République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Abderrahmane Mira de Béjaïa
Faculté des Sciences Exactes
Département d'Informatique

Mémoire de fin de cycle

En vue de l'obtention du diplôme de Master Recherche en Informatique

Option : **Systèmes d'information avancés**

Thème :

Vers une conception sûre des systèmes embarqués basée sur les SR-models
Use case : IDCT

Présenté par

HAMMICHE Imen
HAMITI Taous

Encadré par

Dr Chabane-Mechiouri Sarah

Jury

Mr Amroun Kamal : Président
Mr Khammari Mohammed : Examinateur
Mme Boukerram Samira : Examinatrice
Mme Adel Karima : Examinatrice

Année universitaire 2024 / 2025

Dédicaces

Nous dédions ce mémoire

À nos chers parents,
Pour leur amour, leur patience et leurs sacrifices inestimables.

À nos frères et sœur ,
Pour leur soutien constant et leur présence rassurante.

Que ce travail soit le reflet de toute notre reconnaissance.

Remerciements

Nous tenons à exprimer notre profonde gratitude à **Madame Mechiouri Sarah**, notre encadrante, pour sa disponibilité, sa rigueur et son accompagnement bienveillant.
Son investissement constant, ses conseils avisés et son exigence scientifique ont été déterminants dans l'avancement et l'aboutissement de ce projet.
Nous lui sommes extrêmement reconnaissantes pour le temps qu'elle nous a consacré, ainsi que pour sa patience et son engagement tout au long de ce travail.
Ce mémoire est aussi le vôtre.

Nous remercions sincèrement les membres du jury : **Monsieur Amroun Kamal**, **Monsieur Khammari Mohammed**, **Madame Boukerram Samira** et **Madame Adel Karima**, pour avoir accepter d'évaluer notre travail.

Nos remerciements les plus profonds vont à nos parents.
À nos mamans, véritables piliers dans nos vies, pour leur amour infini, leurs sacrifices silencieux, leurs mots réconfortants et leur force admirable.
À nos papas, pour leur confiance, leur courage et leur soutien constant, même dans les moments les plus exigeants.
Merci de nous avoir appris à persévérer, à croire en nous, et à viser toujours plus haut.
Ce chemin, nous l'avons parcouru grâce à vous.

Nous exprimons aussi toute notre gratitude à nos frères et sœur.
Merci pour vos encouragements, vos sourires, vos blagues pour détendre l'atmosphère et votre présence rassurante.

Résumé

Ce mémoire s'inscrit dans le contexte du développement de systèmes embarqués critiques, dont l'intégration est aujourd'hui omniprésente grâce à l'essor des systèmes cyber-physiques et de l'Internet des objets. Ces systèmes, toujours plus complexes, doivent répondre à des exigences croissantes en matière de performance, de fiabilité et de rapidité de développement. Face à cette complexité, la réutilisation de composants, associée à des approches de conception rigoureuses, constitue une solution efficace. La modélisation formelle, notamment, permet de garantir des propriétés de sûreté dès les premières phases de conception.

Dans ce cadre, une attention particulière est portée aux systèmes réactifs, qui interagissent en permanence avec leur environnement. Après avoir étudié leurs caractéristiques, leurs principaux domaines d'application et les approches classiques de modélisation, nous nous sommes intéressées aux automates d'entrée/sortie (I/O automata), largement utilisés pour représenter les comportements de ces systèmes. Nous avons examiné les extensions existantes ainsi que les principes de la conception modulaire, afin d'identifier les leviers permettant d'améliorer la fiabilité et la vérification des systèmes réactifs.

Sur cette base, nous proposons un nouveau cadre de modélisation appelé *SR-Dep* (Synchronous Reactive Dependency model), qui enrichit les SR-modèles existants — une extension des I/O-automata — en introduisant explicitement des règles de dépendance entre les entrées et les sorties. Cette approche, orientée composants, vise à construire des systèmes corrects par construction, tout en assurant la préservation de leur sémantique comportementale. Pour démontrer la faisabilité et la pertinence de notre proposition, nous avons réalisé une implémentation concrète, illustrée par une étude de cas sur le composant IDCT. Les résultats obtenus confirment l'intérêt du modèle SR-Dep proposé pour la modélisation fiable et modulaire de systèmes réactifs embarqués.

Abstract

This thesis is set in the context of the development of critical embedded systems, whose integration is now ubiquitous due to the rise of cyber-physical systems and the Internet of Things. These increasingly complex systems must meet growing demands in terms of performance, reliability, and fast development. To address this complexity, component reuse combined with rigorous design approaches offers an effective solution. In particular, formal modeling enables the guarantee of safety properties from the earliest design phases.

In this context, special attention is given to reactive systems, which continuously interact with their environment. After studying their characteristics, main application domains, and classical modeling approaches, we focused on input/output automata (I/O automata), which are widely used to represent the behavior of such systems. We examined existing extensions as well as the principles of modular design in order to identify ways to improve the reliability and verification of reactive systems.

Based on this analysis, we propose a new modeling framework called *SR-Dep* (Synchronous Reactive Dependency model), which enriches existing SR-models—an extension of I/O automata—by explicitly introducing dependency rules between inputs and outputs. This component-oriented approach aims to build systems that are correct by construction, while ensuring the preservation of their behavioral semantics. To demonstrate the feasibility and relevance of our proposal, we developed a concrete implementation, illustrated by a case study on the IDCT component. The results obtained confirm the relevance of the proposed SR-Dep model for the reliable and modular modeling of embedded reactive systems.

Table des matières

Introduction générale **8**

1 Généralités sur les systèmes réactifs **10**
1.1 Les systèmes réactifs . . . 10
1.2 Les caractéristiques des systèmes réactifs . . . 11
1.3 Les domaines d'application des systèmes réactifs . . . 12
1.3.1 Systèmes de contrôle industriel . . . 12
1.3.2 Supervision et surveillance . . . 13
1.3.3 Traitement du signal . . . 14
1.3.4 Interfaces homme-machine (IHM) . . . 16
1.3.5 Systèmes embarqués dans l'automobile . . . 16
1.3.6 Aéronautique et spatial . . . 17
1.3.7 Systèmes médicaux . . . 18
1.3.8 Robotique . . . 19
1.3.9 Jeux vidéo et simulations . . . 20
1.4 Approches de conception des systèmes réactifs . . . 21
1.4.1 L'approche synchrone . . . 21
1.4.2 L'approche asynchrone . . . 23
1.5 Comparaison entre les systèmes synchrones et asynchrones . . . 24
1.6 Modèles de conception des systèmes réactifs . . . 24
1.6.1 Automates déterministes . . . 24
1.6.2 Modèles basés sur les réseaux de Petri . . . 25
1.6.3 Modèles basés sur les tâches . . . 25
1.6.4 Processus communicants . . . 25
1.7 Fiabilité des systèmes réactifs . . . 25
1.7.1 Méthodes de vérification des systèmes . . . 26

2 Etat de l'art sur les Input/Output automata **29**
2.1 Problématiques liés à la modélisation des systèmes réactifs . . . 29
2.1.1 Complexité croissante . . . 29
2.1.2 Contraintes de performance . . . 30
2.1.3 Problématiques de sécurité et sûreté (fiabilité) . . . 30
2.2 Modélisation des systèmes réactifs . . . 30
2.2.1 Réseaux de processus . . . 30
2.2.2 Réseaux de Pétri . . . 31
2.2.3 Les automates d'Entrée/Sortie . . . 31
2.2.4 Les automates d'interfaces . . . 31
2.2.5 Les automates Mode (Mode-automata) . . . 31
2.2.6 Les langages Synchrones . . . 32
2.3 Approche de conception par composition . . . 32

2.3.1 Objectifs de la conception par composition . . . 33
2.3.2 Caractéristiques de la conception par composants . . . 33
2.3.3 Plateformes et outils de conception basée composants . . . 34
2.4 Introduction aux automates d'entrée/sortie (I/O Automata) . . . 40
2.4.1 L'Orientation des I/O Automata vers l'Asynchrone . . . 41
2.5 Les différentes variantes des I/O automata . . . 41
2.5.1 Timed I/O Automata (TIOA) . . . 41
2.5.2 Hybrid I/O Automata (HIOA) . . . 42
2.5.3 Probabilistic I/O Automata . . . 42
2.5.4 Dynamic I/O Automata . . . 43
2.5.5 Probabilistic Dynamic I/O Automata (PDIOA) . . . 44
2.5.6 Interface I/O Automata . . . 44
2.6 Positionnement de nos travaux . . . 44

3 Contribution
I/O-Dépendance : une nouvelle dimension pour les SR-modèles 46
3.1 Systèmes synchrones réactifs . . . 46
3.2 Modélisation . . . 47
3.2.1 Abstraction des données . . . 48
3.2.2 Paramètres des SR-Systèmes . . . 48
3.3 SR-modèles(Synchronous reactive models) . . . 49
3.3.1 Choix des SR-modèles . . . 49
3.3.2 Définition d'un SR-modèle . . . 50
3.3.3 Sémantique des SR-modèles . . . 50
3.3.4 Représentation des états d'un SR-modèle . . . 51
3.3.5 Exemples SR-modèles . . . 52
3.4 Problématique . . . 53
3.5 Proposition . . . 54
3.6 Contribution : SR-Dep modèle (Synchrounous reactive dependancy model) 54
3.6.1 Propriétés de dépendance des entrées/sorties . . . 55
3.6.2 Exemples de SR-Dep Modèles . . . 56

4 Évaluation du SR-Dep modèle par étude de cas : IDCT component 59
4.1 Implémentation des SR-Dep modèles . . . 59
4.1.1 Structures et bibliothèques utilisées . . . 59
4.1.2 Utilisation de listes et fonctions auxiliaires . . . 60
4.1.3 Fonction `construct` : génération du LTS SR-Dep . . . 61
4.1.4 Ajout des dépendances Id et Od . . . 61
4.2 Etude de cas : composant IDCT . . . 63
4.2.1 IDCT (Inverse Discret Cosine Transform) . . . 63
4.2.2 Interface . . . 63
4.2.3 Architecture . . . 64
4.3 Modélisation des composants d'IDCT avec les SR-Dep modèles . . . 66
4.3.1 Vérification formelle de la sémantique des SR-Dep modèles . . . 68
4.3.2 Discussion . . . 72

Conclusion générale 73

Table des figures

1.1 Système réactif. . . . 11
1.2 Exemples de systèmes réactifs dans un système de contrôle . . . 13
1.3 Les feux de signalisation intelligents . . . 14
1.4 Plateforme de test expérimentale pour Siri . . . 15
1.5 Organigramme du système de surveillance vidéo intelligent . . . 15
1.6 Exemple de tableau de bord du logiciel JIRA. . . . 16
1.7 Principe de fonctionnement de l'ABS. . . . 17
1.8 Radar de surveillance. . . . 18
1.9 Réseaux corporels pour le domaine médical. . . . 19
1.10 Classification en robotique industrielle. . . . 20
1.11 Illustration de l'usage d'une signalétique réaliste dans le jeu vidéo Portal . 21
1.12 Valeur RMS de la moyenne synchrone d'un signal microphonique . . . 22
1.13 Processus du model checking . . . 27
1.14 Processus de la preuve par théorème . . . 28

3.1 Une vue en boite noire d'un composant . . . 48
3.2 SR-modèle du composant avec $M = 1$, $L = 1$ . . . 52
3.3 SR-modèle du composant avec $M = 2$, $L = 2$ . . . 53
3.4 Un SR-composant avec Dépendance d'entrées/sorties . . . 54
3.5 SR- Composant avec deux entrées (i_1, i_2) et une sortie (o) . . . 56
3.6 SR-Dep modèle pour $M = 2$, $L = 2$, $Id = 2$, $Od = 1$ . . . 56
3.7 Composant SR-Dep avec deux entrées (i_1, i_2) et deux sorties (o_1, o_2) . . . 57
3.8 Automate SR-Dep pour $M = 2$, $L = 2$, $Id = 2$, $Od = 2$ . . . 57

4.1 Fonction auxilière tous_elements_egaux. . . . 60
4.2 Fonction auxilière count_$\bar{i}, o_predecessors$. . . . 61
4.3 Code python : Contrainte sur les entrées. . . . 62
4.4 Code python : Condition sur les sorties. . . . 62
4.5 Architecture interne du composant IDCT. . . . 64
4.6 SR-Dep modèle du composant INR et POST . . . 66
4.7 SR-Dep modèle du composant ACCU. . . . 67
4.8 Partie 1 du SR-Dep modèle du composant TRSP . . . 67
4.9 Partie 2 du SR-Dep modèle du composant TRSP . . . 68

Liste des tableaux

1.1 Comparaison entre les systèmes synchrones et asynchrones 24

2.1 Comparaison des plateformes de modélisation 45

4.1 Vérification formelle de la Propriété 1 pour le SR-Dep modèle d'INR et POST 69
4.2 Vérification formelle de la Propriété 2 pour le SR-Dep modèle d'INR et POST 69
4.3 Vérification formelle de la Propriété 1 pour tous les états d'ACCU 70
4.4 Vérification formelle de la Propriété 2 pour le SR-Dep modèle d'ACCU . . 70
4.5 Vérification formelle de la Propriété 1 pour tous les états de TRSP 71
4.6 Vérification formelle de la Propriété 2 pour le SR-Dep modèle de TRSP . . 71

Introduction générale

La croissance rapide des technologies embarquées et leur intégration croissante dans des domaines critiques tels que l'aéronautique, l'automobile, la médecine ou encore l'industrie, posent aujourd'hui d'importants défis en matière de fiabilité, de sécurité et de performances. Ces systèmes, appelés systèmes réactifs embarqués, sont caractérisés par leur interaction continue avec leur environnement et par leur exigence de réponse en temps contraint. Assurer leur conception correcte et fiable est devenu un enjeu central, notamment face à la complexité croissante de leur architecture et de leur comportement.

Le caractère critique des systèmes embarqués impose aux concepteurs des exigences de fiabilité particulièrement élevées. L'adoption de méthodes formelles s'avère particulièrement rentable, permettant à la fois de réduire les coûts et délais de validation tout en améliorant significativement la fiabilité. Cette approche assure l'adéquation entre le comportement réel du système et sa spécification formelle préalable, permettant une diminution substantielle des coûts et délais de commercialisation par l'identification anticipée des défauts durant la phase de conception.

L'utilisation des méthodes formelles nécessite une théorie qui permet de représenter le comportement des systèmes à vérifier. De nombreux modèles sont proposés dans la littérature. Dans le contexte de ce travail nous nous intéressons aux SR-modèles (Synchronous Reactive models) qui sont une extension des Input/Output automata qui se distinguent par leur capacité à capturer à la fois la dynamique interne des composants et les contraintes temporelles associées aux communications des systèmes réactifs.

Dans ce contexte,nous proposons d'enrichir les SR-modèles classiques par l'introduction d'une nouvelle dimension basée sur la dépendance I/O (d'entrées/sorties) , afin de mieux représenter les relations causales entre les actions du système. Cette contribution se traduit par la définition d'un nouveau modèle SR-Dep (Synchronous Reactive Dependency model), permettant d'intégrer les propriétés de dépendance entre les entrées et les sorties, tout en conservant la structure et les propriétés fondamentales des SR-modèles.

Pour valider notre approche, nous l'avons évaluer sur un cas d'étude réel, à savoir le composant IDCT (Inverse Discrete Cosine Transform), utilisé dans le cadre de traitements multimédia embarqués. Ce choix s'explique par la nature fortement réactive de ce composant, son architecture modulaire et sa sensibilité aux contraintes de synchronisation. L'étude de ce cas pratique permet d'illustrer les apports du modèle SR-Dep en termes de précision de modélisation, de vérification sémantique et de préservation des propriétés de sûreté.

Ce mémoire est structuré de la manière suivante :

— Le premier chapitre présente les notions fondamentales relatives aux systèmes réactifs.

— Le second chapitre est consacré à un état de l'art sur les modèles formels de description des systèmes réactifs, avec un intérêt particulier sur les I/O-Automata.

— Le troisième chapitre présente notre contribution, à travers la formalisation des SR-Dep modèles.

— Le quatrième et dernier chapitre évalue notre proposition à travers une modélisation complète du composant IDCT.

— Nous finirons ce mémoire avec une conclusion générale et les perspectives de ce travail.

Chapitre 1

Généralités sur les systèmes réactifs

Introduction

Dans le paysage actuel de l'informatique, où les systèmes sont appelés à interagir en temps réel avec des environnements souvent imprévisibles, les systèmes réactifs représentent une catégorie incontournable. Ce type de systèmes se distingue par sa capacité à répondre de manière continue et immédiate aux sollicitations extérieures. Contrairement aux systèmes dits transformationnels, qui produisent un résultat à partir de données d'entrée fixes, les systèmes réactifs fonctionnent dans un cycle permanent d'observation et de réaction face aux événements externes.

Apparus dans les années 1980, les systèmes réactifs ont été théorisés afin de répondre aux exigences croissantes des applications critiques, où un simple retard ou une mauvaise réponse peut engendrer des conséquences graves, tant sur le plan matériel qu'humain. De nombreux domaines, tels que l'industrie, les transports, la santé ou les télécommunications, reposent sur ces systèmes pour assurer des fonctions de contrôle, de supervision ou d'automatisation.

Ce chapitre vise à introduire les concepts fondamentaux liés aux systèmes réactifs, à clarifier leur positionnement par rapport à d'autres types de systèmes, et à mettre en lumière les contraintes spécifiques qu'ils doivent respecter — notamment en matière de temps réel et de déterminisme. Ces éléments fourniront un socle théorique solide en vue d'explorer, dans les chapitres suivants, les approches de modélisation et de vérification adaptées à la conception fiable de ces systèmes.

1.1 Les systèmes réactifs

Le concept de systèmes réactifs a été introduit dans les années 1980 par D. Harel et A. Pnueli [35] afin de désigner une catégorie spécifique de systèmes informatiques conçus pour maintenir une interaction continue avec leur environnement. Contrairement aux systèmes transformationnels, qui exécutent un traitement à partir de données initiales pour produire un résultat final, les systèmes réactifs fonctionnent en répondant en permanence aux sollicitations extérieures.

Cette notion, développée par D. Harel et al.[35], oppose les systèmes réactifs aux systèmes interactifs. Les premiers doivent réagir immédiatement aux sollicitations externes

selon un rythme imposé par l'environnement, tandis que les seconds gèrent leurs interactions à leur propre cadence, comme c'est le cas des systèmes d'exploitation.

Avec l'avènement de la programmation synchrone, cette distinction s'est précisée : les systèmes interactifs peuvent moduler le moment de leurs échanges, alors que les systèmes réactifs doivent impérativement répondre sans délai aux stimuli reçus.

Ces systèmes fonctionnent selon un cycle de réactions successives aux entrées reçues, chaque réponse pouvant influencer l'environnement. Leur implémentation doit donc garantir la prise en compte de tous les stimuli en temps voulu. Par exemple, dans un système de contrôle industriel, des capteurs récupèrent des informations depuis l'environnement et les transmettent au système de contrôle, qui réagit en conséquence via des actionneurs exécutant des commandes en retour.

Les systèmes réactifs sont souvent associés aux systèmes temps réel, qui imposent des contraintes strictes sur les délais de réponse. Un retard dans le traitement d'un événement peut compromettre l'efficacité du système, rendant ses résultats inutilisables. Ainsi, le délai de réponse n'est pas seulement une mesure de performance, mais une contrainte essentielle pour assurer le bon fonctionnement du système.

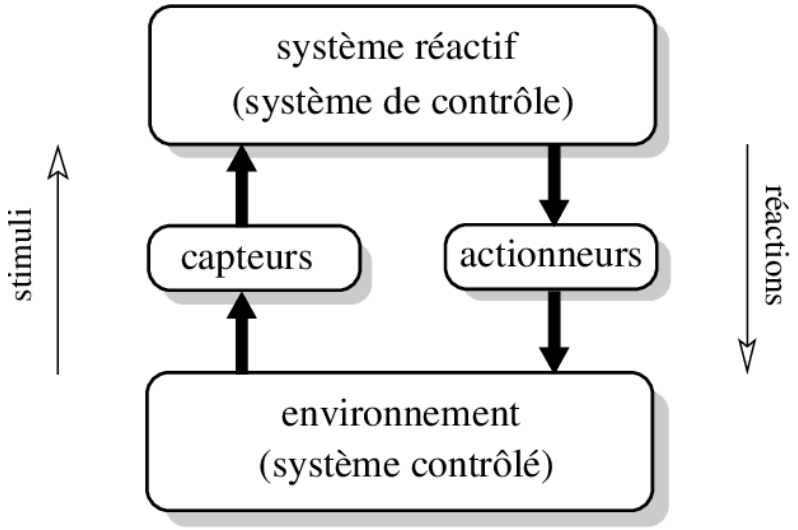

FIGURE 1.1 – Système réactif.
[42]

1.2 Les caractéristiques des systèmes réactifs

Pour mieux cerner la nature et le fonctionnement des systèmes réactifs, il est essentiel de s'intéresser aux éléments qui les définissent et les différencient des autres catégories de systèmes ; voilà les caractéristiques qui leur sont propres.[32]

— **Concurrence** : Interaction constante avec l'environnement et exécution en parallèle de plusieurs composants pour assurer le comportement souhaité. La décomposition logique d'un système en processus parallèles ne correspond pas nécessairement à son implémentation physique ou à une amélioration des performances.

— **Contraintes temporelles** : Respect strict des délais d'entrée et de sortie. Les

contraintes temporelles doivent être définies dans les spécifications, prises en compte lors de la conception et vérifiées à l'implémentation. Une exécution efficace repose sur une évaluation précise des temps de traitement.

— **Déterminisme** : Les sorties dépendent uniquement des valeurs et des instants d'entrée, contrairement aux systèmes interactifs qui sont souvent non déterministes. Ce déterminisme simplifie la conception, l'analyse et le débogage, et doit être préservé dans l'implémentation.

— **Critique (Fiable)** : Toute erreur peut avoir des conséquences graves (humaines ou financières), nécessitant des méthodes de conception rigoureuses et une vérification formelle pour garantir la sûreté du système.

— **Implémentation mixte matériel/logiciel** : Certains systèmes sont encore entièrement matériels pour des raisons de performance ou de coût. D'autres combinent matériel et logiciel, avec une séparation tardive entre ces deux parties au cours de la conception.

— **Réactivité élevée** : Capacité à réagir instantanément aux événements externes. Un système réactif doit traiter les entrées en temps réel ou quasi-réel pour assurer un fonctionnement fluide et éviter des délais critiques.

