illustrates how the violation of the never-claim occurs, and thus illustrates how the original test purpose is fulfilled.

Manual specification of test purposes can lead to efficient test cases, but to ensure complete coverage of all possible system behaviors remains difficult. Many different techniques have been proposed to automatically and systematically derive complete sets of test cases. Most of them follow the idea of never-claims. A popular approach is to automatically create properties based on coverage criteria (see Sec. 4.1.1), since coverage criteria are very commonly used test purposes. The properties based on coverage criteria are initially called "trap properties" [91], for each item that should be covered, one trap property is generated. A trap property states that certain item could never be covered by any execution. Model checking the trap property results in a counterexample illustrating how the item can be covered, if the trap property is violated.

The idea of coverage-based test generation using model-checking has already been studied by a few researchers. [92] proposes using a model-checker to generate a test case that covers certain areas of the program. [146] uses model-checking to perform reachability analysis, i.e. whether certain areas can ever be covered by any test case. In [81], model-checking is used to derive test cases from all uncovered branches to enhance structural coverage. [92] and [146] explain how one test case can be built as [81] presents a procedure for dealing with every uncovered branch. To answer the question "how to design a test case using model-checking", [147] presents a framework where structural coverage criterion MC/DC is formalized as temporal logic formula used to challenge a model-checker to find test cases. Also in [78], are demonstrated techniques of using model-checker to generate test cases from coverage criteria including branch coverage and MC/DC.

The majority of coverage based test generation approaches use some structural coverage criterion based on the model. It is sometimes desirable to create test cases based on other test purposes than coverage criteria. Examples of different techniques include requirements based approach and mutation based approach.

Requirements based test generation [51, 170] uses requirement properties as test purposes. Challenging the model checker with the negation of a requirement property results in a counterexample. But this counterexample is not always nec-

essarily useful regarding the property. For example, negating a safety property might result in a counterexample consisting of only one state (the initial state).

Mutation based approach involves intentionally modifying a program or specification in small ways. Each mutated version is called a mutant. Then a model checker is used to show how much those mutants differ from the original model. The difference is shown with sequences that can be used as test cases. Mutation testing was originally applied to source code [70], and later then to specification [49]. Regarding testing with model checkers, Ammann and Black [3] first introduced specification mutation for coverage analysis; Ammann *et al.* [15] initially suggested the use for test case generation. More details regarding different test generation techniques using model checkers can be found here [86, 87].

After choosing the strategy of triggering counterexamples generation, next challenge is to interpret these counterexamples as test cases. This depends on several factors: the types of the system under test, the abstraction level of the model with respect to the system, etc. In the literature, testing with model checkers is most commonly applied to reactive systems [97]. A reactive system communicates with an environment, reading input values and sending output values, as shown in Fig. 1.2. The model of a reactive system usually consists of vectors of input and output variables, and possibly internal variables. For this kind of model, counterexamples generated by model checkers are sequences of valuations for input, output and internal variables. Such counterexamples are directly usable as test cases, consisting of input data and the expected output data. Although testing with model checkers is not limited to reactive systems, the mapping from counterexamples to test cases is less obvious for other kind of systems [106, 107].

The execution of test cases also depends on the type of the system under test and the abstraction level of the model used for test generation. In the literature, a common scenario of execution on reactive systems is preformed in a cyclical manner: in each cycle the system receives stimuli (inputs), performs some calculations and sends the results as outputs. The observation of an execution leads to the problem of **oracle** [24]. An oracle specifies the expected outcome of a test as applied to a tested object. The use of oracles involves comparing the expected output for a specified input to the observed output sent by the system under test. The result can either be fail if the observed output does not match the oracle; or pass if otherwise.

2.3 Synchronous approach for real-time systems

Let's first briefly present reactive systems before talking about synchronous languages. The term "reactive systems" was first introduced in 1985 by Harel and Pnueli [103]. Now it is commonly accepted to designate a system permanently interacting with a physical environment. The reactive system receives input from the environment (sensors, actuators, human operators, etc) and responds with calculated output to the environment (Fig. 1.2. It is important to precise that reactive systems have to react to an environment which **can not wait**. This distinguishes reactive systems from *interactive systems*. Interactive systems permanently communicate with an environment, at their own speed (make the environment wait). Concurrent processes in operating systems or in data-base management are generally interactive. Reactive systems also differ from *transformational systems*, which are classical programs beginning with initial data and terminating with providing calculated results (a compiler for example).

In the literature, the specific features of reactive systems have been pointed out several times [35, 31][97]:

- Deterministic concurrency: unlike most interactive systems, reactive systems are generally deterministic and their description involves concurrency. The concurrency comes from several reasons: reactive systems run in parallel with their environment; their implementation is quite often distributed for reasons of performance and fault tolerance, etc. Most of the time, even if the systems are implemented in a centralized way, it is convenient to describe them as parallel modules.

- Synchrony: reactive systems should support the simple and commonly used implementation schemes in Fig. 2.6, where all mentioned actions are assumed to take finite memory and respect time constraints. Time constraints concern input frequencies and input-output response time. They are introduced by the environment and should be imperatively satisfied.

- Critical reliability: most of reactive systems are highly safety-critical, such as control/command system in a nuclear power plant or a commercial aircraft flight control system. The application domain of reactive systems requires very careful design and very stringent verification. As mentioned at the beginning of this chapter, formal methods should be used with higher priority for safety-critical systems.

Fig. 2.6 illustrates two common execution schemes as a way to implement a reactive system by a single loop [97]. The program scheme is either "event driven", if each reaction is triggered by an input event; or "sampling driven", which

consists in periodically sampling the inputs. In both cases the program typically implements an *automaton*: the states are the valuations of the memory and the transitions correspond to the reactions. With their simplicity and efficiency, automata are useful tools to describe reactive systems, but they are very difficult to design by hand. As a result, synchronous languages [26] have been proposed around the nineties aiming to provide high level and modular constructs to make design easier.

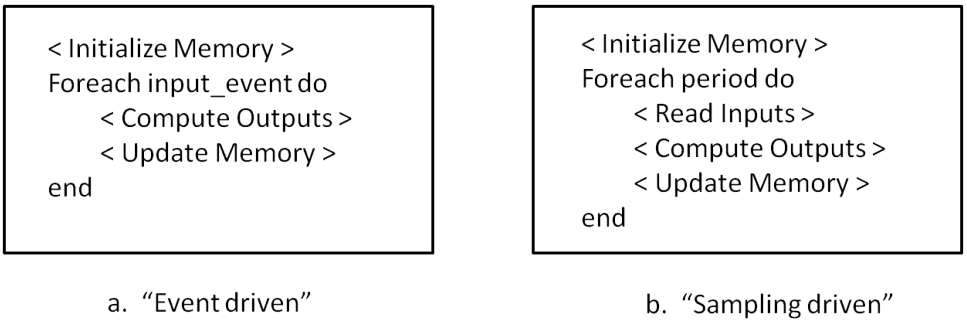


Figure 2.6: Two common execution schemes for reactive systems

Statecharts [102] proposed by Harel in the early eighties are probably the first and most popular formal language for the design of reactive systems. However, statecharts were proposed more as a specification formalism than as a programming languages. Moreover, determinism is not ensured and many semantic problems were raised [23]. Around the nineties, three synchronous programming languages were proposed by French academic groups: Esterel [36, 32], Signal [122, 28, 29] and Lustre [52, 101]. All these programming languages share some same principles: synchronous hypothesis and deterministic concurrency. In practice the synchronous hypothesis assumes that the program is able to react to an external event before any further event occurs. These synchronous languages also differ from each other in style, which results in different application domains. In the following we first introduce separately the three french synchronous languages and then focus on Lustre, since it's the language chosen in the project "CONNEXION" as well as in this thesis.

2.3.1 Esterel

Esterel is an imperative synchronous language developed at Ecole des Mines and Inria Sophia Antipolis. Esterel provides usual constructs such as assignments of

variables, making control decisions, etc (see Table 2.4 for a sampling of the complete language [27]). An Esterel program consists of nested, concurrently running threads whose execution is synchronized to a single global clock. At the beginning of each reaction, each thread starts its execution or resumes if it was paused in the last reaction, performs imperative code and finally terminates or pauses for the next reaction.

Esterel has been successfully applied in design of synchronous circuits. Most hardware description languages(for example Verilog [161, 6] and VHDL [8]) are very useful in designing the data part of a circuit, however, they are of little help when it comes to describe complex hardware controllers. This explains the success of Esterel in this field. Some references about the application of Esterel can be found here [155, 153, 163].

Statement	**Interpretation**
emit S	Make signal S present immediately
present S then p else q end	If signal S is present, perform P otherwise q
pause	Stop this thread of control until the next reaction
p; q	Run p then q
loop p end	Run p; restart when it terminates
await S	Pause until the next reaction in which S is present
p ‖ q	Start p and q together; terminate when both have terminated
abort p when S	Run p up to, but not including, a reaction in which S is present
suspend p when S	Run p except when S is present
sustain S	Means loop emit S; pause end
run M	Expands to code for module M

Table 2.4: Some basic Esterel statements

2.3.2 Signal

Another data-flow synchronous language Signal was developed at IRISA Rennes. Based on the synchrony hypothesis, the semantics of Signal is defined via a mathematical model of **multiple-clocked** flows. Signal handles (possibly infinite) sequences of data and events with respect to a discrete time scale and such sequences are referred to as *signals*. At a given instant, signals may have the state *absent* or the state *present*. All the signals taking the state *present* simultaneously

are said to *possess the same clock*, and they are said to possess different clocks otherwise. so clocks can be considered as equivalence classes of signals that are always present simultaneously. Signal offers operators relating clocks to the values of various signals involved in a given dynamic system. Such systems have been referred to as Multiple-Clocked Recurrent Systems (MCRS) [26] and Signal is a **multiclock** language. Using Signal to specify this kind of system releases the programmer from the burdens of handling explicitly multiple time indices. Every signal has a associated clolck and an implicit time index and the operators define relations between these time indices.

Table 2.5 gives a sampling of Singal operators. The state *absent* of a signal at a given instant is marked by $\perp$. X_τ denotes the status ($\perp$ if absent or actual carried value if present) of a signal X in an arbitrary reaction τ. The first two statements concern signal possessing the same clock. They are called single-clocked statements. Integer k represents the instants at which signals are present. Note that the index k is implicit and does not appear in the syntax. *op* denotes a basic operator operating pointwisely to the sequence of values of a signal (e.g. +, *, ...). The third and forth statements concern signals possessing different clocks. They are referred to as multiclocked statements. More details regarding Signal semantics can be found elsewhere [122, 27].

	Statement	**Meaning**
Single-clocked	`Z := X op Y`	$Z_\tau \neq\perp\Leftrightarrow X_\tau \neq\perp\Leftrightarrow Y_\tau \neq\perp$, $\forall k : Z_k = \text{op}\ (X_k, Y_k)$
	`Y := X$1`	$X_\tau \neq\perp\Leftrightarrow Y_\tau \neq\perp$, $\forall k : Y_k = X_{k-1}(\text{delay})$
Multiclocked	`X := U when B`	$X_\tau = U_\tau$ when $B_\tau = true$, otherwise $X_\tau = \perp$
	`X := U default V`	$X_\tau = U_\tau$ when $U_\tau \neq\perp$, otherwise $X_\tau = V_\tau$

Table 2.5: A sampling of Signal operators

2.3.3 Lustre

Lustre is a synchronous data-flow language developed at CNRS Grenoble. Lustre is based on two fundamental notions:

- *flow*: each Lustre variable and expression is considered as a flow, i.e., a (possibly infinite) sequence of values of a given type.

- *clock*: a clock represents a sequence of times

Each flow is associated implicitly to a clock: the flow takes the n^{th} value of its sequence at the n^{th} instant of its clock. If the behavior of a system can be described in a cyclic manner, then that cycle defines a sequence of times called the global clock (also called the basic clock). Any flow whose clock is the global clock takes the n^{th}value at the nth execution cycle. Other different clocks can be defined with respect to the global clock using a boolean-valued flow: for example the clock is a sequence of times when the boolean flow takes the value *true*.

The notion of "clock" introduces a discrete time scale. For a synchronous language, the time granularity must be adapted **a priori** to the time constraints imposed by the environment to which the system reacts. The time granularity concerns the input frequencies and the input-output response time. Both times should be considered negligible with respect to one cycle of the global clock. In other words, between two successive cycles, there are no external events (new inputs coming from the environment) and the time to compute outputs from inputs is considered negligible to the duration of one cycle. As a result, inputs are taken into account and outputs are computed "at the same time" with respect to the discrete time scale. The synchrony hypothesis should be validated a posteriori. The functional behavior of a system described in Lustre does not depend on the clock cycle. Therefore, it is possible, and probably more convenient to perform a functional validation of the system description on another machine (e.g. the development machine) rather than the target machine. The time validation, however, must be performed on the target machine. If the input-output computation time is less than the time interval between two successive instants on the global clock, the synchrony hypothesis is considered satisfied. The computation time depends on software and hardware performance. The time constraint (interval between two successive instants) is usually imposed in the requirement specification.

A Lustre program describes the relations between its inputs and outputs through variables and equations. Each variable is a flow, so variable X and Y are respectively $(x_1, x_2, \cdots, x_n)$ and $(y_1, y_2, \cdots, y_n)$. The equation $X = Y$ denotes $x_i = y_i$ with $0 \leq i \leq n$. When two variables are considered identical, they have the same sequence of values and the same clock. In a Lustre program, the order of the equations does not matter, which is one important principle of the language: the substitution principle. An equation such as $X = Y$ is oriented in the sense that it defines X and the way X is used in other equations can not give it more properties than those coming from its definition. As a result, X can be substituted to Y anywhere in the program and vice versa. Extra (internal) variables can be created just to give names to expressions. Lustre offers a few basic data types:

`boolean, integer, real` and a type constructor `tuple`. Other complex types can be imported from a host language (e.g. C/C++) and handled as abstract types. It is also possible to import functions from a host language. Constants are flows have constant sequences of values following the global clock. Their types can be the basic types or imported complex types.

An equation is expressed through variables and operators. Lustre offers usual operators over basic types:

- arithmetic operators: `+, -, *, /, div, mod`
- boolean operators: `not, and, or`
- relational operators: `=, <, <=, >=, >`
- control operator: `if then else`

These operators are called data operators and they only operate on variables or expressions sharing the same clock. Data operators work in a pointwise manner on the sequences of values of their operands. Lustre also has two "temporal" operators:

- operator `pre` ("pevious") refers to the value of its operand at the previous cycle: for a flow $m = (m_1, m_2, \cdots, m_n, \cdots)$, $pre(m) = (nil, m_1, m_2, \cdots, m_n, \cdots)$ where nil denotes a undefined value.
- operator `->` ("initialization") is used to assign to its operand the value at the initial cycle: for two flows $m = (m_1, m_2, \cdots, m_n, \cdots)$ and $s = (s_1, s_2, \cdots, s_n, \cdots)$, m `->` s denotes a flow $(m_1, s_2, \cdots, s_n, \cdots)$. Actually the flow m `->` s is equal to the flow s except for the first cycle.

The data types and operators mentioned above constitute the so-called "Lustre core" [112], which is the common syntax shared by different versions of Lustre. The two most commonly used version are Lustre V4 and Lustre V6. We will get into more details on this subject in the following section. The semantics of Lustre can also be given in terms of linear temporal logic [135].

Fig. 2.7 gives as example a simple counter described using Lustre. The counter is graphically presented in a style very familiar to control engineers. The counter has two integer inputs m and n and one integer output s. At every execution cycle, the counter calculates the product of m and n and add the result to the value of s at the previous cycle. We immediately recognize the two temporal operators `->` and `pre`. A constant integer 0 initialize the flow s. As a reminder, a constant is

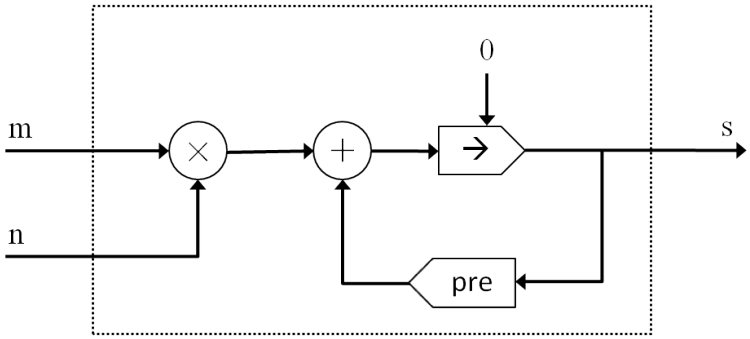


Figure 2.7: A simple counter described using Lustre operators

also a flow so $0 = (0, 0, ...,)$. The "initialization" operator `->` actually only effects the first (initial) cycle so it can be considered "disappeared" for the following cycles.

The same counter (Fig. 2.7) described in Lustre program looks like the following. An internal variable p is defined as the result of $m * n$. Table 2.6 gives the values of all the flows in an execution of the counter.

```
node counter (m, n: int) returns (s:int);
var p: int;
let
p = m * n;
s = 0 -> p + pre(s);
tel
```

Cycle	1	2	3	4	5	...
m	1	3	5	7	9	...
n	2	4	6	8	10	...
p	2	12	30	56	90	...
s	0	12	42	98	188	...

Table 2.6: Example of an execution of the counter

In a Lustre program, we distinguish two types of structures:

- operator: the basic operators (including temporal operators) described just above; operator can be seen as a low-level node.
- node: a sub-program defining relations between its output parameters and input parameters, through a set of unordered equations and possibly local

variables. Once declared, a node can be freely instantiated anywhere in the program, just as a basic operator.

Through the notions of operator and node, Lustre naturally offers a **hierarchical description** and **component reuse**. A Lustre program is thus structured into a hierarchical network of nodes and operators, as shown in Fig. 2.8.

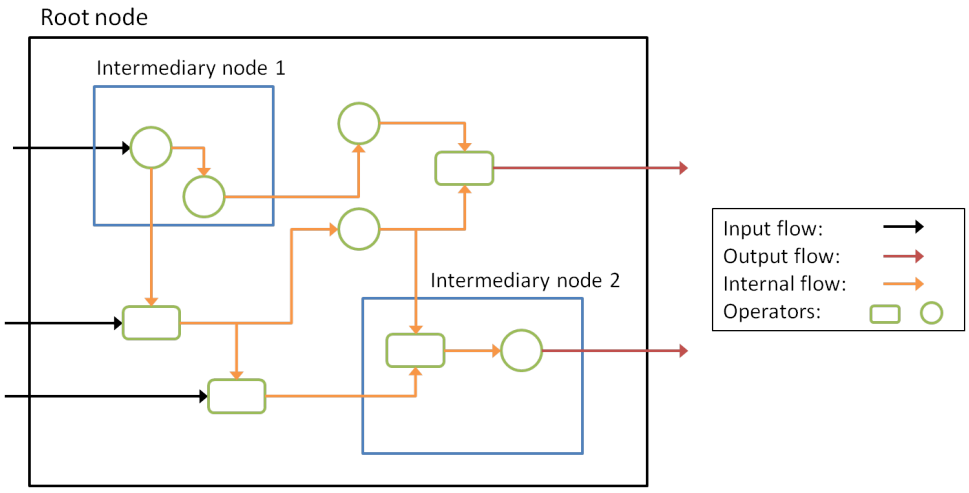


Figure 2.8: Lustre program structure

Now we can easily describe the function HOP (Fig. 2.1) as a Lustre node:

```
node HOP (HP, E: bool) returns (yL:bool);
let
yL = 0 -> if HP then false
          else if E then true
               else pre(yL);
tel
```

The automaton in Fig. 2.1 does not impose any condition on the initial state, however in the Lustre program, the first value of yL is set to $false$ i.e. 0. An example of execution is given in Table 2.7.

2.3.4 A brief summary

Nowadays, synchronous languages are widely accepted as a technology of choice for specifying, modeling, validating, and implementing real-time systems. The

Cycle	1	2	3	4	5	...
HP	false	false	false	false	false	...
E	true	true	true	false	true	...
pre(yL)	nil	false	true	true	true	...
yL	false	true	true	true	true	...

Table 2.7: Example of an execution of the function HOP

paradigm of synchrony has emerged as an engineer friendly design method based on mathematically sound tools. In 1991, proceedings of the IEEE devoted a special section [25] to the synchronous languages including Esterel, Signal and Lustre. At the time, the three French synchronous languages were well defined and had seen some industrial use, but were still under development. In the following decade, the languages have been improved, gained a much larger user community, and have been successfully commercialized. A little summary of the three languages are given in Table 2.8.

Language	Style	Commercial tool	Industrial use
Esterel	Imperative	Esterel	**Avionics**: Dassault, **CAD**: Cadence, Synopsys, **Telecom**: Thomson, TI.
Signal	Data-flow	Sildex	
Lustre	Data-flow	SCADE	**Avionics**: Airbus, Honeywell, **Nuclear plants**: EDF, Schneider-Electric

Table 2.8: Comparison of Esterel, Signal and Lustre

The first commercial version of Esterel was marketed in 1998 by the French software company Simulog[2]. This division was spun off in 1991 to form the independent company Esterel Technologies[3]. In 2001 Esterel Technologies acquired the commercial SCADE environment for Lustre programs to combine two complementary synchronous approaches.

Signal was licensed to a French software company TNI in the early nineties. TNI developed and marketed the Sildex tool in 1993. Several versions have been issued since then, most recently Sildex-V6. In 1999, TNI merged with Valiosys, a startup operating in the area of validation techniques and became TNI-Valiosys[4].

[2] `https://www.ercim.eu/publication/Ercim_News/enw24/simulog.html`
[3] `http://www.esterel-technologies.com/`
[4] `http://www.tni-valiosys.com/`

In the 1980s, two big industrial projects including safety-critical software were launched independently in France: the N4 series of nuclear power plants and the Airbus A320. Facing with the challenge of designing highly safety-critical software, the two companies, Schneider-Electric and Airbus decided to build their own tools since they failed to find suitable existing tools. As consequence, both their tools were based on synchronous data-flow formalism and used the Lustre language. After some years of successfully using these tools, both companies were faced with the problem of maintaining and improving them. Eventually, the software company Verilog undertook the development of a commercial version of Lustre and built the SCADE (Safety-Critical Application Development Environment). In 2001, Esterel Technologies purchased the SCADE business unit[5].

2.4 The story of Lustre

2.4.1 SCADE and Lustre

The SCADE development environment offers a synchronous model-based approach to the design, validation and implementation for embedded software. Combining block diagrams and hierarchical state machines, SCADE allows users to create high-level graphical models with rigorous formal specifications. The SCADE KCG compiler, offering C code generation from high-level designs, is certified at avionics norm DO-178B highest level A. SCADE also supports visual simulation and animation, test coverage analysis and formal verification techniques. With the complete SCADE tool set, users are allowed to generate correct-by-construction implementation from high-level formal specifications. Being executable, these specifications can be thoroughly simulated and verified before being implemented. The SCADE tool set is now widely used in the safety-critical industries such as avionics, automotive, railway, etc.

In this thesis, we use SCADE for the software environment and Scade for the underlying formalism. The current version of SCADE formalism is Scade 6 [65]. Scade 6 offers two specification formalisms for cycle-based design: *block diagrams* and *safety state machines*. Lustre is the root textual language of Scade block diagrams.

- *Block diagrams*, are a formalism familiar to control engineers, for specifying continuous control. Continuous control means sampling the inputs at reg-

[5] `http://www.esterel-technologies.com/`

ular time intervals, performing computations on the inputs and outputting calculated results. In SCADE, continuous control is graphically presented using block diagrams; the root textual language comes from Lustre. Fig. 2.9 depicts the previously discussed function HOP in Scade using block diagrams.

- *Safety state machines (SSMs)* [33], are the Scade state machines for specifying discrete control. Discrete control means changing behaviors according to external events. The events could originate either from discrete environment inputs or from internal program events. SSMs evolved from the Esterel language and SyncCharts [16, 17] which is a synchronous version of Statecharts [102].

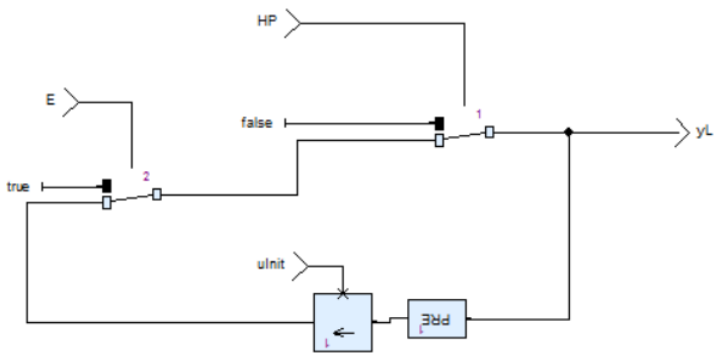


Figure 2.9: Function HOP depicted in Scade block diagram

The Scade 6 formalism provides its users with a full integration of block diagrams and SSMs. For example, a state in a state machines may contain either another state machine or a block diagram. It is therefore possible to implement the mode-automata [125] where the same flow can switch from one continuous control computation to another according to boolean conditions.