— **Sécurité et tolérance aux pannes** : Conception prenant en compte la gestion des erreurs et la continuité de service. Des mécanismes de redondance, de récupération et de détection d'anomalies sont souvent mis en place pour éviter les défaillances.

— **Optimisation des ressources** : Utilisation efficace des ressources (processeur, mémoire, énergie) pour garantir des performances optimales, surtout dans les systèmes embarqués ou contraints en ressources.

— **Complexité de validation et de test** :Vérification difficile en raison de l'interaction avec l'environnement et des multiples scénarios possibles. Des méthodes de test avancées, comme la simulation et la vérification formelle, sont souvent nécessaires.

1.3 Les domaines d'application des systèmes réactifs

Les systèmes réactifs trouvent des applications dans un grand nombre de secteurs où l'interaction en temps réel avec l'environnement est essentielle. Leur capacité à répondre rapidement et de manière fiable à des événements externes en fait des outils incontournables dans les contextes critiques. Voici quelques-uns des domaines majeurs où ces systèmes sont largement utilisés :

1.3.1 Systèmes de contrôle industriel

Dans l'industrie, les processus de production automatisés reposent sur des systèmes capables de surveiller l'état des machines, de capter des mesures en temps réel et d'agir immédiatement si des seuils critiques sont atteints. Les automates programmables industriels (API) exécutent des programmes cycliques de contrôle, tandis que les systèmes SCADA [57] collectent des données à distance et permettent aux opérateurs humains de superviser l'ensemble du système via une interface graphique.

Exemples de systèmes réactifs dans un système de contrôle :

— Une chaîne d'embouteillage détecte un bouchon mal positionné grâce à un capteur optique ; un actionneur est déclenché instantanément pour retirer la bouteille.

— Un lave-linge automatique ajuste la température, la vitesse d'essorage et la durée du cycle selon le programme sélectionné.

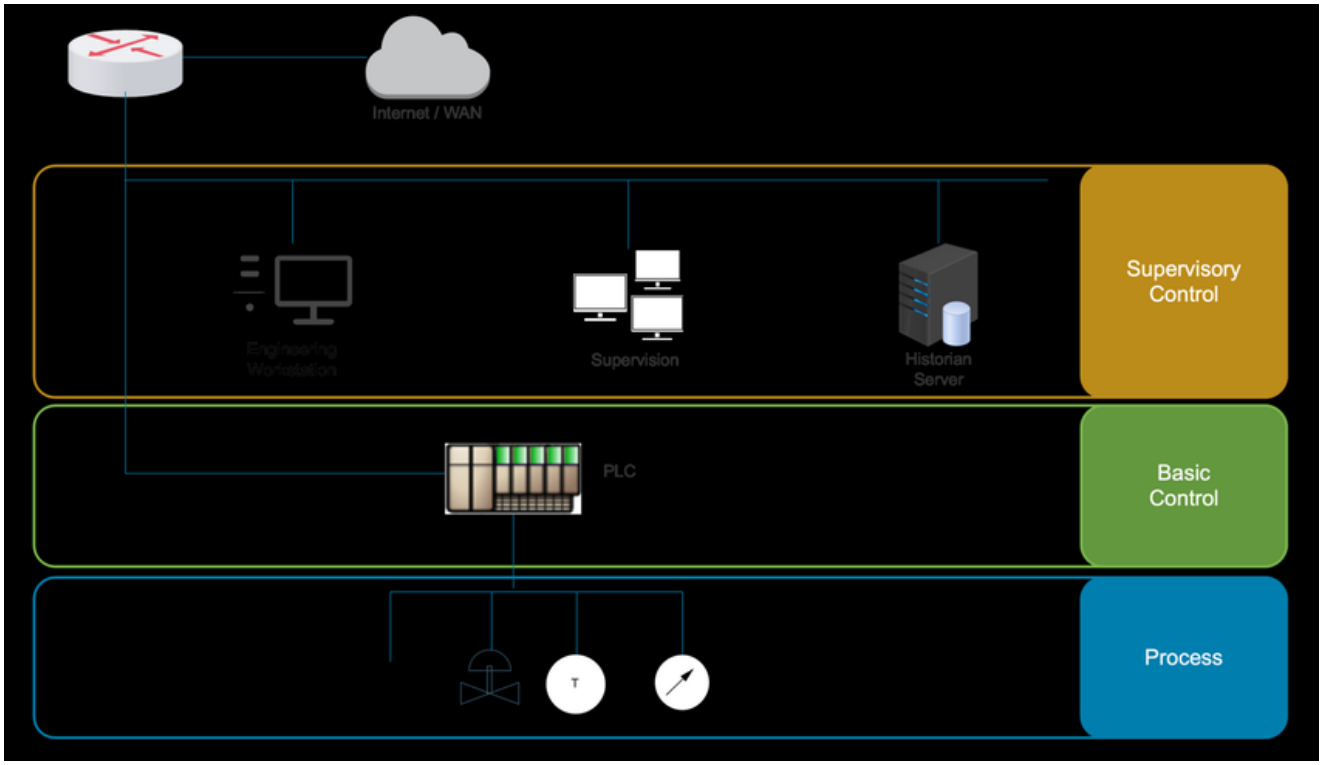

FIGURE 1.2 – Exemples de systèmes réactifs dans un système de contrôle [38]

1.3.2 Supervision et surveillance

Dans des environnements sensibles comme les centrales électriques, réseaux ferroviaires ou usines chimiques, la supervision repose sur des systèmes capables de détecter et réagir rapidement à toute défaillance. Ces systèmes assurent la continuité de service, préviennent les accidents et déclenchent des plans de secours automatiquement.

Exemples de systèmes réactifs dans un système de supervision

— Dans un réseau de distribution d'électricité, une chute de tension est automatiquement compensée par une redistribution de la charge.

— Les feux de signalisation intelligents adaptent la durée des cycles selon le trafic détecté.

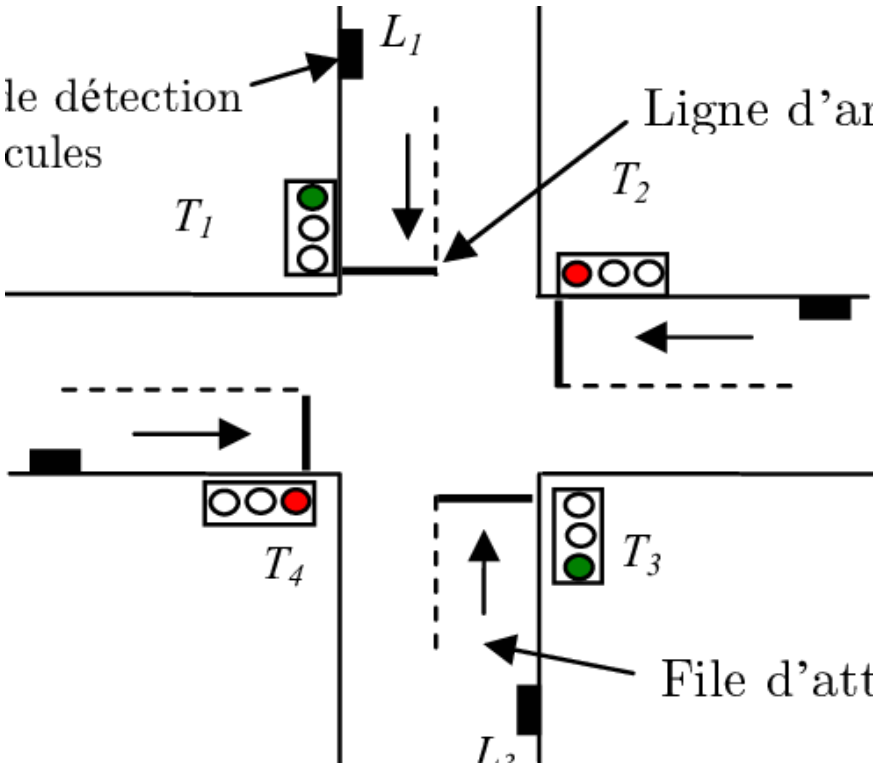

FIGURE 1.3 – Les feux de signalisation intelligents
[72]

1.3.3 Traitement du signal

Les systèmes de traitement du signal audio, vidéo ou capteurs intègrent des fonctions réactives pour traiter les flux d'information en continu. Par exemple, dans les systèmes de vidéoprotection intelligents, des algorithmes détectent des comportements suspects ou des anomalies en temps réel, et déclenchent des alertes automatiques.

a. Exemples de systèmes réactifs dans un système de traitement de signal

— Un assistant vocal comme Siri ou Google Assistant reconnaît une commande dès qu'elle est prononcée et y répond en temps réel.

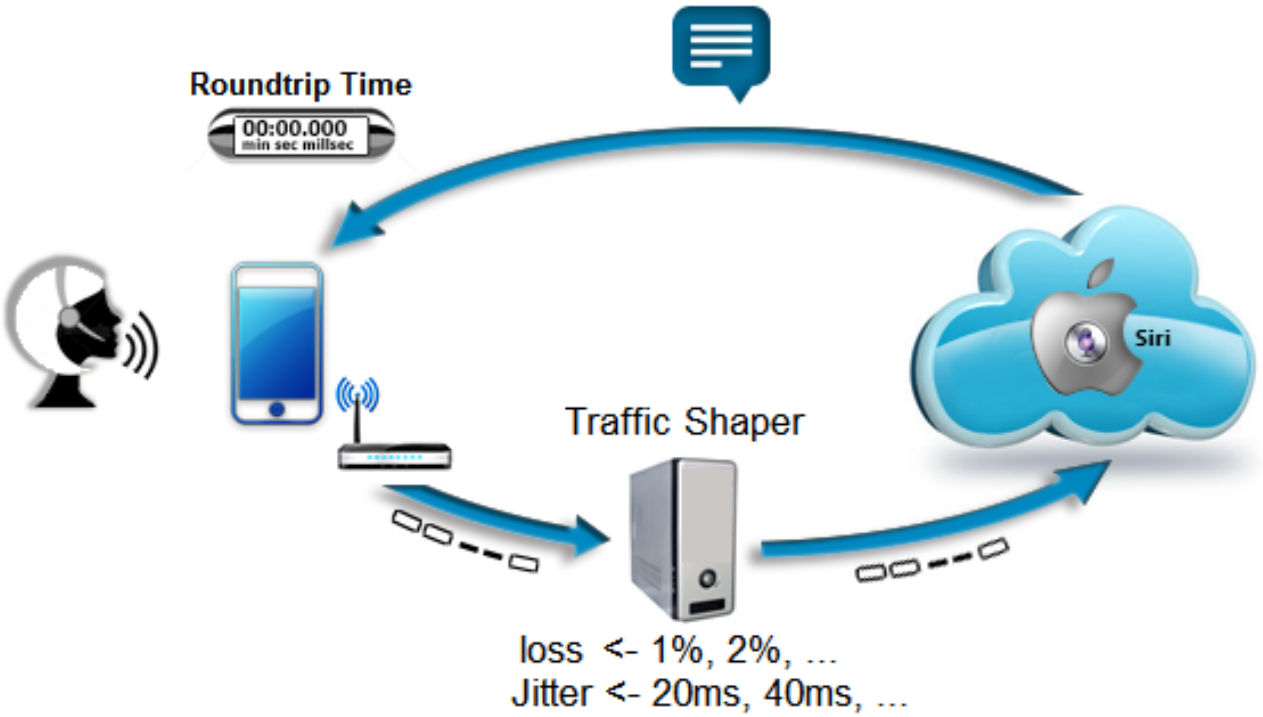

FIGURE 1.4 – Plateforme de test expérimentale pour Siri
[3]

— Une caméra de vidéosurveillance intelligente détecte un comportement suspect et déclenche une alerte instantanée.

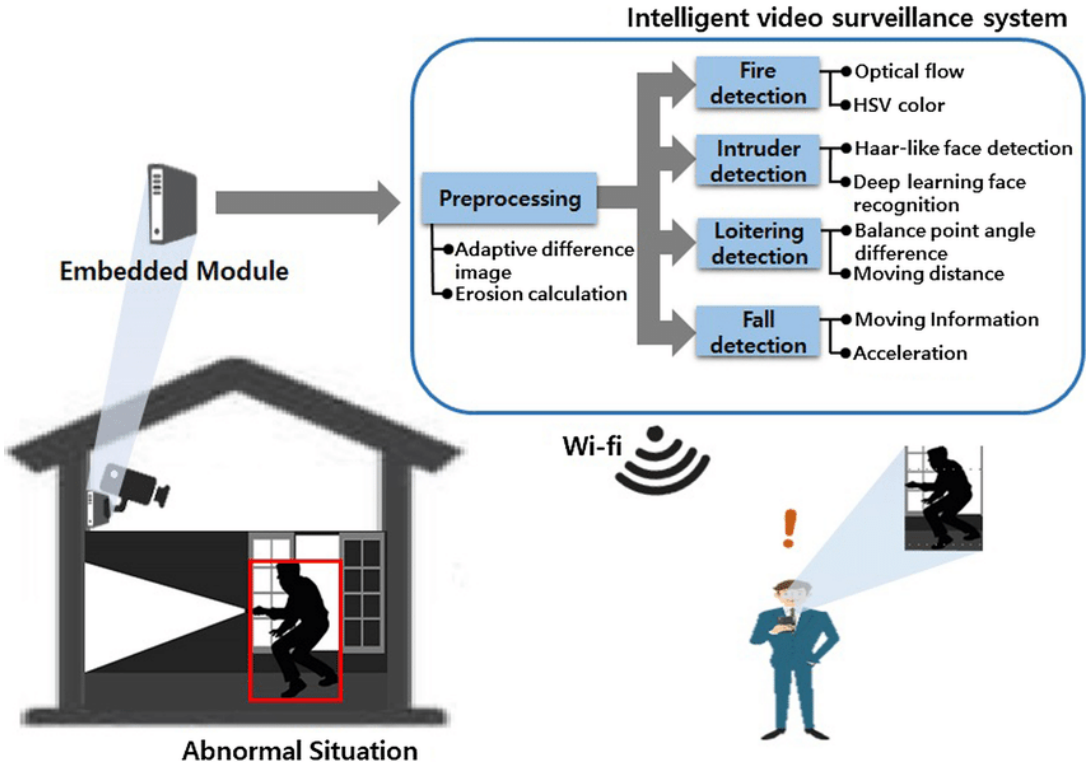

FIGURE 1.5 – Organigramme du système de surveillance vidéo intelligent
[45]

1.3.4 Interfaces homme-machine (IHM)

Les systèmes embarqués interactifs, comme les tableaux de bord numériques ou les écrans tactiles, nécessitent une réactivité parfaite à chaque action de l'utilisateur. Les systèmes doivent être capables de traiter rapidement les entrées pour modifier dynamiquement l'affichage ou le comportement du système.

Exemples de systèmes réactifs dans des IHM :

— Un conducteur règle la température via un écran tactile ; le système de climatisation s'ajuste immédiatement et met à jour l'affichage.

— Les tableaux de bord numériques des véhicules réagissent en temps réel aux commandes et aux conditions de conduite.

FIGURE 1.6 – Exemple de tableau de bord du logiciel JIRA.
[60]

1.3.5 Systèmes embarqués dans l'automobile

Le secteur automobile regorge de systèmes réactifs intégrés : systèmes d'assistance à la conduite, freinage d'urgence, gestion moteur, etc. Ces composants doivent collecter en continu les données de capteurs (accélération, freinage, position), et répondre en quelques millisecondes pour garantir la sécurité des usagers.

Exemples de systèmes réactifs dans les systèmes embarqués automobiles

— Le système ABS ajuste la pression sur les freins plusieurs fois par seconde pour éviter le blocage des roues lors d'un freinage brutal.

— Les airbags se déclenchent quelques millisecondes après la détection d'un choc.

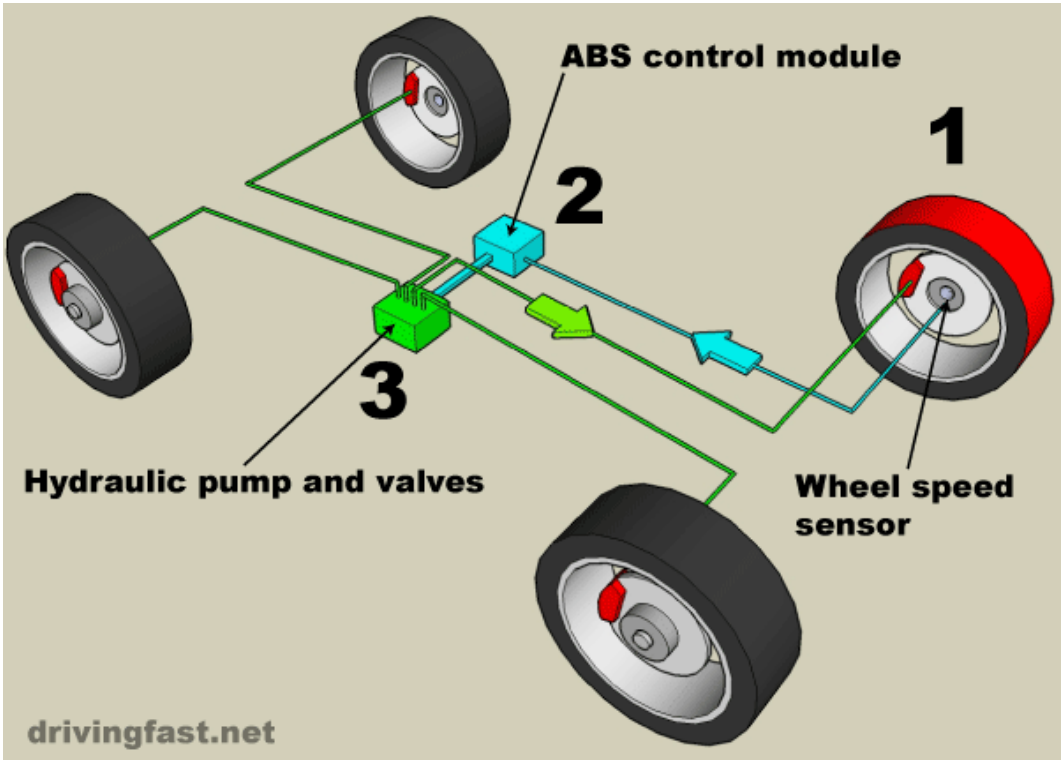

FIGURE 1.7 – Principe de fonctionnement de l'ABS.
[68]

1.3.6 Aéronautique et spatial

Le secteur automobile regorge de systèmes réactifs intégrés : systèmes d'assistance à la conduite, freinage d'urgence, gestion moteur, etc. Ces composants doivent collecter en continu les données de capteurs (accélération, freinage, position), et répondre en quelques millisecondes pour garantir la sécurité des usagers.

Exemples de systèmes réactifs en Aéronotique :

— Les systèmes de pilotage automatique ajustent l'altitude et la direction en fonction des capteurs de bord.

— Les radars de surveillance détectent la position des avions et ajustent les trajectoires pour éviter les collisions.

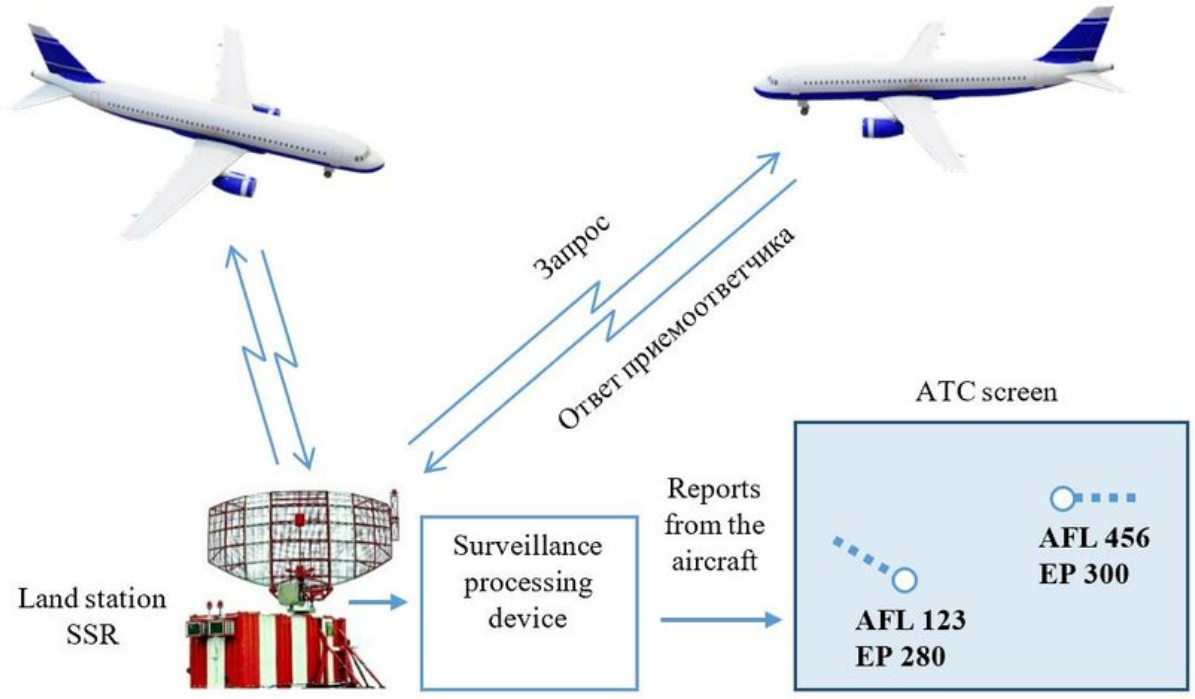

FIGURE 1.8 – Radar de surveillance.
[63]

1.3.7 Systèmes médicaux

Dans les environnements hospitaliers, les dispositifs médicaux réactifs surveillent en permanence les paramètres vitaux des patients (tension, fréquence cardiaque, glycémie) et réagissent automatiquement en cas de danger.

Exemples de systèmes réactifs dans les systèmes médicaux :

— Une pompe à insuline intelligente adapte la dose administrée selon la glycémie mesurée en temps réel.

— Un moniteur cardiaque déclenche une alarme en cas de rythme irrégulier.

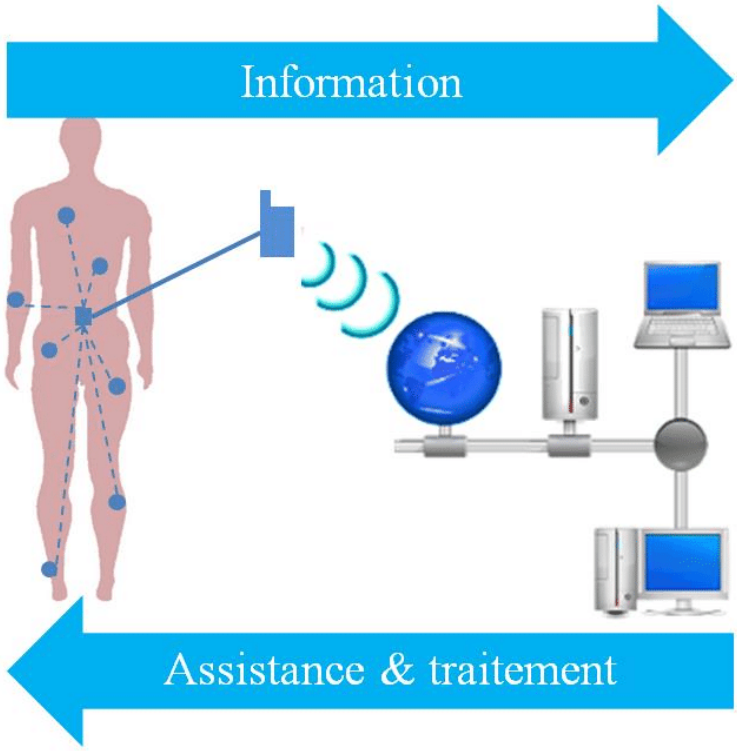

FIGURE 1.9 – Réseaux corporels pour le domaine médical.
[70]

1.3.8 Robotique

Les robots modernes doivent percevoir, décider et agir en quelques millisecondes. Qu'il s'agisse de robots de production industrielle ou de robots mobiles, la réactivité est indispensable pour ajuster les mouvements, éviter les obstacles ou collaborer avec des humains.

Exemples de systèmes réactifs dans la robotique

— Un robot industriel ajuste ses mouvements en temps réel pour suivre une pièce sur une chaîne de montage.

— Un robot collaboratif s'arrête instantanément dès qu'il détecte un humain dans sa zone d'action.

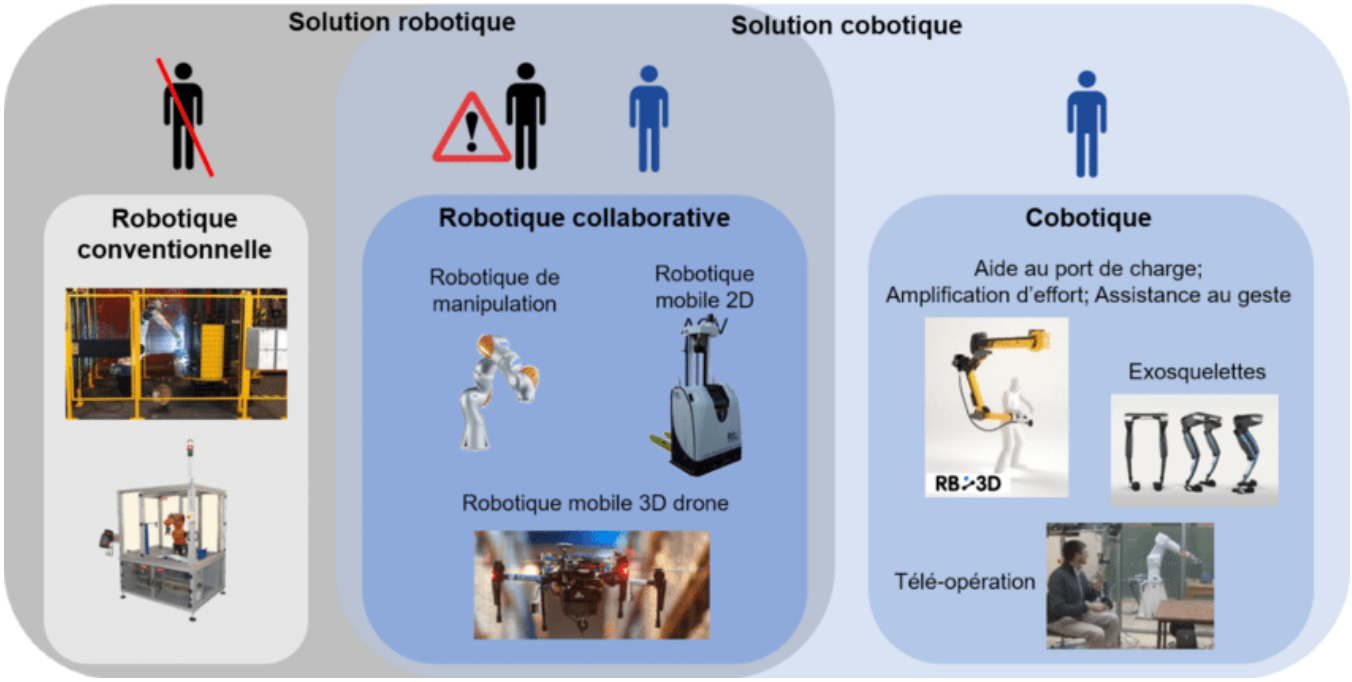

FIGURE 1.10 – Classification en robotique industrielle.
[14]

1.3.9 Jeux vidéo et simulations

Les moteurs de jeux utilisent des systèmes réactifs pour gérer les actions du joueur, mettre à jour l'environnement virtuel, et maintenir une synchronisation avec les entrées. Le moindre délai peut nuire à l'expérience de jeu, notamment dans les jeux compétitifs ou immersifs (réalité virtuelle, simulateurs de vol).

Exemples de systèmes réactifs dans les jeux vidéos et simulations

— Dans un jeu multijoueur, les actions de déplacement ou de tir sont traitées instantanément et synchronisées avec le serveur.

— Un simulateur de vol adapte immédiatement les conditions virtuelles en fonction des actions du pilote.

FIGURE 1.11 – Illustration de l'usage d'une signalétique réaliste dans le jeu vidéo Portal [23]

1.4 Approches de conception des systèmes réactifs

Concevoir des systèmes réactifs constitue un défi considérable, car leur conception nécessite de prendre en compte plusieurs exigences : la réactivité aux stimuli externes, le déterminisme des comportements, la gestion efficace de la concurrence, et le respect strict des contraintes temporelles imposées.

Pour relever ces défis, deux paradigmes majeurs se sont imposés dans le domaine : l'approche synchrone et l'approche asynchrone. Ces deux paradigmes fournissent des cadres conceptuels distincts et des méthodes d'implémentation différentes. Chacun présente ses propres avantages et limitations, les rendant plus ou moins appropriés selon les contextes d'application spécifiques.

1.4.1 L'approche synchrone

L'approche synchrone a fait son apparition avec les langages synchrones [9] qui offre aux concepteurs une abstraction permettant d'aborder la complexité temporelle inhérente aux systèmes qui interagissent continuellement avec leur environnement. L'approche synchrone repose sur un paradigme : « Réaction instantanée » ,i.e. un programme réagit sans délai perceptible aux évènements externes, ce qui introduit un déterminisme dans l'exécution. Chaque évènement généré par le système, qu'il soit interne ou visible depuis l'extérieur, possède une datation précise par rapport à la séquence des événements d'entrée. Ce qui donne un double déterminisme : un déterminisme sur l'aspect fonctionnel par rapport aux résultats produits et un déterminisme sur l'aspect temporel par rapport à l'ordre de production de ces résultats.

L'une des contributions théoriques majeures de l'approche synchrone réside dans sa reconceptualisation du temps. Le modèle s'affranchit de la notion traditionnelle de temps physique (chronométrique) pour adopter une vision centrée sur les relations d'ordre entre événements. Dans ce cadre conceptuel, seules deux relations temporelles sont considé-

rées comme significatives : la simultanéité et la précédence. Cette abstraction temporelle présente une propriété remarquable : le temps physique perd son statut privilégié pour devenir un événement externe parmi d'autres, au même titre que n'importe quel signal provenant de l'environnement.

La structure temporelle dans le modèle synchrone s'articule autour de la notion d'instant logique. Cette structure temporelle offre un cadre formel puissant pour raisonner sur les comportements temporels complexes.

a. Propriétés des modèles synchrones

Les modèles synchrones partagent les propriétés suivantes :

— Les événements qui surviennent au même instant logique sont considérés comme strictement simultanés

— Les événements apparaissant à des instants logiques différents sont ordonnés selon l'ordre total des instants concernés

— Aucun phénomène ne peut survenir entre deux instants logiques consécutifs

— Tous les processus constituant le système disposent d'une connaissance commune (qu'on nomme Horloge) des événements survenant à chaque instant logique.

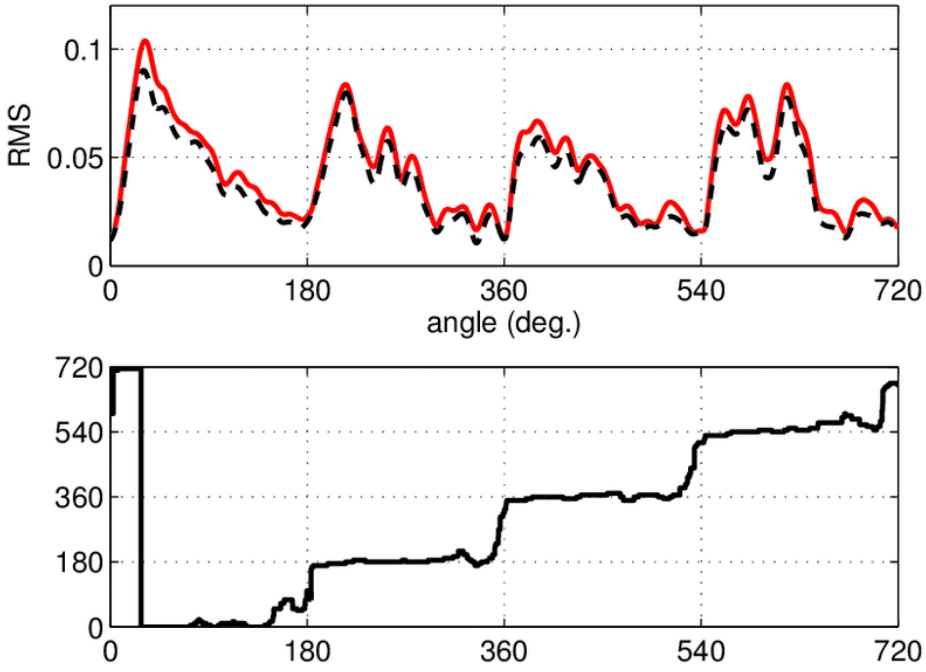

FIGURE 1.12 – Valeur RMS de la moyenne synchrone d'un signal microphonique [50]

La mise en pratique de ce modèle repose sur l'hypothèse synchrone, selon laquelle le système est assez rapide pour traiter tous les événements externes dans le bon ordre avant l'arrivée des suivants.

1.4.2 L'approche asynchrone

Face à la rigueur conceptuelle et au déterminisme de l'approche synchrone, les langages et systèmes asynchrones proposent une vision alternative de la programmation réactive. Là où le modèle synchrone privilégie la précision temporelle et la prévisibilité absolue, l'approche asynchrone favorise la flexibilité et la scalabilité, au prix d'une plus grande complexité d'analyse. [74]

Dans le paradigme asynchrone, les tâches s'exécutent indépendamment, sans la structure temporelle ordonnée qui caractérise l'approche synchrone. Les événements surviennent non pas selon un flux logique parfaitement maîtrisé, mais en fonction de facteurs variables comme la disponibilité des ressources ou les délais d'exécution. Cette flexibilité introduit une indétermination temporelle absente du modèle synchrone. Un programme asynchrone fonctionne en arrière-plan, poursuivant plusieurs opérations simultanément. Le traitement parallèle qui en résulte peut améliorer la réactivité apparente du système, mais compromet la certitude temporelle qui fait la force de l'approche synchrone. Les événements sont traités selon des délais réels et variables, rendant pratiquement impossible la vérification formelle des propriétés temporelles du système - un atout majeur de l'approche synchrone.

Pour compenser ces limitations conceptuelles, les systèmes asynchrones s'appuient sur des mécanismes spécifiques comme les callbacks, les promesses ou les coroutines. Ces constructions, nécessaires à la coordination des tâches, ajoutent une couche de complexité absente des langages synchrones où l'ordonnancement est intrinsèquement géré par le modèle.

Les systèmes asynchrones trouvent leur utilité dans certains contextes spécifiques : architectures distribuées, applications web ou environnements où la charge est imprévisible. Ils offrent une utilisation pragmatique des ressources en évitant le blocage des processus lors de tâches longues, comme les requêtes réseau ou les opérations sur base de données.

Leur principal avantage réside dans leur capacité à gérer un grand nombre de tâches concurrentes sans surcharger le système. Cependant, cette capacité s'obtient au détriment de la transparence et de la vérifiabilité qui caractérisent l'approche synchrone. L'analyse et les tests deviennent plus complexes en raison du non-déterminisme inhérent au modèle asynchrone.

Certes, les approches asynchrones offrent des outils pour optimiser la gestion des tâches en arrière-plan, ils ne peuvent égaler la simplicité conceptuelle et la rigueur formelle qui caractérisent l'approche synchrone, particulièrement précieuse dans les domaines où la correction temporelle et le déterminisme sont essentiels.

1.5 Comparaison entre les systèmes synchrones et asynchrones

Critère	Système synchrone	Système asynchrone
Définition	Tous les processus progressent en "lockstep", sous une même horloge globale [1].	Chaque processus possède sa propre horloge et avance indépendamment [1].
Exemples d'application	Multiprocesseurs centralisés, puces VLSI, visioconférences [1].	Réseaux distribués, e-mails, forums, plateformes auto-dirigées [1].
Communication	Forte coordination, communications synchrones [1].	Faible coordination, délais ou asynchronisme possibles [1].
Performance (efficacité temporelle)	Plus rapide pour les tâches nécessitant coordination stricte [1].	Nécessite au moins un facteur $\log(n)$ de temps en plus pour certains problèmes [1].
Tolérance aux retards	Moins robuste aux défaillances de synchronisation [1].	Plus tolérant aux délais variables [1].
Exigences techniques	Nécessite bon équipement audio/vidéo et connexion stable [17].	Moins exigeant techniquement [17].

TABLE 1.1 – Comparaison entre les systèmes synchrones et asynchrones

1.6 Modèles de conception des systèmes réactifs

Pour concevoir des applications critiques, les modèles de conception des systèmes réactifs offrent des approches rigoureuses permettant d'assurer la fiabilité, la réactivité et la maintenabilité des systèmes. Ces méthodes sont fondamentales pour garantir un comportement stable et prévisible, particulièrement dans des contextes où une défaillance pourrait entraîner des conséquences graves. Elles visent à concevoir des architectures capables de résister aux pannes, de réagir rapidement aux événements et d'évoluer facilement dans le temps. [33]

1.6.1 Automates déterministes

Les automates sont souvent utilisés pour implémenter le noyau de contrôle d'un système réactif [43]. Étant donné un ensemble de valeurs d'entrée, l'automate sélectionne une transition à partir de son état courant, appelle les tâches séquentielles correspondantes, puis change d'état pour sa réaction suivante.
Cette approche conduit généralement à d'excellentes performances, mesurables : une réaction est un segment linéaire de code (sans boucle, récursivité, interruption, ni surcharge liée à la gestion de processus), dont le temps d'exécution maximal peut être précisément borné. De plus, les automates sont des objets mathématiques bien connus, pour lesquels des techniques de vérification formelle existent (évaluation de formules de logique temporelle, réduction et observation).

1.6.2 Modèles basés sur les réseaux de Petri

Ces modèles sont principalement utilisés pour programmer des contrôleurs industriels [43]. Leur concurrence inhérente permet de réduire la complexité de la description du système. Toutefois, en raison du manque de hiérarchie, ils sont difficiles à appliquer à des systèmes de grande taille. De plus, leur sémantique temporelle est souvent floue ou mal définie.

1.6.3 Modèles basés sur les tâches

Dans cette approche, on conçoit un système comme un ensemble de tâches séquentielles, activées et contrôlées par un système d'exploitation temps réel. Le système est décomposé en tâches qui communiquent généralement par une mémoire partagée. [43]
Selon les auteurs [43], il s'agit d'une approche de bas niveau. Les contraintes temporelles ne sont pas directement exprimées dans la description ; elles ne peuvent être satisfaites que par des instructions de planification (interruptions, priorités, etc.) fournies au système d'exploitation. Cette méthode soulève des problèmes de portabilité, rend l'analyse du système difficile (en raison du non-déterminisme et de l'absence de vue globale), et peut dégrader les performances à cause de la gestion dynamique des tâches.

1.6.4 Processus communicants

Des langages parallèles généralistes comme Ada [43] ou Occam offrent des primitives de haut niveau pour structurer les programmes et les données. Les mécanismes de communication et de synchronisation (rendez-vous, files d'attente, etc.) sont plus propres que l'usage de mémoire partagée. Ces langages ont été conçus pour améliorer la portabilité des programmes.

Cependant, cette portabilité est obtenue au prix du non-déterminisme. Pour qu'un programme ait un comportement indépendant de l'architecture cible (mono ou multiprocesseur), les hypothèses sur la synchronisation entre processus sont réduites au minimum. Même si certains de ces langages disposent de primitives temps réel, leur sémantique reste souvent vague.

1.7 Fiabilité des systèmes réactifs

Les systèmes réactifs sont fréquemment utilisés dans des domaines critiques comme l'aéronautique, le ferroviaire, les dispositifs médicaux ou les infrastructures industrielles, où toute défaillance peut entraîner des conséquences graves, voire irréversibles. Leur fiabilité représente donc un enjeu fondamental tout au long de leur développement.

Après les phases de modélisation et de conception, l'étape de vérification et validation (V&V) devient indispensable avant toute mise en œuvre. Elle permet de s'assurer que le système respecte bien les exigences formelles définies (vérification), tout en garantissant qu'il répond effectivement aux besoins fonctionnels et opérationnels des utilisateurs finaux (validation). Cette étape est d'autant plus cruciale que les systèmes réactifs doivent souvent répondre à des contraintes strictes de synchronisation, de cohérence temporelle et de déterminisme, afin d'assurer un comportement fiable et prévisible dans tous les scénarios d'exécution.

1.7.1 Méthodes de vérification des systèmes

La vérification des systèmes prend 40% à 70% de l'effort de développement total d'un système, et il existe de nombreuses méthodes de vérification. Ces méthodes sont regroupées en deux familles : Simulation et vérification formelle.

a. Simulation

La simulation, également appelée vérification dynamique, est l'une des méthodes les plus utilisées dans le processus de validation des systèmes. Elle s'applique généralement au produit final, consistant à exécuter une description du système à un certain niveau d'abstraction sur un ensemble de scénarios types. Ainsi, la qualité des résultats obtenus dépend fortement de la qualité des tests appliqués en entrée.

Les simulations peuvent porter soit sur les fonctions globales du système, soit sur sa structure détaillée, en reproduisant les connexions et interactions entre les éléments du système réel [27].

Les simulations peuvent porter soit sur les fonctions globales du système, soit sur sa structure détaillée, en reproduisant les connexions et interactions entre les éléments du système réel [27].

Parmi les techniques de simulation les plus courantes, on retrouve notamment :

La méthode de Monte Carlo [75], utilisée depuis les années 1960, qui repose sur l'exécution répétée d'expériences avec des variables aléatoires pour estimer des probabilités de résultats.

La méthode Bootstrap [25], qui améliore la fiabilité des estimateurs statistiques, notamment sur de petits échantillons.

Ces techniques se sont largement développées depuis la fin des années 1980 et sont aujourd'hui très répandues.

La simulation permet de détecter efficacement les erreurs fonctionnelles, mais elle présente aussi plusieurs limites. Elle peut nécessiter une puissance de calcul importante, rendant son exécution coûteuse en temps, ce qui justifie les travaux de recherche visant à améliorer ses performances [40] [13]. De plus, elle ne constitue pas une preuve formelle : le système est testé sur un ensemble limité de scénarios, sans garantie de couverture complète. Les hypothèses sous-jacentes aux modèles peuvent également être incomplètes ou incorrectes.

En résumé, bien que la simulation soit une méthode essentielle de validation, elle ne peut prouver l'absence d'erreurs, seulement révéler leur présence. De plus, elle intervient souvent tard dans le cycle de conception, ce qui peut nuire à la détection précoce des défauts. Pour pallier ces limites, des approches complémentaires comme la vérification formelle permettent de vérifier les propriétés du système dès les premières étapes de développement.

b. Vérification formelle

La vérification formelle permet de réduire le temps de vérification en offrant des techniques de vérification éffectives qui s'intègrent très tôt dans le processus de conception.

Il existe principalement deux types de vérification formelle, la technique de Model-Checking et la preuve par théorèmes (Theorem Proving). Nous donnons un aperçu des deux techniques.

Model-checking

La technique de Model-checking, apparue dans les années 80 [64], est largement utilisée pour la vérification automatique des systèmes. Elle repose sur la modélisation du système par une machine à états finis, puis sur la vérification systématique de propriétés logiques via des algorithmes spécialisés. Comme le montre la figure 1.13, le processus comporte trois étapes principales : la spécification du système dans un langage formel, la traduction en un automate ou un graphe d'états, et enfin la vérification de propriétés exprimées en logique temporelle, comme LTL ou CTL. Si la propriété est vérifiée, le model-checker retourne "vrai", sinon il fournit un contre-exemple. Bien que cette technique soit exhaustive, elle peut souffrir du problème d'explosion combinatoire pour les systèmes complexes. Pour y remédier, des méthodes comme l'abstraction ou le model-checking symbolique avec des structures comme les BDD ou SAT ont été proposées. Des outils comme CADP [30] ou SPIN [37] sont couramment utilisés pour mettre en œuvre ces techniques.