The SCADE tool set now includes:

– SCADE Suite: an editor offering high-level graphical and textual descriptions of a system as well as visual simulation.

– SCADE KCG: a C code generator certified by authorities for qualified software production.

– SCADE QTE: a qualified testing environment including a test coverage analysis tool (SCADE MTC). The conventional coverage metrics need to be modified when applying to data-flow languages such as Scade and Lustre (see Sec. 4.1.1).

– SCADE Design Verifier (DV) [13]: an interface to formal verification tools such as Prover plug-in[6].

The SCADE environment and the Lustre Language have been chosen in "CONNEXION" to specify models of the I&C systems.

2.4.2 Lustre versions

The Lustre language was first designed more than twenty years ago, at the beginning of the nineties. Lustre has been successfully commercialized and resulted in an industrial software development tool: SCADE [33, 98]. SCADE is now being used by many major companies developing safety-critical embedded systems (avionic, energy, transportation,...). On the academic side, Lustre is continuously being developed and upgraded, resulted in two commonly used versions: Lustre V4 and Lustre V6.

Fig. 2.10 illustrates the relationship between different academic Lustre versions and the Scade 6 formalism. The Lustre syntax and semantics previously presented in Sec. 2.3.3 correspond to the so-called Lustre core (or the basic Lustre), a common part shared by Lustre V4 and V6. Lustre V4 and V6 also both support arrays. The Lustre V6 specific features not supported by Lustre V4 include:

– V6 offers a new operator *array iterators* to replace the use of V4 homomorphic extension [148] and to provide a (restricted) notion of higher-order programming.

– V6 supports structured data types such as struct, records, enumerations.

– V6 introduces a package system which aims at introducing a new level of structuration and modularity as well as name space facilities.

Some Lustre V4 features not supported in Lustre V6:

– recursive arrays slices are replaced by array iterators

– complex data are replaced by structures and arrays, e.g., [int, real] becomes struct; [int, int] becomes int^2, etc;

[6] `http://www.prover.com/`

The strong evolutions of the language from V4 to V6 are prototyped thanks to Marc Pouzet's "Lucid synchrone" [53, 54], which is a higher order extension of Lustre [98].

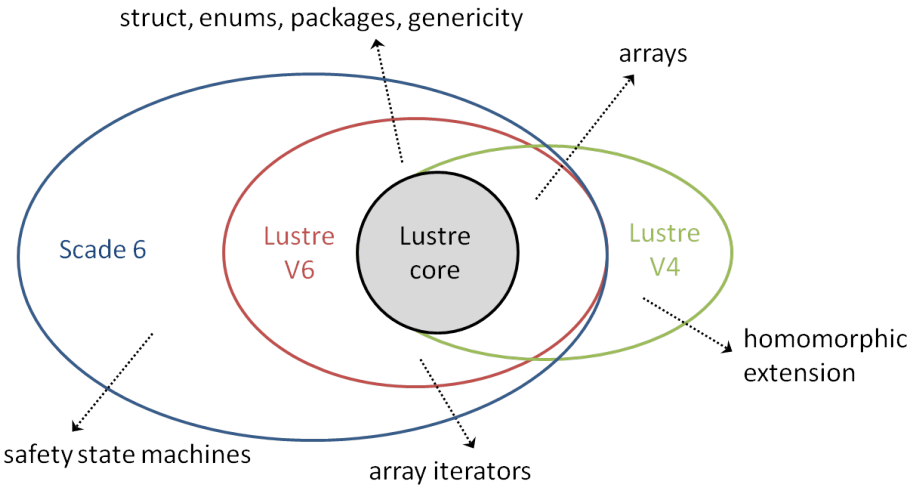


Figure 2.10: Relationship between academic Lustre versions and Scade 6

The Scade 6 formalism consists of block diagrams and safety state machines. Lustre V6 is the underlying textual language of block diagrams. SSMs evolved from the Esterel language and the SyncCharts synchronous statecharts model. SSMs have been proved to be scalable to large control systems [33].

2.4.3 Model checking tools for Lustre

As a synchronous data-flow language, Lustre has been commonly accepted by academic and industrial users for specification and verification of reactive systems. Lustre has also been successfully commercialized and resulted in the SCADE environment. Reactive systems are usually highly safety-critical systems requiring very careful design and stringent verification. As consequence, formal verification techniques play an important role since they allow rigorous prove or disprove of critical safety properties. Combined with model-based system engineering approaches, they offer early verification and detection of defects during the development phases.

The Lustre language therefore already benefits from various verification tools. They include several open-source academic model checkers: Lesar [99, 100], Kind 2 [57], PKind [114], jKind [89], Zustre (a Lustre front end for the Z3-based model checker Spacer [117]). These tools are oriented to the traditional model checking

problem: property verification. There are also some tools more oriented to test generation: Lutess [44, 72, 140], Lurette [149, 111, 109, 110] and GATeL [126, 127]. Lutess and Lurette both consider the testing of reactive programs in a black-box framework. Lustre is not used for the system specification but for the specification of the environment where the program is embedded. GATeL offers test sequence generation for Lustre based programs with finely tunable test objectives.

On the SCADE side, SCADE DV (Design Verifier) [13] uses Prover plug-in verification engine to perform symbolic model checking. The user provides temporal assertions about program behavior, derived from application requirements. Assertions are expressed in the Scade formalism, technically as boolean flows that should always stay true. Compared to conventional model checking problem where properties are specified in temporal logic, the SCADE DV releases its user from learning specific property specification languages. While SCADE DV performs very well for certain verification tasks, it can fail badly for others due to complexity problems. When a verification task is infeasible, SCADE DV provides the user with little or no information regarding the causes. This makes it almost impossible to decide which further measures to take: try to make the verification task feasible, or to settle for a weaker verification result, or to abandon formal verification and invest in testing [22]. This is a disadvantage both for practical application in industry and for academic research pertaining to formal verification with SCADE DV. A recent study [108] on formal safety analysis of two industrial SCADE models may explain the industries' indecision towards adopting formal verification in productive processes. In [22], an alternative open-source SMT-based model checking method for Scade models is presented. The method introduces LAMA as an intermediate language for translating Scade programs into SMT solver input models with experiment results using the Z3 solver.

Fig. 2.11 illustrates the relationship between Lustre based formalisms and model checking tools. The academic Lustre V4 benefits from several model checking tool: GATeL is more oriented to test generation while Lesar and Kind 2 more oriented to property verification. Each tool works on a slightly different version of Lustre V4 as input modeling language. Lesar, the first academic model checker for Lustre, takes standard V4. The front-end of Lesar is a compiler translating Lustre V4 to the format EC (expanded code). GATeL takes an extension of V4 and Kind 2 a V4 extended with part of V6. Lustre V6 can be automatically translated to the format LIC (Lustre internal code) which is equivalent to EC for Lustre V4. Lustre V6 programs are therefore amenable to Lesar through LIC. The formalism Scade 6 is based on Lustre V6 extended with safety state machines. SCADE Design Verifier (based on Prover Plug-in) is the commercial formal verification tool

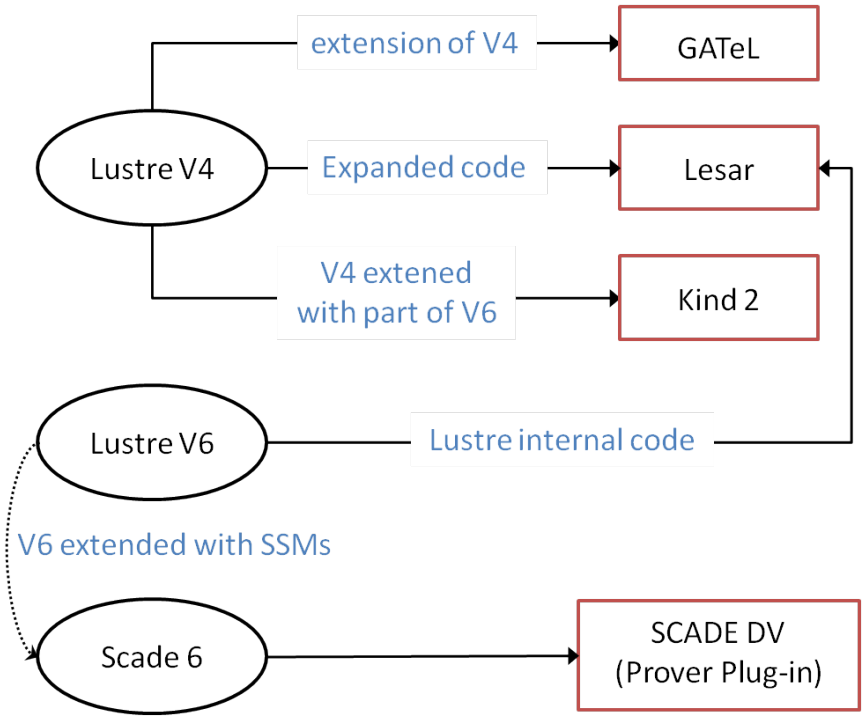


Figure 2.11: Model checking tools for Lustre and Scade

for Scade models.

SCADE DV is not included as part of the project "CONNEXION". In this thesis, we focus on three academic model checking tools for the Lustre language: GATeL, Lesar and Kind 2. GATeL has been chosen in the project "CONNEXION". Lesar is a standard powerful model checker for Lustre V4 programs. Lesar performs explicit and symbolic model checking and has been applied to verification of real time properties of a landing gear control system [43]. Kind 2 relies on state-of-the-art SMT solvers, capable of handling systems of infinite state space. Kind 2 is used in academia and in a variety of industrial settings [18]. We therefore chose these tools as candidates to apply our methodology. Each tool works on a slightly different extension or subset of the Lustre language. The fact that Lustre language has several academic versions make the story more complicated. We also consider another alternative of translating Lustre programs to other modeling languages so as to benefit from more powerful tools (Sec. 2.4.4).

Lesar

Lesar [99, 100] is a formal verification tool for Lustre V4 programs with respect to safety properties. Verification is performed on a finite state abstraction of the program, which models only the behavior of boolean variables. Lesar performs two kinds of verification techniques:

1. an exhaustive exploration of the state space of the model, similar to explicit model checking.

2. a symbolic construction of the set of states that satisfy the property, analogous to symbolic model checking [139].

GATeL

GATeL [127, 126], developed in CEA List and selected in "CONNEXION", is a model checking tool supporting test sequence generation for an extension of Lustre V4. GATeL takes as input a Lustre model of the system under test, a test objective and a specification of environment. A test objective can be a safety property or a declarative characterization of some interesting states of the SUT. A specification of the environment describes the possible evolution of the inputs that can be considered during test sequence generation. Like a classical model checker, GATeL performs formal verification of the model with respect to the test objective, considering the specification of environment in the same time. According to the test objective, e.g., a safety property or a particular state, GATeL returns test sequences as counterexamples illustrating the violation of property or how to reach the particular state.

Compared to most model-based testing tools, GATeL offers mechanisms allowing the definition of customized test selection strategies for the SUT. Through test objectives, a user could define his own test selection strategy. For each test case derived from the selection strategy, GATeL automates the generation of input sequences. It also systematically calculates, from the Lustre model, the inputs, outputs and variables involved in the test objective. This provides the information needed to construct an oracle: these computed outputs constitute the expected values that should be compared to actual outputs of the SUT.

The kernel of GATeL is a resolution procedure for constraints built from an interpretation of Lustre constructions over boolean variables, variables with integer intervals (real numbers or floating-point number arithmetic are not considered yet), and a special synchronization constraint for the status of each cycle (whose

value is either `initial`, or `non_initial`). Resolution proceeds by successive elimination of all constraints. A non-deterministic instantiation procedure (called "labelling" in the logic programming community) instantiates the variables involved in the constraints. In order to avoid erroneous valuations, a constraint propagation mechanism continuously checks constraint satisfiability. The instantiation of a variable "awakes" the propagation of related constraints, which can disappear (when solved) or awake/create other constraints [127]. More details on this mechanism of constraint propagation can be found in [126].

It is not always possible to predict how long each test sequence will need to be, i.e., of how many cycles it consists. However, this length is bounded by a parameter (maximum number of cycles) tunable by the user. GATeL generates test sequences in a *backward* manner: the state where the objective is reached is considered as the final state of test sequence. Many other tools are based on *forward* generation techniques: test sequences are generated from the initial state and they are limited to the test of safety properties (invariant properties).

Kind2

Kind 2 [57] (successor of PKind [114]) is a multi-engine, SMT-based model checker developed at the University of Iowa. Kind 2 takes as input models described in an extension of the Lustre V4 language that allows the specification of assume-guarantee-style contracts [56] for system components. Kind 2 relies on off-the-shelf SMT solvers (Z3 [67], Yices [73] and CVC4 [20])to prove(or disprove)quantifier-free regular safety properties. For those properties that are falsified, Kind 2 produces input sequences illustrating the violations in XML format. Kind 2 runs a combination of different induction-based model checking engines in parallel. By default Kind 2 runs a process for k-induction [154, 94, 93], two processes for invariant generation, and a process for IC3 [46].

As mentioned in Sec. 2.3.3, Lustre programs are structured into nodes and operators and naturally offer a hierarchical description and component reuse. Kind 2 takes advantage of this by performing two following techniques:

1. *modular reasoning*: each node can be assigned its own properties and verified individually. The results of the verification process (e.g., proven properties and auxiliary invariants) can be reused in the analysis of other components calling that node.

2. *compositional reasoning*: the user is allowed to specify assume-guarantee style contracts for each node. A contract has typically less states than the

node it specifies. When verifying a node, compositional reasoning consists in abstracting the complexity of its components (sub-nodes) by their contracts.

Compositional reasoning is usually applied in conjunction with modular reasoning, since a successful compositional proof of a node does not guarantee the correctness of the un-abstracted node. More details regarding k-induction, invariant generation, IC3, modular reasoning and compositional reasoning can be found in [12].

SMT solvers applies the cutting edge model checking techniques and have been developed in academia and industry with increasing scope and performance [21]. However, the most notable restriction of SMT solvers is that nonlinear expressions are forbidden, meaning that no multiplication or division is allowed between variables. Kind 2 uses off-the-shelf SMT solvers and encounters difficulties when applied to our case study from "CONNEXON": the model of the SUT contains several non-linear expression such as multiplication and division between local (internal) variables. One solution Kind 2 offers relies on the "CoCoSpec" extension: a language providing assume-guarantee style contracts for every component of the Lustre program. Non-linear expressions can be abstracted away using contracts specification. Then through compositional reasoning, the complexity of components containing non-linear expressions can be abstracted by the contracts.

An example of assume-guarantee style contracts for a multiplication expression is given below. An assume-guarantee contract (A,G) for a node is a set of assumptions A and a set of guarantees G. Assumptions describe how the node must be used, while guarantees specify how the node behaves. In the following example, the node `times` outputs a real number `z` as multiplication of its two real inputs `x` and `y`. The complexity of this non-linear expression is abstracted away by the `guarantee` expressions defined in the contract. Note the absence of `assume` expressions indicates that the node `times` always behave like `guarantee`. Kind 2 is able to perform compositional reasoning, which consists in replacing the non-linear expression `z=x*y` by the contract every time the node `times` is called in the program.

```
node times(x, y: real) returns (z: real) ;
(*@contract
var abs_x: real = if x < 0.0 then -x else x ;
var abs_y: real = if y < 0.0 then -y else y ;
var abs_z: real = if z < 0.0 then -z else z ;
-- Neutral.
guarantee (z = y) = ((x = 1.0) or (y = 0.0)) ;
```

```
guarantee (z = x) = ((y = 1.0) or (x = 0.0)) ;
-- Absorbing.
guarantee (z = 0.0) = ( (x = 0.0) or (y = 0.0) ) ;
-- Sign.
guarantee (z > 0.0) = (
( (x > 0.0) and (y > 0.0) ) or
( (x < 0.0) and (y < 0.0) )
) ;
guarantee (z < 0.0) = (
( (x > 0.0) and (y < 0.0) ) or
( (x < 0.0) and (y > 0.0) )
) ;
-- Loose proportionality.
guarantee (abs_z >= abs_y) = ((abs_x >= 1.0) or (y = 0.0)) ;
guarantee (abs_z >= abs_x) = ((abs_y >= 1.0) or (x = 0.0)) ;
guarantee (abs_z <= abs_y) = ((abs_x <= 1.0) or (y = 0.0)) ;
guarantee (abs_z <= abs_x) = ((abs_y <= 1.0) or (x = 0.0)) ;
*)
let
z = x * y ;
tel
```

2.4.4 Lustre translators

Lustre, EC and LIC

The Luster V4 distribution[7] is a tool kit containing the model checker Lesar and other tools for compilation, simulation, etc. Those tools deal with several formats; some of them are executable, while others are just scripts and shortcuts. The Lustre V6 release consists mainly of front-end complier. Fig. **??** gives a simplified overview of these formats and tools. More details are available in [148, 168].

The front end of all Lustre V4 tools is a pre-processor *lus2ec*. This is actually a compiler transforming a Lustre V4 file (.lus) into a Lustre expended-code (EC) file (.ec). A Lustre V4 file is structured into nodes and operators with modularity, arrays, recursions etc; while a EC file is a single node without arrays or recursions. A EC file can be considered as a "flattened" version of its Lustre V4 file. All other tools in the tool kit (compilers, simulators) are running on the EC format. The low-level target format in Lustre V4 is Ansi-C. This code can be obtained directly

[7] `http://www-verimag.imag.fr/The-Lustre-Toolbox.html?lang=en`

from the EC code, using the compiler *ec2c*. *ecexe* is a unix-filter style simulation tool. It interprets EC code, reading on standard input and writing to standard output. The Lustre V4 distribution provides in general shell scripts combining the front-end *lus2ec* with the various back-ends. The Lustre model checker *ecverif* is combined with *lus2ec* into the shell Lesar. In this thesis, Lesar and Lesar/ecverif are interchangeable.

The Lustre V6 release consists of a compiler *lus2lic*, translating Lustre V6 to LIC (Lustre internal code). LIC can be considered as the equivalent of EC for Lustre V6. The compiler *lus2lic* serves therefore as front end for Lustre V6 programs, translating them into LIC programs amenable to various Lustre V4 tools. Fig. 2.12 gives an overview of Lustre V4 and V6 formats and tools.

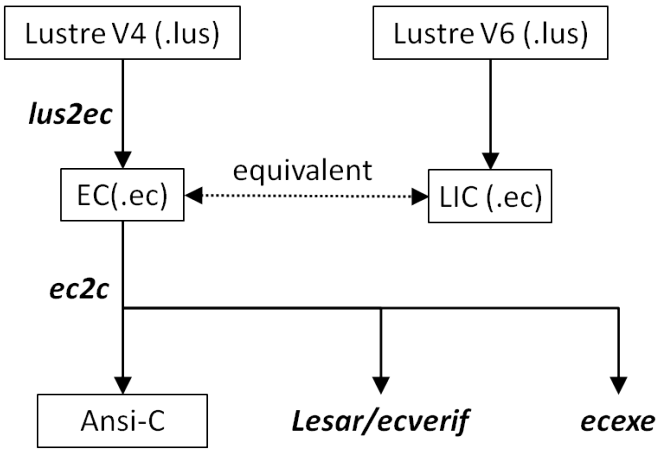


Figure 2.12: An overfiew of Lustre V4 and V6 formats and tools

Lustre to SMV

There exists a translator from EC format to SMV [128] input, developed by Gordon Pace[8]. Combined with *lus2ec*, this translator is able to convert a Lustre V4 program to a SMV program. However, the tool was developed in 2001 and is no longer maintained by the author.

A translator framework

Fig. 2.13 summarizes a translator framework developed by Rockwell Collins and the Critical Systems Research Group at the University of Minnesota [169, 134].

[8] `http://www.cs.um.edu.mt/gordon.pace/tools.html`

With the growing popularity of model-based development for the design of embedded systems, tools such as Simulink[9] and SCADE Suite[10] are achieving widespread use in industry (in particular avionics and automotive). The graphical models produced by these tools provide a (nearly) formal specification that is often amenable to formal analysis. The translator framework is developed with the motivation to bridge the gap between these commercial modeling tools and the input languages of the formal verification tools.

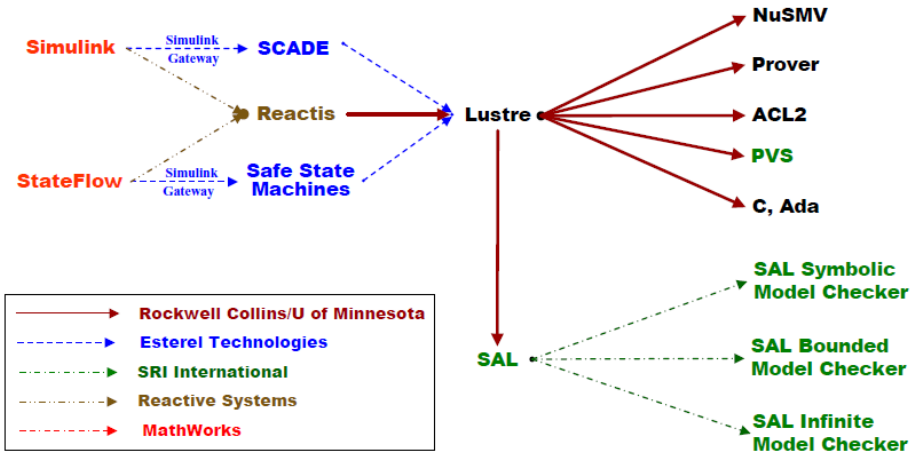


Figure 2.13: Translator Framework

The translator framework uses primarily the Lustre language but this is hidden from the user. For SCADE models, the initial translation into Lustre is immediate since Lustre is the underlying specification language of SCADE. Simulink and StateFlow users can translate their models into Lustre using the Simulink Gateway provided by Esterel Technologies. Another alternative is to import their models into the Reactis[11] test case generator developed by Reactive Systems and using a translator developed by Rockwell Collins.

Once in Lustre, the specification is read into an abstract syntax tree (AST) and a number of transformation passes are applied to it. Each transformation pass produces a valid Lustre AST that is syntactically closer to the target specification language and preserves the semantics of the original Lustre specification. This allows all Lustre type checking and analysis tools to be used after each transformation pass. When the AST is sufficiently close to the target language, a pretty

[9] https://www.mathworks.com/products/simulink.html
[10] http://www.esterel-technologies.com/products/scade-suite/
[11] http://www.reactive-systems.com/

printer is used to output the target specification [133].

The translator framework is actually a product family of translators in that many transformation passes are reused in the translators for each target language. The translators of Lustre into other input languages of various model checking tools allow to implement and extend our methodology. However, these translators are Rockwell Collins proprietary and the project with University of Minnesota is no longer supported.

Chapter 3

Project "CONNEXION"

Towards a complete testing environement

Since 2012, the main industrial and academic partners of the French nuclear industry initiated an ambitious R&D program called "CONNEXION". Regrouping a number of projects, "CONNEXION" [71] aims to improve the development process of the Instrumentation & Control (I&C) system of nuclear power plants (NPPs). "CONNEXION" is based on the existing expertises of major operators in French nuclear industry (EDF, ALSTOM, AREVA and RRCN) and various software tools provided by other industrial and academic partners (CEA, CORYS, ESTEREL Technologies and ALL4TEC).

The objectives of "CONNEXION" concern nuclear Control Systems and Operational Technologies to maintain a high level of safety, to offer new services improving the efficiency and the effectiveness of operational activities. With the current system engineering approach, increasingly detailed models describing the behavior of the I&C system have been built in early design phases. The models are specified in a formal language dedicated to the nuclear sector, using predefined blocks. During development phases, these models of the I&C system are subject to verification with respect to the functional requirements, which is defined as Functional Validation (FV) in IEC standard 61531. Functional validation of these models are currently performed manually since these models are formal but not directly executable. Automated testing is only performed at the implementation level. The actual approach, clearly fulfills the high quality requirements defined in the standards of nuclear sector. However the cost is very high considering the inherent evolution during the development process.

To automate the functional validation as much as possible, "CONNEXION"

gathers together a unique and complete set of modeling and verification tools. Integrating recent progress in MBSE, "CONNEXION" proposes to reinforce the V-model development lifecycle by introducing two sub-cycles of functional validation. The two sub-cycles functional validation perform verification of design models with respect to functional requirements, resulting in an innovative triple V-model development lifecycle. Development process is aligned with IEC standard [5], encouraging test automation tools. The current lifecycle relies on a document-centric system engineering approach; "CONNEXION" enables the transition to a model-centric practice as advocated by the INCOSE [7]. The introduction of the two V sub-cycles answers for a key challenge of Industrial Cyber-Physical Systems: early stage of Verification and Validation [82].

As real-time reactive systems (Sec. 1), the I&C system consists of a control/-command system maintaining permanent reaction to a physical environment. The control/command system and its environment are specified using different modeling languages. "CONNEXION" proposes a platform integrating the models, tools and processes related to functional validation. The platform allows closed-loop execution of the I&C system where models exchange data and synchronize with each other via FMI/FMU (Sec. 1.2). In our opinion, it is advantageous that the platform could benefit from an information system of tracebility (IST). The IST accompanies activities of verification and validation and saves not only the data but also the historical and traces between them. The IST is expected to provide a better understanding of the test coverage and minimize the effort to test non-regression [24]. We present in Sec. 3.4 the design of IST in UML using the open-source platform StarUML 2[1].