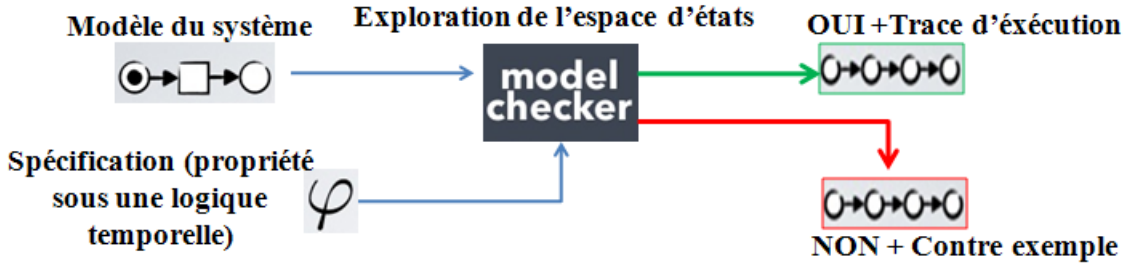

FIGURE 1.13 – Processus du model checking [52]

Preuve par théorèmes

La vérification par preuve consiste à vérifier qu'une implémentation est conforme à sa spécification, comme montré dans la figure 1.14. La spécification et l'implémentation sont décrites dans la même logique, permettant ainsi leur vérification. Ces descriptions sont souvent rédigées dans une logique de premier ou d'ordre supérieur, où le comportement du système est modélisé par des prédicats dépendant des entrées du système. Une fois cette description logique établie, un prouveur de théorèmes est utilisé pour construire semi-automatiquement des preuves mathématiques. Les prouveurs s'appuient sur des solveurs SAT et SMT pour effectuer leurs vérifications. Les solveurs SAT résolvent le problème de satisfiabilité booléenne (SAT), tandis que les solveurs SMT étendent cette approche en incluant d'autres théories du premier ordre, comme l'arithmétique, permettant ainsi la vérification de systèmes infinis. Des outils comme Z3 [59]et COQ [12] sont fréquemment utilisés pour ces vérifications formelles. Ces techniques sont particulièrement appliquées à des domaines comme la planification des ressources et la vérification des systèmes cryptographiques.

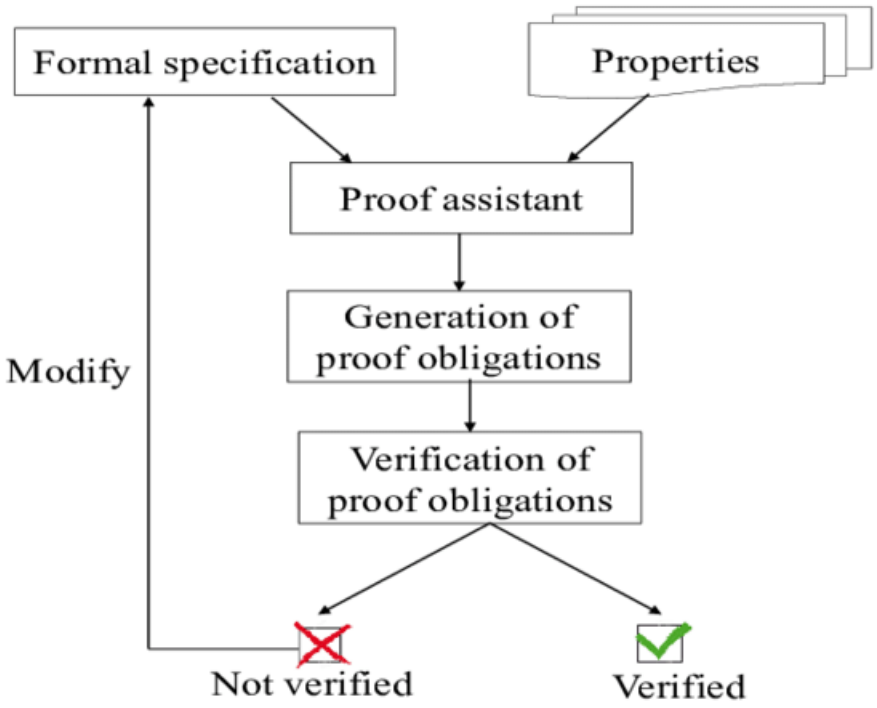

FIGURE 1.14 – Processus de la preuve par théorème
[28]

Conclusion

Dans ce chapitre, nous avons présenté le cadre conceptuel de notre étude en explorant l'univers des systèmes réactifs. Nous avons mis en lumière leurs caractéristiques, leurs domaines d'application variés, ainsi que les approches majeures qui guident leur conception et leur modélisation. Notre analyse a également porté sur les principaux modèles employés dans ce domaine, en soulignant leurs atouts respectifs et leurs limitations.

Le chapitre suivant sera consacré aux automates d'entrée/sortie (Input/Output automata), un formalisme qui s'est imposé comme une référence incontournable dans la modélisation des systèmes réactifs, offrant un cadre mathématique rigoureux pour représenter et raisonner sur ces systèmes complexes.

Chapitre 2

Etat de l'art sur les Input/Output automata

Introduction

La modélisation des systèmes réactifs est une étape essentielle dans la conception des systèmes interactifs et temps réel. Elle permet de représenter formellement le comportement d'un système face à des événements externes et de vérifier sa conformité aux exigences fonctionnelles et temporelles. Cette modélisation facilite l'analyse, la simulation et la validation des propriétés essentielles du système avant son implémentation.

Dans ce chapitre, nous allons introduire les automates d'entrées/sorties (Input/Output (I/O) Automata), modèle sur lequel se base notre travail. Nous allons tout d'abord présenter l'origine de ce formalisme, conçu pour modéliser des systèmes concurrents et distribués. Ensuite, nous décrirons le modèle de base des I/O automata, en détaillant sa structure, ses actions, et ses règles d'exécution. Nous explorerons également les principales variantes développées à partir du modèle original, telles que les Timed I/O Automata pour la modélisation des systèmes temporels, les Hybrid I/O Automata pour les systèmes mixtes discrets-continus, les Probabilistic I/O Automata pour l'incorporation d'incertitude, ainsi que les Dynamic I/O Automata qui permettent la création et la suppression dynamique de composants. Cette présentation nous permettra de poser les fondations théoriques nécessaires à la compréhension et à la construction de notre propre modèle.

2.1 Problématiques liés à la modélisation des systèmes réactifs

La modélisation des systèmes réactifs soulève plusieurs défis majeurs, liés à la nature même de ces systèmes et à leur évolution constante. Dans cette section, nous présentons les principales problématiques rencontrées :

2.1.1 Complexité croissante

Les systèmes réactifs modernes, qu'ils soient embarqués, distribués ou cyber-physiques, deviennent de plus en plus complexes. Cette complexité résulte de l'interconnexion croissante entre de nombreux composants logiciels et matériels, souvent hétérogènes, et de la nécessité de gérer de multiples flux d'interactions en parallèle. La modélisation doit ainsi être capable de représenter fidèlement non seulement le comportement interne de

chaque composant, mais aussi leurs interactions dans un environnement dynamique. Cela nécessite des abstractions précises mais suffisamment expressives pour capturer tous les aspects critiques du système sans engendrer une explosion de la taille ou de la difficulté de l'analyse.

2.1.2 Contraintes de performance

Les systèmes réactifs doivent généralement répondre à des événements externes dans des délais stricts. Par conséquent, la modélisation ne peut pas se limiter à décrire uniquement les séquences d'actions possibles, mais doit aussi prendre en compte des contraintes temporelles. Dans des applications temps réel, comme l'aéronautique ou les systèmes industriels, il est indispensable de vérifier que certaines réponses interviennent dans une fenêtre temporelle garantie. Cela complexifie la modélisation, car il faut intégrer des notions de chronométrage, d'ordonnancement, et parfois de ressources limitées (processeur, mémoire).

2.1.3 Problématiques de sécurité et sûreté (fiabilité)

Les erreurs dans les systèmes réactifs peuvent avoir des conséquences graves, notamment dans les domaines critiques tels que le médical, l'automobile ou les infrastructures de transport. Il est donc essentiel que la modélisation permette de vérifier des propriétés de sécurité (par exemple : "aucun état interdit ne peut être atteint") ainsi que des propriétés de sûreté (par exemple : "le système peut toujours atteindre un état sûr en cas de panne"). La modélisation doit aussi prendre en compte la robustesse face aux comportements inattendus, comme les défaillances partielles ou les attaques malveillantes, ce qui introduit la nécessité d'analyser non seulement les comportements normaux, mais aussi les scénarios d'exception.

2.2 Modélisation des systèmes réactifs

Dans ce travail, nous nous intéressons à la modélisation des systèmes réactifs .De nombreux modèles de composants ont été proposés pour représenter ces derniers à différents niveaux d'abstraction. Chaque modèle met l'accent sur certains aspects en fonction des objectifs visés. Il est donc essentiel de sélectionner un modèle adapté à ses besoins, qui soit à la fois précis et conforme aux propriétés du système ciblé. Parmi les modèles les plus connus,nous pouvons citer :

2.2.1 Réseaux de processus

Les réseaux de processus sont principalement utilisés pour modéliser les applications à flux de données. Chaque processus représente un programme avec ses flux d'entrée et de sortie, et la communication entre processus se fait via des canaux FIFO (First In First Out) non bornés. Cette absence de limite permet aux processus d'écrire à tout moment, mais la lecture est bloquante si le canal est vide ou en concurrence. Toutefois, ces réseaux ne garantissent pas l'exécution en temps réel et posent des problèmes de gestion mémoire. Pour y remédier, le modèle de flot de données synchrones (SDF) a été proposé , où la

quantité de données produites et consommées est déterminée statiquement. Ce modèle est à la base de langages synchrones comme LUSTRE [34] et SIGNAL.

2.2.2 Réseaux de Pétri

Les réseaux de Pétri sont un modèle mathématique destiné à la spécification des opérations dans les systèmes concurrents et temps réel. Les transitions (opérations) sont représentées par des rectangles connectés par des flèches unidirectionnelles. Ce modèle est particulièrement adapté à la modélisation de systèmes dynamiques multitâches. Plusieurs extensions ont été développées pour intégrer des aspects temporels [76]. Néanmoins, l'un des inconvénients majeurs est la complexité de la décomposition d'une application en réseaux de Pétri, qui peut s'avérer coûteuse [47].

— **Systèmes de transitions étiquetés :** Les systèmes de transitions étiquetés (LTS) sont utilisés pour décrire de manière abstraite le comportement des systèmes. Ils définissent la manière dont ces systèmes peuvent passer d'un état à un autre en utilisant des transitions associées à des étiquettes définissant les opérations accomplies durant la transition [2].

2.2.3 Les automates d'Entrée/Sortie

Un automate d'entrée/sortie (de l'anglais I/O automaton), introduit par N.A. Lynch et M.R. Tuttle [55], est un modèle mathématique connu, initialement utilisé pour modéliser les systèmes concurrents et distribués à événements discrets. Un I/O automaton est une machine à états ; à chaque réaction, elle reçoit des données en entrées depuis l'environnement et produit des sorties qui représentent les nouvelles valeurs des données consommées. L'utilisation des I/O automata a par la suite été élargie et a été utilisée afin de décrire et raisonner sur plusieurs types de systèmes. Les I/O automata temporisés [43] étendent les I/O automata avec des transitions discrètes qui représentent les événements instantanés, et des trajectoires continues qui modélisent l'évolution d'un état au fil du temps. Par la suite, il y a eu les I/O automata hybrides, qui permettent la spécification des systèmes hybrides. Les I/O automata hybrides incluent la notion de comportement externe, qui capture les interactions continues et discrètes entre le système et son environnement.

2.2.4 Les automates d'interfaces

Les automates d'interfaces modélisent le comportement d'un composant en se basant sur des hypothèses sur son environnement, utilisées pour représenter les restrictions liées, par exemple, à l'ordre d'appel des procédures. En effet, ces automates n'acceptent que certaines entrées générées par l'environnement et ne décrivent le comportement du composant que sur ces entrées. Contrairement aux automates E/S, où chaque état doit être réceptif à toute entrée possible, les automates d'interfaces, à chaque état, peuvent avoir des entrées illégales.

2.2.5 Les automates Mode (Mode-automata)

Les automates Mode [56], destinés à exprimer les structures modes des systèmes réactifs, sont des automates dont chaque état est labellisé par un programme de flot de données, qui représente un mode d'exécution du système. Les opérations sur l'automate sont tirées d'ARGOS , et les équations de flots de données sont tirées de LUSTRE [34].

2.2.6 Les langages Synchrones

L'approche synchrone a donné naissance à plusieurs langages de programmation spécifiquement conçus pour modéliser et développer des systèmes réactifs. Ces langages prennent en entrée des flots de valeurs, exécutent des calculs en fonction de transitions d'états, et produisent en sortie des flots synchronisés. Parmi les plus notables :

— ESTEREL [10] : langage à syntaxe impérative orienté sur la gestion du flux de contrôle dans les systèmes réactifs. Il permet une description précise des réactions à des événements synchrones et a été largement utilisé dans des systèmes embarqués critiques.

— LUSTRE / SCADE [34] : LUSTRE repose sur un modèle de flux de données et s'appuie sur des systèmes d'équations. SCADE (Safety Critical Application Development Environment), basé sur Lustre, est un environnement graphique destiné aux systèmes critiques, notamment dans l'aéronautique (Airbus A340, A380, etc.). Il permet la conception, vérification et optimisation des systèmes de contrôle à haute fiabilité [24].

— SIGNAL / SILDEX / POLYCHRONY [8] : SIGNAL est un langage synchrone basé sur les équations de signaux. Chaque signal possède sa propre horloge, ce qui rend le langage plus expressif que Lustre, mais également plus complexe à manipuler.

— ZELUS [15] : langage qui combine le temps logique discret des langages synchrones classiques avec le temps continu via l'intégration d'équations différentielles ordinaires (ODEs). Il étend Lustre pour modéliser des systèmes hybrides.

2.3 Approche de conception par composition

Dans la conception des systèmes réactifs, l'approche par composition joue un rôle fondamental. Elle consiste à modéliser une application comme un assemblage de composants élémentaires interconnectés. Ces composants interagissent via des canaux d'entrée et de sortie, et peuvent être organisés en série ou en parallèle pour former un système global cohérent.

Dans ce travail, nous nous intéressons particulièrement à la composition séquentielle, où les composants sont connectés en série. Cette configuration permet de raisonner plus facilement sur des propriétés globales telles que la latence totale et la capacité mémoire, calculées comme la somme des latences et capacités individuelles des composants.

Cependant, composer correctement ces éléments tout en respectant la sémantique propre aux systèmes réactifs soulève plusieurs défis, notamment en termes de synchronisation, de cohérence temporelle et de déterminisme. L'opérateur de composition issu du formalisme des I/O Automata constitue une base théorique puissante pour modéliser ces interactions, mais il ne garantit pas toujours une préservation correcte du comportement attendu dans un système pipeline, comme l'illustrent certains contre-exemples.

La conception orientée composants est issue du génie logiciel et s'appuie sur trois concepts fondamentaux. D'abord, le principe de séparation des préoccupations, né dans les années 1970 [61], qui recommande de fractionner un logiciel en modules fonctionnels cohérents et autonomes. Ensuite, les langages de description d'architecture [29], développés plus

tardivement, permettant l'analyse de structures complexes représentées comme des assemblages de composants interconnectés. Finalement, le développement par composants, introduit par le secteur industriel [67], qui définit un composant comme une unité autonome de composition et de déploiement avec des interfaces contractuelles précises. Cette méthodologie encourage la réutilisation des composants dans différents contextes applicatifs et facilite la création de systèmes sophistiqués par assemblage d'éléments préexistants.

La méthode orientée composants se distingue par plusieurs caractéristiques essentielles [69]. Chaque composant dispose d'une interface qui masque sa complexité interne tout en fournissant les points d'accès nécessaires pour interagir avec son environnement. Les composants sont autonomes, pouvant être conçus et déployés comme des entités distinctes et isolées, ainsi que substituables, ce qui permet leur remplacement sans perturber le reste du système. La séparation des préoccupations constitue un principe central : les composants prennent en charge des fonctions spécifiques et limitées pour garantir leur compatibilité et prévenir les interférences, tandis que l'interface libère l'utilisateur de la nécessité de comprendre les mécanismes internes. La composition, quant à elle, facilite l'agrégation de plusieurs composants pour former des structures plus élaborées, grâce à des interfaces clairement définies qui spécifient les opérations offertes par chaque composant.

2.3.1 Objectifs de la conception par composition

La conception par composants offre beaucoup d'avantages aux concepteurs parmi ces avantages, nous pouvons citer :

— **Réutilisabilité** : cette approche permet de faciliter l'emploi de composants développés indépendamment dans divers contextes d'utilisation.

— **Productivité accrue** : cette approche permet de réduire le temps de conception, de développement et de mise sur le marché des systèmes.

— **Flexibilité et hétérogénéité** : cette approche permet l'intégration de composants développés par différentes équipes, utilisant des technologies et langages variés.

— **Évolution et maintenabilité** : cette approche permet d'améliorer la capacité d'adaptation du produit final grâce à l'interchangeabilité des composants.

— **Fiabilité renforcée** : cette approche permet d'augmenter la robustesse des systèmes par la réutilisation de composants déjà développés et validés, réduisant ainsi les efforts de vérification.

2.3.2 Caractéristiques de la conception par composants

Nous pouvons résumer les caractéristiques principales de l'approche orientée composant par les points suivants[69] :

— **Interface :**le composant est équipé d'une couche supérieure qui rend les détails internes et l'implémentation du composant inaccessibles aux autres composants ou à l'environnement. Cette couche contient les informations nécessaires à la communication du composant avec son environnement ou avec d'autres composants.

— **Indépendance :**les composants peuvent être développés et déployés indépendamment, comme des parties isolées d'un système. Par conséquent, un composant est considéré comme une unité atomique de déploiement.

— **Interchangeabilité :**si un composant doit être mis à jour ou modifié, cela peut être fait sans altérer les autres composants du système.

— **Séparation des préoccupations :**un composant doit assurer des fonctionnalités les plus élémentaires possible afin de garantir sa compatibilité avec d'autres composants et éviter les conflits. La séparation des préoccupations signifie aussi que l'interface dispense l'utilisateur de connaître le fonctionnement interne du composant pour pouvoir l'utiliser. Elle permet également à différentes équipes de développer, en parallèle, divers composants d'une même application.

— **Composition :**les composants peuvent être assemblés avec d'autres composants pour créer des systèmes plus complexes. L'assemblage doit être réalisé à partir d'interfaces de composants bien spécifiées, décrites comme une liste d'opérations implémentées par le composant.

2.3.3 Plateformes et outils de conception basée composants

Nous présentons un ensemble de plateformes de conception orientée composants, en mettant en avant celles couramment utilisées dans les systèmes embarqués. Ces environnements ont été conçus pour faciliter la modélisation par composition tout en offrant des garanties formelles sur le comportement du système. Ils intègrent des mécanismes spécifiques permettant d'orchestrer la communication et la synchronisation entre composants. Toutefois, les I/O automata restent particulièrement adaptés pour formaliser ce type de conception, grâce à leur expressivité et à leur structure modulaire.

a. Ptolemy II

Le projet Ptolemy II [26] a été développé par le département de génie électrique et informatique de l'université de Berkeley, sous la supervision du professeur Edward Lee. L'objectif de ce projet est de proposer un ensemble complet d'outils et de documentations pour la modélisation, la simulation et la conception de systèmes concurrents, temps réel et embarqués.

Concept.

Ptolemy II repose sur un paradigme de modélisation et de conception de systèmes appelé hétérogénéité hiérarchique. Un système est dit hétérogène s'il est composé de sous-systèmes aux caractéristiques différentes, pouvant communiquer de plusieurs manières (synchrone, asynchrone, etc.).

Il existe plusieurs façons de résoudre le problème de communication entre composants hétérogènes. L'une des approches est l'approche unifiée, qui propose une sémantique commune pour la spécification d'un système complet. L'autre est l'approche hétérogène, qui combine différentes sémantiques correspondant aux modèles des sous-systèmes.

Dans [51], les auteurs détaillent l'utilisation de la hiérarchie dans le modèle Ptolemy, qui repose sur le principe selon lequel un modèle complexe peut être divisé en une arbores-

cence de sous-modèles imbriqués, où chaque niveau représente un modèle de composant particulier.

Modèle de composant.

La plateforme Ptolemy II génère un modèle représentant une architecture hiérarchique, obtenue en assemblant des modèles ou des composants (appelés acteurs), et contenant un domaine, qui décrit précisément l'interaction et l'exécution des composants et modèles contenus dans un modèle parent. Cela permet de faire interagir des composants ayant des fonctionnements différents. Ptolemy II fournit plus d'une dizaine de domaines , permettant de concevoir des systèmes hétérogènes.

Vérification.

La plateforme permet la simulation des modèles construits, mais la vérification formelle n'est pas encore intégrée.

b. AADL

Proposé par Honeywell au début des années 90 sous le nom de MetaH , initialement développé pour le domaine de l'avionique, le langage AADL (Architecture Analysis & Description Language) [29] a été standardisé par la SAE (Society of Automotive Engineers) en 2004, et une version 2.1 du standard a été publiée en 2012 par une communauté issue du domaine de l'ingénierie des véhicules. Le langage AADL concerne aujourd'hui les systèmes embarqués en général, notamment dans les domaines de la robotique, des dispositifs médicaux et de l'aéronautique. De nombreux outils développés par les milieux académique et industriel reposent sur le langage AADL.

Concept.

Le langage AADL permet la conception et l'analyse de systèmes complexes, critiques et temps réel. Il fournit une notation textuelle et graphique standardisée pour décrire les architectures matérielles et logicielles. Il permet également différentes formes d'analyses afin de déterminer le comportement et la performance du système modélisé. Il a été conçu de manière à être extensible, afin d'élargir le type d'analyses effectuées.

Modèle de composant.

Le modèle généré par les langages AADL est une architecture de composants. Un composant peut être logiciel ou matériel, et appartient à une catégorie. Il existe cinq catégories pour les composants logiciels applicatifs : les données, sous-programmes, threads, groupes de threads, et processus. Et quatre catégories pour les plateformes d'exécution : les mémoires, processeurs, périphériques, et les bus.

Un composant AADL contient une interface et un contenu (implantation). Les interfaces assurent les communications entre les composants. Ces interfaces se composent de ports de données, ports à messages, appels synchrones à des sous-programmes, et accès aux données d'autres composants. Quant au contenu, il comporte les déclarations des appels aux sous-programmes.

AADL offre aussi la possibilité d'assigner aux composants un ensemble de propriétés exprimant des aspects non fonctionnels.

Vérification.

Il existe dans la littérature un grand nombre de travaux dédiés à l'analyse et à la vérification des modèles AADL. On peut citer l'analyse d'ordonnancement intégrée dans la boîte à outils OCARINA [49]. OSATE [39], un outil largement utilisé dans la communauté AADL, permet l'analyse des flots de données. ADES [17], qui permet la simulation des systèmes AADL, est un outil limité mais permet principalement de simuler l'ordonnancement de threads.

c. PECOS

La plateforme PECOS a été proposée en 2002 , à la suite d'un projet regroupant l'Université de Bern, et des industriels allemands (ABB, FZI), et hollandais (OTI). L'environnement PECOS permet la spécification, la composition, la configuration, et le déploiement des systèmes embarqués à base de composants.

Modèle de composant.

Un composant PECOS est doté d'un contenu, des ports, et des propriétés. Le contenu représente le comportement du système et/ou l'ensemble de ses sous-composants. Les ports représentent le seul moyen de communication entre les composants et permettent le partage des données. Les propriétés d'un composant peuvent représenter des échéances, des périodicités ou des temps d'exécution au pire cas.