This chapter is organized as follows: Sec. 3.1 presents the project "CONNEXION" focusing on its objectives related to functional validation of early design models; the unique and complete set of modeling and verification tools in "CONNEXION" are discussed in Sec. 3.2; Sec3.3 reveals several challenges and constraints related to the solution and methodology of "CONNEXION"; Sec. 3.4 concludes the chapter by our design of the IST specified in UML diagrams as well as a prototype of implementation.

3.1 Functional validation objectives

The I&C system, together with plant operations personnel, serves as the "central nervous system" of a nuclear power plant (NPP). Through its various constituent

[1] `http://staruml.io/`

elements such as sensors and actuators, the I&C system monitors all the aspects of performance and safety of the NPP and allows the operators to act on the process in any situation. It also responds to failures and off-normal events, thus ensuring goals of efficient power production and safety. Essentially, the purpose of the I&C system of a NPP is to enable and ensure safe and reliable power generation. Therefore it is required to be a system of high quality to ensure a resilient and efficient exploitation of the plant. At the same time, it is also characterized by a very long life cycle: the design and implementation phase lasts about 10 years and the operating life lasts at least 40 years.

Continuous technological advances in computer science and electronic engineering offer favorable conditions for completing and/or gradually replacing analogue devices of the original design with digital components for nuclear I&C systems. In addition, instead of building a completely customized I&C system, it is now possible to purchase components off-the-shelf (COTS) and then to integrate these ready-to-use elements. In addition, during its long life cycle, the I&C system is subject to partial renovations resulting from either the evolution of requirements or the modification of physical equipment. These partial evolution and modifications can be frequent and integrate to different phases of system life cycle. This brings not only productivity gains but also a challenge for the nuclear industry to ensure that the new system meets all safety, reliability and performance requirements.

Since 2012, the main industrial and academic partners of the French nuclear industry initiated an ambitious R&D program called "CONNEXION". Regrouping a number of projects, "CONNEXION" [71] aims to improve the development process of the I&C system of nuclear power plants. "CONNEXION" is based on the existing expertises of major operators in French nuclear industry (EDF, ALSTOM, AREVA and RRCN) and various software tools provided by other industrial and academic partners (CEA, CORYS, ESTEREL Technologies and ALL4TEC). In the following we first present the approach used today at EDF for the development of I&C system and then focus on the needs of the project and the challenges associated.

Triple V system life cycle

The I&C system of a nuclear power plant is composed of several hundreds of **Elementary Systems (ES)**, controlling with a very high safety level thousands of remote controlled actuators: about 8000 binary signals and 4000 analog signals sent to the control room, concerning over 10 000 I&C sub-functions and over 300 I&C cabinets. Each ES is a set of circuits and components, performing an essential

function to the operation of the nuclear power plant. Each ES is documented by an elementary system dossier (ESD), containing documents detailing the ES in different aspects: operation, control/command, equipment, etc.

Every elementary system is composed of two sub-systems:

- the **Process** represents the physical infrastructure and equipment, e.g., heat exchangers, valves, pipes, etc.
- the **control/command (CC)** is a real-time reactive system permanently interacting with the Process. It is responsible for protection, control, and supervision of functioning of the Process.

The Process and the control/command depict different aspects of the elementary system and correspond to different documents in the ESD. The control/command, describing the functional aspect of the ES, is specified using **functional diagram (FD)**. It is a formal language dedicated to the nuclear sector based on predefined blocks. The specification of this language complies with IEC (International Electrotechnical Commission) standard 61804-1 [4] and the implementation is defined by elementary blocks with reference to IEC standard 61131-3 [11].

To develop the control-command of one ES, the current approach practiced at EDF gradually transforms a purely mechanical representation into a functional diagram. The FD is then implemented into executable code by automata dedicated to this purpose. The approach integrates expertise of several domains including mechanics, functioning and control engineering. The current verification and validation process of the control-command leads to functional tests carried out on a set of interconnected previously validated systems. These systems are automata implemented with executable code of control/command applications. See [71] for more details.

The original V-model life cycle, corresponding to the current approach, contains phases 1, 2, 3, 7, 8, 9 and 10 (Fig. 3.1). The left side of the V-model (1, 2, 3, 7) represents the design and implementation of the system and the right side (8, 9, 10) represents the system verification and validation. It is important to note that the left side of the V-model relates to a single elementary system but the verification and validation activities are performed by integrating this ES with an abstraction of its environment [71]. Moreover, the granularity of the model of the Process used in co-simulation is adapted to the validation objectives specific to each of these last three phases (8, 9 and 10).

The design begins in phase 1 with a global modeling of the functional specification of the ES from the simplified diagram of the Process and its various operating configurations. The simplified diagram of the Process is a functional specification for both the physical environment ant its control/command [71]. In phase 2, a functional specification dedicated to control/command is developed in the form of a functional diagram by system engineers. This FD represents an explicit specification of the expected control/command behavior consistent with the Process diagram and the description of functional requirements produced in phase 1. This design specification is then progressively detailed into a refined functional diagram (RFD) in phase 3, which is ready to be transformed into programs implemented in phase 7. The system V&V begins in phase 8 by verifying the implementation (at the output of phase 7) with respect to its design specification (phase 3). In phase 9, the control/command of the ES is integrated with the other elementary systems already validated. At the end of the life cycle (phase 10), Hardware-In-the-Loop (HIL) [30] techniques is applied to all interconnected and validated ESs in phase 9. Through a virtualized real environment, developers are allowed to verify the functioning of implemented automata before their actual on-site integration and the corresponding final validation tests.

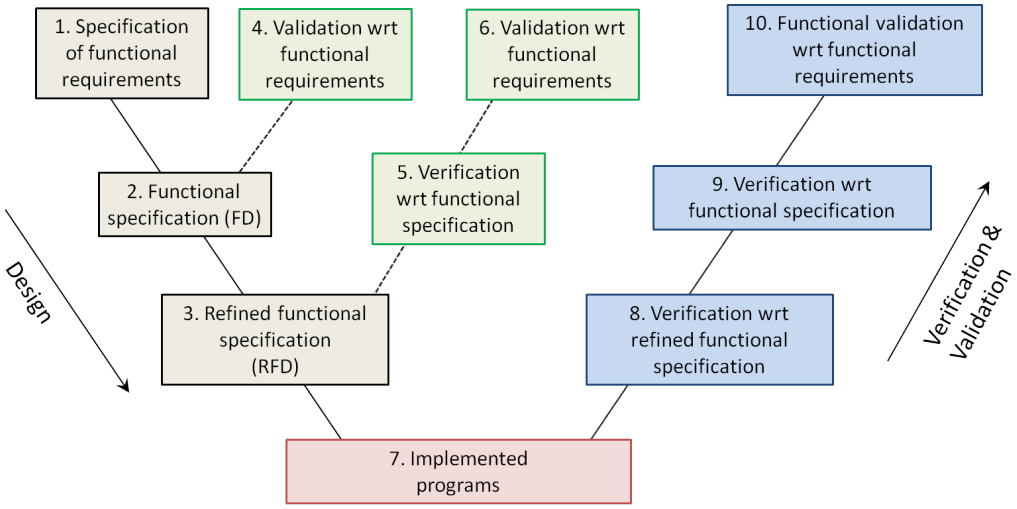


Figure 3.1: Triple V system life cycle of the I&C system

The current approach, corresponding to the left side of the V-model, provides the elements for the production of elementary dossier. This approach is based on various expertise knowledge and produces control/command models in form of functional diagrams. The FDs are formally verifiable but not directly executable. As consequence, functional validation of the FDs are manually performed. Au-

tomated testing is performed only at the level of implementation. The current approach, clearly fulfills the high quality requirements defined in the standards of nuclear sector. However the cost is very high considering the inherent evolution during the development process.

The "CONNEXION" project searches to improve functional validation efficiency by introducing the first V sub-cycle (phases 1, 2 and 4) and the second V-sub-cycle (phases 1, 2, 3, 5 and 6), as shown in Fig. 3.1. The two sub-cycles first introduce **executable** control/command models: at phase 2 and 3, executable functional specification corresponding to the FD and RFD are developed. Phase 1 introduces a model describing the high level elementary system. The executable model of the FD produced at phase 2 is verified with respect to the functional requirements (phase 4). The executable model of the RFD produced at phase 3 will be (i) verified with respect to the model of functional specification developed at phase 2 (phase 5) and (ii) verified with respect to the functional requirements (phase 6). The two V sub-cycles results in an innovative triple V system life cycle. A similar approach referred to as "multilevel testing for design verification" [152] has been proposed for embedded systems.

The "CONNEXION" approach, similar to the current approach, still requires integration of multiple domain expertise. However, by introducing a model in the loop (MIL) [30] approach, functional validation can be automated as much as possible thanks to executable models of FD and RFD as well as appropriate tools. Development process is aligned with IEC standard [5], encouraging test automation tools. The current life cycle relies on a document-centric system engineering approach; "CONNEXION" enables the transition to a model-centric practice as advocated by the INCOSE [7]. The introduction of the two V sub-cycles answers for a key challenge of Industrial Cyber-Physical Systems: early stage of Verification and Validation [82].

Towards a complete verification platform

Another objective of "CONNEXION" is to create a complete verification platform supporting functional validation methodologies presented above. The platform would accompany all the V&V activities required in the development of the I&C system throughout its life cycle. For one elementary system, the platform integrates the Process model, the control/command model as well as a human machine interface (HMI) dedicated to control operators. Supported by model-based testing tools, the platform is able to generate automatically test cases and to execute these test cases by co-simulation of the Process and the control/command. Since

the Process and the control/command are modeled using different languages, they are integrated to the platform via Functional Mock-up Interface (FMI). This independent standard is designed to support both the exchange of models between partners and the co-simulation of dynamic models.

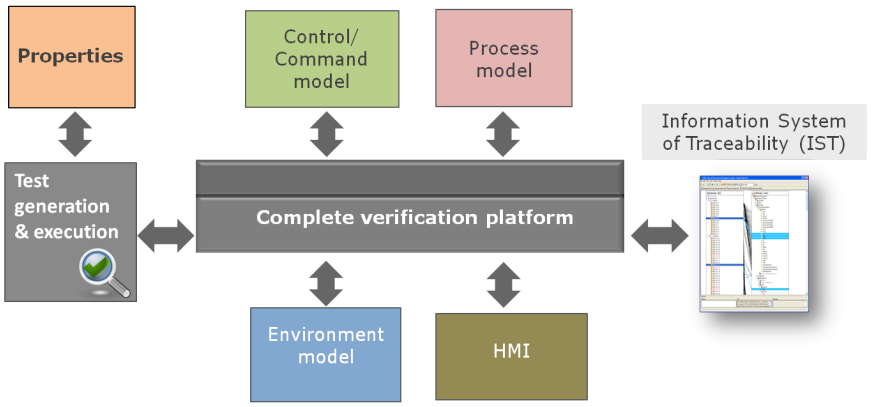


Figure 3.2: A complete verification platform

Fig. 3.2 depicts the principle of such a platform at the level of one elementary system. Through the co-simulation of the Process and the control/command, this platform enables system engineers to verify various aspects of the system during its development phases. The model of the expected properties of the system and the environment model, as perceived by the control/command, are also necessary to describe in particular the requirements that the elementary system must respect and the constraints on its solicitation. We specify that the environmental model introduces the constraints that come from other interconnected elementary systems to the elementary system under test. It is quite possible to perform a context-free verification of the ES under test, which in this case does not require any model of the particular environment. Finally, we think the platform should be supported by an information system of traceability (IST). The IST accompanies activities of verification and validation and saves not only the data but also the history and relationship between them.

The verification activities are based on the ALICES simulator[2] provided by CORYS[3]. ALICES is able to manage the interaction between several elementary systems. For one elementary system, ALICES allows the co-simulation of Process and control/command as well as HMI animation. The co-simulation environment is not full scale, but it is realistic. ALICES can integrate different simulation or

[2]http://www.corys.com/en/alicesr
[3]http://www.corys.com/en

animation modules, either in its proprietary technology, or by interfacing with Scade models for the control/command and Modelica [88] models for the Process.

3.2 Unique and complete tool box

The tools brought by partners of "CONNEXION" can be divided into two categories: modeling tools and verification tools.

Modeling tools

We hereby briefly present the modeling tools chosen by "CONNEXION". More details can be found in [159].

Process model:

- Dymola[4] is a commercial modelling and simulation environment based on the open-source Modelica language. *Dymola* is used to model the Process.

High-level model of the ES:

- Based on Eclipse platform, Papyrus [90] from CEA offers an open-source graphical editor for modeling in the formal language SysML [10]. In "CONNEXION", a high-level model of the ES is developed in SysML with *Papyrus*. This model has a high abstraction level, corresponding to that of the specification of functional requirements.

Models of the control/command:

- SCADE Suite [58] from ESTEREL Technologies has been chosen. Based on the Scade formalism [34], SCADE Suite is specialized for designing safety-critical systems as its code generator is qualified under several certifications. In "CONNEXION", SCADE Suite is used to specify models of the control/-command at different abstraction levels, corresponding to the FD and RFD.

Verification tools

Verification tools automate as much as possible various activities such as formal verification, test generation, test execution, coverage analysis, etc. Table 3.1 gives an overview of these tools.

[4]http://www.3ds.com/products-services/catia/products/dymola

MaTeLo	Model-based test generation tool
GATeL	Model checker for Lustre models
SCADE QTE	Coverage analysis for Scade models
ALICES	Co-simulation platform
ARTiMon	Real-time test observer

Table 3.1: Overview of "CONNEXION" verification tools

- MaTeLo [58] from ALL4TECH[5] is a model-based test generation tool for statistical usage testing [123], allowing testing the system from the user's angle. Test generation is based on a usage model specified using markov chains, created manually from the specification of functional requirements. One test case corresponds to a path randomly chosen from the usage model. In "CONNEXION", the usage model is built according to the functional requirements of the ES and the test cases generated are intended for closed-loop simulation.

- Since the Process model and the control/command model are in different formats, the closed-loop simulation is performed on a platform: ALICES [136] from CORYS. ALICES is responsible for the exchange and synchronization of data between models through the open source standard FMI/FMU.

- The test suite generated by MaTeLo does not include the expected outputs (oracles) to be compared with the actual outputs. The oracles are performed by a test observer also integrated to ALICES: ARTiMon [90] from CEA. The properties to verify are manually formalized in ARTiMon. During the co-simulation, ARTiMon provides real-time observation of the execution results so that any violation of the properties will be recorded.

- After the execution of test cases, the exchange of data between the Scade model and the Process model is documented. These records are necessary to measure the structural coverage of the test suite by another tool from the SCADE tool kit: SCADE QTE (Qualified Testing Environment) [1].

- The model checker GATeL from CEA, works on models specified using the synchronous Lustre language. It verifies properties that are invariant or characterizations of reachable states of the system. With a test objective expressed in an extended version of Lustre, GATeL generates test data which drives the system to a state satisfying the test objective. Scade models can be automatically transformed into Lustre models by the *s2d* tool[6].

[5] `http://www.all4tec.net/`

[6] The *s2d* tool is developed and provided by *Laboratoire Sûreté des Logiciels, CEA/DRT/DT-*

3.3 “CONNEXION”: challenges and constraints

Sub-systems modeled in different languages

For each elementary system, the Process is modeled using Dymola and the control/command system modeled using SCADE Suite. The test generation tool MaTeLo generates closed-loop test based on the functional requirements; the oracles are provided by ARTiMon, a real time observer integrated to the execution platform ALICES; the structural coverage is measured on the control/command model by SCADE QTE; finally the model checker GATeL takes the control/command model in Lustre, obtained by an automatic transformation of the control/-command model in SCADE Suite.

GATeL is a model checking tool oriented to test generation. It is able to perform conventional model checking problem, i.e., property verification. It also offers user-definable test objectives which can be used to force test sequence generation. GATeL takes as input the control/command model and therefore generates open-loop test sequence. In our methodology, we use coverage-based test objective to force GATeL to generate open-loop test sequence. However, a suitable test case for functional validation must be closed-loop and related to functional requirements. The passage from open-loop to closed-loop and from coverage-based to requirement-based demands collaboration of testers and system engineers. Our methodology includes a process refining test generation using GATeL (Sec. 4.2.2), so as to help with the passage from open-loop to closed-loop.

Multiplication of models and tools

The “CONNEXION” approach proposes a multi-model and multi-tool environment for the development of I&C system. Each test model and associated tools should help improve productivity and increase confidence in the verification and validation cycle. The advantages of this approach are obvious: automation of functional validation at early design stages can detect a defect as soon as possible. Moreover, the choice of formal models and powerful verification tools ensures a high level of quality of the system under consideration.

However, we believe that the multiplication of models and tools can quickly become counter-productive. Each model and each tool ”speaks” their own language, which requires extra time spent in learning and understanding. It may also be necessary to spend time in transformations from one model to another. Moreover, there may be a redundant overlay between the capabilities of different tools. It is

SI/SOL, 91191 Gif sur Yvette, France

therefore important to address these challenges by proposing a clear methodology for the use of these tools. For example, choose carefully and even parsimoniously any additional model. Each new model or tool from this first selection must show a value of both verification efficiency and productivity. Always know how, when and why to use a model. Finally, ideally, it is necessary to understand how a multi-model environment can work "seamlessly" in a fluid way by moving from one tool to another in a flexible and clear workflow.

Another aspect concerns the transformations between models: recreate the FD and RFD in SCADE Suite and transform them into Lustre models for GATeL. We assume for now that these models are equivalent. The proof of equivalence remains an open question.

Modification request

A test can show two kinds of problems: false negatives or false positives [130]. A problem can be located in various places in the verification chain: not only in the model under test, but also in the test itself, e.g., the desired property may be poorly formulated. It may also due to a defect of the verification tool itself or to the semantics of the model [131]. The multiplicity of tools can be beneficial in this respect: a verification tool may not find exactly the same result as another tool, because the properties to be verified have not been described in a completely equivalent way. Therefore verification by a tool at an abstraction level may shed light on the verification by another tool. It is thus conceivable that it is even possible to detect a false positive: a real problem in the model which is not detected at a phase when it already exists and which is just detected at another phase by another tool.

Finding the same problem at several levels of abstraction of the description can thus be an integral part of a global diagnostic methodology and allows classification in terms of the severity of this problem. Reproducing the same problem under a series of tests rather than just one will also be part of the methodology to understand the extent and locate the origin of the problem.

3.4 Information system of traceability

We think that a complete verification environment should include an Information System of tracebility (IST). The IST is expected to improve the verification efficiency especially when the system is being upgraded. The IST should save not only the data but also the history and relationship between them. Moreover, the IST must keep records of the operating conditions of the simulation environment.

That is to say, the control/command and the Process configuration related to the state of the NPP unit: starting, operating at full power or at reduced power, cold stop, etc.). Here are some basic elements that the IST should deal with:

- models of the system, properties under test, testing environment and data, e.g., values, scenarios, simulation history, etc.
- traceability between one model and another
- development process of different states and variables
- what tools of which versions are used to perform what tests, etc

It will provide a better understanding of the test coverage and minimize the effort to test non-regression.

HP Quality Center (QC)[7] from HP is a leading commercial software testing management tool. HP QC offers an environment supporting essential aspects of testing management: requirements management, test planning, analysis of results and management of defects and problems. The "CONNEXION" methodology contains multiple models and verification tools and the traceability is therefore more complicated than what HP QC is able to manage.

Design of the IST using UML diagrams

The design of our IST is specified in UML language on the open-source platform StarUML 2. UML is a graphical modeling language allowing to specify, visualize, modify and construct documents necessary for the development of an object-oriented software. It offers a modeling standard to represent the software architecture. The 14 charts proposed by UML can be divided into two categories: structural diagrams and behavioral diagrams. We present two diagrams as the IST models: a use case diagram and a class diagram.

The use case diagram of the IST (Fig. 3.3) belongs to behavioral diagrams. It allows to identify the possible interaction between the IST and the actors outside the IST. In other words, the use case diagram describes all the functionality that the IST must provide. The IST first proposes to the user functions of models and requirements management, e.g., registration, modification and search by keywords, etc. Then, to verify a model with respect to certain requirements, the user must set a test objective such as verification of a requirement under certain

[7] `https://saas.hpe.com/en-us/software/quality-center`

configuration of the NPP. During this activity, depending on the user needs, other functions may be called: saving or canceling this test objective or search for tests with similar objectives (verification of the same requirement under different NPP configurations for example). Test execution starts with choosing a verification tool while recording some information Of the tool: name, version and installation date, etc. As mentioned previously, closed-loop test execution are performed on the ALICES platform. We precise that the verification tools and simulation platform are external entities of the IST and therefore the same are their activities including test generation and execution. The IST is only capable of managing the results of these activities such as test case files and test results management.

The class diagram of the IST (Fig. 3.4) is part of structural diagrams. It represents classes involved in the system as well as all the relationships between them.

- The **Model** class contains several necessary attributes to keep the traceability: name, version, development date, type (Process or control/command) and the underlying formalism. The **ModelsManagement** class generates all objects from the **Model** class and offers functions such as search for one or more models by keywords, modify a model and add or delete a model.

- The **Requirement** class has a structure similar to the **Model** class. All objects in the **Requirement** class are managed by the **RequirementsManagement** class in a similar way.

- The **TestCase** class represents the smallest unit of test. A **TestCase** contains a nuclear unit configuration and a data set. A configuration is defined by the nuclear unit status, e.g., start, normal or degraded functioning, stop; as well as environmental conditions, e.g., hot/cold period, the status of other systems in interface, etc. Verification of one **Requirement** requires usually more than one **TestCase**. A collection of **TestCase** compose one **TestSuite** which has a well defined test objective. The **TestSuite** class also contains a reference to an object **TestTool** which represents the verification tools related to the corresponding **TestSuite**. Each **TestSuite** produces a **TestResult** which specifies the test result status (successful, unsuccessful or unknown), the date of test execution and an error description if necessary. All the tests and results are managed by **TestTrackManagement**. It proposes functions to save or delete one **TestCase**, one **TestSuite** or one **TestResult**. In addition, the association between **Requirement** and **TestSuite** and that between **Requirement** and **Model** allow to keep the traceability between all the key elements of the verification activity (model,

requirements, tool, tests performed and results). For example, in **TestTrackManagement**, one can assemble by a search query all the **TestSuite** related to one **Model**, or all the **TestResult** associated with verification of one **Requirement**, etc.

Finally, the features of **ModelsManagement**, **RequirementsManagement** and of **TestTrackManagement** are assembled by **UIManagement**, the interface between the user and the IST. It associates the actions of the user with the functions provided by these three management classes and brings the results to the user with a proper visualization.

Implementation Prototype

Fig. 3.5 shows the structure of this prototype at code level. The code is organized in two "layers": a storage system and a Java application. The communication between the two layers is accomplished by JDBC (Java DataBase Connectivity)[8] which is a programming interface allowing Java applications to access databases.

The storage system is divided into two parts: a system of files for simply storing the files of SCADE Suite models and Modelica models, the files of test cases, etc; another system stores previously defined classed in the UML class diagram (Fig. 3.4) in the form of tables with a well-defined structure. The latter requires a DBMS (Database Management System) such as MySQL[9], an open-source commonly used database management software.

At the Java application level, **TrackManagement** offers functions such as saving, deleting, updating, searching for objects (models, requirements, tools, tests and results). The user interface **UIManagement** sends the user's actions to the functions of **TrackManagement** and retrieve and then display the results.

[8] `https://fr.wikipedia.org/wiki/Java_Database_Connectivity`
[9] `https://www.mysql.com/`

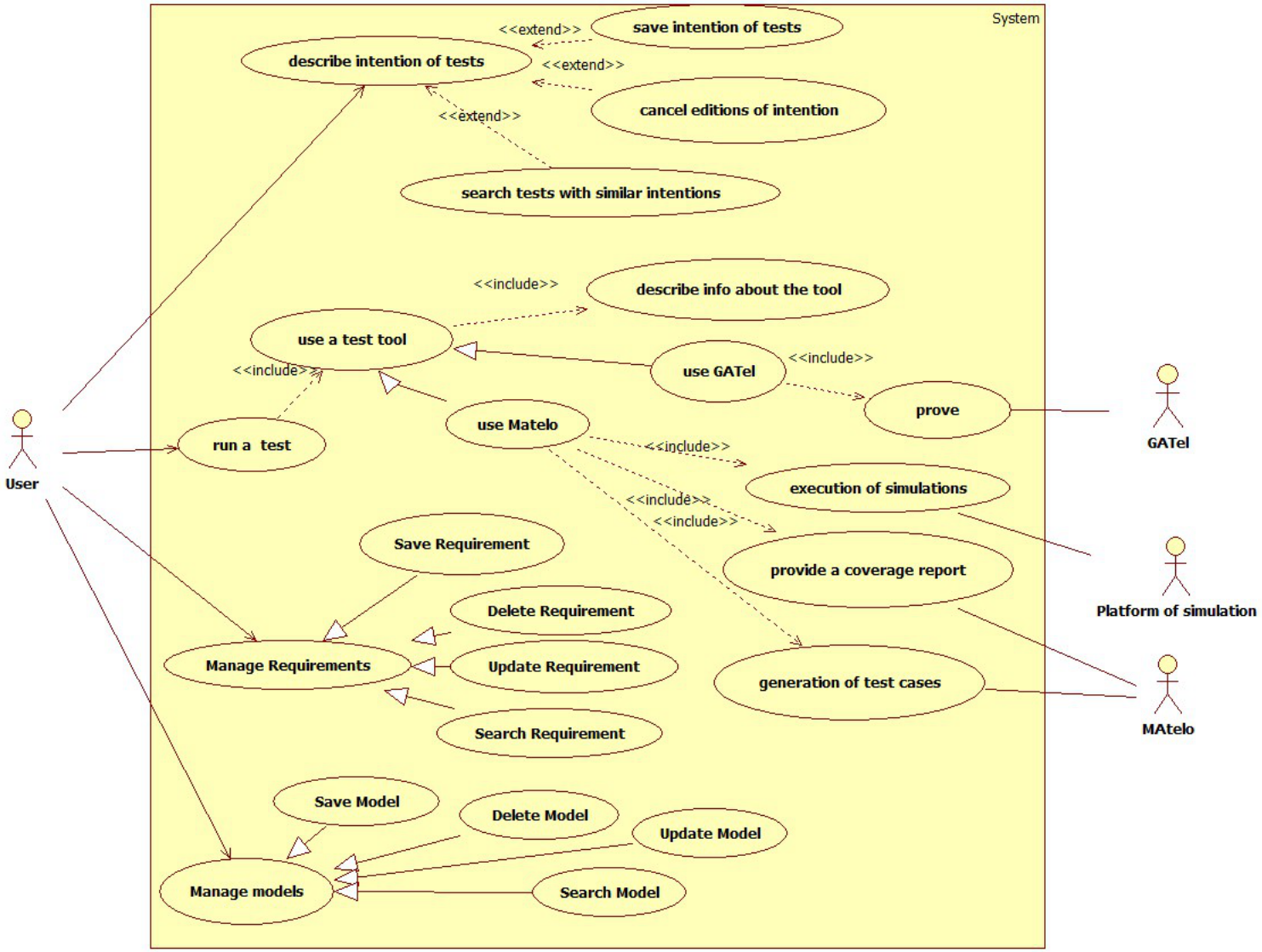


Figure 3.3: UML use case diagram of the Information System

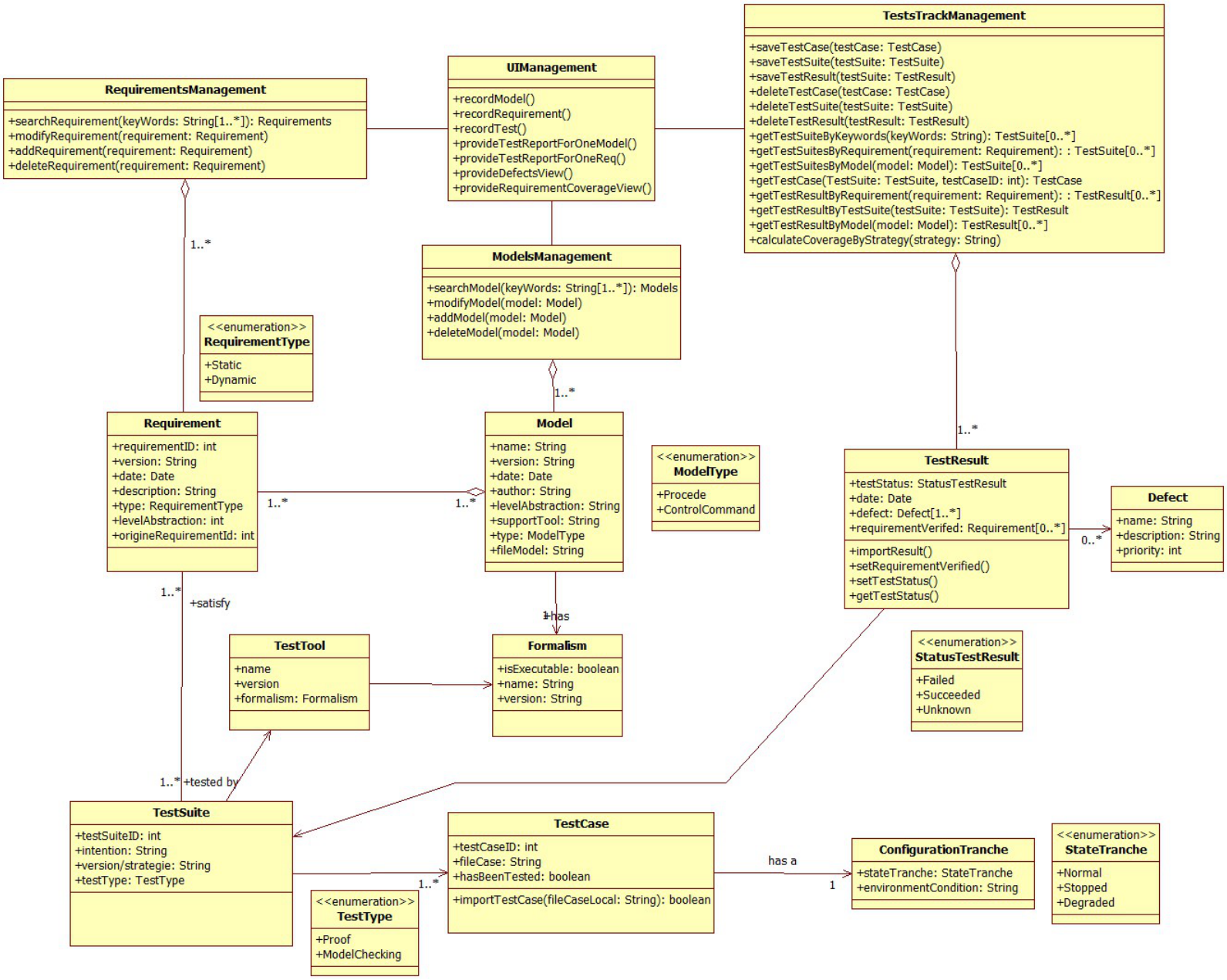


Figure 3.4: UML class diagram of the Information System

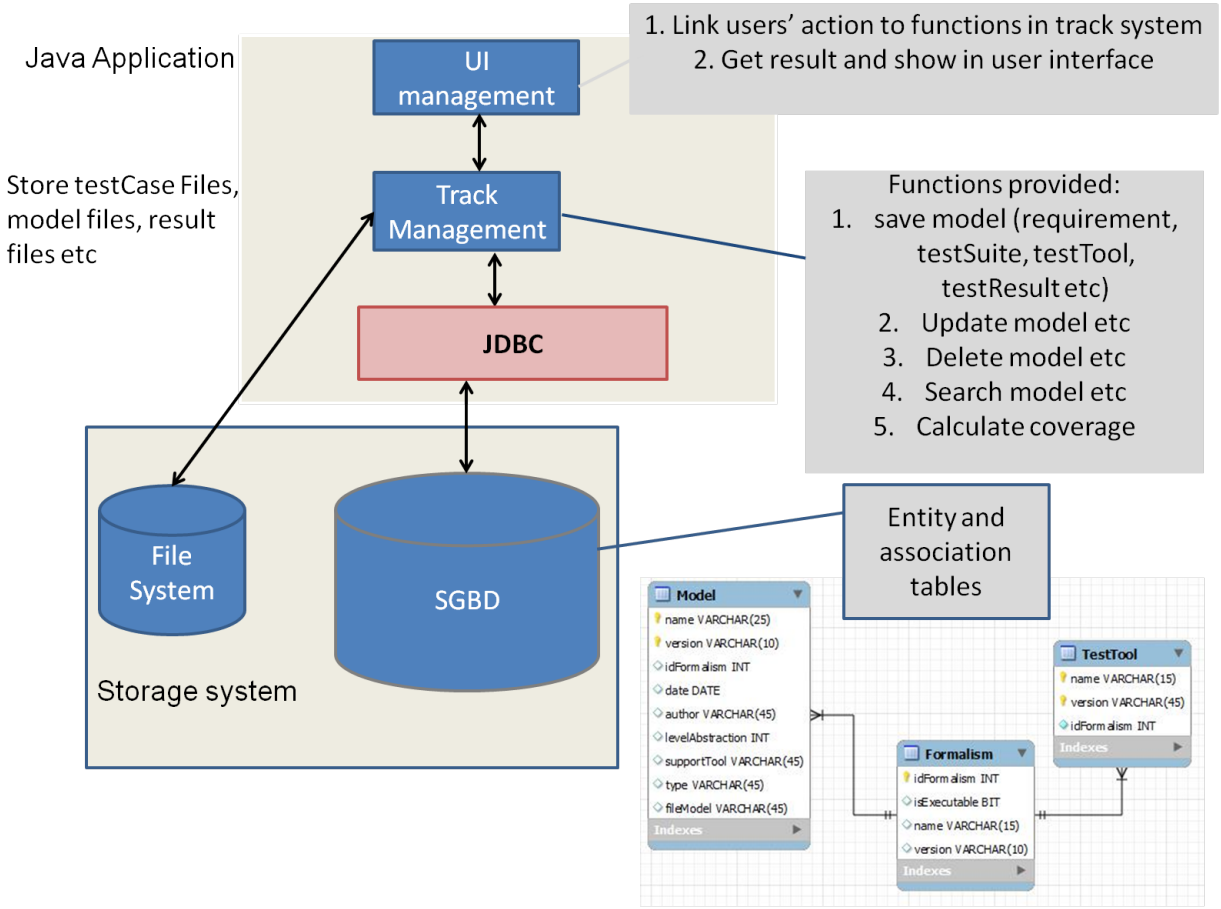


Figure 3.5: Prototype

Chapter 4

Model-based testing for functional validation

Towards hybrid verification

In this thesis, a methodology is defined as a collection of related processes, methods, and tools. A model-based testing methodology is therefore the collection of related processes, methods, and tools used to support the verification and validation of a system in a model-based context [79]. Sec. 4.1 presents the generic model-based testing process and a few testing-related concepts. Sec. 4.1.2 discusses our methodology including terminology, processes, tools and the "hybrid verification" heuristic. Sec. 4.2 illustrates a new technique to refine test generation with model checking, by gradually adding constrains from the physical environment.

4.1 Model-based testing process

Model-based testing (MBT) is about generating tests from a model of the system under test (SUT). Utting *et al.* defined the generic process of model-based testing in [166]. Fig. 4.1 illustrates this process. MBT tools are software tools that automate MBT activities.

Step 1: A model of the SUT is developed from the requirements of the system. This model represents behaviors of the SUT with a level of abstraction. The model must be simpler than the SUT, otherwise the validation of this model will be equivalent to that of the SUT. However the model must be close enough to the SUT so as to generate relevant test cases.

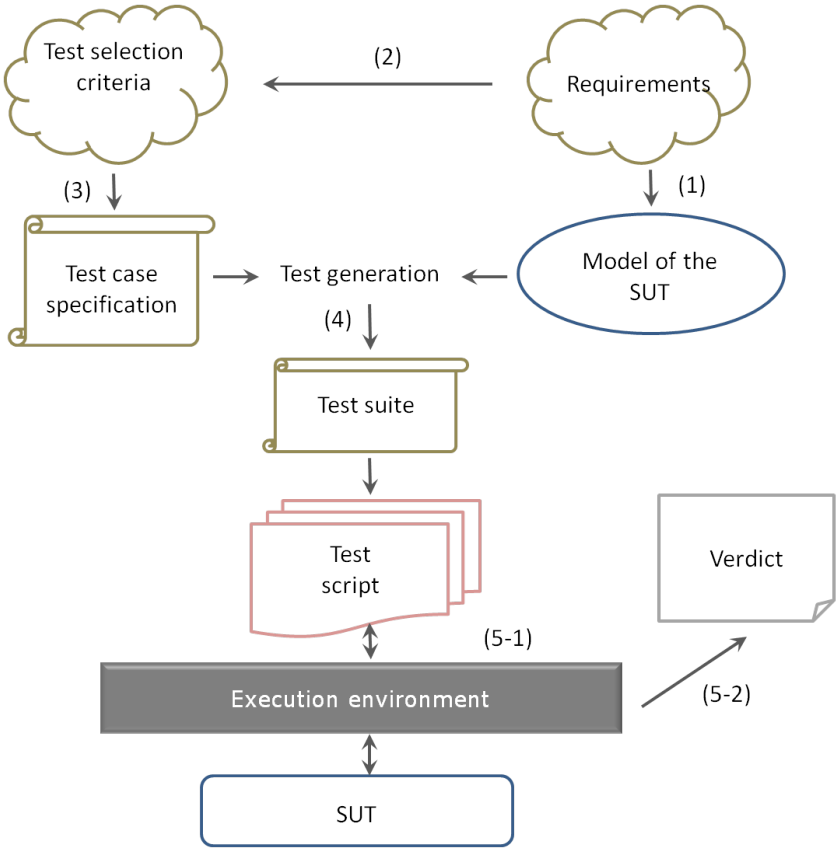


Figure 4.1: Model-based testing process

Step 2: The test selection criteria are defined. These criteria describe **informally** the objectives of the test cases that will be generated by the MBT tools. These criteria may refer to a certain functionality of the system (criterion based on requirements) or to the structure of the model (state coverage, transition coverage, i.e. non-functional criterion); or to stochastic characterizations.

Step 3: The test selection criteria are then translated into test case specification which formalizes the criteria and make them operational.

Step 4: The model of the SUT and the test case specification being well defined, the MBT tool is capable of generating test suites. A test suite is a collection of test cases satisfying one test case specification. Test cases are interpreted as pairs of inputs/outputs of the model: the outputs are those expected from the system under test.

Step 5: The test cases are executed. The execution consists of two steps:

Step 5-1: Test cases are manually translated into executable scripts and sent to an execution environment. The execution of a test case involves applying to the SUT the inputs of the test case and recording the responses of the SUT which are the actual outputs.

Step 5-2: A verdict is the result of comparing the actual outputs of the SUT with the expected outputs defined in the test suite/script. A verdict can produce three possible outcomes: (1)OK if the actual outputs consistent with those expected; (2)Not OK if the actual outputs do not consistent with those expected; (3)unknown if the result is not clear enough. Recall that the model represents the behavior of the SUT with a level of abstraction. Since test cases are generated from the model, applying the inputs of test suite to the SUT requires a concretization of them. Similarly to compare the actual outputs of the SUT with the expected outputs, it is necessary to map real outputs to test suite outputs [144].

Model-Based Testing has been more and more applied to industrial projects. For example, to automate functional testing of a particular class of programmable logic controllers[162]. Testing with model checker (Sec. 2.2) qualifies as model-based testing approach. Our methodology focus on exploring the test sequence generation ability of GATeL. Therefore, the model checker GATeL can also be considered as a model based testing tool.

Testing strategies

In general, there are two mainstream testing strategies: black-box testing (or functional testing) and white-box testing (or structural testing).

The goal of black-box testing is to find errors in the program by testing the functions listed in the specification. Designing test set does not require analyzing the details of the program but using the specification. On the contrary, white-box testing focuses on the structure of the program. A test set in which every line of the program under test is executed at least once is an example of white-box testing.

A mixture of different strategies can be used to improve the effectiveness of testing. For example, since black-box testing does not require knowledge of the structure of the program, there may be some parts of the program unreachable because they are defective or insensitive to certain inputs, these problems may not

show up in functional testing.

The "CONNEXION" project focuses on functional validation of the models of I&C system developed in early design phases. Test cases for functional validation should verify these models with respect to the functional requirements. Meanwhile the I&C system is highly safety critical. Therefore the functional test cases are also required to "cover" (execute) as much as possible the structure of models (or at least some parts). This inspired us to design a methodology combining black-box and white-box testing strategies. These two strategies naturally lead to different coverage criteria discussed in the following.

4.1.1 Coverage criteria

Coverage criteria indicate how adequately the testing has been performed. According to the testing strategies presented just above, there are at least two categories of coverage criteria : requirement coverage and structural coverage. We hereby adopt the definitions of different coverage criteria given in [165].

Definition 16. *Requirement. A requirement is a testable statement of some functionality that the system must perform.*

Definition 17. *Requirement coverage. It demands that all requirements are covered in the functional test set. In other words, a measurement of the requirements that are covered in the test set indicates how well the functional testing has been performed.*

For structural coverage, many criteria have been discussed in the literature [173]. Statement coverage and branch coverage are two most widely used criteria in practice. A measurement of statements or branches covered in the test set indicates the test adequacy.

Definition 18. *Statement coverage. The test set must execute every reachable statement in the program. The coverage rate of a test set is the ratio of the number of executed statements to the total number of statements.*

Definition 19. *Decision coverage (or branch coverage). The test set must ensure that each reachable decision is made true and false at least once.*

Testing of a decision depends on the structure of that decision in terms of conditions: a decision contains one or more conditions combined by logic operators (*and, or, not, etc.*). Several decision-oriented coverage criteria are hence derived.

Definition 20. *Multiple condition coverage (MCC). A test set achieves MCC if it exercises all possible combinations of condition outcomes in each decision. This requires up to 2^N test cases for a decision with N conditions.*

Definition 21. *Modified condition/decision coverage (MC/DC). A test set achieves MC/DC when each condition in the program is forced to true and to false in a test case where that condition independently affects the outcome of the decision. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. For a decision containing N conditions, a maximum of 2N test cases are required to reach MC/DC.*

Indeed, these different structural coverage criteria are not equally strong. For example if we reach MCC then we reach MC/DC since MCC requires more test cases than MC/DC. Similarly decision coverage is stronger than statement coverage. More detailed information about the hierarchy of coverage criteria can be found in [165].

Requirement coverage and structural coverage are somewhat independent to the sense that a 100% requirement coverage does not guarantee a 100% structural coverage and vice versa. In practice, the functional testing of large scale system is usually performed by executing a set of functional test cases in a harness.

4.1.2 A new MBT methodology for safety-critical systems

In "CONNEXION" this methodology is being applied to functional validation of the I&C system models developed in early design phases. We recall that the I&C system of a nuclear power plant is composed of several hundreds of Elementary Systems (ES) and each ES is composed of a Process (physical environment) and a control/command system (RTS) reacting permanently to the Process.

Terminology

Definition 22. *Structural unit (SU). A SU is the unit of coverage measurement on the model regardless of the coverage criteria chosen.*

Definition 23. *Reachability check. For a given structural unit, a model checker verifies formally whether this structural unit can be executed by any test.*

We also recall some reappearing definitions that have been given in introduction.

Definition 24. *Open-loop test. In an open-loop test, only the real-time system itself is executed. Behavior of the environment permanently interacting with the RTS is not taken into account.*

Definition 25. *Closed-loop test. In a closed-loop test, the real-time system and its environment are executed together, also called co-execution. The behavior of the RTS is thus influenced by the environment.*

Workflow and tools

Our methodology is composed of three main phases.

- **Phase 1:** Generation of test cases based on test objectives derived from high-level functional requirements. These functional test cases are then executed through co-simulation of the environment and the RTS. A model-based functional testing tool is utilized for test generation (MaTeLo in "CONNEXION"). A simulator supporting closed-loop test execution is another requirement (the ALICES platform in "CONNEXION").

- **Phase 2:** After the execution, structural coverage of these test cases is measured (MC/DC coverage in "CONNEXION"). It is important to notice that the coverage is measured only on the RTS. Uncovered structural units are collected. A coverage monitoring tool is required in this phase (SCADE QTE in "CONNEXION").

- **Phase 3:** For each uncovered SU, a model checker is utilized to generate test sequences executing the SU under consideration. The model checker takes as input the RTS model and generates open-loop test sequences. With the help of system experts, closed-loop test cases can be developed from these open-loop test sequences. New test cases should not only improve the structural coverage but also are related to the functional requirements. This process is iterated on every uncovered SU until the coverage criteria are satisfied. Tools required in this phase includes a model checker (GaTeL in "CONNEXION"), for coverage-based test generation. The functional (closed-loop) test generation tool used in phase 1 is also needed for building a functional realistic test case covering the SU under consideration.

Let's now take a closer look at phase 3. For each uncovered structural unit, the model checker is used to perform a reachability check in order to prove whether this SU can be covered by any test. If a SU is found reachable, the model checker generates test sequence that covers this particular SU and, interestingly enough this test could also cover other SUs. It is possible that the SU could not be reached

by the model checker, either because it is truly unreachable or because the model checker encounters a "time out" (TO). In the first case, this SU will be recorded for further analysis since it can be dead code or even the manifestation of a bug, which would require a fix. The second case is also (imperfectly) addressed since we propose using hybrid verification similar to the work presented in [131] combining model checking and simulation (Sec. 4.1.3).

For the nuclear I&C system, the Process and its control/command(CC) are modeled using different languages. Test sequences generated by the model checker can be interpreted as open-loop test, i.e., executable only on the CC system. An open-loop test does not take into account how the Process reacts to the outputs produced by the CC system; or what kind of data the Process can realistically send as inputs to the CC. Our research work concentrates on using model checkers to produce open-loop test sequences. How to develop closed-loop test from open-loop data requires expertise from the I&C system experts and is not in the scope of the thesis. In the practice, closed-loop test is more realistic and valuable than open-loop test and sometimes necessary or even required. Therefore our methodology proposes some techniques to help with the passage from open loop to closed loop. These techniques are based on formalizing properties describing behavior of the physical environment as constraints for the model checker and using these constraints to progressively refine the generation of open-loop test (Sec. 4.2)

Considering the length of the methodology, we decide to break it into two parts. Part 1 deals with the SUs that have been found either reachable or not reachable in the reachability check. Part 2 discusses the third possible outcome of reachability check: the model checker has TO. The methodology of part 1 has been tested on the "CONNEXION" case study with support of "CONNEXION" tools (Sec. 5.1). To implement hybrid verification proposed in part 2, we have tested several academic model checkers but none of them could offer all the required techniques (Sec. 5.3).

MBT methodology: Part 1 of 2

Fig. 4.2 illustrates the methodology part 1 including methods, processes and tools that are requested for application. The process can be generalized for various structural coverage metrics.

A functional test generation tool is first used to derive a functional test suite (TS) according to the functional requirements (step 1). This test suite is then executed in an environment on the model of the system (step 2). The structural coverage (SC) of the test suite is measured by a coverage monitoring tool after

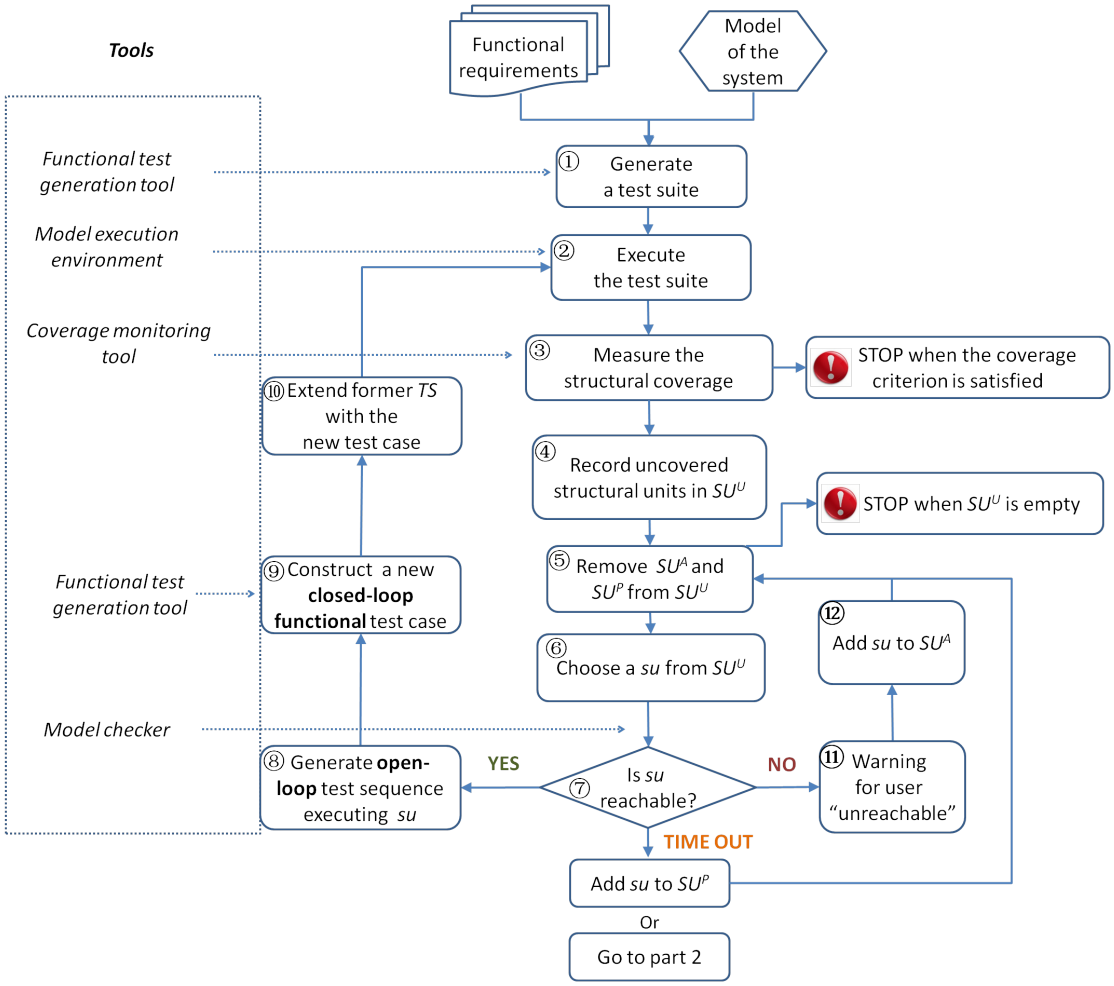


Figure 4.2: Model-based testing methodology: part 1 of 2

the execution (step 3). We define SU^U as the set of all uncovered structural units after the execution of TS (step 4). SU^A is the set of all **actually unreachable** structural units and SU^P the set of all **potentially unreachable** structural units, for which the method did not succeed to answer the reachability question. Initially they are both empty. Take an uncovered structural unit su from SU^U (step 6) and apply a model checker to check if su is reachable (step 7):

- If su is not reachable, send a warning message to user (step 11) and record su as an **actual** unreachable structural unit (step 12): $SU^A_j = SU^A_{j-1} \cup su$. Go to step 5 and continue the following steps.
- If su is reachable, the model checker must have produced open-loop test data that forces the system to reach su. These data will be used to construct

a closed-loop functional test case (denoted ntc) that covers this particular structural unit and possibly others (possibly in SU^P which needed to be computed again) (step 9). At this point a return to the high-level functional requirements is required to ensure the functional reality of ntc. For systems composed of different sub-systems modeled by different formats, the functional test generation tool may also be necessary to prepare a closed-loop test case. Complete the former test suite TS with this new test case (step 10): $TS_i = TS_{i-1} \cup ntc$. Go to step 2 and continue the following process.