Comme pour Ptolemy, PECOS repose sur un modèle hiérarchique, où des composants peuvent être vus comme des super-structures pour l'exécution de leurs sous-composants. Contrairement à Ptolemy, qui sépare le comportement du modèle de l'exécution dépendant du domaine, PECOS définit les communications et les activités, mais laisse l'ordonnancement des exécutions aux composants composites, qui définissent l'ordonnancement des fils.

Vérification.

Il existe dans la littérature un grand nombre d'études reposant sur le modèle PECOS. On peut citer l'étude , qui décrit la vérification du comportement temporel des systèmes PECOS, en utilisant les réseaux de Pétri pour modéliser le comportement et l'ordonnancement du système.

d. TinyOS

TinyOS est un environnement de conception des systèmes événementiels pour les réseaux de capteurs embarqués, proposé par l'université de Berkeley. Une première version était disponible en 2000. TinyOS dispose d'une base de codes et d'utilisateurs importante, et il est souvent utilisé comme référence pour l'implantation des systèmes pour les réseaux de capteurs.

Concept.

L'environnement TinyOS repose sur le langage de programmation NesC , un dérivé du langage C pour la programmation par composants.

Modèle de composant.

Dans le langage NesC, on retrouve deux types de composants : les modules, qui représentent le comportement par du code nesC pour une ou plusieurs interfaces, et les configurations, qui représentent les sous-composants connectés entre eux par leurs interfaces et par des liaisons. Chaque type d'interface est associé à deux types de méthodes : les commandes, qui vont du client vers le serveur, et les événements, qui vont du serveur vers le client. Les événements représentent aussi les interruptions émises par le matériel.

Vérification.

NesC, étant un dérivé du C, sa compilation est monolithique et ne permet pas de compiler des parties du système indépendamment, mais uniquement l'intégralité du système. TinyOS permet la simulation des systèmes compilés, il permet aussi l'évaluation de la consommation d'énergie des nœuds en prenant en compte les caractéristiques du matériel de la plateforme d'exécution et de simulation.

TinyOS a été intégré à d'autres méthodes existantes comme Viptos, qui utilise l'interface graphique de Ptolemy II pour la conception et la simulation de systèmes TinyOS. Cette étude propose de fournir une modélisation plus précise de certaines parties des systèmes, en exploitant d'autres domaines de Ptolemy II (temps continu, dataflow, etc.). Il est aussi possible de traduire les systèmes TinyOS dans le langage BIP, ce qui permet d'effectuer des simulations et d'utiliser des méthodes d'analyse puissantes comme le Model-checking. TinyOS n'offre pas de méthode de vérification.

e. Fractal

Le modèle FRACTAL a été réalisé dans le cadre d'un consortium OBJEC-TIFWEB2 par FRANCE TELECOM R&D et l'INRIA, dans le but de proposer une définition, une configuration et une reconfiguration dynamique d'une architecture à base de composants.

Concept.

FRACTAL, destiné à l'administration de systèmes logiciels, permet la conception, l'implantation, le déploiement et la reconfiguration des systèmes. Il est défini comme un système extensible de relations entre concepts, où un composant peut être doté de plusieurs formes de contrôle. Il repose sur la séparation des aspects de conception et d'implémentation, ainsi qu'une séparation entre l'aspect fonctionnel des systèmes (interfaces, assemblage, ...) et l'aspect contrôle associé à ces composants.

Modèle de composant.

Un composant Fractal est composé de deux parties : le contenu (de l'anglais content), qui gère les aspects fonctionnels, et la membrane, qui gère tous les aspects non-fonctionnels (configuration, sécurité, ...). Plusieurs implémentations reposent sur le modèle de composant Fractal, on peut citer Julia en Java, Think en C. Fractal est aussi la base de plusieurs plateformes comme GCM , qui offre une extension du modèle de composant Fractal pour prendre en compte les applications distribuées.

Vérification.

Fractal et Think ne fournissent aucun outil ni technique d'analyse, pour la simulation ou la vérification.

f. GCM

GCM, pour Grid Component Model, a été proposé par le réseau d'excellence européen CoreGrid [7], comme une extension du modèle Fractal dédiée aux systèmes distribués, incluant des capacités de contrôle.

Concept.

La plateforme GCM permet la conception, l'implantation, l'exécution et le déploiement d'applications basées sur un composant hiérarchique et reconfigurable à grande échelle.

Modèle de composant.

Chaque composant GCM est caractérisé par un nom et un ensemble d'interfaces. Il existe deux types de composants : primitifs et composites. Un composant primitif est la vue en boîte noire d'un composant qui encapsule le code d'implémentation et fournit des fonctionnalités. Chaque composant primitif contient une membrane responsable du contrôle et de la gestion. De plus, un composant primitif peut contenir des attributs de type primitif qui sont accessibles de l'extérieur. Un composant composite englobe des sous-composants internes et est séparé en deux parties : une membrane contenant tous les sous-composants, et prenant en charge le contrôle et la gestion de l'application ; et un contenu "content", qui englobe les sous-composants qui implémentent la logique métier "Business logic".

Vérification.

Il existe une plateforme dédiée à la spécification, l'analyse, la vérification et la validation des systèmes GCM appelée VerCors [36]. La vérification sur VerCors suit le principe suivant : (1) L'utilisateur spécifie l'architecture de l'application basée sur GCM, et le comportement des composants primitifs ; (2) puis un modèle comportemental de l'application est généré par des réseaux de Petri, ou des systèmes de transitions étiquetées ; (3) ensuite, le modèle est traduit vers un langage accepté par des techniques de Model-checking, afin de vérifier la correction du modèle.

g. SOFA 2

Le projet SOFA (SOFtware Appliances) [44], initié en 1998, avait pour objectif de traiter les problèmes des applications à composants qui peuvent être déployées sur un réseau. Plus tard, en 2006, est apparu SOFA 2 [44], la plateforme qui a fait le succès de SOFA.

Concept.

SOFA 2 est un système de composants distribués basé sur des composants assemblés hiérarchiquement. Il offre un support complet pour toutes les étapes de développement et de déploiement des applications.

Modèle de composant.

Dans SOFA 2, le modèle de composant est défini par ses méta-modèles. Un répertoire stocke les informations sur le modèle de composant de l'application et gère leur visionnage ainsi que l'accès aux outils de développement et à l'environnement d'exécution.

Vérification.

SOFA 2 utilise des protocoles de comportement [62] pour spécifier formellement le comportement d'un composant, ce qui permet la vérification de la composition durant la conception de l'architecture de l'application. Elle offre également la possibilité de vérifier l'implémentation d'un composant primitif en la comparant au protocole.

h. BIP

La plateforme BIP pour Behavior, Interactions, Priority [6], a été développée par VERIMAG. Elle offre un langage et des outils supportant une modélisation rigoureuse des systèmes complexes. La plateforme BIP est largement utilisée dans le milieu académique et industriel. Pour ne citer que quelques projets : ASCENS, COMBEST, PRO3D, et les études de cas : Dala Robot, Heterogeneous Communication System.

— **Le comportement :**Le modèle comportemental de BIP contient deux catégories de composants : les composants atomiques, qui sont représentés par un comportement, des données locales, et des ports pour interagir avec d'autres composants ; et les composants composites qui sont représentés par leurs sous-composants. Le modèle comportemental étend la théorie des automates temporisés avec l'expression d'urgence [19].

— **L'interaction :**Cette couche permet la synchronisation des modèles, dans le but de contraindre les comportements de plusieurs composants atomiques. La synchronisation est spécifiée

Concept.

BIP est basé sur une théorie permettant la modélisation et le développement de systèmes temps réel hétérogènes. BIP repose sur une sémantique formelle qui décrit la composition des composants entre eux pour générer des composants complexes.

Modèle de composant.

Un composant en BIP est obtenu par la superposition de trois couches : le comportement, spécifié par deux protocoles : le rendez-vous, une forte synchronisation symétrique, et la diffusion (broadcast), une faible synchronisation asymétrique. L'échange de données entre les composants se fait à travers des ports qui relient différents composants entre eux, ainsi qu'un connecteur qui définit l'ensemble des interactions possibles.

— **La priorité :**C'est un mécanisme de contrôle qui permet de réduire l'ensemble des interactions exécutables afin de résoudre les conflits, modéliser et composer les politiques d'ordonnancement.

Vérification.

Dans [16], les auteurs étendent les modèles BIP pour l'analyse d'une classe particulière de systèmes synchrones : les systèmes synchrones ordonnancés, sans interblocage et fortement synchronisés par une action commune qui initie les actions de chaque composant. Une action est modélisée par un réseau de Pétri acyclique, étendu par un ensemble de positions initiales et finales, avec des priorités.

2.4 Introduction aux automates d'entrée/sortie (I/O Automata)

Les automates d'entrée/sortie (I/O automata) ont été introduits par Nancy Lynch et Mark Tuttle [55] au Massachusetts Institute of Technology (MIT) dans les années 1980, comme un modèle puissant pour la modélisation des systèmes concurrents et distribués à événements discrets.
Ce modèle a été conçu pour répondre aux besoins de formalisation des interactions entre plusieurs composants fonctionnant de manière asynchrone, notamment dans des domaines tels que les algorithmes d'allocation de ressources réseau, les algorithmes de communication, les bases de données concurrentes, les objets atomiques partagés et les architectures de flux de données.

Contrairement aux automates traditionnels qui calculent une fonction en se basant sur une entrée donnée avant de s'arrêter, les I/O automata sont conçus pour interagir continuellement avec leur environnement. Chaque composant d'un système est représenté par un automate qui réagit aux entrées externes et produit des sorties de manière autonome. Les actions des automates sont classifiées en trois catégories :

— **Les actions d'entrée** : qui sont imposées par l'environnement.

— **Les actions de sortie** : qui sont générées par l'automate et transmises instantanément à l'environnement.

— **Les actions internes** : qui sont invisibles pour l'environnement et servent au fonctionnement interne de l'automate.

Une caractéristique fondamentale des I/O automata est leur propriété d'input-enabled, ce qui signifie qu'ils ne peuvent pas bloquer les entrées fournies par l'environnement, contrairement à d'autres modèles comme Communicating Sequential Processes (CSP) [55] où le blocage des entrées est possible.

2.4.1 L'Orientation des I/O Automata vers l'Asynchrone

À l'origine, le modèle des automates I/O (Input/Output Automata) a été introduit dans le but de formaliser le comportement des algorithmes distribués dans des environnements asynchrones. Comme le précise l'article "Hierarchical Correctness Proofs for Distributed Algorithms" de Nancy Lynch et Mark Tuttle [54], ce modèle a été conçu spécifiquement pour surmonter les difficultés rencontrées dans la vérification des algorithmes distribués — notamment la gestion de l'interaction complexe entre processus, messages, et événements.
Le caractère asynchrone de ce modèle se reflète dans plusieurs de ses propriétés fondamentales :

— Les actions d'entrée (input) sont toujours activées, ce qui signifie qu'un composant peut recevoir une entrée à tout moment, indépendamment de son état actuel. Cela reflète le comportement d'un système piloté par les événements, typique des réseaux asynchrones.

— Une distinction claire est faite entre les actions contrôlées localement (sorties et internes) et les actions contrôlées par l'environnement externe (entrées), ce qui est essentiel pour modéliser des systèmes distribués ouverts et non bloquants.

— Les transitions sont non déterministes, ce qui permet de capturer les incertitudes inhérentes aux environnements distribués, comme les délais de transmission et la concurrence entre processus.

Contrairement à d'autres formalismes comme CSP (Communicating Sequential Processes) ou CCS (Calcul des Communications Séquencielles) [54], qui ne distinguent pas formellement les actions internes et externes, le modèle I/O Automata permet de structurer rigoureusement la communication entre composants, facilitant ainsi des preuves de correction modulaires et hiérarchiques pour les systèmes asynchrones complexes.
Enfin, ce modèle a permis de formaliser et vérifier des algorithmes distribués classiques (comme l'arbitre distribué ou les algorithmes de consensus) tout en conservant une sémantique d'automate propre, claire et adaptée à la composition hiérarchique.

2.5 Les différentes variantes des I/O automata

Les automates I/O (Input/Output Automata) sont une base formelle puissante pour la modélisation des systèmes réactifs. Plusieurs variantes ont été développées pour mieux représenter certains aspects spécifiques, notamment le temps, la concurrence, ou les comportements hybrides.

2.5.1 Timed I/O Automata (TIOA)

Face aux limites des automates I/O classiques, notamment leur incapacité à modéliser des contraintes temporelles, une première extension a été proposée avec les Timed I/O Automata (TIOA) [43]. Ce modèle introduit la notion de temps continu dans la structure des automates I/O tout en conservant leurs propriétés fondamentales comme l'input enabling (toute entrée est toujours possible) et la composition modulaire.

Les TIOA permettent à un automate d'alterner entre des transitions discrètes et des trajectoires continues, ce qui les rend adaptés à la modélisation de systèmes temps réel. Une transition discrète correspond à une réaction instantanée à un événement, tandis qu'une trajectoire continue décrit l'évolution d'un système dans le temps (ex. : position d'un train, niveau d'eau, etc.). Ces automates sont capables de raisonner sur des propriétés comme la faisabilité I/O, la réceptivité, l'absence de comportements Zeno (infinité d'actions en un temps fini), ou encore la sécurité temporelle.
Parmi les exemples présentés dans l'article associé [43], nous trouvons un canal FIFO à délai borné et un protocole de synchronisation d'horloges, illustrant la capacité du TIOA à modéliser les comportements temporels avec précision tout en respectant l'architecture I/O.

2.5.2 Hybrid I/O Automata (HIOA)

Le modèle des Hybrid I/O Automata (HIOA) marque une avancée majeure dans la formalisation des systèmes discrets-continus. Il s'agit d'une généralisation des TIOA, destinée à modéliser des systèmes hybrides complexes tels que les systèmes embarqués, les véhicules autonomes, les robots, ou encore les systèmes de contrôle en temps réel. Ces systèmes combinent des composants numériques (calculs discrets) et des composants physiques (dynamique continue), souvent soumis à des contraintes strictes de sécurité et de fiabilité.
Un HIOA est un automate à états qui intègre à la fois :

- Des variables d'état internes (modélisant l'aspect logiciel ou logique du système),
- Des variables externes d'entrée et de sortie (modélisant l'interaction continue avec l'environnement),
- Des trajectoires définissant l'évolution dans le temps des variables (souvent continues),
- Des transitions discrètes permettant la synchronisation entre composants via des actions partagées.

Ce modèle repose sur une séparation claire entre les mécanismes de communication discrets (via des actions partagées) et continus (via des variables partagées), ce qui simplifie la modélisation et la vérification. Il prend en charge la composition parallèle, la simulation, les relations d'implémentation, et garantit des propriétés de réceptivité, assurant que le passage du temps n'est jamais bloqué.[53]

2.5.3 Probabilistic I/O Automata

Pour répondre aux besoins de modélisation des systèmes distribués randomisés, notamment dans des contextes comme les protocoles cryptographiques ou les blockchains, les Probabilistic I/O Automata (PIOA) ont été introduits [66]. Ce modèle étend les automates I/O classiques en intégrant des transitions probabilistes, permettant ainsi de capturer l'incertitude et le comportement aléatoire des systèmes.

Un PIOA est défini par un ensemble d'états, une signature d'actions (entrées, sorties, internes) et un ensemble de transitions probabilistes discrètes, où chaque transition est

associée à une mesure de probabilité discrète sur les états cibles. Les PIOA conservent les propriétés fondamentales des automates I/O, telles que l'input enabling (toute action d'entrée est toujours activable) et la composition modulaire, mais ils introduisent des concepts comme les distributions de traces pour analyser le comportement externe probabiliste.

Ce modèle est particulièrement adapté à la vérification formelle de systèmes distribués randomisés, comme les algorithmes de consensus ou les protocoles de sécurité. Par exemple, les PIOA ont été utilisés pour analyser des protocoles cryptographiques [18], en modélisant les interactions entre participants comme des automates composés en parallèle, avec des transitions probabilistes reflétant les choix aléatoires ou les incertitudes environnementales.

2.5.4 Dynamic I/O Automata

Pour répondre aux besoins de modélisation des systèmes distribués dynamiques, notamment dans des environnements tels que les réseaux mobiles, les systèmes adaptatifs, ou encore les blockchains, les Dynamic I/O Automata (DIOA) ont été introduits. Ce modèle étend les automates I/O classiques en permettant une évolution dynamique de la structure du système au cours de l'exécution, capturant ainsi des comportements complexes tels que l'apparition, la disparition ou la reconfiguration des composants.

Un DIOA est défini par un ensemble d'états, une signature d'actions (entrées, sorties, internes) dépendante de l'état courant, et un ensemble de transitions pouvant inclure des actions spéciales comme la création ou la suppression d'automates, la modification de rôles ou la reconfiguration des connexions. Cette variabilité permet de changer dynamiquement les actions disponibles et donc de modéliser des interactions évolutives, la mobilité des agents ou encore la variabilité structurelle du système.

Les DIOA conservent les principes fondamentaux du modèle I/O, notamment la composition modulaire, tout en introduisant des opérateurs avancés comme la composition parallèle, le masquage d'actions, le renommage ou encore la sous-typisation comportementale via l'inclusion de traces. Ils sont également hiérarchiques : un ensemble d'automates dynamiques en interaction peut être représenté comme un unique automate composite, facilitant l'analyse à différents niveaux d'abstraction.

Ce modèle est particulièrement adapté à la vérification formelle de systèmes distribués évolutifs, comme Bitcoin ou Ethereum, où les participants rejoignent ou quittent dynamiquement le réseau, et où des sous-chaînes peuvent être créées ou supprimées. Les DIOA offrent ainsi un cadre rigoureux pour raisonner sur la correction, la compatibilité et la sécurité de ces systèmes complexes.

Ce modèle se révèle particulièrement pertinent pour des systèmes comme Bitcoin ou Ethereum, où les participants peuvent rejoindre ou quitter le réseau dynamiquement, et où de nouvelles chaînes ou sous-chaînes peuvent être créées ou supprimées. Les DIOA fournissent alors un cadre formel robuste pour analyser la correction, la compatibilité et la sécurité de tels systèmes distribués complexes[22] [4].

2.5.5 Probabilistic Dynamic I/O Automata (PDIOA)

Une extension des DIOA, les Probabilistic Dynamic I/O Automata (PDIOA), combine la dynamique des DIOA avec les transitions probabilistes des PIOA [22]. Cette variante est conçue pour modéliser des systèmes à la fois dynamiques et probabilistes, comme les blockchains , où les participants changent dynamiquement et les décisions sont souvent probabilistes.

Un PDIOA est défini par un ensemble d'états, une signature dépendante de l'état, et des transitions probabilistes discrètes. Les PDIOA étendent les opérateurs des DIOA (composition, masquage, renommage) au contexte probabiliste et introduisent des notions comme la monotonicité de l'implémentation par rapport à la création et la destruction d'automates. Ce modèle est crucial pour la vérification formelle de protocoles combinant dynamisme, probabilité et cryptographie, posant les bases pour des outils comme la secure-emulation composable dans des contextes dynamiques.

2.5.6 Interface I/O Automata

Face aux limitations des automates I/O classiques, notamment leur incapacité à capturer les interactions entre les composants logiciels de manière suffisamment modulaire et explicite, une extension a été proposée avec le modèle des Interface I/O Automata (IIOA) [46]. Ce modèle constitue une évolution du modèle d'Interface Automata d'Alfaro et Henzinger, visant à formaliser les interactions comportementales entre composants logiciels. Les Interface I/O Automata introduisent une séparation claire entre les hypothèses (assumptions) faites par un composant sur son environnement et les garanties (guarantees) qu'il offre, distinction absente dans la version initiale.

Cette extension, proposée par Larsen, Nyman et Wasowski, introduit également un langage d'interface pour les Input/Output Automata de Lynch, largement utilisé pour la modélisation de systèmes distribués asynchrones. En intégrant cette séparation explicite des hypothèses et des garanties, le modèle des IIOA offre un cadre plus rigoureux et modulaire pour la vérification de la compatibilité entre composants. Il permet également de formaliser la composition et l'affinement des interfaces, apportant ainsi une meilleure modularité et une approche plus précise pour la vérification des systèmes distribués.

Les Interface I/O Automata sont particulièrement adaptées à la modélisation de systèmes complexes, où l'interaction entre différents composants nécessite une attention particulière aux hypothèses et aux garanties, notamment dans des contextes de systèmes distribués asynchrones et de conception de logiciels composables. Cette approche permet de formaliser avec rigueur la manière dont les composants interagissent, facilitant ainsi l'analyse et la vérification de leurs comportements dans des architectures modulaire et flexible [48]

2.6 Positionnement de nos travaux

Nos travaux s'inscrivent dans le cadre de la conception sûre des systèmes réactifs. Nous proposons une approche basée sur les composants, combinée à l'utilisation des méthodes formelles.

La conception orientée composants est devenue une approche établie, qui a fait ses preuves dans un grand nombre de domaines. Nous synthétisons dans le tableau 2.1 les plateformes

les plus connues pour la conception par composants, en spécifiant quel langage est utilisé pour la modélisation des composants, ainsi que la prise en charge ou non de la simulation et des vérifications formelles.

Plateforme	**Modèle de composant**	**Simulation**	**Vérification formelle**
Ptolemy II	Langage C	Oui	Non
AADL	Langage C, Ada	Oui : OCARINA, OSATE, ADES	Non
PECOS	Langage C	Oui	Oui
TinyOS	Langage C adapté à Ptolemy et BIP	Oui : Viptos	Non
Fractal	Think : langage C	Oui	Non
GCM	Langage C	Oui : Fractal	Oui : VerCors
SOFA 2	Méta-modèles	Oui	Oui
BIP	Systèmes de transitions étiquetées	Oui	Oui
SR-modèle	Extension des I/O-automata	Oui	Oui
SR-Dep modèle	Extension des SR-modèle	Oui	Oui

TABLE 2.1 – Comparaison des plateformes de modélisation

Nous proposons une approche modulaire pour la conception et l'analyse de systèmes réactifs corrects. Contrairement aux plateformes orientées composants existantes, généralement inadaptées à l'analyse réactive, ou aux extensions spécialisées comme [16]nécessitant une expertise en langages dédiés, notre méthode permet la spécification d'applications sans accès au code source.
Notre contribution repose sur un modèle formel, le SR-modèle (modèle synchrone réactif), qui étend les I/O-automates avec une sémantique comportementale dédiée aux systèmes synchrones réactifs. Cette approche permet à la fois la spécification paramétrique des systèmes et la construction de systèmes corrects-par-construction.