MBT methodology: Part 2 of 2

Fig. 4.3 depicts the part 2 of our methodology. The third possibility of the reachability check of a SU (step 7) is TO: the model-checker stops its execution without giving an answer. Our solution is first to increase TO then to apply hybrid verification (i.e. a combination of model checking and simulation) similarly to [131] to check the reachability of this su (step 13). If the su is reachable then go to step 8 and continue the following process. Hybrid verifcation can also "time out" which leads to sending an "abandon" message to the user (step 15) and then recording su as **potentially** unreachable (step 16): $SU_k^P = SU_{k-1}^P \cup su$. Then go to step 5 and continue the following process. Notice that in step 5, we have $SU_i^U = SU_i^U - SU_j^A - SU_k^P$.

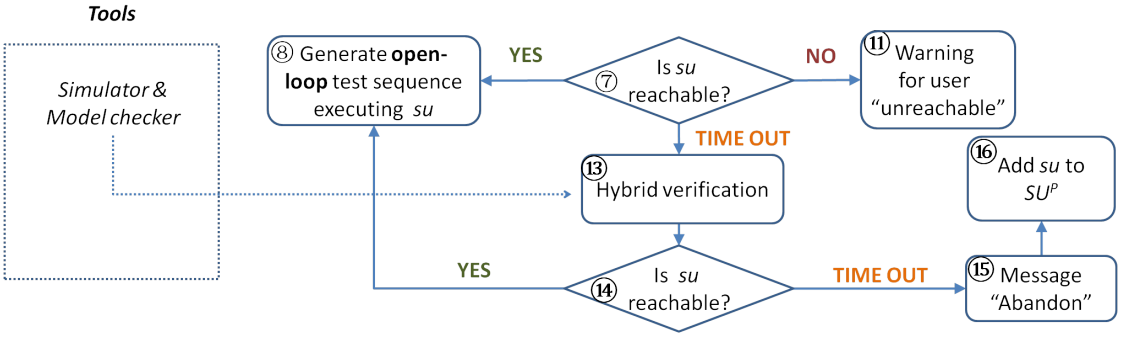


Figure 4.3: Model-based testing methodology: part 2 of 2

This process converges when either the structural coverage criterion is satisfied or there are no more unexplored uncovered structural units i.e. $SU_i^U = \emptyset$. Since $TS_i \supset TS_{i-1}$, that leads to $SU_i^U \subset SU_{i-1}^U$ and $SC_i > SC_{i-1}$ because at least one more structural unit is covered.

It is possible that the process terminates immediately after execution of the initial functional test suite TS_0 if the corresponding SC_0 is already satisfying.

Otherwise, at the end of the process, if the loop at left is executed at least once, we have an improved test coverage; if the loop at right is executed at least once, i.e. $SU^A \cup SU^P \neq \emptyset$, further analysis with the authors of the specification is required since at least one structural unit is suspected to be dead code or even a bug.

4.1.3 A heuristic: hybrid verification

Hybrid verification [158, 157, 55] is a technique combining model checking and simulation. Model checking tries to explore all the possible states while simulation explores partially the entire state space. Fig. 4.4 depicts the principle of hybrid verification. Hybrid verification requires various techniques such as exhaustive state-space exploration, memorizing exploration traces, forward/backward trace generation, step-by-step simulation, etc.

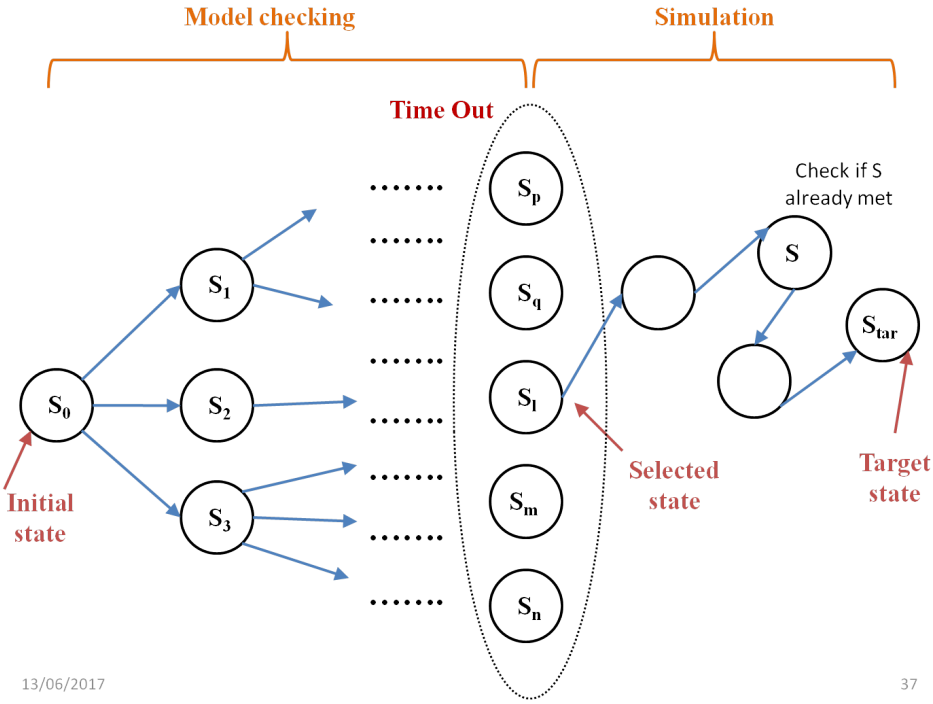


Figure 4.4: Principles of hybrid verification

Supposing that we have a model of the SUT and the problem to verify is: starting from an initial state S_0, can we find a test case that finally arrives at the target state S_{tar}. In the context of our methodology, an uncovered structural unit can be considered as the target state S_{tar}. The model checker performs model checking until it runs out of time or memory (time out). We suppose that the

model checker is able to memorize the already explored state traces just before TO. Say, for example, the model checking at a computation cycle where five states S_p,S_q, S_l, S_m and S_n have been explored. We select from the five states a candidate state, say S_l, as the candidate state to start a step-by-step simulation. The selection of candidate state is based on an informal user-defined "distance" between each potential candidate state and the target state. The definition of such a selection remains the most delicate and difficult part of the hybrid technique. Some heuristics can be easily found, however, we doubt that a formal distance can be established. If that was the case, this could be incorporated to the model checker.

Raffinement par ajout progressif des contraintes

Cette section présente une technique de raffinement de la génération de test en boucle ouverte discutée ci-dessus. Cette technique est illustrée avec le model checker GATeL. GATeL prend le modèle Lustre du système contrôle/commande comme une entrée. Il permet également deux autres entrées: un objectif de test et une description de l'environnement. Notre objectif de test est de vérifier si une unité structurelle donné peut être couvert par n'importe quel test, c'est-à-dire l'accessibilité de cette unité. Au cas où l'unité est accessible, GATeL génère des séquences de test atteignant l'unité au dernier cycle de calcul. La description de l'environnement est composée de booléen expressions destinées à sélectionner parmi toutes les valeurs possibles de variables les correspondants aux réactions réalistes du système. Chaque expression de sélection est indiquée comme une directive assert "qui doit être vraie à chaque cycle de séquences générées. Ces expressions sont utilisées par GATeL pour dériver des contraintes dé nir des entrées / sorties des relations.

4.1.4 An first example: cruise control

The following experiment results are based on a simple example: cruise control system. It is a system that automatically controls the speed of a motor vehicle by taking over the throttle of the car to maintain a steady speed as set by the driver.

The model of the system is built in SCADE Suite, coupling data-flow block diagrams and hierarchical SSMs (Safe State Machines) [34]. MaTeLo serves as the functional test generator and GATeL as the model-checker. SCADE QTE (Qualified Testing Environment) [1] is responsible for executing test cases and measuring the coverage.

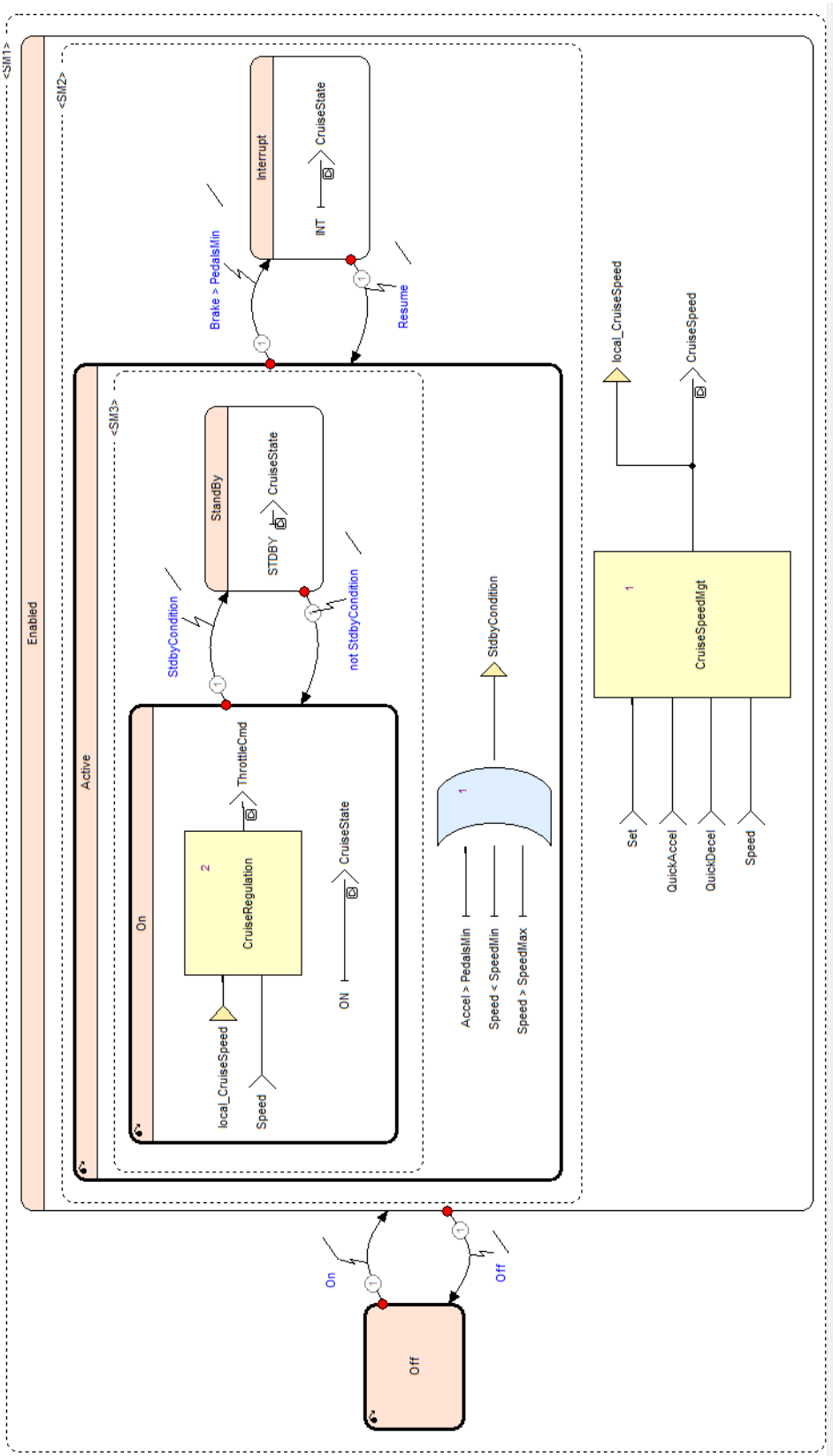


Figure 4.5: Cruise control model in SCADE Suite

The cruise control system presented in Fig. 4.5 consists in three nested state machines, each having two states: the cruise control can be either *Off* or *Enabled*; while *Enabled*, the second automaton *SM2* may be in state *Interrupt* or *Active*; a third automaton *SM3* defines two computation modes when in state *Active*: state *On* or state *StandBy*. The system depends on these inputs: *On* (enable the cruise control), *Off* (disable the cruise control), *Resume* (resume from state *Interrupt*), *Set* (set current car speed as new cruise speed), *QuickDecel* and *QuickAccel* (decrease/increase the cruise speed), *Accel* (accelerator pedal sensor), *Brake* (brake pedal sensor) and *Speed* (car speed sensor). At each cycle, depending on its active state, the system computes three outputs: *CruiseSpeed* (cruise speed value), *ThrottleCmd* (throttle command) and *CruiseState* (state of cruise control, it can be OFF, ON, STDBY, or INT).

The original functional test set has three records. After execution, the model coverage MC/DC is measured on each operator composing the model [80], as shown in figure 4.6. The color green indicates a 100% coverage while yellow a partial coverage. Note the boolean operator "or" highlighted by a red arrow: in figure 4.5 it corresponds to the operator calculating the internal variable *StandBy Condition*. This operator has three inputs:

- input i1 $Accel > PedalsMin$ is true when accelerator pedal is pressed;
- input i2 $Speed < SpeedMin$ is true when current car speed is less then the minimum speed value;
- input i3 $Speed > SpeedMax$ is true when current car speed is greater then the maximum speed value.

This operator describes a functional requirement of cruise control system: when the accelerator pedal is pressed or the car speed is out of range, cruise control should be in state *StandBy* and the output *CruiseState* is valued STDBY.

A full coverage of this operator requires 4 test cases while the original test set only provide 2. We choose "only input i2 is true" as an uncovered SU and apply the model-checker GATeL. As mentioned before, GATeL works on models in Lustre (a synchronous data-flow language). It also provides an extension to handle state machine specifications [38]. The graphical SCADE model of cruise control is first equivalently transformed into a textual Lustre model with help of the "s2d" tool[1]. Then in GATeL we define a test node to describe the test objective as well as environmental conditions to reduce the domain of inputs' values (see Fig. 4.7).

[1]The tool is developed and provided by *Laboratoire Sûreté des Logiciels, CEA/DRT/DTSI/-SOL, 91191 Gif sur Yvette, France*

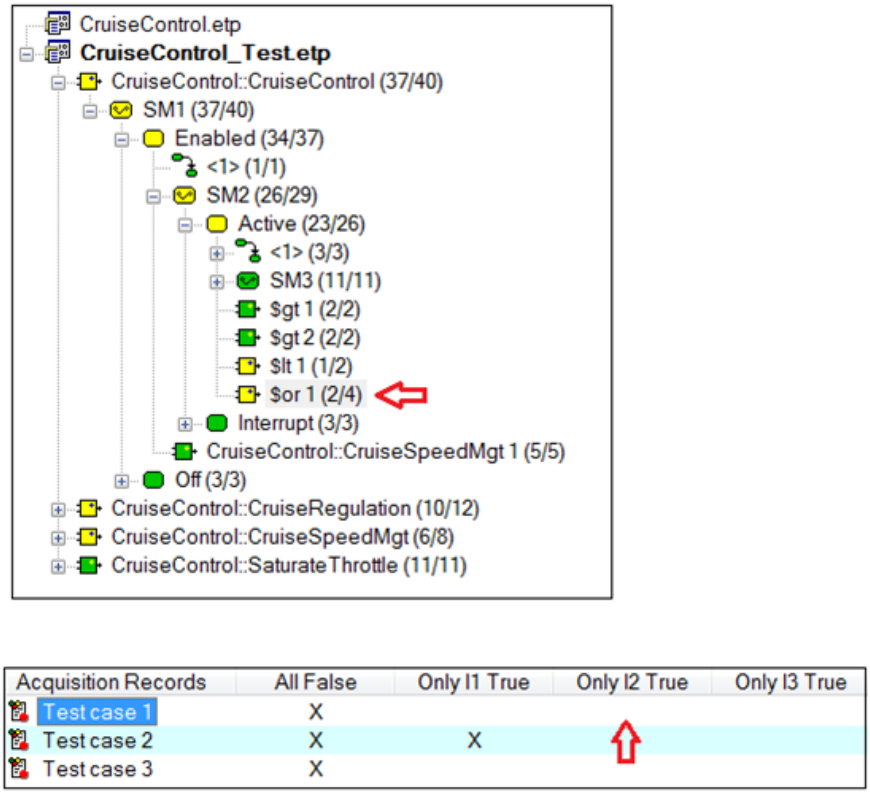


Figure 4.6: Original structural coverage of cruise control model

We ask GATeL to reach a state where the car speed is less then the minimum speed value and the accelerator pedal is not pressed. Under this circumstance the output *CruiseState* should be STDBY. The last line of test objective assures that the test case generated contains at least 2 cycles. As shown in Fig. 4.8, GATeL generates a test case where the objective is reached at the last cycle (cycle 0).

Following our testing methodology defined in Sec. 4.1.2, the original test set is enriched by the new test case generated by GATeL. After execution of new test set, its structural coverage is measured and the results are presented in Fig. 4.9. The coverage rate of operator "or" is enhanced from 2/4 to 3/4. Note that the operator "lt" just above the operator "or" has also changed from partial coverage (yellow) to full coverage (green). It corresponds to the calculation of $Speed < SpeedMin$. As SCADE operators are hierarchical, i.e. one operator can contain other operators, it is advisable to start from the "deepest" operator when choosing an uncovered structural unit.

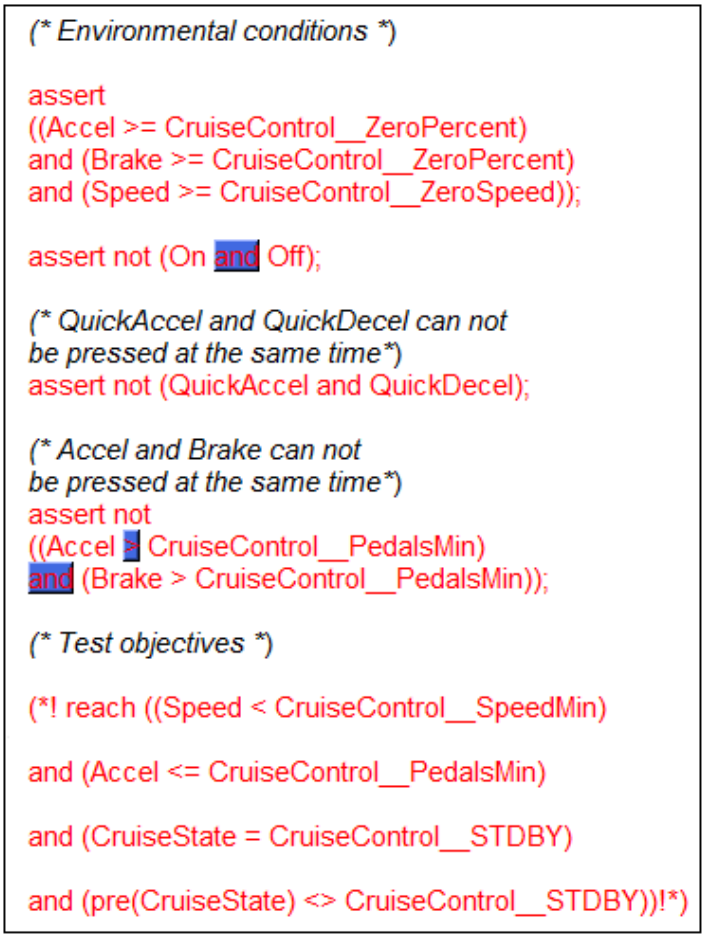


Figure 4.7: GATeL interface: node of test

Suppose an error is introduced in the SCADE model: the operator "or" with three inputs is replaced by a combination of an operator "and" and an operator "or", each having two inputs, as defined in Fig. 4.10. The rest part of model remains unchanged. The original test set will not be able to identify this bug because it doesn't execute the modified branches.

The SCADE model with error is then transformed into an equivalent Lustre model, taken by GATeL as input. We create the same test node as described in previous subsection, but this time GATeL is not able to generate a sequence satisfying our test objective, see Fig. 4.11. According to our methodology, this SU is considered as unreachable. A simple code inspection guided by this result will easily identify the design bug.

Experiment results on cruise control model somewhat validate our testing

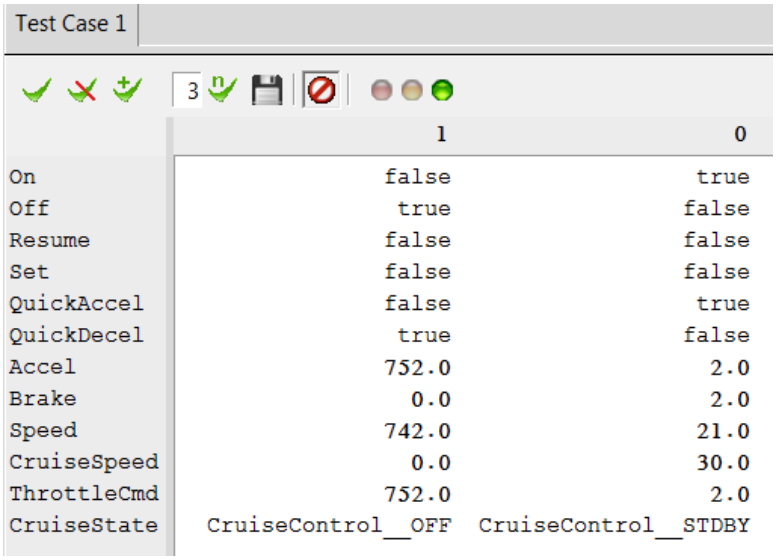


Figure 4.8: GATeL interface: test case generated

methodology as an approach to enhance coverage and detect design bugs using model-checking. The dimension of this model is too limited to challenge the model-checker with a "time out" situation. So part 2 of methodology could not be tried with this example.

Complexity of Scade models

In the domain of embedded systems, projects are increasingly adopting model-based engineering tools, such as SCADE or Simulink, to specify the functional architecture. One important impact of model-based engineering is automatic generation of certified code from models, with the result that the code metrics no longer match engineering efforts. New methods must be developed to evaluate the quality and complexity of these models.