Conclusion

Dans ce chapitre, nous avons présenté les différentes méthodes de modélisation des systèmes réactifs, en mettant en évidence leurs caractéristiques, leurs domaines d'application, ainsi que les approches adoptées pour leur conception. Nous avons également étudié les principaux modèles utilisés, en soulignant leurs avantages et leurs limites. Enfin, nous avons introduit les automates d'entrée/sortie (I/O Automata), un formalisme rigoureux largement utilisé pour la spécification des systèmes réactifs.

Dans le chapitre suivant, nous présenterons notre contribution à travers l'introduction des SR-modèles, une extension des I/O Automata adaptée aux systèmes réactifs synchrones. Nous y introduirons également la notion de I/O-dépendance, une nouvelle dimension permettant de mieux capturer la dynamique de ces systèmes.

Chapitre 3

Contribution I/O-Dépendance : une nouvelle dimension pour les SR-modèles

Introduction

Le chapitre précédent a présenté un état de l'art détaillé sur la modélisation des systèmes réactifs, en insistant particulièrement sur les automates d'entrée/sortie (I/O automata) et leurs variantes. Nous avons mis en évidence les limites des modèles existants face à la complexité croissante des systèmes et aux exigences de fiabilité et de performance. Nous avons également discuté des approches de conception par composition, en analysant les plateformes et outils actuels, ainsi que la manière dont notre travail se positionne par rapport à ces solutions.

Dans ce chapitre, nous introduisons notre contribution. Nous commençons par une présentation des systèmes réactifs synchrones, catégorie spécifique des systèmes réactifs qui constitue le cœur de notre démarche. Ensuite, nous détaillons le cadre de modélisation retenu, à savoir les I/O automata, qui offrent une base formelle robuste pour la spécification des interactions entre composants. Nous proposons ensuite une extension des SR-modèles (Synchronous Reactive Models), en intégrant une nouvelle dimension de dépendance d'entrées/sorties.

3.1 Systèmes synchrones réactifs

Un système réactif, aussi appelé système à base d'événements, est généralement une collection d'un grand nombre d'éléments fonctionnels distribués qui interagissent entre eux en utilisant des événements. Dans ces systèmes, certains éléments produisent des événements et d'autres en consomment. Ces systèmes interagissent avec leur environnement avec une séquence d'étapes appelée réactions. Chaque réaction est déclenchée quand un événement arrive et affecte l'état du système. Un état du système est sa condition à un moment donné, il affecte comment le système réagit aux événements.

Les principes de l'approche réactive peuvent être utilisés comme approche méthodologique pour la conception et le développement de systèmes dans un environnement distribué, où les éléments de conception sont des composants d'un système global. Ce style de

conception permet une meilleure intégration des composants autonomes et hétérogènes dans des systèmes complexes, ce qui les rend facilement évolutifs.

Dans notre travail, nous nous focalisons sur les systèmes réactifs synchrones (SR-systèmes), où une horloge est distribuée tout au long d'un circuit pour guider l'exécution. De plus, les réactions sont simultanées et instantanées à chaque clic d'une horloge globale.[41] Un SR-système interagit avec son environnement externe ou avec d'autres systèmes en échangeant des données. Il peut accepter des données en entrée, et produire des données en sortie. À chaque front d'horloge, les sorties des composants sont (conceptuellement) simultanées avec leurs entrées. Ce type de systèmes est sujet à des contraintes de réactions comme une mémoire statique limitée et un temps de réponse, c'est-à-dire que le système peut s'exécuter avec une mémoire limitée et le délai de traitement doit être délimité, et ce malgré le fait que le traitement peut continuer pour un temps illimité.

Un SR-système peut être spécifié par certaines caractéristiques comme la capacité de stockage, la réactivité et l'élasticité. Dans ce travail, nous considérons qu'un SR-système est spécifié par deux paramètres : la latence (notée L), est le délai entre l'arrivée des entrées et la première sortie correspondante à ces entrées ; la mémoire (notée M) est la capacité disponible pour le stockage des entrées du composant. Alors, le traitement d'un SR-composant est une séquence infinie d'états où chaque état est déterminé par une liste de données consommées par le composant et la durée d'attente de chaque donnée pour le traitement. Les SR-systèmes sont souvent déterministes.[11] Ce qui signifie qu'il existe au plus une réaction possible à chaque cycle d'horloge. De plus, tant qu'ils interagissent avec l'environnement, ces systèmes sont aussi réceptifs, ce qui implique que le composant ne bloque pas d'entrées depuis l'environnement, et réciproquement, l'environnement ne bloque pas de sorties depuis le composant.

Les systèmes synchrones réactifs font partie d'un grand nombre de domaines comme les processus de contrôle industriels, les systèmes embarqués, pilotes de périphériques ou traitement de signal. Pour une analyse formelle des systèmes réactifs, plusieurs modèles sémantiques ont été proposés et appliqués à différents groupes de recherche. On peut citer les réseaux de Petri et leurs variantes, qui sont des formalismes classiques, ils sont utilisés pour la modélisation des systèmes réactifs(Ex : [16], [73]). De base, la sémantique des réseaux de Petri n'est pas intrinsèquement compositionnelle. Cependant, il existe des études qui abordent la modélisation compositionnelle des systèmes réactifs, par exemple dans [71], où les auteurs proposent des modèles basés sur les réseaux de Petri pour modéliser des systèmes réactifs qui peuvent interagir entre eux d'une manière asynchrone. La correction de tout le système est prouvée via des transformations structurelles. Par ailleurs, il existe aussi des modèles sémantiques basés sur les automates d'entrée/sortie (de l'anglais I/O automata) pour ce genre de systèmes[31], mais aucune de ces études citées n'est dédiée à la vérification compositionnelle des systèmes réactifs synchrones.

3.2 Modélisation

Pour pouvoir intégrer et utiliser des techniques de vérification formelle dans une approche de conception des systèmes, il est nécessaire d'exprimer ces systèmes dans un formalisme permettant l'analyse formelle. Dans cette section, nous introduisons les modèles choisis pour décrire de manière naturelle le comportement d'un SR-composant (Synchronous reactive). Ces modèles sont utilisés pour vérifier, par model-checking, la satisfaction

de propriétés fonctionnelles. Aussi, ces modèles sont adaptés pour une conception de systèmes basée sur une approche compositionnelle, ils peuvent être combinés pour former des modèles complexes, cohérents et complets.

3.2.1 Abstraction des données

Dans ce travail, nous proposons un formalisme de haut niveau pour modéliser les systèmes synchrones réactifs à un niveau transactionnel, à savoir que nous faisons abstraction des valeurs réelles des données, en nous intéressant uniquement aux événements de transfert de données. Cette modélisation est adaptée pour les systèmes de traitement de signal où les traitements des flux de données ne dépendent pas de leurs valeurs. Elle est aussi adaptée pour les systèmes où le flux d'entrées contrôle les traitements réalisés.

Dans notre démarche, les composants sont vus comme des boîtes noires. Un modèle en boîte noire permet d'appréhender un objet sans connaître nécessairement sa constitution interne, le composant n'est représenté que par ses interfaces d'entrées et sorties (voir figure 3.1). Un composant est placé dans un environnement qui peut lui fournir des données en entrée, et accepter les données proposées en sortie par le composant ; ce dernier peut accepter ou non des entrées selon son état interne.

FIGURE 3.1 – Une vue en boite noire d'un composant

Nous représentons les systèmes synchrones réactifs à l'aide de deux caractéristiques fondamentales : la latence (L) et la mémoire (M). À cela s'ajoute une fonction de dépendance des entrées/sorties, qui permet de formaliser plus précisément les relations entre les événements d'entrée et les réactions du système.

Ces paramètres nous permettent de décrire le comportement du composant, puisqu'ils permettent de contrôler les flux d'entrées et de sorties.

Dans la section suivante, nous allons présenter le formalisme proposé pour la modélisation des systèmes synchrones réactifs.

3.2.2 Paramètres des SR-Systèmes

Afin de modéliser avec précision le comportement d'un SR-composant, il est essentiel de s'intéresser à sa sémantique comportementale. Celle-ci décrit les différentes situations

dans lesquelles le composant peut se trouver au cours de son exécution, en fonction de ses interactions avec l'environnement et de son propre état interne. Pour cela, certains paramètres fondamentaux doivent être définis afin de caractériser la manière dont le composant traite les données et gère son évolution dans le temps. Dans le cadre de ce travail, nous nous intéressons à trois paramètres qui sont essentiels dans les systèmes réactifs.

a. Latence (L)

Représente la durée minimale qu'une donnée doit passer dans le composant avant d'être éligible à la sortie. Autrement dit, une fois qu'une donnée est introduite, elle doit séjourner dans le système pendant au moins L unités de temps (ou cycles d'horloge) avant de pouvoir être extraite. Cette contrainte reflète le délai de traitement nécessaire au fonctionnement du composant, et garantit que chaque donnée est maintenue pendant une durée suffisante pour que les opérations internes puissent s'effectuer correctement.

b. Mémoire (M)

Désigne le nombre maximal de données que le composant est capable de stocker simultanément. Tant que la capacité mémoire n'est pas entièrement utilisée, le composant peut continuer à accepter de nouvelles entrées. En revanche, lorsque cette limite est atteinte, il cesse temporairement d'accepter d'autres données jusqu'à ce qu'une ou plusieurs d'entre elles soient traitées et libèrent de l'espace. La mémoire joue ainsi un rôle déterminant dans la régulation du flux d'informations et conditionne la disponibilité du composant à recevoir de nouvelles entrées.

c. Dépendance des données

Décrit la relation entre les données d'entrée et les données de sortie du composant. Ce paramètre permet d'indiquer quelles sorties dépendent directement de quelles entrées, en tenant compte du contexte temporel et de l'état interne du composant. La fonction de dépendance permet ainsi de formaliser le lien de causalité entre événements d'entrée et de sortie, et constitue un outil clé pour raisonner sur la validité, la prévisibilité et la traçabilité du comportement du système réactif.

3.3 SR-modèles(Synchronous reactive models)

Dans le cadre de la modélisation formelle des composants matériels, nous avons opté pour une variante des I/O automata : les SR-modèles (Synchronous Reactive Models), adaptés à la représentation des systèmes synchrones à comportement réactif. Ce choix repose sur leur adéquation avec la problématique centrale de l'article de Baclet et Pacalet [5], à savoir la simulation précise et efficace des composants dans des architectures complexes de type SoC (System-on-Chip) et plus généralement des systèmes réactifs.

3.3.1 Choix des SR-modèles

Les SR-modèles [21] se basent sur les I/O-automata, formalisme dérivé des automates finis déterministes, pour représenter les séquences d'entrées/sorties des composants.

Les I/O-automata présentent un grand inérêt pour notre approche, d'une part, c'est un formalisme parfaitement adapté pour représenter la sémantique d'un système réactif, d'autre part, il supporte le principe de compositionnalité : la théorie des automates propose des mécanismes maitrisés pour combiner des automates.

Les objectifs principaux de ce type de modélisation sont :

— Fournir un modèle précis des échanges d'entrées/sorties pour vérifier l'équivalence de deux implémentations.

— Générer automatiquement des programmes de simulation efficaces en temps de calcul et en occupation mémoire

— Proposer une base solide pour la validation formelle par Model Checking ou Theorem Proving.

Les SR-modèles permettent de modéliser le comportement synchrone des composants : chaque événement est aligné sur un signal d'horloge global (the rising edge of a clock), ce qui est une contrainte centrale des composants réactifs synchrones. Ils introduisent une abstraction où les valeurs de données sont ignorées, ne conservant que les interactions d'entrées/sorties entre les composants qui représente le comportement du système.

3.3.2 Définition d'un SR-modèle

Les SR-modèles [21] étendent les I/O-automata avec des paramètres de mémoire(M) et de latence (L). En incluant ces paramètres, ces modèles permettent de représenter le comportement des systèmes réactifs synchrones.

Le comportement d'un SR-composant peut être modélisé à travers l'ensemble des états dans lesquels il est susceptible de se trouver. Ces états sont régis par trois règles fondamentales :

— **Règle d'acceptation des données :** Le composant peut recevoir de nouvelles données de son environnement externe uniquement si sa capacité de stockage n'est pas saturée, c'est-à-dire tant qu'il dispose d'un espace mémoire suffisant.

— **Règle de latence minimale :** Chaque donnée en entrée doit respecter un temps de séjour minimal correspondant à la latence du système avant que la sortie associée puisse être générée.

— **Règle de contrôle de sortie :** Lorsqu'une donnée a terminé son traitement et est prête à être transmise, le composant dispose d'une flexibilité décisionnelle lui permettant soit de procéder immédiatement à sa sortie, soit de différer cette transmission si l'environnement n'est pas prêt à accepter la donnée.

3.3.3 Sémantique des SR-modèles

La sémantique comportementale d'un SR-composant définit formellement les interactions entre le composant et son environnement. Cette sémantique s'articule autour de la gestion temporelle des flux de données et du contrôle des ressources mémoire disponibles.

Le comportement repose sur les deux paramètres constants : la mémoire M qui représente le nombre maximum d'entrées stockables dans le composant, et la latence L qui correspond au temps nécessaire pour traiter une entrée. Chaque état du composant est représenté par une liste ordonnée d'éléments entiers de longueur maximale M, où chaque élément code la durée depuis l'entrée d'une donnée. Ces éléments sont inférieurs ou égaux à L puisqu'il n'est pas nécessaire de mesurer le temps d'attente au-delà de la latence.

Le modèle utilise plusieurs fonctions pour analyser le comportement, notamment la fonction m(s) qui calcule le nombre de données stockées dans un état en comptant les entrées moins les sorties, et la fonction lfst(s) qui détermine l'âge de la donnée la plus ancienne. Le nombre total d'états possibles est calculé par une fonction combinatoire qui tient compte des contraintes imposées par les paramètres M et L. Cette approche permet de modéliser précisément le comportement temporel et la gestion mémoire des SR-composants dans un cadre formel basé sur les I/O-automates.

Définition 1 (SR-modèle) *Un SR-composant C ayant une mémoire M et une latence L est modélisé par un* ***SR-modèle*** *[21], qui est défini par un automate*

$$A = (S, s_0, \Sigma, \rightarrow)$$

où :

— *S est un ensemble fini d'états,*

— *s_0 est l'état initial,*

— *$\rightarrow$ est une fonction de transition :*

$$\rightarrow : S \times \Sigma \times S$$

où $\Sigma = \{(i, o), (i, \bar{o}), (\bar{i}, o), (\bar{i}, \bar{o})\}$ représente l'alphabet des événements synchrones.

Informellement, un SR-modèle est un I/O-automate contraint par le nombre d'états atteignables et l'ensemble de transitions qui peuvent être exécutées à chaque état. l'ensemble Σ contient les événements d'entrée/sortie. Ces événements sont testés pour leur présence i et o ou leur absence $\bar{\imath}$ et $\bar{o}$.

Intuitivement :

— i/o : signifie qu'une entrée et une sortie se produisent simultanément ;

— i/$\bar{\text{o}}$ signifie qu'une entrée se produit, mais aucune sortie ;

— $\bar{\text{ı}}$/o signifie qu'il n'y a pas d'entrée mais qu'une sortie est produite ;

— $\bar{\text{ı}}$/$\bar{\text{o}}$ signifie qu'il n'y a ni entrée ni sortie. Nous considérons que ces événements sont instantanés et qu'ils sont exécutés à chaque cycle d'horloge.

3.3.4 Représentation des états d'un SR-modèle

L'implémentation d'un SR-modèle repose sur un algorithme [20] qui représente les états d'un SR-système par des listes d'entiers où chaque état est une liste $s = \langle i \rangle^{\,i \in [1..L]}$, dont chaque élément i représente le temps d'attente de la donnée correspondante.
Le processus débute avec une liste vide $\langle\rangle$ et évolue selon les règles suivantes :

— A chaque nouvelle entrée, un élément de valeur 1 s'ajoute à la liste pour l'étendre.

— A chaque sortie, l'élément le plus ancien dont la valeur atteint ou dépasse le seuil L est retiré de la liste.

— Le passage d'un état à un autre s'accompagne d'une incrémentation systématique (+1) de tous les éléments présents dans l'état source, cette modification étant répercutée sur l'état cible. Cette mécanisme traduit la progression temporelle durant le fonctionnement du composant.

— Pour éviter la génération infinie d'états, le temps d'attente d'une entrée est plafonné à la valeur de latence L par convention. L'incrémentation ne s'applique qu'aux éléments n'ayant pas encore atteint cette valeur limite.

3.3.5 Exemples SR-modèles

a. Exemple 1 : $M = 1$, $L = 1$

Prenons un composant ayant une mémoire $M = 1$ et une latence $L = 1$. Le SR-modèle représentant ce composant est défini dans la figure 3.2.

Ce composant peut retenir au plus une seule donnée, et chaque donnée doit être traitée en un cycle. Dans ce cas, les états du SR-modèle sont représentés par des listes contenant au plus un seul entier, correspondant au temps d'attente de la donnée.

— L'état initial est la liste vide $\langle\rangle$, indiquant qu'aucune donnée n'est en mémoire.

— Lorsqu'une entrée est reçue sans production de sortie (i/$\bar{o}$), on passe à l'état $\langle 1\rangle$, signifiant qu'une donnée vient d'être introduite et attend 1 cycle. Les transitions possibles sont :

- Si une sortie est produite (o), la donnée est extraite et l'état redevient $\langle\rangle$.
- Si aucune entrée ni sortie n'a lieu ($\bar{\text{i}}$/$\bar{\text{o}}$), on reste dans le même état avec une éventuelle incrémentation du temps d'attente (ici non applicable car L = 1).

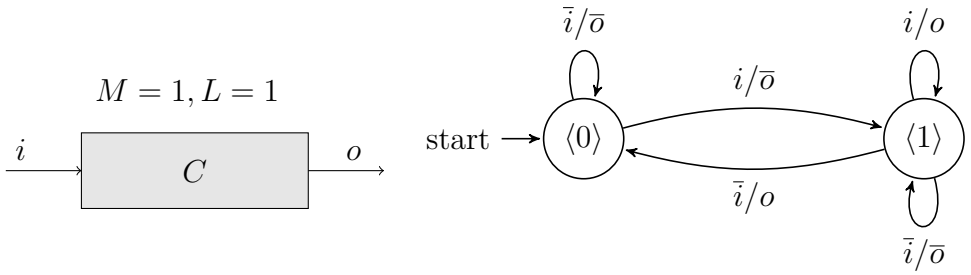

FIGURE 3.2 – SR-modèle du composant avec $M = 1$, $L = 1$

b. Exemple 2 : $M = 2$, $L = 2$

Prenons un composant ayant une mémoire $M = 2$ et une latence $L = 2$. Le SR-modèle représentant ce composant est défini dans la figure 3.3.

— L'état initial, noté $\langle\rangle$, correspond à un composant vide, depuis cet état, une entrée sans sortie $(i, \bar{o})$ fait passer le composant à l'état $\langle 1\rangle$, indiquant qu'une donnée vient d'être reçue et attend depuis un cycle. Si aucune entrée ni sortie n'a lieu $(\bar{i}, \bar{o})$, le composant reste dans l'état vide.

— Depuis l'état $\langle 1 \rangle$, deux transitions sont possibles : soit une entrée i sans sortie $(i, \bar{o})$, qui mène à l'état $\langle 2\,;1 \rangle$, soit aucune entrée et aucune sortie $(\bar{i}, \bar{o})$, menant à l'état $\langle 2 \rangle$ (calcul du temps de traitement).

— Depuis l'état $\langle 2\,;1 \rangle$, trois transitions existent : entrée avec sortie (i, o) restant dans le même état, aucune entrée mais sortie $(\bar{i}, o)$ qui ramène à l'état $\langle 2 \rangle$, et aucune entrée ni sortie $(\bar{i}, \bar{o})$ restant dans $\langle 2\,;1 \rangle$.

— Depuis l'état $\langle 2 \rangle$, quatre transitions sont possibles : une entrée avec sortie simultanée (i, o) qui revient à l'état $\langle 1 \rangle$, une entrée sans sortie $(i, \bar{o})$ menant à l'état $\langle 2\,;1 \rangle$, aucune entrée mais sortie $(\bar{i}, o)$ vers un l'état vide, et aucune entrée ni sortie $(\bar{i}, \bar{o})$ restant dans l'état 2.

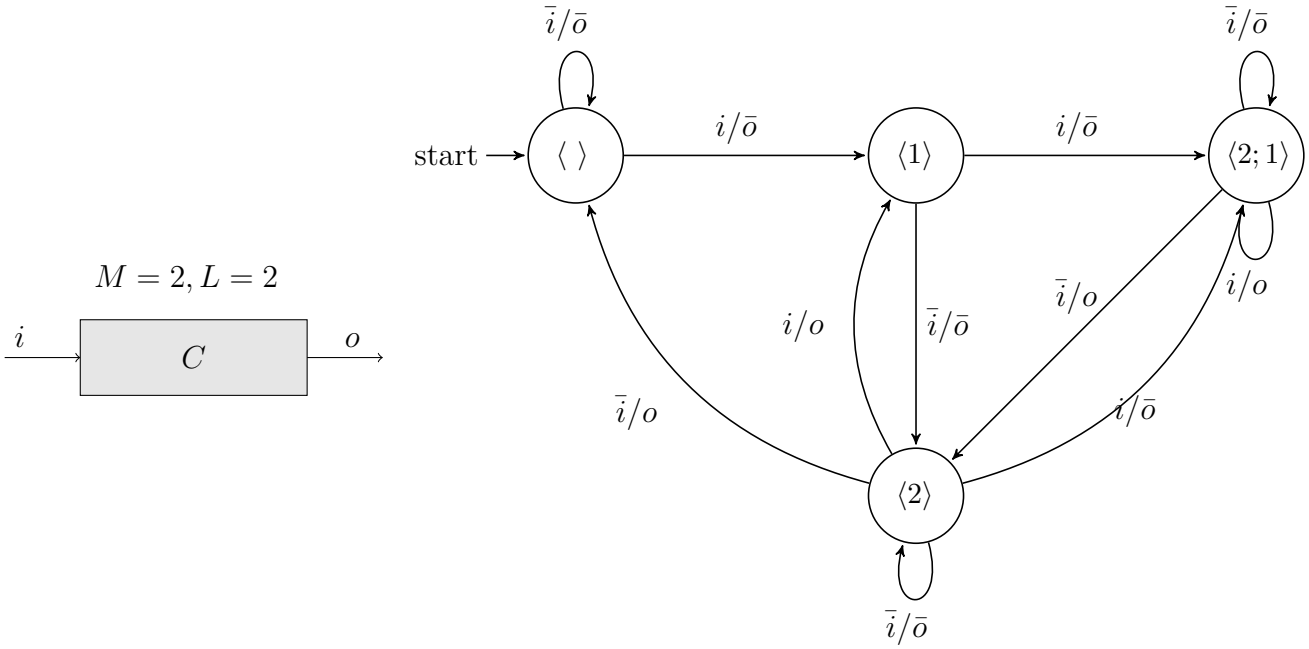

FIGURE 3.3 – SR-modèle du composant avec $M = 2$, $L = 2$

3.4 Problématique

Les SR-modèles, fondés sur les I/O-automates, fournissent un cadre formel robuste pour l'analyse des systèmes réactifs synchrones ainsi qu'une base fiable pour assurer la composition correcte de leurs composants. Ce modèle intègre deux paramètres essentiels, la capacité de stockage M et la latence L, qui permettent de maîtriser le flux des entrées et des sorties.