The research project ERACES [2] performed at the Carnegie Mellon Software Engineering Insitute aims at designing methods and tools to measure and reduce complexity in software models. To evaluate complexity of SCADE models, implementation of existing metrics and introduction of new metrics for data-flow language have been proposed in [69]. Tools designed to compute complexity using these notations have been released as plug-ins for SCADE tool set under an open-source license. We computed complexity of cruise control model under *Nesting Level* metrics and *Data Flow* metrics. The former goes through modeling compo-

[2]https://github.com/cmu-sei/eraces

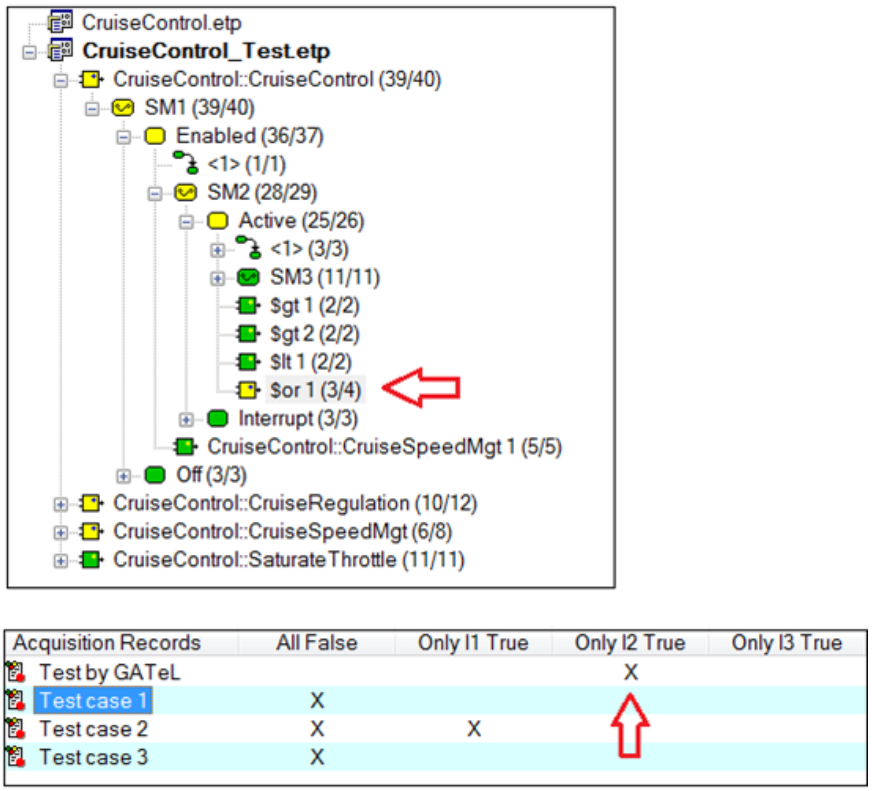


Figure 4.9: New structural coverage of cruise control model

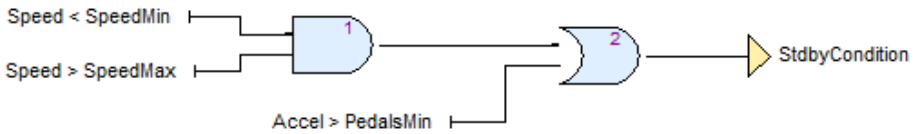


Figure 4.10: Operator modified in cruise control model

nents and counts the depth of non-predefined operators and state machine states; the latter allows users to easily follow how each input data is broadcasted in the model. Computation results are partially presented is Fig. 4.12 and Fig. 4.13.

Model complexity has a significant impact on maintenance costs during system lifecycle. It matters particularly to safety-critical system which are usually main-

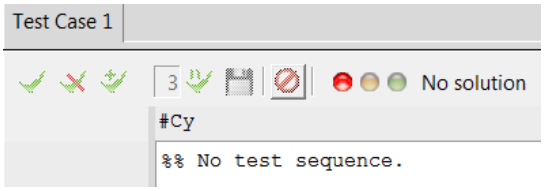


Figure 4.11: GATeL interface: unreachable branch detected

Script Item	Level
Off	1
Enabled	1
CruiseSpeedMgt	2
LimiterUnSymmetrical	3
Active	2
Interrupt	2
On	3
CruiseRegulation	4
SaturateThrottle	5
Gain	5
Gain	5
StandBy	3

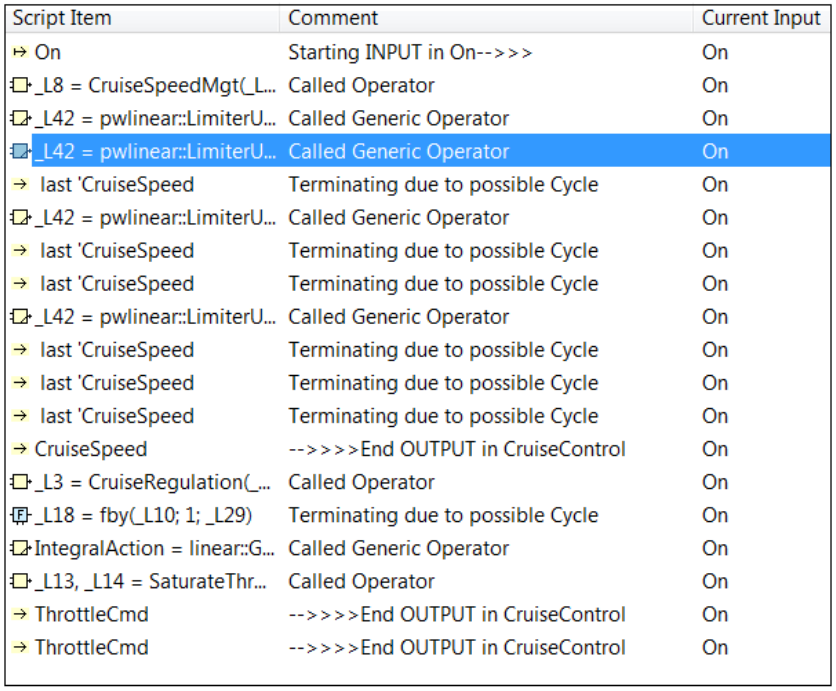
Figure 4.12: Cruise control model complexity: *Nesting Level* metrics

tained for more than 20 years. In this thesis, we settle for calculating complexity with the SCADE extension developed by ERACES project. How to include this aspect into application of our testing methodology will be one subject of future work.

4.2 Refinement by gradually adding constraints in GATeL

This section describes a refinement of the open-loop test generation by model checking presented in the above methodology. The technique is illustrated with the model checker GATeL.

The model checker GATeL takes the Lustre model of the control/command system as one input. It also allows two more user inputs: the test objective and



Script Item	Comment	Current Input
On	Starting INPUT in On-->>>	On
_L8 = CruiseSpeedMgt(_L...	Called Operator	On
_L42 = pwlinear::LimiterU...	Called Generic Operator	On
_L42 = pwlinear::LimiterU...	Called Generic Operator	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
_L42 = pwlinear::LimiterU...	Called Generic Operator	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
_L42 = pwlinear::LimiterU...	Called Generic Operator	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
last 'CruiseSpeed	Terminating due to possible Cycle	On
CruiseSpeed	-->>>>End OUTPUT in CruiseControl	On
L3 = CruiseRegulation(...	Called Operator	On
_L18 = fby(_L10; 1; _L29)	Terminating due to possible Cycle	On
IntegralAction = linear::G...	Called Generic Operator	On
_L13, _L14 = SaturateThr...	Called Operator	On
ThrottleCmd	-->>>>End OUTPUT in CruiseControl	On
ThrottleCmd	-->>>>End OUTPUT in CruiseControl	On

Figure 4.13: Cruise control model complexity: *Data Flow* metrics

an environment description. Our test objective is to verify whether a given SU of the model can be covered by any test, i.e the reachablity of this SU. In case that the SU is reachable, GATeL generates test sequences reaching the desired SU at the last computation cycle. The environment description is composed of boolean expressions intended to select from all possible values of variables those corresponding to realistic reactions of the system. Each selecting expression is stated as an "assert" directive that must be true at each cycle of generated sequences. These expressions are used by GATeL to derive constraints defining inputs/outputs relationships.

The refinement of test generation concerns adding progressively more constraints in the environment description. The constraints are divided into three categories, including physical conditions, initialization conditions and functional requirements. This decides the order to add them into the environment description. After every addition of a new constraint, all the possible results are considered and necessary decisions are taken.

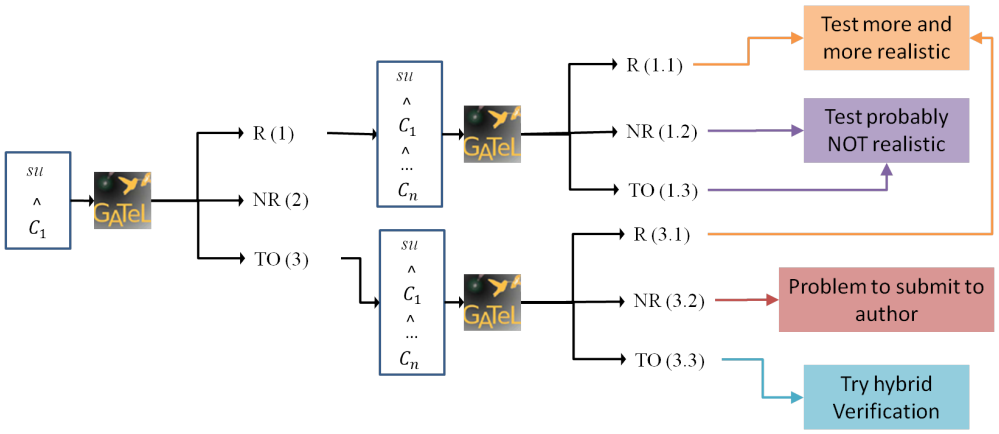


Figure 4.14: Refining test generation by adding constraints to the model checker

4.2.1 Three categories of constraints

We distinguish three categories of constraints:

- **Physical constraints** only concern inputs of the model. Physical constraints filter out the values that could not appear in the actual physical system where the model is applied. Such as incompatible values for two input flows or an input domain restrained by physical law (the water temperature is between 0 °C and 100°C for example).

- **Initialization constraints** only concern inputs of the model. Initialization constraints define input values at the initial cycle. These constraints are not necessarily true after the first cycle. Although an expression stated by "assert" directive must be true at each cycle, the Lustre language proposes temporal operators such as delay (`pre`) and initialization ($\rightarrow$), which make initialization constraints possible.

- **Requirement constraints** concern inputs and outputs of the model. Requirement constraints are derived from the functional requirements of the system, defining relationships between inputs and outputs.

The order to progressively add the constraints is physical constraints, then initialization constraints and finally requirement constraints. Note that not all functional requirements can be translated as invariant constraints. In this thesis we are only handling invariant requirements.

4.2.2 Refinement by adding progressively the constraints

Fig. 4.14 illustrates the principle of refining test generation in GATeL. Given an uncovered structural unit *su*, GATeL verifies if *su* is reachable while respecting the constraints defined in the environment description. At first the environment description contains only one constraint C_1 and the reachability check on *su* can produce three outcomes:

- (1) *su* is found Reachable (R).
- (2) *su* is found Non-Reachable (NR). At this point there is no need to continue adding more constraints. *su* will be recorded as a non-reachable branch.
- (3) GATeL encounters Time-Out (TO).

For the outcome(1) or (3), new constraints are then added progressively to the environment description, one at a time. After each addition of a new constraint, GATeL verifies the reachability of *su*. With more constraints in the environment description, the possible outcomes are:

- (1.1) *su* is still found reachable. In this case GATeL generates test sequences where *su* is covered at the last cycle. The sequences construct a new open-loop test case satisfying the constraints $C_1 \wedge C_2 \wedge ...C_n$. It helps to construct a functional closed-loop test case covering the structural unit *su*.
- (1.2) *su* becomes NR with the newly added constraint, which means the previous result R(1) was not "realistic" This could suggest that *su* is an unreachable structural unit which requires further analysis. This could also indicate a bug somewhere: the constraint may not be correctly formalized or there may be a violation of the requirement described by the constraint in the model.
- (1.3) GATeL has a TO. In general adding constraints means reducing the state space that GATeL needs to explore. One possible explanation for this situation is that the previously generated test sequences in (1) are eliminated by the newly added constraint and GATeL, exactly like in (1.2), can not find another new trace covering *su* in a restricted period of time.
- (3.1) With more constraints added to the environment description, GATeL generates realistic test sequences covering *su*.
- (3.2) We have a stronger hypothesis that *su* is unreachable.
- (3.3) A Time Out. Following Fig. 4.3 hybrid verification is the last resolution to consider.

Chapter 5

"CONNEXION" Case study: SRI

The case study proposed in "CONNEXION" is SRI: an Elementary System present in the I&C system of all the French nuclear power plants. SRI is briefly presented in Sec. 5.1. The part 1 of our methodology has been tested on SRI and the results are discussed in Sec. 5.2. Although the part 2 of the methodology has not been successfully tested, we tried three academic model checkers on SRI and compared them with respect to capacities required for hybrid verification. This part is described in Sec. 5.3.

5.1 Description of SRI

The main function of the SRI is to ensure the refrigeration of several other ESs, referred to as the clients of SRI. The SRI also interfaces with a cooling source SEN through its heat exchangers.

The SRI Process includes two heat exchangers working in parallel, where the cold water from SEN and the hot water from its clients are mixing. The water temperature at the exit of the heat exchangers is regulated by three parallel valves, varying by their opening the flow rate and thus the heat exchange. According to different operational modes, a user can be connected to the SRI or not. A water tank is used to compensate the eventual leakage of the circuit. Three pumps in parallel ensure the water circulation in the system (pump 3 is spare). Fig. 5.1 demonstrates a simplified schema of the Process.

The control/command of SRI functions at several levels:

- regulating the water temperature at the exit of the heat exchangers;
- regulating the level in the water tank;

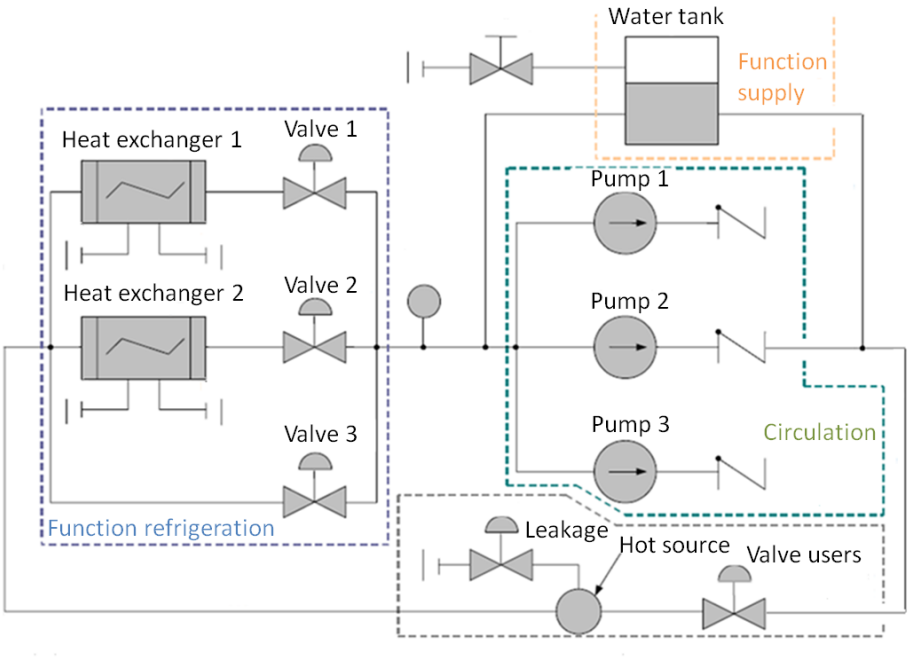


Figure 5.1: A simplified schema of SRI

- automatic start-up of a pump in case of malfunction of the pumps in service.

The control/command sub-system of SRI is modeled by SCADE Suite. A part of the block diagram is shown by Fig. 5.2.

5.2 Experimentation results of part 1

The experimentation results are organized into three phases, same as when we presented the methodology in previous chapter.

Phase 1

The activities performed in phase 1 include functional test generation with the MaTeLo tool and closed-loop test execution with the ALICES platform.

Based on the functional requirements, AREVA, a project partner of"CONNEXION", has created a usage model of the SRI in MaTeLo. They have used MaTeLo to generate a functional test suite containing 10 test cases. Then we execute these 10 test cases by closed-loop simulation on ALICES. We obtain the scripts recording data exchange during simulation between the Process model and the control/command

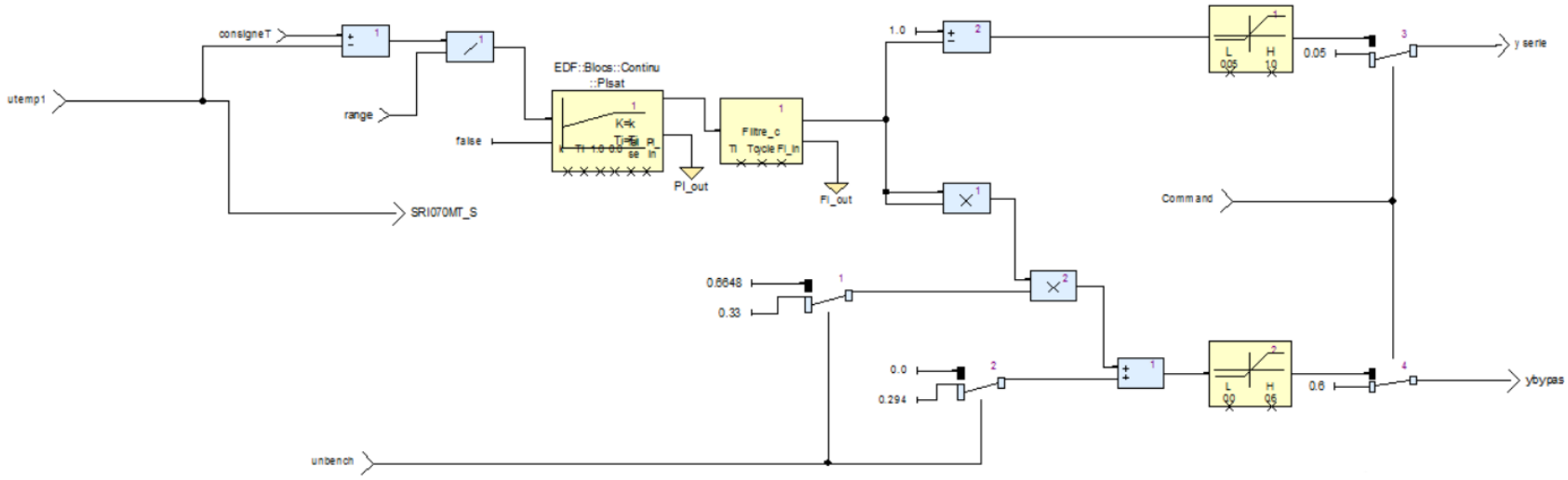


Figure 5.2: A part of the SRI control/command in SCADE Suite

model.

Phase 2

The scripts obtained in phase 1 are now used to measure the MC/DC coverage on each operator composing the control/command model, as shown in Fig. 5.3. More details regarding coverage metrics of Scade models can be found in [80].

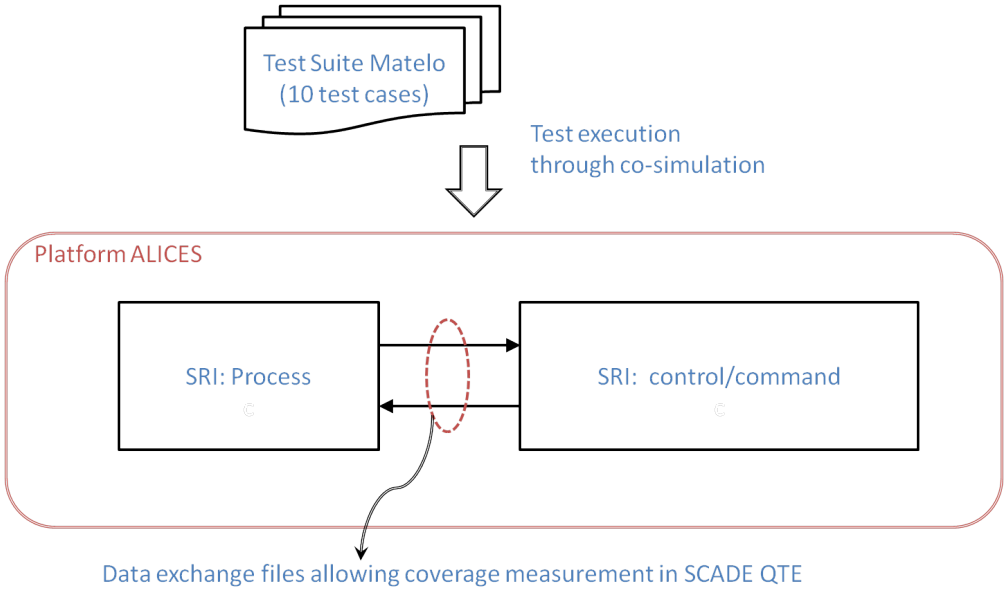


Figure 5.3: Coverage measurement

Fig. 5.4 shows the coverage result measured by SCADE QTE. Green color indicates a 100% coverage, yellow a partial coverage and red a zero coverage. The Scade model of control/command system is structured as an hierarchy of operators. The top level operator (the root operator) is called "ControlCommande". The MC/DC coverage rate measured at this level is about 50%, as highlighted in the figure.

Phase 3

With a 50% coverage rate at the top level, we need to choose an uncovered structural unit to perform reachability check. While choosing an uncovered structural unit, we suggest to start with a "deepest" operator who does not contain other

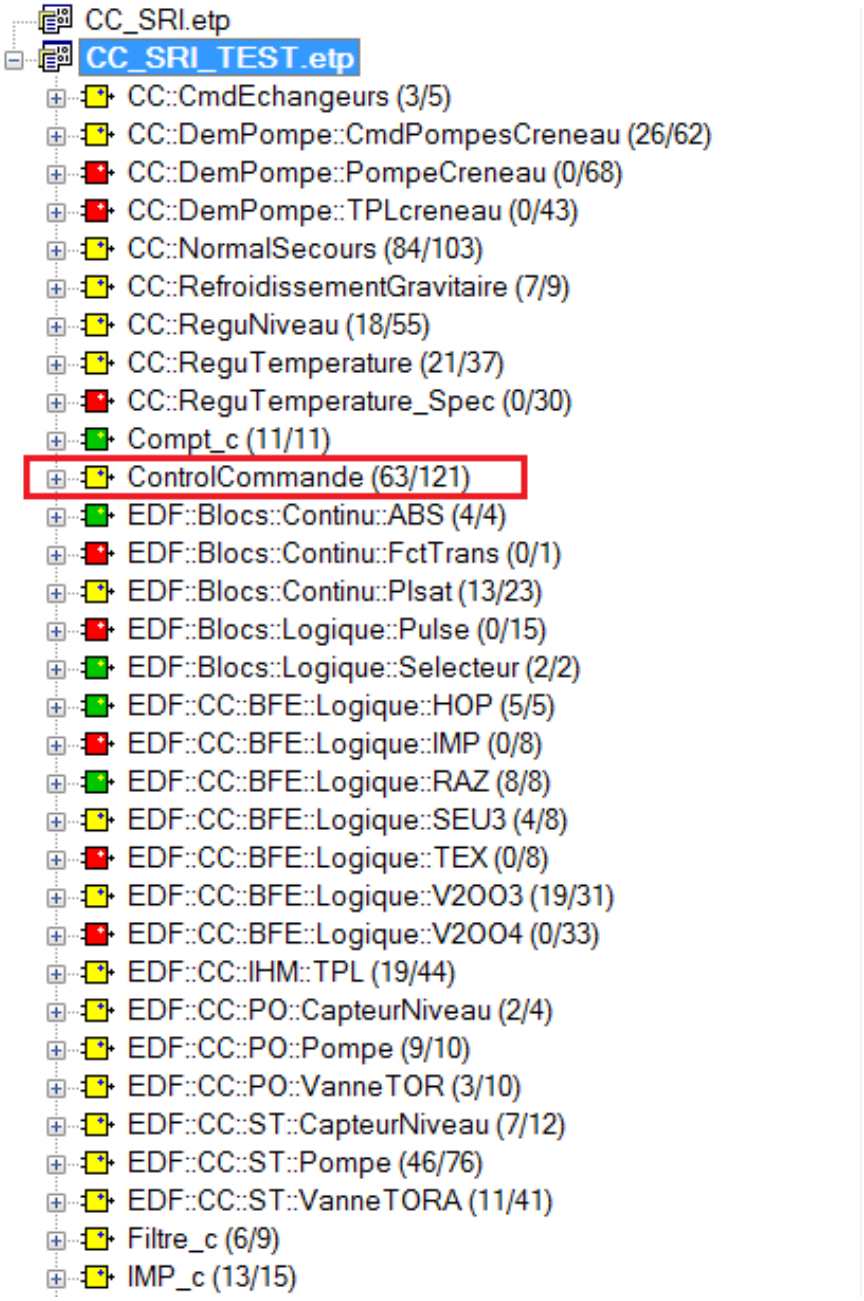


Figure 5.4: MC/DC coverage on each operator of the control/command model

predefined operators. The advantage is that a test executing this "deepest" SU may as well covers other higher-level SU that have not been covered. The uncovered structural unit su that we have chosen is three-level deeper with respect to the top operator: the two boolean inputs $i1$ and $i2$ of an "and" logic have never

been both true (Fig. 5.5).

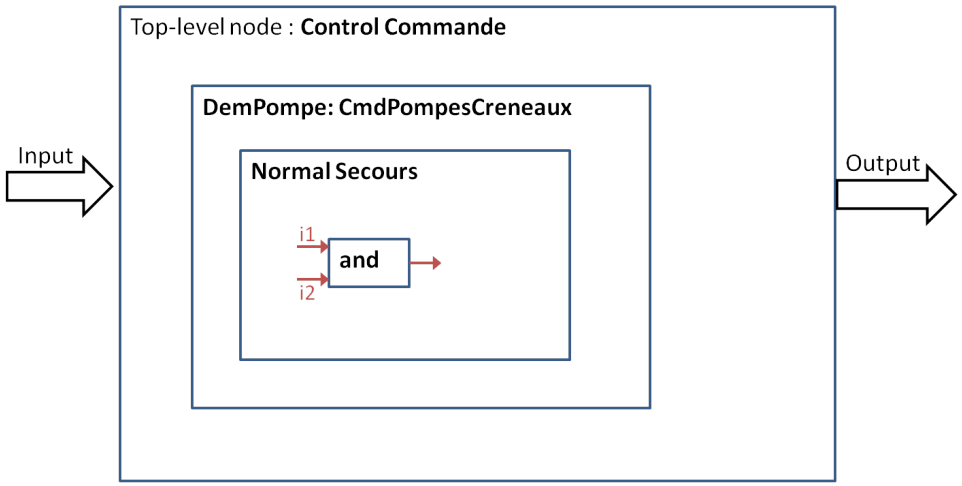


Figure 5.5: The uncovered SU chosen for experiment

We have used GATeL to perform reachability check following the refinement technique: progressively adding assumptions about the behavior of the environment. These assumption are translated into invariant constraints for GATeL. These constraints can be found in appendix A. We first challenged GATeL with this *su* under a physical constraint C_1. This constraint assumes three things:

1. the water temperature is confined between 0°C and 100°C;
2. the water level in the tank is confined between 0 and the maximum tank level;
3. the boolean variables representing pump status and pump default can not be true at the same time (if the status variable is true that means the pump does not have a default).

Under $C1$, *su* has been found reachable and a test sequence containing 3 cycles have been generated. We have measured again the MC/DC coverage, including this newly generated test. Fig. 5.6 and Fig. 5.7 shows the MC/DC coverage at the *su* level and the top node level. $TS0$ is the original functional test suite. $localVar2$ is the new test generated by GATeL to cover the previously uncovered *su* that we have chosen. The new test has not only covered the *su* but also raised the top level coverage rate from 50% to 80%. This means that other uncovered structure units have also been executed by the new test.

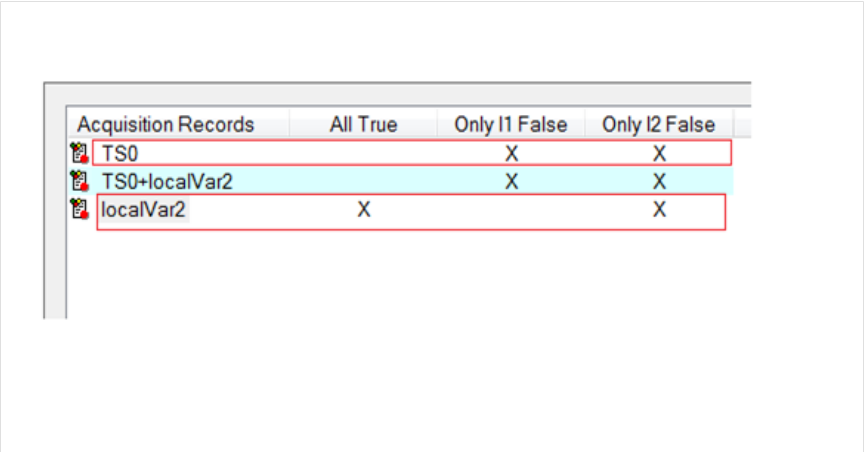


Figure 5.6: MC/DC coverage at the *su* level

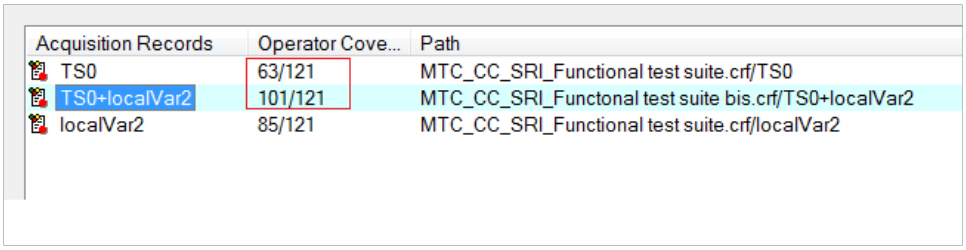


Figure 5.7: MC/DC coverage at the top level: before and after

Then an initialization constraint C_2 has been added to the environment description. $C2$ assumes that in the initial state of the system, both the two heat-exchangers are functioning. We encountered TO at first: the execution was aborted without giving a result because of the configuration in GATeL regarding the maximum number of cycles of test generation. It had been set to 20, which is not long enough under the assumptions $C_1 \wedge C_2$. We have thus increased this configuration to 30 and obtained a test sequence of 23 cycles. The calculation time is 6391 seconds and memory used 272309 kilobytes.

This result indicates that the 3-cycle test sequence obtained at first is not realistic. This proves that the refinement by adding gradually constraints does help enhancing functional reality of the test generation. We admit that based on the test sequence of 23 cycles, preparing a relevant closed-loop test case would not be easy. But the depth of the test also suggests that manually designing a test covering the structural unit *su* can be very difficult.

5.3 Lustre model checkers toward hybrid verification

We have tested three academic model checking tools for the Lustre language, GATeL, Lesar and Kind 2, to implement the complete methodology including hybrid verification in particular. Lesar and Kind 2 are model checkers oriented towards the conventional property verification problem. Lesar takes as input the standard Lustre V4 models while Kind 2 accepts as input modeling language Lustre v4 extended with a part of V6. Both Lesar and Kind 2 are used to verify safety properties. GATeL is a model checking tool oriented towards test sequences generation. It accepts a specific extension of Lustre V4. Able to perform traditional model checking problem, GATeL offers user-definable test objectives. A test objective can be an invariant property such as a safety property, or some particular states to be reached of the system under test. GATeL is able to generate test sequences illustrating how the test objective can be satisfied. GATeL also takes as input a specification of the environment describing possible evolution of the inputs that can be considered during test sequence generation. This environment description allows refining test sequence generation by progressively adding constraints as presented in Sec. 4.2.

The first step of this experimentation is to build a proper input model of the SUT for these three model checkers. We started with the textual Scade model of the SRI control/command (CC) which is automatically generated from the graphical Scade model by SCADE KCG. The s2d tool provided by CEA/List then translates the Scade code of the SRI CC to an extended Lustre V4 model which can be directly input to GATeL. Once the SRI CC model in GATeL, we define the test objective as checking the reachability of the unexecuted SU that has been identified in Fig. 5.5. This is the same procedure previously used in Sec. 5.2.

The uncovered SU chosen for experiment is hierarchically three-level "deeper" with respect to the top-level node. The inputs and output of the `and` operator are considered as internal or local variable for the top-level node. However, for Kind 2 and Lesar, it is necessary to specify a target property concerning only the inputs and outputs of the top-level node. Therefore, we decide to modify the graphical Scade model of SRI CC so as to bring out `i1` and `i2` to top-level outputs `o1` and `o2`, as depicted in Fig. 5.8. Then following the above procedure, we obtain a modified model of SRI CC for GATeL.

The GATeL input model is not directly readable by Kind 2 and Lesar, since each tool works on a slightly different version of the Lustre language. Fig. 5.9

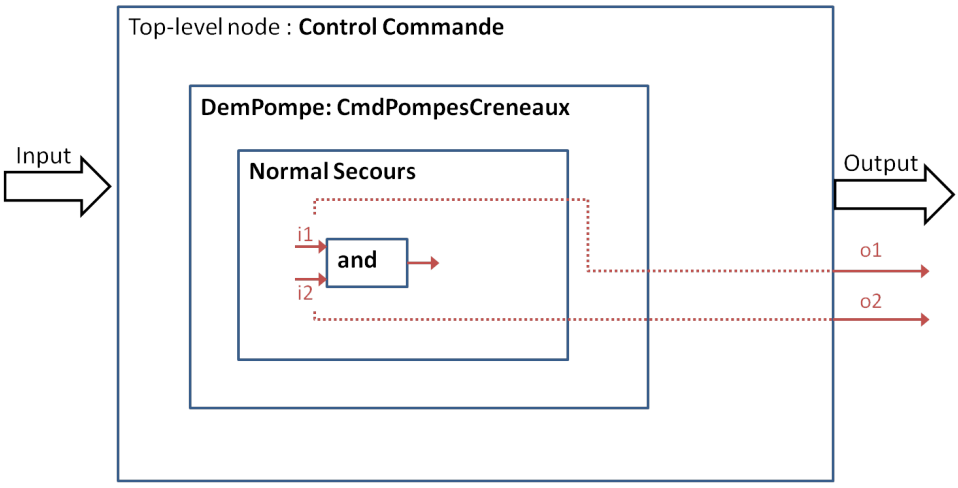


Figure 5.8: The uncovered SU chosen for experiment: modify the top level node

shows how the Scade model has to be transformed to become readable by our three model checkers. Luckily the GATeL model of SRI CC does not contain syntax not included neither in Lustre V4 nor in V6. However it is still subject to some syntax modification including:

- For complex data definition e.g., `type T_NS = {T1: real, T2: real, T3: real};` all the commas `,` need to be replaced by semi-colons `;`.
- GATeL allows writing `assume expression_1` to indicate that `expression_1` is always true. In Lustre, the `assume` needs to be replaced by `assert` with the same semantics.
- Scade and GATeL offers user ID `#1`, `#2`, `#3`, etc to identify different instances of the same operator or node. This definition does not exist in Lustre V4 or V6.
- In Scade and GATeL, the evaluation of a structured variable can be performed through an operator `make` such as `make T = (v1, v2, v3)` where `T` is a `T_NS` type variable. In Lustre this should write as `T.T1=v1; T.T2=v2; T.T3=v3`.
- Scade and GATeL offers an operator `caseof` as a swith statement, allowing the value of a variable or expression to change the control flow of program execution via a multiway branch. In Lustre, `caseof` must be equivalently replaced by several nesting `if then else`.

After these modifications, the GATeL model has been translated to an input model for both Kind 2 and the compiler *lus2lic*. During our experimentation we

have identified a defect of *lus2lic*. For the conversion of a real number to an integer number, *lus2lic* calls for a function `real2int` which is not declared in the .ec file generated. The tools for EC therefore return a segmentation error. The conversion function actually exists in .ec under a different syntax `int`. A Unix filter is good enough to correct the problem. We have informed the Lustre authors of this defect. Finally we obtain an input model for each of the three model checking tools. Fig. 5.9 gives an overview of the relationship between various models.

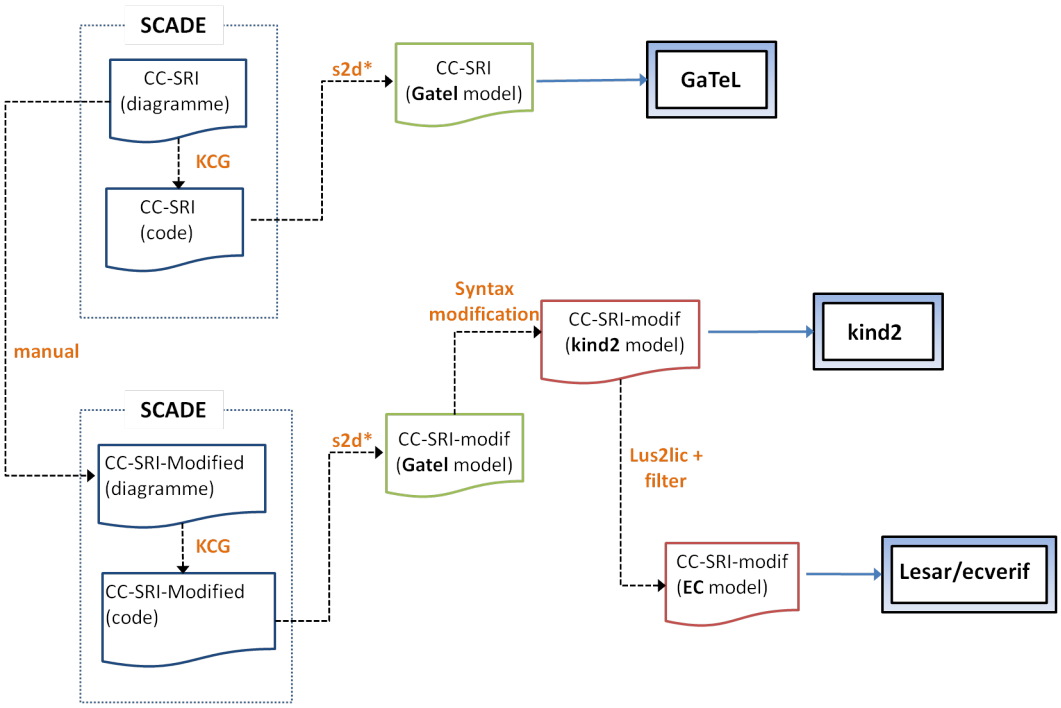


Figure 5.9: Applying different model checkers to the SRI CC

The three model checkers performs quite differently to the same property verification (or test generation) problem.

- Lesar runs out of memory, i.e. time out. In fact Lesar is a model checking tool handling exclusively boolean values. It is therefore clearly not adapted for the model of SRI CC, resulting in state explosion. Lesar does not record any temporary results during state space exploration. As a consequence, it is not possible to perform hybrid verification, which requires memorizing state exploration traces.

- GATeL succeeds in generating test sequences which improve the structural coverage, as discussed in previous section. However GATeL only offers back-

ward test generation so it's not suitable for hybrid verification, which requires forward and backward generation.

- Kind 2 relies on off-the-shelf SMT solvers. The SMT solvers are not adapted for the non-linear expressions of the SRI CC such as multiplication/division of variables. One solution is to replace these non-linear expressions by abstractions defined in assume-grantee style contracts (Sec. 2.4.3). The abstracted model may be suitable for property verification, if it's proved equivalent to the original model, but not amenable for test generation.

We also searched for translators of Lustre towards other modeling language so as to benefit from other powerful model checking tools such as NuSMV [59]. The Lustre translator framework presented in Sec. 2.4.4 is Rockwell Collins proprietary and the project with University of Minnesota is no longer supported. Another academic translator from EC to SMV (Sec. 2.4.4) is no longer maintained. The SCADE Design Verifier (based on the model checker Prover)(Sec. 2.4.1) is a product that we could not experience with since its license was not included in "CONNEXION". As a result we have come to the conclusion that to implement the complete methodology presented in this thesis, an integration of several model checking tools may be necessary. An alternative solution is to develop a new model checker with the following features:

- forward and backward counterexample generation;

- dump unexplored states after TO;

- resume exploration after simulation.

Chapter 6

Conclusion

The verification and validation of safety-critical real-time system (RTS) are subject to stringent standards and certifications. Integration of recent progress in model-bases system engineering (MBSE) into the life cycle of such systems constitutes the background of this thesis. MBSE approaches, such as model-based design (MBD) and model-based testing (MBT), encourage performing verification activities in the early design stages, allowing early detection of defects. This is very cost-effective for safety-critical RTS since the cost of defects found later in the actual system can be extremely high.

Since 2012, the main industrial and academic partners of the French nuclear industry initiated an ambitious R&D program called "CONNEXION". Regrouping a number of projects, "CONNEXION" [71] aims at improving the development process of the Instrumentation & Control (I&C) system of nuclear power plants (NPPs). "CONNEXION" is based on the existing expertises of major operators from the French nuclear industry (EDF, ALSTOM, AREVA and RRCN) and various software tools provided by other partners (CEA, CORYS, ESTEREL Technologies and ALL4TEC). Several objectives regarding functional validation (FV) have been proposed such as an (innovative) triple V development life cycle where two additional FV sub-cycles are introduced on the models developed at early design phases. Another objective is to create a complete verification platform supporting triple V life cycle through integration and co-simulation of various models and tools. The partners of "CONNEXION" provide a rich and unique tool set to automate as much as possible the functional validation.

The ambitious objectives of "CONNEXION" come with various challenges. The control/command system and its Process are modeled in different languages whereas most verification tools can only deal with one modeling language. Test data generated from the control/command model is therefore referred to as open-

loop. However, a suitable test case for functional validation must be closed-loop and related to functional requirements. Secondly the environment of multi-models and multi-tools can quickly become counter-productive, considering the time spent in error prone transformations back and forth from one model to another and the eventual redundant overlay between the capabilities of different tools. Finally we believe that the complete verification platform should be supported by an information system of traceability (IST) to keep records of verification activities. That is to say, models, properties, verification tools, environment configurations, test data, results, as well as history and relationship between these elements. We proposed a preliminary design of the IST using UML diagrams. We also proposed an implementation prototype structured into two "layers": a storage system for keeping files and a Java application for user interface. The two layers communicate through JDBC tools such as MySQL.

We propose a MBT methodology directed by structural coverage and functional requirements. The methodology relies on a repetitive use of model checkers to generate coverage-based open-loop test sequences. The passage from coverage-based open-loop test sequences to requirement-based closed-loop test cases requires expertise from system engineers and is not in the scope of our research. However, we propose a refinement of test generation using model checkers through progressively adding constraints of the environment. These constraints are derived from physical conditions, initialization conditions and functional requirements. They require the generated test sequences to represent functionally realistic behaviors with respect to the complete SUT. Our methodology also considers that a model checker can TO due to state explosion problem and proposes a heuristic referred to as hybrid verification which combines model checking and simulation. The principle is to collect the states explored at the computation cycle just before TO, select a candidate state which should be the "closest" state to the target state, and begin a step-by-step simulation from this candidate state trying to reach the target state.

The thesis is part of "CONNEXION" and therefore benefits from an actual case study of the I&C system. Our methodology (except hybrid verification) has been tested with support of "CONNEXION" tools on the case study SRI. GATeL is used as the model checker for test generation. The open-loop test sequence generated by GATeL improve the structural coverage of the functional test suite. Moreover, the refinement technique through progressively adding constraints also help to produce more realistic functional test case with respect to the target SUT.

The synchronous data-flow language Lustre has been chosen in "CONNEXION" to model the control/command system. To implement the complete method-

ology, in particular hybrid verification, we review the history of Lustre including various academic versions and related model checking tools. Lustre has been successfully commercialized and resulted in SCADE tool set which includes SCADE DV based on the engine of Prover Plug-in. However SCADE DV is not included in "CONNEXION". As consequence, we have tested three academic model checking tools Lesar, GATeL and Kind 2. Each tool takes as input modeling language a different extension of Lustre. These tools are not able to fulfill 100% of the methodology requirements. We also review the translators of Lustre towards other modeling languages, so as to benefit from other model checking tools such as NuSMV.

The methodology proposed in this thesis is very costly and therefore adapted exclusively for highly safety-critical systems. The verification and validation of this kind of systems are subject to stringent and strongest certifications and standards. Such systems require a very high level of coverage beyond fulfilling the safety requirements. Our methodology offers a solution to automate as much as possible coverage-based test generation, which is a pure manual process in the current engineering approach. Secondly, the methodology includes a refinement of open-loop test generation. The passage from open-loop to closed-loop requires collaboration from system experts. Our refinement can help system experts to more efficiently intervene in preparing closed-loop functional test cases. Our methodology is therefore cost-effective since it saves the time of system experts. Finally we have reviewed various academic model checking tools for the synchronous dataflow language Lustre to implement the methodology. We come to the conclusion that an integration of several tools may be necessary. In our opinion it is also advantageous to revisit the academic Lustre versions and the related model checking tools. However, this is beyond the work of this thesis.

Future work concerns (1) define and possibly develop a new tool to support sufficiently hybrid verification; (2) propositions of selection strategy used in hybrid verification, i.e. how to define the distance between two states in the state space of a transition system; (3) generalized framework of the open-loop test generation refinement; (4) explore the test generation by Kind 2 using assume-garantee style contracts with respect to implementing hybrid verification.

Appendices

Appendix A

Constraints coded in GATeL to refine test generation

```
%physical constraints for certain input values of the
   control/command;

node phyConstr(SRI001BA_Z, SRI071MT_S, SRI072MT_S,
   SRI073MT_S: real; SRI010PO_D, SRI010PO_E, SRI020PO_D,
   SRI020PO_E,
SRI030PO_D, SRI030PO_E: bool)
returns (result:bool);
var v1, v2, v3: bool;
let

%the level in the tank is confined between 0 m and the
   maximum tank level;

v1= (SRI001BA_Z > 0.0) and (SRI001BA_Z < 5.0);

%the water temperature is confined between 0 and 100
   degrees;

v2 = (SRI071MT_S > 273.15 and SRI071MT_S < 373.15 ) and (
   SRI072MT_S > 273.15 and SRI072MT_S < 373.15 ) and
(SRI073MT_S > 273.15 and SRI073MT_S < 373.15 );

%pump status and the pump default can not be true at the
   same time;
```

```
v3 = not (SRI010PO_D and SRI010PO_E) and not(SRI020PO_D and
    SRI020PO_E) and not (SRI030PO_D and SRI030PO_E);
result = v1 and v2 and v3;
tel;

%initialization conditions

% All the three pumps are automatically controlled (no
   human intervention);
assert (not SRI010PO_AM -> true);
assert (not SRI020PO_AM -> true);
assert (not SRI030PO_AM -> true);

%Neither of the pump presents a default;
assert (not SRI010PO_D -> true );
assert (not SRI020PO_D -> true );
assert (not SRI030PO_D -> true );

%The two heat-exchangers are both working initially;
assert (SRI050VN_C -> true);
assert (SRI060VN_C -> true);

%functional requirements concerning the control/command
   system than can be translated as invariant constraints;

%SRI water temperature is between 15 and 38 celsius degree
   (in the following code kelvin temperature is used) when
   SRI is functioning normally;

node req8 (PumpCom1, PumpCom2, PumpCom3: bool; sg10,
sg20: bool; clients: T_Clients; temp: real)
returns (result: bool);
var condition, action: bool;
let
condition = functioning (PumpCom1, PumpCom2, PumpCom3,
sg10, sg20, clients);
action = (temp > 288.15) and (temp < 311.15);
result = true -> (if condition then action else pre(result)
   );
```

```
tel;

%when the water temperature is less then 16 celsius degree
   an alarm is triggered when SRI is functioning (not
   necessarily normally);

node req17 (PumpCom1, PumpCom2, PumpCom3: bool; sg10, sg20:
    bool; clients: T_Clients; temp: real; Alarmes :
   T_Alarmes)
returns (result: bool);
var condition, action: bool;
let
condition = functioning (PumpCom1, PumpCom2, PumpCom3, sg10
   , sg20, clients) ;
action =  Alarmes.SRI901KA and temp < 289.15;
result = true -> (if condition then action else pre(result)
   );
tel;

%when the water temperature is greater than 30 celsius
   degree an alarm is triggered when SRI is functioning (
   not necessarily normally);

node req18 (PumpCom1, PumpCom2, PumpCom3: bool; sg10, sg20:
    bool; clients: T_Clients; temp: real; Alarmes :
   T_Alarmes)
returns (result: bool);
var condition, action: bool;
let
condition = functioning (PumpCom1, PumpCom2, PumpCom3, sg10
   , sg20, clients) ;
action =  Alarmes.SRI071KA and temp > 303.15;
result = true -> (if condition then action else pre(result)
   );
tel;

%When SRI is in the low-temperature start-up after total
   stop status, and the temperature is less than 7 celsius
   degree, the 2 exchangers should not be working.
```

```
node req22 (PumpCom1, PumpCom2, PumpCom3: bool; sg10, sg20:
    bool; clients: T_Clients; temp: real; SRI002VN_C: real)
returns (result: bool);
var condition, action: bool;
let
condition = startUpLT(PumpCom1, PumpCom2, PumpCom3, sg10,
   sg20, clients) ;
action = SRI002VN_C = 1.0 and temp < 280.15;
result = true -> (if condition then action else pre(result)
   );
tel;

%When SRI is in the high-temperature start-up after total
   stop status, and the temperature should be between 7 and
    15 celsius degrees;

node req23 (PumpCom1, PumpCom2, PumpCom3: bool; sg10, sg20:
    bool; clients: T_Clients; temp: real; SRI002VN_C: real)
returns (result: bool);
var condition, action: bool;
let
condition = startUpHT(PumpCom1, PumpCom2, PumpCom3, sg10,
   sg20, clients) ;
action = SRI002VN_C = 1.0 and temp > 280.15 and temp
   <290.15;
result = true -> (if condition then action else pre(result)
   );
tel;

%When SRI is in degrades stop, the draining valve should be
    open;

node req24 (pump1, pump2, pump3, sg10, sg20: bool; clients:
    T_Clients; SRI005VN_C: bool)
returns (result: bool);
var condition, action: bool;
let
condition = degradedStop (pump1, pump2, pump3, sg10, sg20,
   clients);
action = SRI005VN_C;
```

```
result = true -> (if condition then action else pre(result)
   );
tel;

%When SRI is functioning normally, opening of the bypass
   valve is limited to 60 percent;

node req34 (pump1, pump2, pump3, sg10, sg20: bool; clients:
    T_Clients; SRI002VN_C: real)
returns (result: bool);
var condition, action: bool;
let
condition = normFunct(pump1, pump2, pump3, sg10, sg20,
   clients);
action = SRI002VN_C <= 0.6;
result = true -> (if condition then action else pre(result)
   );
tel;

%When SRI is functioning, the valve SRI002VN_C is always
   closed if the warter temperature is greater than 17
   celsius degree;
node req47 (pump1, pump2, pump3, sg10, sg20: bool; clients:
    T_Clients; temp, SRI002VN_C: real)
returns (result: bool);
var condition, action: bool;
let
condition = functioning(pump1, pump2, pump3, sg10, sg20,
   clients);
action = temp > 290.12 and SRI002VN_C < 0.05;
result = true -> (if condition then action else pre(result)
   );
tel;

%SRI is in the status degraded stop meaning no pump nor
   heat-exchanger is working and only one client (an
   elementary system called APP) is connected to SRI;

node degradedStop (pump1, pump2, pump3, sg10, sg20: bool;
   clients: T_Clients)
```

```
returns (status: bool);
var numP, numE: int; numU: bool;
let
numP = PumpCounter(pump1, pump2, pump3);
numE = ExchCounter (sg10, sg20);
numU = only_APP(clients);
status = numP = 0 and numE = 0 and numU;
tel;

%SRI is in the status low-temperature start-up after total
   stop meaning no heat-exchanger working, one or two pumps
    are working and no client connected;

node startUpLT (PumpCom1, PumpCom2, PumpCom3: bool; sg10,
   sg20: bool; clients: T_Clients)
returns (status: bool);
var numP, numE: int; numU: bool;
let
numP = PumpCounter (PumpCom1, PumpCom2, PumpCom3);
numE = ExchCounter (sg10, sg20);
numU = no_client (clients);
status = (numP = 1 or numP=2) and (numE = 0) and numU;
tel;

%SRI is in the status high temperature start-up after total
    stop, meaning one or pumps working, no heat-exchanger
   working and no client connected or only SAP connected.

node startUpHT (PumpCom1, PumpCom2, PumpCom3: bool; sg10,
   sg20: bool; clients: T_Clients)
returns (status: bool);
var numP, numE: int; numU1, numU2: bool;
let
numP = PumpCounter (PumpCom1, PumpCom2, PumpCom3);
numE = ExchCounter (sg10, sg20);
numU1 = no_client (clients);
numU2 = only_SAP (clients);
status = (numP = 1 or numP = 2) and (numE = 0) and (numU1
   or numU2);
tel;
```

```
%SRI is functioning meaning (two pumps and two heat-
   exchagers working or one pump and two heat-exchangers
   working or two pumps and one exchanger working) and at
   least one client is connected;

node functioning (PumpCom1, PumpCom2, PumpCom3: bool; sg10,
    sg20: bool; clients: T_Clients)
returns (status: bool);
var numP, numE: int; numU: bool;
let
numP = PumpCounter (PumpCom1, PumpCom2, PumpCom3);
numE = ExchCounter (sg10, sg20);
numU = at_least_one (clients);
status = ((numP=2 and numE=2) or (numP=1 and numE=2) or (
   numP=2 and numE=1)) and numU;
tel;

%Verify if the SRI is functioning normally meaning 2 pumps
   and 2 heat-exchangers working and at least one client
   connected;

node normFunct (PumpCom1, PumpCom2, PumpCom3: bool; sg10,
   sg20: bool; clients: T_Clients)
returns (status: bool);
var numP, numE: int; numU: bool;
let
numP = PumpCounter (PumpCom1, PumpCom2, PumpCom3);
numE = ExchCounter (sg10, sg20);
numU = at_least_one (clients);
status = (numP = 2) and (numE = 2) and numU;
tel;

%count the number of pumps that are functioning;

node PumpCounter (PumpCom1, PumpCom2, PumpCom3: bool)
returns (PumpNumber: int);
var p1, p2, p3: int;
let
p1= if PumpCom1 then 1 else 0;
```

```
p2= if PumpCom2 then 1 else 0;
p3= if PumpCom3 then 1 else 0;
PumpNumber = p1+p2+p3;
tel;

%count the number of exchangers that are functioning;
node ExchCounter (sg10, sg20: bool)
returns (ExchNumber: int);
var ex1, ex2: int;
let
ex1 = if sg10 then 1 else 0;
ex2 = if sg20 then 1 else 0;
ExchNumber = ex1+ex2;
tel;

%At least one clinet is connected to SRI;

node at_least_one (clients: T_Clients)
returns (result: bool);
let
result = ((clients.APP=true) or (clients.CEX=true) or (
   clients.GFR=true) or (clients.GHE=true) or (clients.AGR=
   true) or (clients.GRHe=true) or (clients.GSY=true) or (
   clients.SAP=true) or (clients.CET=true) or (clients.GGR=
   true) or (clients.GRHh=true) or (clients.GST=true));
tel;

%only APP is connected to SRI;

node only_APP(clients: T_Clients)
returns (result: bool);
let
result=((clients.APP=true) and (clients.CEX=false) and (
   clients.GFR=false) and (clients.GHE=false) and (clients.
   AGR=false) and (clients.GRHe=false) and (clients.GSY=
   false) and (clients.SAP=false) and (clients.CET=false)
   and (clients.GGR=false) and (clients.GRHh=false) and (
   clients.GST=false));
tel;
```

```
%only SAP is connected to SRI;

node only_SAP(clients: T_Clients)
returns (result: bool);
let
result=((clients.APP=false) and (clients.CEX=false) and (
   clients.GFR=false) and (clients.GHE=false) and (clients.
   AGR=false) and (clients.GRHe=false) and (clients.GSY=
   false) and (clients.SAP=true) and (clients.CET=false)
   and (clients.GGR=false) and (clients.GRHh=false) and (
   clients.GST=false));
tel;

%no client is connected to SRI;

node no_client(clients: T_Clients)
returns (result: bool);
let
result=((clients.APP=false) and (clients.CEX=false) and (
   clients.GFR=false) and (clients.GHE=false) and (clients.
   AGR=false) and (clients.GRHe=false) and (clients.GSY=
   false) and (clients.SAP=false) and (clients.CET=false)
   and (clients.GGR=false) and (clients.GRHh=false) and (
   clients.GST=false));
tel;

%calculate the absolute value of the input;

node abs(input: real)
returns (output: real);
let
output = if input >= 0.0 then input else -input;
tel;
```