Cependant, bien que la mémoire et la latence soient des paramètres fondamentaux pour décrire le comportement des SR-systèmes, cette approche reste limitée à une catégorie restreinte de systèmes réactifs. En effet, les SR-modèles reposent sur l'hypothèse que chaque sortie dépend exclusivement d'une seule entrée, ce qui ne reflète pas toujours la complexité des dépendances réelles entre entrées et sorties.

La problématique abordée dans ce mémoire consiste donc à enrichir les SR-modèles existants en y intégrant un nouveau paramètre de dépendance des données. L'objectif est de proposer une méthode qui intègre cette dimension supplémentaire dans la modélisation

des systèmes réactifs synchrones, afin d'améliorer la représentation du comportement des composants SR tout en garantissant leur sûreté et la correction de leur composition.

Cette extension vise à offrir un cadre de modélisation plus complet, capable de mieux refléter les interactions complexes entre les entrées et les sorties dans les systèmes réactifs.

3.5 Proposition

Notre proposition vise à enrichir les SR-modèles classiques [21], qui se basent uniquement sur les paramètres de mémoire et de latence, en y introduisant la notion de dépendance de données. Plutôt que de modifier la structure de base des SR-modèles, nous proposons d'ajouter des règles de dépendance lors de leur implémentation. Ces règles permettent de capturer les contraintes d'ordre spécifiques aux dépendances entre données, reflétant ainsi des comportements plus fidèles à la réalité qui ne sont pas pris en compte par l'abstraction basée sur M et L. Cette approche a pour but d'améliorer la fidélité de la modélisation, en couvrant une classe plus large de SR-systèmes, tout en garantissant la compatibilité avec les outils et méthodes existants.

Pour intégrer les dépendances de données dans notre modèle, nous introduisons des règles basées sur une fonction $D(I, O) = (Id, Od)$ où Id et Od sont initialisés, qui associe certaines entrées aux sorties qui en dépendent. Ces règles permettent d'exprimer des contraintes d'ordre liées aux relations causales entre événements, enrichissant ainsi la modélisation sans modifier la structure des SR-modèles. La figure 3.4 illustre ce principe de manière schématique.

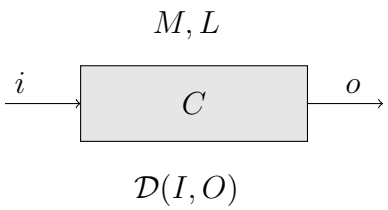

FIGURE 3.4 – Un SR-composant avec Dépendance d'entrées/sorties

3.6 Contribution : SR-Dep modèle (Synchrounous reactive dependancy model)

Le modèle SR-Dep étend les SR-modèles classiques en ajoutant deux propriétés de dépendance essentielles. La première tient compte de la mémoire actuelle pour déterminer si une entrée peut être traitée immédiatement ou doit être différée via une boucle silencieuse. La seconde règle limite l'activation des transitions en fonction du nombre de sorties déjà engagées, empêchant certaines interactions tant qu'un seuil n'est pas atteint. Ces mécanismes permettent de mieux représenter les contraintes de dépendance dans les systèmes réactifs.

Définition 2 (SR-Dep Modèle) *Un SR-composant C ayant une mémoire M, une latence L et une fonction de dépendance $\mathcal{D}(I,O)$ est modélisé par un* ***SR-Dep modèle****, qui est défini par un automate*

$$A = (S, s_0, \Sigma, \rightarrow)$$

où :

- *S est un ensemble fini d'états,*
- *s_0 est l'état initial,*
- *$\Sigma = \{(i,o),(i,\bar{o}),(\bar{i},o),(\bar{i},\bar{o})\}$ représente l'alphabet des évènements synchrones.*
- *$\rightarrow$: la relation de transition minimale que doit respecter chaque état atteignable.*
- *$\rightarrow$ repose sur des conditions de production qui assurent le respects des contraintes liées à la dépendance des données : $\forall s \in S.\ Conforme(s, M, L, D(I,O))$*

3.6.1 Propriétés de dépendance des entrées/sorties

Nous allons maintenant définir la sémantique comportementale d'un SR-Dep modèle afin de nous assurer que tous les états sont conformes $\forall s \in S.\ Conforme(s, M, L, D(I,O))$.
...D'abord vous dites que les SR-Dep modèles respectent les propriétés d'un SR-modèle classique (Déterminisime et receptivité) vous définissez en une phrase chaque notion (vous trouverez sur la thèse).
ET vous dites en plus nous introduisons ces nouvelles propriétés qui sont propres a la dépendance)

Propriété 1 (Input dependency)

$$\forall s \in S,\ m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)$$

Ajoutez : Cette propriété signifie que le SR-système doit rester en attente (Une boucle sur le même état) tant qu'il n'a pas reçu toutes les données nécessaires au traitement, ce nombre correspond au paramètre Id

Propriété 2 (Output dependency)

$$\forall s \in S,\ \exists s' \in s,\ s \xrightarrow{\bar{i}/o} s' \wedge nombre\_O(s) < Od - 1 \Rightarrow \nexists\ s'',\ s \xrightarrow{i/\alpha} s'',\ où\ \alpha \in \{o, \bar{o}\}$$

- nombre_O(s) : désigne le nombre de sorties qu'il y a eu avant l'état s.

Cette propriété signifie que tant que le nombre de données produites n'a pas atteint le paramètre Od, on ne permet pas au composant d'accepter une entrée afin de respecter la dépendance entre les entrées et les sorties.

3.6.2 Exemples de SR-Dep Modèles

a. Exemple 1 : $M = 2$, $L = 2$, $Id = 2$, $Od = 1$

Considérons un composant ayant une mémoire $M = 2$, une latence $L = 2$, $Id = 2$ et $Od = 1$ (Figure 3.5). Le SR-Dep modèle correspondant est représenté dans la figure 3.6.

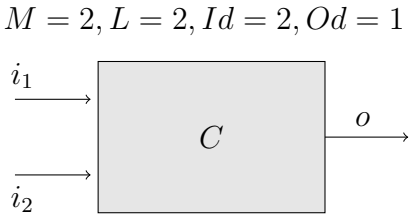

FIGURE 3.5 – SR- Composant avec deux entrées (i_1, i_2) et une sortie (o)

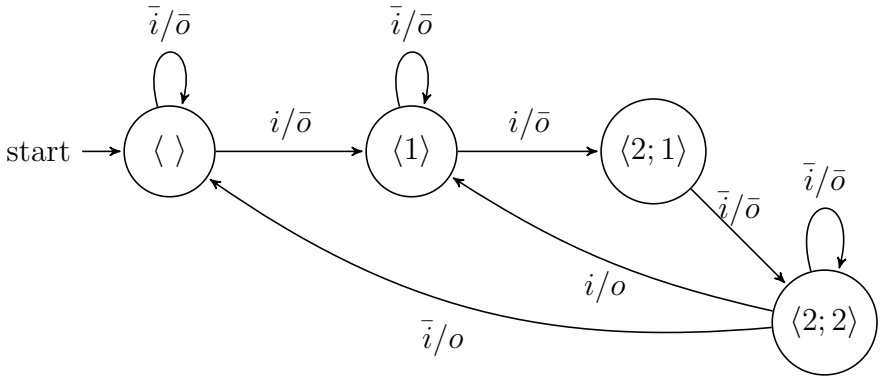

FIGURE 3.6 – SR-Dep modèle pour $M = 2$, $L = 2$, $Id = 2$, $Od = 1$

— L'état initial, $\langle\ \rangle$, représente un composant vide, sans entrée ni sortie en attente. Deux transitions en partent :
  — Une boucle silencieuse $(\bar{i}, \bar{o})$: aucune entrée ni sortie, le composant reste vide.
  — Une entrée seule $(i, \bar{o})$: une donnée est reçue sans produire de sortie, ce qui mène à l'état $\langle 1 \rangle$.

— Depuis l'état $\langle 1 \rangle$, où une première entrée est stockée :
  — Une boucle silencieuse $(\bar{i}, \bar{o})$ maintient l'état inchangé car il n'a pas reçu toutes les données $m(\langle 1 \rangle < Id$(Propriété 1)
  — Une seconde entrée $(i, \bar{o})$ remplit complètement la mémoire (M=2) et mène à l'état $\langle 2; 1 \rangle$.

— Depuis l'état $\langle 2; 1 \rangle$, le composant est plein, la première donnée à atteint la latence :
  — Une transition silencieuse $(\bar{i}, \bar{o})$ fait passer à l'état $\langle 2; 2 \rangle$, Vu que Od=1, les deux entrées doivent avoir atteint la latence avant de produire une sortie, le composant continue le calcul.

— Depuis l'état $\langle 2; 2 \rangle$, trois transitions sont possibles :
  — Une transition (i, o) : une nouvelle entrée est reçue et une sortie est émise simultanément, Ce qui amène le composant à l'état 1 car Id=2 et Od=1, une sortie vide le composant de toutes les données utilisées dans le traitement.

— Une sortie sans entrée $(\bar{i}, o)$: on libère les deux données de la mémoire, et l'on retourne à l'état initial $\langle\ \rangle$.

— Une boucle silencieuse $(\bar{i}, \bar{o})$: aucune entrée ni sortie, le composant reste dans l'état $\langle 2; 2\rangle$.

b. Exemple 2 : $M = 2$, $L = 2$, $Id = 2$, $Od = 2$

Considérons un composant ayant une mémoire $M = 2$, une latence $L = 2$, $Id = 2$ et $Od = 2$ (Voir la figure 3.7). Le SR-Dep modèle correspondant est représenté dans la figure 3.8.

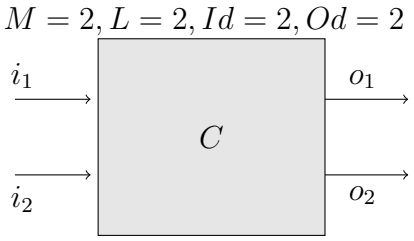

FIGURE 3.7 – Composant SR-Dep avec deux entrées (i_1, i_2) et deux sorties (o_1, o_2)

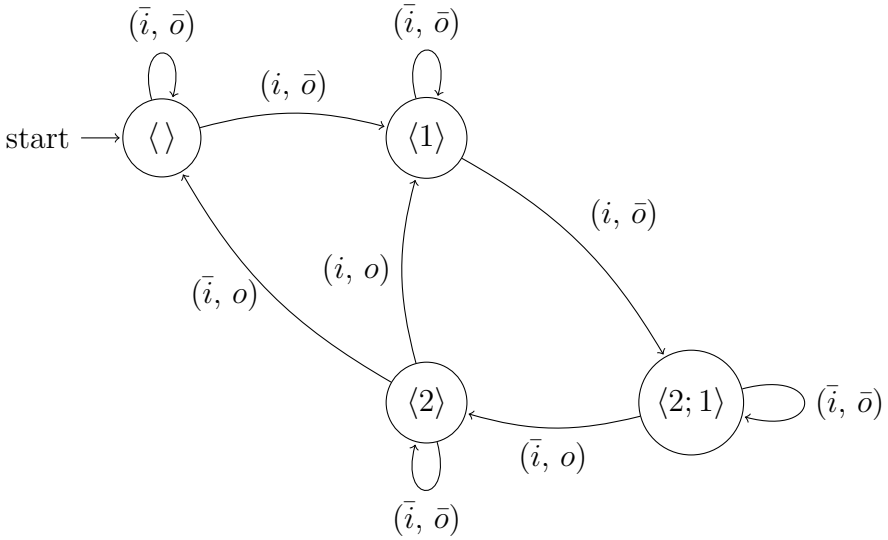

FIGURE 3.8 – Automate SR-Dep pour $M = 2$, $L = 2$, $Id = 2$, $Od = 2$

— L'état initial, $\langle\ \rangle$, représente un composant vide. Deux transitions en partent :
  — Une boucle silencieuse $(\bar{i}, \bar{o})$: aucune entrée ni sortie, le composant reste vide.
  — Une entrée $(i, \bar{o})$: une donnée est reçue sans produire de sortie, ce qui mène à l'état $\langle 1\rangle$.

— Depuis l'état $\langle 1\rangle$, deux transitions sont possibles :
  — Une boucle silencieuse $(\bar{i}, \bar{o})$ maintient l'état inchangé accord avec la propriété 1 :$m(1) < Id$.
  — Une seconde entrée $(i, \bar{o})$ remplit complètement la mémoire (M=2) et mène à l'état $\langle 2; 1\rangle$.

— Depuis l'état $\langle 2;1\rangle$, le composant est plein :
  — Depuis l'état $\langle 2;1\rangle$ Une transition silencieuse $(\bar{i},\bar{o})$ permet de rester dans le même état.

  — Une transition $(\bar{i},o)$ permet de rejoindre l'état $\langle 2\rangle$ en produisant une sortie. Aucune entrée n'est permise conformément à la propriété 2, car $nombre\_O(\langle 2;1\rangle) < Od-1=1$.

  — Depuis l'état $\langle 2\rangle$, trois transitions sont possibles :
    — Une boucle silencieuse $(\bar{i},\bar{o})$: aucune activité n'a lieu, le système reste dans l'état $\langle 2\rangle$.

    — Une transition (i,o) : une entrée est reçue et une sortie est produite simultanément, ce qui ramène le système à l'état $\langle 1\rangle$.

    — Une transition $(\bar{i},o)$: une sortie est produite sans nouvelle entrée, ce qui vide totalement le composant et le ramène à l'état initial $\langle\ \rangle$.

Concrètement : Ces deux exemples illustrent comment des modèles ayant des paramètres identiques de mémoire et de latence peuvent générer des comportements distincts selon leur fonction de dépendance entrées/sorties. L'introduction de contraintes de dépendance différentes produit ainsi des SR-Dep modèles aux structures différentes. L'analyse comparative de l'état $\langle 2;1\rangle$ dans les deux modèles (Figure 3.6 et Figure 3.8) le démontre clairement :
— Dans le modèle de la figure 3.6 avec la dépendance Id=2 Od=1, il n'y a qu'une transition d'attente $(\bar{i},\bar{o})$car la deuxième donnée doit atteindre la latence.

— Alors que dans le modèle de la figure 3.8 avec la dépendance Id=2, Od=2. Il produit une sortie sans accepter d'entées $\bar{i}/o$ car il produira les sorties sans accepter d'entrées tant qu'il n'a pas atteint $Od-1$ sorties.

Cette différence montre que l'introduction des propriétés liées à la dépendance des entrées/sorties permet de représenter le comportement d'une classe plus large de composants synchrones réactifs.

Conclusion

Dans ce chapitre, nous avons introduit les SR-modèles, adaptés à la modélisation des systèmes réactifs synchrones, en utilisant les paramètres de mémoire et de latence. Après avoir identifié leurs limites, notamment l'absence de prise en compte des dépendances entre entrées et sorties, nous avons proposé une extension : le SR-Dep modèle. Ce modèle intègre une dimension de dépendance I/O permettant de mieux représenter le comportement réel des composants, tout en restant compatible avec la sémantique comportementale des SR-modèles.

Dans le chapitre suivant, nous mettrons en œuvre cette proposition en appliquant les SR-Dep modèles à un cas concret : la modélisation du composant IDCT (Inverse Discrete Cosine Transform). Nous présenterons l'implémentation de cette approche, ainsi que les résultats issus de cette étude de cas, afin d'évaluer sa pertinence et sa capacité à refléter fidèlement le comportement d'un composant réel.

Chapitre 4

Évaluation du SR-Dep modèle par étude de cas : IDCT component

Introduction

Après avoir présenté dans le chapitre précédent notre proposition d'extension des SR-modèles par l'ajout de la dépendance I/O, nous allons démontrer à travers une étude de cas concrète : Le composant IDCT (Inverse Discrete Cosine Transform), la pertinence et l'efficacité de notre extension dans la modélisation des systèmes réactifs synchrones.

Nous débuterons ce chapitre par présenter le cadre de l'implémentation de notre approche de modélisation dans un outil en Python. Puis nous introduirons l'architecture du composant choisi : IDCT, ainsi que les caractéristiques fonctionnelles de ses différents sous-composants. Nous montrerons par la suite comment chacun des composants peut être modélisé à l'aide des SR-Dep modèles, en mettant en évidence la manière dont notre approche permet de capturer plus précisément les contraintes de dépendance entre entrées et sorties.

4.1 Implémentation des SR-Dep modèles

L'implémentation des SR-Dep modèles repose sur la modélisation d'un système de transitions étiquetées *(Labelled Transition System – LTS)* à l'aide de graphes orientés. Pour cela, nous avons utilisé un code existant [58]comme base de travail, que nous avons adapté et enrichi afin d'y intégrer la notion de dépendance entrée/sortie propre aux SR-Dep modèles. Le développement a été réalisé en Python, en s'appuyant sur la bibliothèque networkx pour la manipulation des graphes, ainsi que sur plusieurs fonctions auxiliaires et structures personnalisées conçues pour répondre aux spécificités de notre modélisation.

4.1.1 Structures et bibliothèques utilisées

— **`networkx`** : bibliothèque principale pour la gestion des graphes orientés (ajout de nœuds, arcs, successeurs, prédécesseurs, etc.).

— **`typing`** : annotations de types statiques pour une meilleure lisibilité et fiabilité (`List`, `Tuple`, `Set`, etc.).

— `copy` : pour la duplication profonde des listes d'états lors de la génération d'états successeurs.

Trois classes principales structurent l'implémentation :

— `LTSVertex` : représente un état, identifié par une **liste d'entiers**, par exemple `[1,2]` ou `[2;1]`.

— `LTSEdge` : représente une transition entre deux états, étiquetée par une paire `(entrée, sortie)` telle que `('i', 'o')`, `('bar(i)', 'bar(o)')`, etc.

— `LTS` : structure principale représentant le graphe, avec des méthodes pour ajouter des états et transitions, et interroger les successeurs/prédécesseurs.

4.1.2 Utilisation de listes et fonctions auxiliaires

Les **états** sont représentés sous forme de `List[int]`, ce qui permet d'encoder la mémoire du composant et d'appliquer les opérations de dépendance. Pour manipuler ces listes efficacement, plusieurs **fonctions auxiliaires** ont été définies :

— `extend`, `fresh`, `incr`, `remove` :utilisées pour construire de nouveaux états à partir des précédents.

— `find_max`, `find_max2`, `least` :utilisées pour extraire certaines propriétés liées au calcul de latence.

— `compare_list`, `state_equiv` : utilisées pour vérifier l'égalité ou l'équivalence entre états, ce qui permet de générer l'automate minimal .

— `mem_twice`, `tous_elements_egaux`, `count_bar i_o predecessors` : utilisées pour appliquer des règles spécifiques à la dépendance. Comme le montre les figures 4.1 et 4.2

```
def tous_elements_egaux(liste):
    return all(x == liste[0] for x in liste) if liste else True
```

FIGURE 4.1 – Fonction auxilière tous_elements_egaux.

```python
def count_bar_i_o_predecessors(self, vertex: LTSVertex) -> int:

    count = 0
    current_vertex = vertex
    visited = set()
    while True:
        pred_edges = [edge for edge in self.get_pred_edges(current_vertex) if edge.label == ('bar(i)', 'o')]
        if not pred_edges:
            break
        # On prend le premier predecesseur
        edge = pred_edges[0]
        if edge.src in visited:
            break
        visited.add(edge.src)
        current_vertex = edge.src
        count += 1
    return count
```

FIGURE 4.2 – Fonction auxilière count_$\bar{i}, o$_predecessors.

4.1.3 Fonction `construct` : génération du LTS SR-Dep

Dans le cadre de ce travail, nous avons modifié la fonction `construct` qui était uniquement valable pour les cas ou une sortie dépend d'une seule entrée. Nous avons intégré de nouvelles contraintes liées à la dépendance des entrées/sorties. Plus précisément, nous avons introduit deux nouveaux paramètres, `Id` et `Od`, permettant de contrôler respectivement certaines transitions et le nombre maximal de sorties réelles admissibles par état. Ces ajouts ont nécessité l'adaptation des conditions internes à la fonction, afin d'assurer le respect des propriétés sémantiques.
La fonction `construct(root, m, l, Id, Od)` prend les paramètres suivants :

— m : La mémoire du composant.

— l : La latence du composant.

— Id : Le nombre d'entrées requises pour produire Od sorties.

— Od : Le nombre de sorties qui dépendent de Id entrées.

Les états sont représentés sous forme de listes d'entiers (type `List[int]`), ce qui permet une manipulation flexible de la mémoire et la latence.