Bibliography

[1] *SCADE Qualified Testing Environment R15 Technical Data Sheet.*

[2] Ieee standard glossary of software engineering terminology. *IEEE Std 610*, 1990.

[3] A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, 08(04):275–299, 2001.

[4] *IEC61804-1: Function blocks (FB) for process control - Part 1: Overview of system aspects*, 2.0 edition, 2003.

[5] *IEC60880: Nuclear power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions*, 2006.

[6] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[7] *INCOSE Systems Engineering Vision 2020.* INCOSE, 2007.

[8] Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, Jan 2009.

[9] *IEC61513: Nuclear power plants - Instrumentation and control important to safety - General requirements for systems*, 2011.

[10] *A Practical Guide to SysML: The Systems Modeling Language.* Morgan Kaufmann/OMG Press, 2011.

[11] *IEC61131-3: Programmable controllers - Part 3: Programming languages*, 3.0 edition, 2013.

[12] *Kind 2 User Documentation, Version v1.0.1*, 2016.

[13] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using scade. In *Proceedings of the First International Conference on Leveraging Applications of Formal Methods*, ISoLA'04, pages 115–129, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the Fifth Annual IEEE Symposium on logic in Computer Science (LICS 90)*, pages 414–425. IEEE Computer Society, 1990.

[15] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, ICFEM '98, pages 46–, Washington, DC, USA, 1998. IEEE Computer Society.

[16] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, IEEE-SMC*, Lille, France, 1996.

[17] C. André. Computing syncharts reactions. *Electron. Notes Theor. Comput. Sci.*, 88:3–19, Oct. 2004.

[18] J. Backes, D. Cofer, S. Miller, e. K. Whalen, Michael W.", G. Holzmann, and R. Joshi. *NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, chapter Requirements Analysis of a Quad-Redundant Flight Control System, pages 82–96. Springer International Publishing, 2015.

[19] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[20] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[21] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, page 825–885. IOS Press, 2009.

[22] H. Basold, H. Günther, M. Huhn, and S. Milius. *An Open Alternative for SMT-Based Verification of Scade Models*, pages 124–139. Springer International Publishing, Cham, 2014.

[23] M. v. d. Beeck. A comparison of statecharts variants. In *Proceedings of the Third International Symposium Organized Jointly with the Working Group*

Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, ProCoS, pages 128–148, London, UK, UK, 1994. Springer-Verlag.

[24] B. Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[25] A. Benveniste and G. Berry. Prolog to the special section on another look at real-time programming. *Proceedings of the IEEE*, 79(9):1268–1269, Sep 1991.

[26] A. Benveniste and G. Berry. Real-time systems design and programming. *Proceedings of the IEEE*, 79(9):1270–1282, Sep 1991.

[27] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan 2003.

[28] A. Benveniste and P. L. Guernic. Hybrid dynamical systems theory and the signal language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[29] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: The signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, Sept. 1991.

[30] D. Bergström and R. Göransson. Model- and hardware-in-the-loop testing in a model-based design workflow, 2016. Student Paper.

[31] G. Berry. Real time programming : special purpose or general purpose languages. Research Report RR-1065, INRIA, 1989.

[32] G. Berry. Proof, language, and interaction. chapter The Foundations of Esterel, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[33] G. Berry. *SCADE: Synchronous Design and Validation of Embedded Control Software*, pages 19–33. Springer Netherlands, Dordrecht, 2007.

[34] G. Berry. SCADE: Synchronous Design and Validation of Embedded Control Software. In *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33. Springer, 2007.

[35] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: An introduction to esterel. In *Proceedings of the First*

Franco-Japanese Symposium on Programming of Future Generation Computers, pages 35–56, Amsterdam, The Netherlands, The Netherlands, 1988. Elsevier Science Publishers B. V.

[36] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, Nov. 1992.

[37] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[38] B. Blanc, C. Junke, B. Marre, P. LeGall, and O. Andrieu. Handling state-machines specifications with gatel. In *Electronic Notes in Theorectical Computer Science*, pages 3–17, 2010.

[39] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J. v. Peetz, S. Wolf, A. S. Gmbh, Q. Berlin, F. Scai, and S. Augustin. The functional mockup interface for tool independent exchange of simulation models. In *In Proceedings of the 8th International Modelica Conference*, 2011.

[40] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[41] B. W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, Jan. 1991.

[42] A. Bonacchi, A. Fantechi, S. Bacherin, and M. Tempestini. Validation process for railway interlocking systems. *Science of Computer Programming*, 128:2–21, 2016.

[43] F. Boniol, V. Wiels, and E. Ledinot. Experiences in using model checking to verify real time properties of a landing gear control system. In *ERTS 2006: 3rd European Congress Embedded Real Time Software*, pages 25–27, 2006.

[44] L. d. Bousquet and N. Zuanon. An overview of lutess: A specification-based tool for testing synchronous software. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ASE '99, pages 208–, Washington, DC, USA, 1999. IEEE Computer Society.

[45] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In *Computer Aided Verification 10th International Conference, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–549, Vancouver, BC, Canada, June 1998. Springer.

[46] A. R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[47] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.

[48] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, Sept. 1992.

[49] T. A. Budd and A. S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, Jan. 1985.

[50] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142 – 170, 1992.

[51] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, August 1996.

[52] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[53] P. Caspi and M. Pouzet. A functional extension to lustre. In *Eighth International Symp. on Languages for Intensional Programming, ISLIP' 95*, Sidney, 1995.

[54] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11:1 – 21, 1998.

[55] E. Cerny, A. Dsouza, K. Harer, P.-H. Ho, and T. Ma. Supporting sequential assumptions in hybrid verification. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 1035–1038, New York, NY, USA, 2005. ACM.

[56] A. Champion, A. Gurfinkel, T. Kahsai, and C. Tinelli. *CoCoSpec: A Mode-Aware Contract Language for Reactive Systems*, pages 347–366. Springer International Publishing, 2016.

[57] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *Computer Aided Verification (CAV) - 28th International Conference*, pages 510–517, Toronto, ON, Canada, July 2016. Part II.

[58] F. Chastrette, F. Vallee, and L. Coyette. Application of model-based testing to validation of new nuclear I&C architecture. ICCSEA 2013.

[59] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, pages 495–499, London, UK, UK, 1999. Springer-Verlag.

[60] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd Conference on Design Automation (DAC)*, pages 427–432. ACM Press: New York, 1995.

[61] E. M. Clarke and E. A. Emerson. The Design and Synthesis of Synchronization Skeletons Using Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131, pages 52–71. Springer-Verlag Lecture Notes in Computer Science, May 1981.

[62] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer: Berlin, 1982. ISBN 3-540-11212-X.

[63] E. M. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *POPL'83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 117–126. ACM Press, 1983.

[64] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[65] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 173–182, New York, NY, USA, 2005. ACM.

[66] O. Coudert, C. Berthet, and J. C. Madre. Verification of snychronous sequential machines based on symbolic execution. In *Proc. Int. Workshop Automatic Verification Methods for Finite State Systems (CAV'89)*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1990.

[67] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] L. de Moura, S. Owre, H. Rueß, Rushby, N. Shankar, M. Sorea, and A. Tiwari. *SAL 2*, pages 496–500. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[69] J. Delange. Managing complexity in software models. SCADE user group conference, 2015.

[70] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[71] C. Devic and P. Morilhat. CONNEXION Contrôle Commande Nucléaire Numérique pour l'Export et la rénovatION - coupler génie logiciel et ingénierie système: source d'innovations. *Génie Logiciel*, 104:2–11, mars 2013.

[72] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: A specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 267–276, New York, NY, USA, 1999. ACM.

[73] B. Dutertre and L. de Moura. The yices smt solver. Technical report, Computer Science Laboratory, SRI International, 2006.

[74] E. A. Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, pages 27–45. Springer-Verlag, 2008.

[75] E. A. Emerson and J. Y. Halpern. Desicion procedures and expressiveness in the temporal logic of branching time. In *STOC'82: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 169–180. ACM Press, 1982.

[76] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[77] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, 1997.

[78] E. P. Enoiu, A. Causevic, T. J. Ostrand, E. J. Weyuker, D. Sundmark, and P. Pettersson. Automated test generation using model-checking: an industrial evaluation. In *ICTSS 2013*.

[79] J. A. Estefan. Survey of Model-Based Systems Engineering (MBSE) Methodologies. Technical Report INCOSE-TD-2007-003-02, INCOSE, 2008.

[80] Esterel Technologies. *SCADE Suite user manual*.

[81] A. Fantechi, S. Gnesi, and A. Maggiore. Enhancing Test Coverage by Backtracking Model-checker Counterexamples. In *Electronic Notes in Theoretical Computer Science*, volume 116, pages 199–211, 2004.

[82] A. Fisher, C. Jacobson, E. Lee, R. Murray, A. Sangiovanni-Vincentelli, and E. Scholte. Industrial cyber-physical systems - iCyPhy. In *Proceedings of the Fourth International Conference on Complex Systems Design & Management*, pages 21–37, 2013.

[83] L. Fix. Fifteen years of formal property verification in intel. *25 Years of Model Checking*, pages 139–144, 2008.

[84] K. Forsberg and H. Mooz. The relationship of system engineering to the project cycle. *INCOSE International Symposium*, 1(1):57–65, 1991.

[85] K. Forsberg and H. Mooz. 4.4.4 application of the 'vee' to incremental and evolutionary development. *INCOSE International Symposium*, 5(1):848–855, 1995.

[86] G. Fraser. *Automated Software Testing with Model Checkers*. PhD thesis, Graz University of Technology, October 2007.

[87] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: A survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, Sept. 2009.

[88] P. Fritzson and P. Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings of the 35th Annual Simulation Symposium*, SS '02, pages 365–, Washington, DC, USA, 2002. IEEE Computer Society.

[89] A. Gacek, A. Katis, M. W. Whalen, J. Backes, and D. Cofer. *Towards Realizability Checking of Contracts Using Theories*, pages 173–187. Springer International Publishing, Cham, 2015.

[90] J. Gallois, J. Pierron, and N. Rapin. Validation test production assistance. ICCSEA 2013.

[91] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, Oct. 1999.

[92] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In *Lecture Notes in Computer Science*, volume 1166, pages 143–158, 1996.

[93] G. Hagen and C. Tinelli. Scaling up the formal verification of lustre programs with smt-based techniques. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, FMCAD '08, pages 15:1–15:9, Piscataway, NJ, USA, 2008. IEEE Press.

[94] G. E. Hagen. *VERIFYING SAFETY PROPERTIES OF LUSTRE PROGRAMS: AN SMT-BASED APPROACH*. PhD thesis, University of Iowa, 2008.

[95] A. A. Haider and A. Nadeem. A survey of safety analysis techniques for safety critical systems. *International Journal of Future Computer and Communication*, 2(2):134, 2013.

[96] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM SYSTEMS JOURNAL*, 41(1):4–12, 2002.

[97] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), jun 1998. LNCS 1427, Springer Verlag.

[98] N. Halbwachs. *A Synchronous Language at Work: The Story of Lustre*, pages 15–31. John Wiley and Sons, Inc., 2012.

[99] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language Lustre. *IEEE Transactions on Software Engineering*, 18(9):785–793, Sep 1992.

[100] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.-C. Glory. *Specifying, programming and verifying real-time systems using a synchronous declarative language*, pages 213–231. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[101] N. Halbwacks, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of IEEE*, 79:1305–1320, Sept. 1991.

[102] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[103] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[104] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32(1):137–161, 1985.

[105] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[106] H. Hong, I. Lee, and O. Sokolsky. Automatic test generation from statecharts using model checking. Technical report, University of Pennsylvania, Janurary 2001.

[107] H. Huang, W. T. Tsai, and R. Paul. Automated model checking and testing for composite web services. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 300–307, May 2005.

[108] M. Huhn and S. Milius. Observations on formal safety analysis in practice. *Sci. Comput. Program.*, 80:150–168, Feb. 2014.

[109] E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont. *Environment-Model Based Testing of Control Systems: Case Studies*, pages 636–650. Springer Berlin Heidelberg, 2014.

[110] E. Jahier, N. Halbwachs, and P. Raymond. Engineering functional requirements of reactive systems using synchronous languages. In *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*, 2013.

[111] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer*, 8(6):517–530, 2006.

[112] E. Jahier, P. Raymond, and N. Halbwachs. *The Lustre V6 Reference Manual*, 2016.

[113] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[114] T. Kahsai and C. Tinelli. Pkind: a parallel k-induction based model checker. In *Proceedings 10th International Workshop on Parallel and Distributed Methods in verification, PDMC 2011, EPTCS*, volume 72, pages 55—62, 2011.

[115] R. M. Karp and R. E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, May 1969.

[116] K. Y. Koh and P. H. Seong. SACS2: A dynamic and formal approach to safety analysis for complex safety critical systems. In *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies, NPIC& HMIT*, pages 5–9, 2009.

[117] A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 17–34, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[118] S. R. Koo, P. H. Seong, J. Yoo, S. D. Cha, C. Youn, and H.-C. Han. Nusee: An integrated environment of software specification and v&v for plc based safetycritical systems. *Nuclear Engineering and Technology*, 38(3):259–276, 2006.

[119] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[120] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[121] J. Lahtinen, J. Valkonen, K. Bjőrkman, J. Frits, I. Niemelä, and K. Heljanko. Model checking of safety-critical software in the nuclear engineering domain. *Reliability Engineering& System Safety*, 105:104 – 113, 2012.

[122] P. Le Guernic, M. Le Borgne, T. Gautier, and C. Le Maire. Programming real time applications with SIGNAL. Research Report RR-1446, INRIA, 1991.

[123] H. LeGuen and T. Thelin. Practical Experiences with Statistical Usage Testing. In *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'04)*.

[124] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL'85: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–107. ACM Press, 1985.

[125] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[126] B. Marre and A. Arnould. Test Sequences generation from LUSTRE Descriptions: GATEL. In *15th IEEE Conf. on Automated SW Engineering*, pages 47–60, 2000.

[127] B. Marre and B. Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111:93 – 111, 2005.

[128] K. L. McMillan. *The SMV System*, pages 61–85. Springer US, Boston, MA, 1993.

[129] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[130] G. Memmi. Conduite du test à chrysalis: un retour d'expérience. *Génie Logiciel*, 72:53–58, 2005. invited conference at ICSSEA'05.

[131] G. Memmi. Integrated circuits analysis, system and method using model-checking. US Patent 7493247, feburary 2009.

[132] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, pages 15–18, 2005.

[133] S. P. Miller. Bridging the gap between model-based development and model checking. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 443–453, Berlin, Heidelberg, 2009. Springer-Verlag.

[134] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, Feb. 2010.

[135] L. Ness. Issues Arising In the Analysis of L.0. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification: 2nd International Conference, CAV'90 Proceedings*, pages 106–115. Springer-Verlag, 1990.

[136] M. Neyret, F. Dormoy, and J. Blanchon. Méthodologie de validation des spécification fonctionnelles du contrôle-commande - Application au cas d'étude du Système de Réfrigération intermédiaire (SRI). *Génie Logiciel*, hors-séries:12–25, mai 2014.

[137] R. D. Nicola and F. Vaandrager. Action versus state based logics for transistion systems. In *Proceedings of the LITP Spring School on Theorectical Computer Science on Semantics of Systems of Concurrent Processes*, pages 407–419. Springer: New York, 1990.

[138] F. Ortmeier, G. Schellhorn, A. Thums, W. Reif, B. Hering, and H. Trappschuh. Safety analysis of the height control system for the Elbtunnel. *Reliability Engineering & System Safety*, 81(3):259–268, 2003.

[139] G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *International Journal on Software Tools for Technology Transfer*, 5(2):158–164, 2004.

[140] I. Parissis. *Test de logiciels synchrones spécifiés en Lustre. (Testing synchronous software specified in Lustre)*. PhD thesis, Joseph Fourier University, Grenoble, France, 1996.

[141] C. Pixley. A theory and implementation of sequential hardware equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(12):1469–1478, 1992.

[142] A. Pnueli. The temporal logic of programs. In *Eighteenth Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Press, 1977.

[143] A. Pnueli. Current trends in concurrency. overviews and tutorials. chapter Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, pages 510–584. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[144] A. Pretschner and J. Philipps. 10 Methodological Issues in Model-Based Testing. In M. B. et al., editor, *Model-Based Testing of Reactive Systems*, pages 281–291, 2005.

[145] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *5th International Symposium on Programming*, volume

137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[146] G. Ratzaby, S. Ur, and Y. Wolfsthal. Coverability Analysis Using Symbolic Model Checking, CHARME 2001. In *Lectured Notes in Computer Science*, volume 2144. Springer-Verlag, 2001.

[147] S. Rayadurgam and M. P. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. *IEEE*, 2001.

[148] P. Raymond. *LUSTRE-V4 manual (draft)*, 2013.

[149] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 200–, Washington, DC, USA, 1998. IEEE Computer Society.

[150] W. W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[151] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[152] S. Schulz, J. W. Rozenbilt, and K. J. Buchenrieder. Multilevel testing for design verification of embedded systems. *IEEE Des. Test*, 19(2):60–69, mar 2002.

[153] E. M. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 428–435, Washington, DC, USA, 1996. IEEE Computer Society.

[154] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, UK, 2000. Springer-Verlag.

[155] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *Proceedings of the 1996 European Conference on Design and Test*, EDTC '96, pages 328–, Washington, DC, USA, 1996. IEEE Computer Society.

[156] A. Spillner. The W-model strengthening the bond between development and test. In *STAReast2002*, 2002.

[157] Y. Sun, G. Memmi, and S. Vignes. Model-based testing directed by structural coverage and functional requirements. In *IEEE conference QRS 2016 Workshop MVV*, Aug. 2016.

[158] Y. Sun, G. Memmi, and S. Vignes. A model-based testing process for enhancing structural coverage in functional testing. In *Complex Systems Design & Management Asia*, pages 171–180. Springer, Feb. 2016.

[159] Y. Sun, G. Memmi, S. Vignes, and F. Daumas. CONNEXION: Éléments de méthodologie de vérification et validation - Épisode 1: découvrir les challenges principaux. *Génie Logiciel*, 109:23–33, juin 2014.

[160] M. Tatar and J. Mauss. Systematic test and validation of complex embedded systems. In *ERTS 2014 - Embedded Real Time Software and Systems*, 2014.

[161] D. E. Thomas and P. R. Moorby. *The VERILOG Hardware Description Language*. Kluwer Academic Publishers, Norwell, MA, USA, 3rd edition, 1996.

[162] M. Tka. *Génération automatique de test pour les contrôleurs logiques programmables synchrones*. PhD thesis, 2016. Thèse de doctorat dirigée par Parissis, Ioannis Informatique Grenoble Alpes 2016.

[163] H. Touati and G. Berry. Optimized controller synthesis using esterel. In *Proceedings of International Workshop on Logic Synthesis*, 1993.

[164] S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.

[165] M. Utting and B. Legeard. *Practical Model Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[166] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. *Working Paper Series*, April 2006.

[167] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332–334. IEEE Computer Society: Silver Spring, MD, 1986.

[168] Verimag. *A Lustre V6 Tutorial*, 2008.

[169] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm. *Integration of Formal Analysis into a Model-Based Software Development Process*, pages 68–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[170] M. Whalen, A. Rajan, and M. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 25–36. ACM, July 2006.

[171] J. Yoo, S. Cha, and E. Jee. A verification framework for fbd based software in nuclear power plants. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 385–392. IEEE, 2008.

[172] J. Yoo, E. Jee, and S. Cha. Formal modeling and verification of safety-critical software. *IEEE software*, 26(3):42–49, 2009.

[173] H. Zhu, P. A. Hall, and J. H. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.