4.1.4 Ajout des dépendances Id et Od

Deux propriétés fondamentales ont guidé l'ajout des conditions dans la fonction `construct` :

a. Propriété 1

Le code de la figure 4.3 illustre l'ajout conditionnel d'une transition `("bar(i)", "bar(o)")` en fonction de la longueur de l'état courant et du paramètre de dépendance Id. Si la mémoire de l'état actuel(`length(s0)`) est supérieure ou égale à Id(Le nombre d'entrées nécessaires au traitement), une transition est ajoutée vers un nouvel état .Ce qui signifie que le composant a reçu toutes les entrées nécessaires

et qu'il peut continuer son calcul pour arriver à la latence. En revanche, si cette mémoire actuelle est strictement inférieure à Id (le composant n'a pas encore reçu toutes les données du traitement), l'état boucle sur lui-même (le composant se met en attente des autres données nécéssaires),en accord avec la propriété 1, que nous rappelons ci-dessous :

$$\forall s \in S,\ m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)$$

```
if length(s0) >= 1 and find_max(s0) < l and length(s0)>=Id:
    s1 = fresh(incr(s0, l))
    v1 = self._get_or_create_vertex_sans_equiv(s1, states, l)
    self.lts.add_edge(LTSEdge(v0, ("bar(i)", "bar(o)"), v1))
elif length(s0) >= 1 and find_max(s0) < l and length(s0)<Id:
    self.lts.add_edge(LTSEdge(v0, ("bar(i)", "bar(o)"), v0))
```

FIGURE 4.3 – Code python : Contrainte sur les entrées.

b. Propriété 2

Le code de la figure 4.4 porte sur la propriété 2 et illustre l'ajout conditionnel des transitions qui permettent une sortie `(`β`, "o")` $\beta \in \{\bar{i}, i\}$, qu'on rappelle ici :

$$\forall s \in S,\ \exists s' \in s,\ s \xrightarrow{\bar{i}/o} s' \wedge \text{nombre\_O}(s) < Od-1 \Rightarrow \nexists\, s'',\ s \xrightarrow{i/\alpha} s'',\ \text{où } \alpha \in \{o, \bar{o}\}$$

Afin de respecter la contrainte liée au paramètre de sorties Od, nous avons intégré plusieurs conditions lors de la génération des transitions de sortie `("bar(i)", "o")` et `("i", "o")`. Deux cas possibles :

— Lorsque $Od = 1$ (qu'une seule sortie dépend de toutes les entrées), la sortie n'est autorisée que si l'état courant `s0` contient des éléments tous identiques (condition vérifiée par la fonction `tous_elements_egaux`), qui signifie que toutes les entrées ont atteint la latence. et quand la sortie est produite, tous les Id doivent être supprimés du composant ($remove\_d(s_0, Id)$)

— Lorsque $Od > 1$, nous avons introduit une méthode `count_bar_i_o_predecessors (s)` qui calcule le nombre de transitions `("bar(i)", "o")` menant à un état donné(s) c'est-à-dire : Combien de sorties ont été produites. La valeur retournée par cette fonction est stockée dans la variable `compteur_O`, et permet de vérifier dynamiquement à chaque sortie si le nombre de sorties O_d à été atteint.

```
if Od==1 and tous_elements_egaux(s0):
    extended= extend(incr(remove_d(s0, Id)))
elif compteur_O >= Od-1 and Od>1 :
    extended = extend(incr(remove(s0, find_max(s0)), l))
s1=fresh(extended)
v1 = self._get_or_create_vertex(s1, states, l)
edge = LTSEdge(v0, ("i", "o"), v1)
self.lts.add_edge(edge)
```

FIGURE 4.4 – Code python : Condition sur les sorties.

4.2 Etude de cas : composant IDCT

Nous appliquons notre approche à un composant réel utilisé dans le traitement de signal : le composant IDCT (Inverse Discrete Cosine Transform).

Nous modélisons les composants internes d'IDCT avec notre modèle afin de montrer que l'approche proposée permet de représenter de manière correcte le comportement des composants réactifs synchrones avec dépendances d'entrées/sorties

4.2.1 IDCT (Inverse Discret Cosine Transform)

La transformée en cosinus discrète (DCT - Discrete Cosine Transform) et sa transformée inverse (IDCT) [65] sont des composants largement utilisés en sciences et en ingénierie.

La transformée en cosinus discrète (DCT - Discrete Cosine Transform) convertit un signal spatial comme les images en représentation fréquentielle. Elle décompose le signal en composantes de différentes fréquences, permettant de séparer les informations importantes (basses fréquences) des détails fins (hautes fréquences). L'IDCT effectue l'opération inverse pour reconstituer le signal original.

Domaines d'application : L'IDCT et DCT sont un standard utilisé en traitement de signal, dans les compressions d'images, vidéos et audios tels que MP3, JPEG, MPEG ou H26x. qui sont utilisés dans plusieurs domaines comme :

— Décodeurs hardware dans les téléviseurs

— GPU modernes avec accélération DCT/IDCT

— Éditeurs d'images (Photoshop, GIMP)

— Applications de streaming vidéo

— Smartphones et tablettes

— Téléviseurs et décodeurs.

L'IDCT et DCT sont omniprésentes dans notre quotidien numérique, présentes dans pratiquement tous les appareils capables de traiter des images ou vidéos compressées.

Comme notre travail porte sur la problématique soulevée par Baclet et al. [5], nous nous basons sur leur représentation d'IDCT.

4.2.2 Interface

L'IDCT est une application synchrone à flux de données, où les opérations sont cadencées par une horloge. Elle dispose d'une interface d'entrée capable de recevoir et de produire des données dans un format approprié, à raison d'une donnée par cycle d'horloge.

Mathématiquement, IDCT divise une image en un sous ensemble de blocs 8×8 noté X, et produit un signal de sortie en deux dimensions de 8 colonnes et 8 lignes (matrice 8×8) après un certain nombre de cycles d'horloges.Le traitement s'effectue en calculant le produit des vecteurs à deux dimensions correspondant au produit A.X^t.A, sachant que A est une matrice donnée de taille 8×8.

4.2.3 Architecture

Nous représentons l'architecture interne de IDCT dans la figure 4.5, dans laquelle est représenté l'algorithme générique de IDCT, où un décodeur reconstruit une image en parcourant une série de sept composants fonctionnels connectés en cascade. Ils communiquent grâce à un protocole de type poignée de main, sachant que chaque composant peut accepter ou non une donnée selon son état interne. Comme illustré dans la figure 4.4, dans un premier temps le sous composant INR1 reçoit des données depuis l'environnement puis transmet les données au composant suivant et ainsi de suite, jusqu'au composant POST2 qui transférera ses sorties vers l'environnement, sous le contrôle du protocole poignée de main. voici la figure :

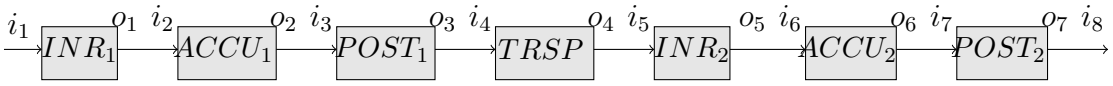

FIGURE 4.5 – Architecture interne du composant IDCT.

Il y a en tout quatre types de composants dans IDCT, chaque composant a ses propres caractéristiques :

— La latence : correspond au temps minimal de traitement d'une entrée.

— La mémoire : correspond au nombre maximal de données contenues dans le composant.

— La dépendance I/O : ce paramètre, que nous introduisons dans notre approche, capture les relations de dépendance entre les entrées et les sorties du composant. Contrairement au SR-modèles classiques [21] qui ne représente que le cas où une sortie dépend d'une seule entrée, cette notion permet de modéliser les cas où une sortie dépend de plusieurs entrées où plusieurs sorties dépendent de plusieurs entrées, reflétant ainsi des comportements plus réalistes des composants synchrones.

Nous décrivons les caractéristiques de chaque sous composant ainsi que l'utilité de chaque composant dans le processus de calcul du produit matriciel. Ne n'allons pas détailler les fonctions de traitement des sous composants mais uniquement, les paramètres permettant de décrire les séquences d'entrée/sortie de ces derniers.

— **Le sous-composant INR** réalise la normalisation d'une matrice à l'aide d'une table de quantification. Il traite une entrée et une sortie par cycle d'horloge, avec une capacité mémoire de 8 échantillons et une latence de 8 cycles (M = 8, L = 8). Chaque bloc de 8 sorties dépend d'un bloc de 8 entrées consécutives.(Id = 8, Od =8)

— **Le sous-composant ACCU** effectue l'opération d'accumulation. Il consomme une entrée par cycle d'horloge et produit huit sorties en parallèle par cycle, une fois qu'il a accumulé huit entrées. Il possède une capacité mémoire de 8 échantillons et une latence de 8 cycles (M = 8, L = 8). Chaque bloc de 8 sorties dépend d'un bloc de 8 entrées consécutives. (Id = 8, Od =1).

— **Le sous-composant POST** effectue l'opération de post-traitement. Il consomme 8 entrées simultanées par cycle d'horloge et produit une sortie par cycle. Il possède une capacité mémoire de 8 échantillons et une latence de 8 cycles(M = 8, L = 8). Chaque sortie produite dépend d'un bloc de 8 entrées consécutives .(Id = 8, Od =8).

— **Le sous-composant TRSP** effectue l'opération de transposition de matrice. Il consomme une donnée par cycle d'horloge et produit également une sortie par cycle. Il dispose d'une capacité mémoire de 64 échantillons et présente une latence de 8 cycles (M = 64, L = 8). (Id = 8, Od =8).

Le composant IDCT a un seul sous-composant TRSP, et a besoin d'une paire de tous les autres sous-composants pour fonctionner. Dans le but d'avoir des modèles de tailles raisonnables, sans perdre en sûreté, nous limitons la taille de la matrice à (2×2) au lieu de (8×8). Les paramètres des sous-composants sont alors comme suit :

INR

— **Mémoire** : 2

— **Latence** : 2

— **Dépendance** : $Id = 2$,$Od = 2$

ACCU

— **Mémoire** : 2

— **Latence** : 2

— **Dépendance** : $Id = 2$, $Od = 1$

POST

— **Mémoire** : 2

— **Latence** : 2

— **Dépendance** : $Id = 2$, $Od = 2$

TRSP

— **Mémoire** : 4

— **Latence** : 4

— **Dépendance** : $Id = 4$, $Od = 1$

4.3 Modélisation des composants d'IDCT avec les SR-Dep modèles

À partir des paramètres propres aux composants de l'IDCT, nous avons généré les SR-Dep modèles de chaque sous-composant, notre implémentation a été réalisée en Python. Ce programme permet de construire automatiquement les SR-Dep modèles à partir des paramètres M, L, Id et Od, en respectant les règles de dépendance introduites dans le chapitre 3 (Définition 2).

INR et POST

Nous illustrons le SR-Dep modèle des composants INR et POST obtenu par le programme dans la figure 4.6 et reflète fidèlement celui de la figure 3.8 représentée au chapitre 3 où nous obtenons un automate avec 4 états et 9 transitions.

```
Component 1:
LTS with 4 vertices and 9 edges:
vertex 1
edges 2
  edge 1 --('i', 'bar(o)')--> 2;1
  edge 1 --('bar(i)', 'bar(o)')--> 1
vertex
edges 2
  edge  --('i', 'bar(o)')--> 1
  edge  --('bar(i)', 'bar(o)')-->
vertex 2
edges 3
  edge 2 --('i', 'o')--> 1
  edge 2 --('bar(i)', 'o')-->
  edge 2 --('bar(i)', 'bar(o)')--> 2
vertex 2;1
edges 2
  edge 2;1 --('bar(i)', 'o')--> 2
  edge 2;1 --('bar(i)', 'bar(o)')--> 2;1
End Graph
```

FIGURE 4.6 – SR-Dep modèle du composant INR et POST

ACCU

Nous illustrons par exemple le SR-Dep modèle de ACCU obtenu par le programme dans la figure 4.7 et reflète fidèlement celui de la figure 3.6 représenté au chapitre 3 où nous obtenons 4 états et 8 transitions.

```
Component 1:
LTS with 4 vertices and 8 edges:
vertex 1
edges 2
  edge 1 --('i', 'bar(o)')--> 2;1
  edge 1 --('bar(i)', 'bar(o)')--> 1
vertex
edges 2
  edge  --('i', 'bar(o)')--> 1
  edge  --('bar(i)', 'bar(o)')-->
vertex 2;1
edges 1
  edge 2;1 --('bar(i)', 'bar(o)')--> 2;2
vertex 2;2
edges 3
  edge 2;2 --('i', 'o')--> 1
  edge 2;2 --('bar(i)', 'o')-->
  edge 2;2 --('bar(i)', 'bar(o)')--> 2;2
End Graph
```

FIGURE 4.7 – SR-Dep modèle du composant ACCU.

TRSP

Nous illustrons par exemple le SR-Dep modèle de TRSP obtenu par le programme dans la figure 4.9

```
Component 1:
LTS with 8 vertices and 14 edges:
vertex 4;4;4;4
edges 3
  edge 4;4;4;4 --('i', 'o')--> 1
  edge 4;4;4;4 --('bar(i)', 'o')-->
  edge 4;4;4;4 --('bar(i)', 'bar(o)')--> 4;4;4;4
vertex 2;1
edges 2
  edge 2;1 --('i', 'bar(o)')--> 3;2;1
  edge 2;1 --('bar(i)', 'bar(o)')--> 2;1
vertex 4;4;4;3
edges 1
  edge 4;4;4;3 --('bar(i)', 'bar(o)')--> 4;4;4;4
vertex 1
edges 2
  edge 1 --('i', 'bar(o)')--> 2;1
```

FIGURE 4.8 – Partie 1 du SR-Dep modèle du composant TRSP .

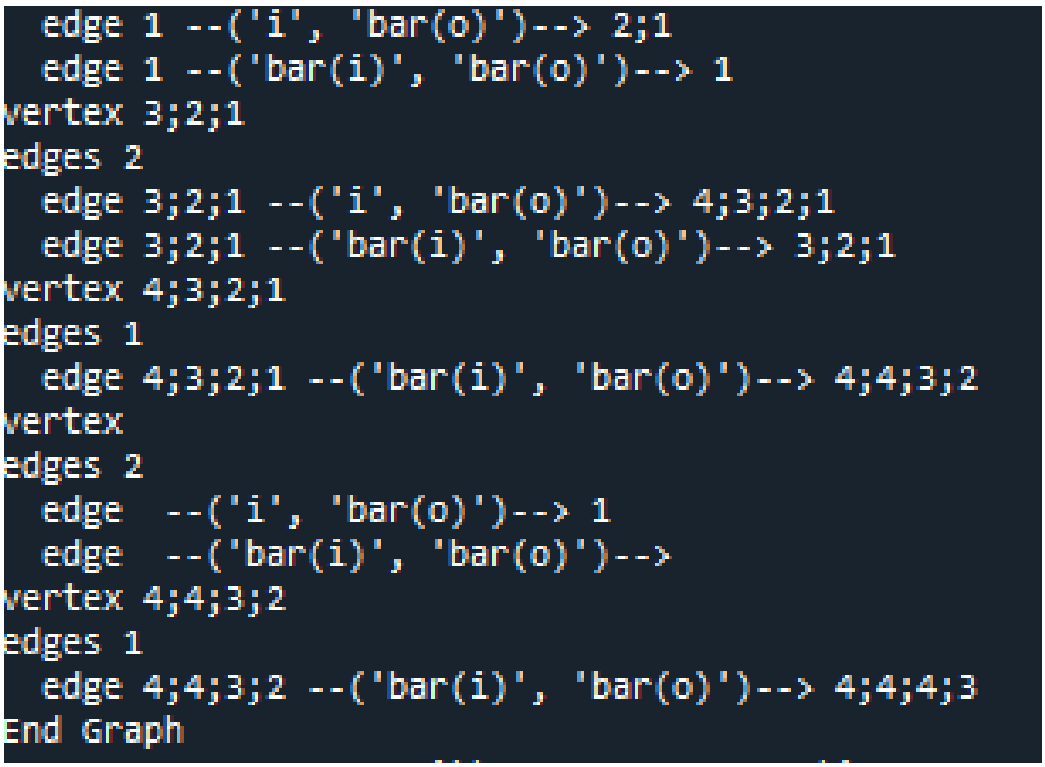

FIGURE 4.9 – Partie 2 du SR-Dep modèle du composant TRSP .

4.3.1 Vérification formelle de la sémantique des SR-Dep modèles

Dans cette section, nous nous intéressons à la vérification formelle des propriétés sémantiques que nous avons énoncées pour les SR-Dep modèles. L'objectif est de démontrer rigoureusement que ces propriétés sont bien respectées dans tous les automates générés pour les composants étudiés, à savoir INR, POST, ACCU et TRSP. Pour ce faire, nous effectuerons une vérification systématique sur les automates associés à chacun de ces composants, en analysant leur structure d'états et de transitions au regard des règles de dépendance introduites. Cette démarche permet d'assurer que notre approche respecte les contraintes théoriques posées, et renforce ainsi la validité et la généralité du modèle proposé. Nous allons définir deux théorèmes qui portent sur les deux propriétés définies dans le chapitre 3.

Théorème 1 (Input Dependency) *Le SR-Dep modèle $\mathcal{A}$ satisfait la propriété 1 ssi :*

$$\forall s \in S, \quad m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)$$

Théorème 2 (Output Dependency) *Le SR-Dep modèle $\mathcal{A}$ satisfait la propriété 2 ssi :*

$$\forall s \in S,\ \exists s' \in s,\ s \xrightarrow{\bar{i}/o} s' \wedge nombre\_O(s) < Od - 1 \Rightarrow \nexists\ s'',\ s \xrightarrow{i/\alpha} s'',\ où\ \alpha \in \{o, \bar{o}\}$$

La validation sémantique des SR-Dep-modèles appliquée aux composants d'IDCT repose sur l'analyse de ces deux propriétés : la dépendance en entrée et la dépendance en sortie. Ces propriétés seront vérifiées à travers l'examen systématique des états de l'automate, en fonction des paramètres de dépendance Id et Od.

a. Vérification de la sémantique du SR-Dep modèle de INR et POST

Preuve du théorème 1 pour INR et POST Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 2, L = 2, Id = 2, Od = 2)$.

État	$m(s)$	$m(s) < Id$	**Transition requise**	**Transition observée**	**Satisfaite**
$\langle\rangle$	0	$0 < 2$	$\langle\rangle \xrightarrow{\bar{i},\bar{o}} \langle\rangle$	Boucle $(\bar{i}, \bar{o})$	✓
$\langle 1\rangle$	1	$1 < 2$	$\langle 1\rangle \xrightarrow{\bar{i},\bar{o}} \langle 1\rangle$	Boucle $(\bar{i}, \bar{o})$	✓
$\langle 2, 1\rangle$	2	$2 \not< 2$	Condition fausse $\Rightarrow$ propriété vraie		✓
$\langle 2\rangle$	1	$1 < 2$	$\langle 2\rangle \xrightarrow{\bar{i},\bar{o}} \langle 2\rangle$	Boucle $(\bar{i}, \bar{o})$	✓

TABLE 4.1 – Vérification formelle de la Propriété 1 pour le SR-Dep modèle d'INR et POST

Conclusion :

$$\forall s \in S = \{\langle\rangle, \langle 1\rangle, \langle 2, 1\rangle, \langle 2\rangle\}, \quad [m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)] = \text{VRAI}$$

La propriété 1 est donc satisfaite pour tous les états de INR et POST.

Preuve du théorème 2 pour INR et POST Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 2, L = 2, Id = 2, Od = 2)$. Notons que :

— T_O : représente $\exists s' \in s,\ s \xrightarrow{\bar{i}/o} s'$

— T_1 représente $nombre\_O(s) < Od-1$: dans le tableau $o(s)$ au lieu de $nombre\_O(s)$

État	$o(s)$	T_0	T_1	$T_0 \wedge T_1$	**Transition non permise**	**Transitions observées**	**Satisfaite**
$\langle\,\rangle$	0	False	$0 < 1$	False	/	$(i, \bar{o})$,$(\bar{i}, \bar{o})$	✓
$\langle 1\rangle$	0	False	$0 < 1$	False	/	$(i, \bar{o})$,,$(\bar{i}, \bar{o})$	✓
$\langle 2, 1\rangle$	0	True	$0 < 1$	True	$\langle 2; 1\rangle \xrightarrow{i/o} \langle 2; 1\rangle$	$(\bar{i}, o)$,$(\bar{i}, \bar{o})$	✓
$\langle 2\rangle$	1	True	$1 \not< 1$	False	/	(i, o), $(\bar{i}, o)$,$(\bar{i}, \bar{o})$	✓

TABLE 4.2 – Vérification formelle de la Propriété 2 pour le SR-Dep modèle d'INR et POST

Conclusion :

Toutes les transitions i/α observées apparaissent uniquement lorsque la condition $T_0 \wedge T_1$ est **fausse**,ce qui est conforme à la propriété 2.

$$\forall s \in S = \{\,\langle\rangle, \langle 1\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \},$$
$$[T_0 \wedge T_1 \Rightarrow \nexists \text{ transition } i/\alpha] = \text{VRAI}$$

La propriété 2 est donc satisfaite pour tous les états de INR et POST.

b. Vérification de la sémantique du SR-Dep modèle de ACCU

Preuve du théorème 1 pour ACCU Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 2, L = 2, Id = 2, Od = 1)$.

État	$m(s)$	$m(s) < Id$	**Transition requise**	**Transition observée**	**Satisfaite**
$\langle\rangle$	0	$0 < 2$	$\langle\rangle \xrightarrow{\bar{i},\bar{o}} \langle\rangle$	Boucle $(\bar{i}, \bar{o})$	✓
$\langle 1\rangle$	1	$1 < 2$	$\langle 1\rangle \xrightarrow{\bar{i},\bar{o}} \langle 1\rangle$	Boucle $(\bar{i}, \bar{o})$	✓
$\langle 2, 2\rangle$	2	$2 \not< 2$	Condition fausse ⇒ propriété vraie		✓
$\langle 2, 1\rangle$	2	$2 \not< 2$	Condition fausse ⇒ propriété vraie		✓

TABLE 4.3 – Vérification formelle de la Propriété 1 pour tous les états d'ACCU

Conclusion :

$$\forall s \in S = \{\langle\rangle, \langle 1\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle\}, \quad [m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)] = \text{VRAI}$$

La propriété 1 est donc satisfaite pour tous les états d'ACCU.

Preuve du théorème 2 pour ACCU Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 2, L = 2, Id = 2, Od = 1)$. Rappelons que :

— T_O : représente $\exists s' \in s,\ s \xrightarrow{\bar{i}/o} s'$

— T_1 représente $nombre\_O(s) < Od{-}1$: dans le tableau $o(s)$ au lieu de $nombre\_O(s)$

État	$o(s)$	T_0	T_1	$T_0 \wedge T_1$	**Transition non permise**	**Transitions observées**	**Satisfaite**
$\langle\,\rangle$	0	False	$0 \not< 0$	False	/	$(i, \bar{o})$,$(\bar{i}, \bar{o})$	✓
$\langle 1\rangle$	0	False	$0 \not< 0$	False	/	$(i, \bar{o})$,$(\bar{i}, \bar{o})$	✓
$\langle 2, 1\rangle$	0	False	$0 \not< 0$	False	/	$(\bar{i}, \bar{o})$	✓
$\langle 2, 2\rangle$	0	True	$0 \not< 0$	False	/	(i, o), $(\bar{i}, o)$,$(\bar{i}, \bar{o})$	✓

TABLE 4.4 – Vérification formelle de la Propriété 2 pour le SR-Dep modèle d'ACCU

Conclusion :

$$\forall s \in S = \{\,\langle\rangle, \langle 1\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \,\}, \\ [T_0 \wedge T_1 \Rightarrow \nexists \text{ transition } i/\alpha] = \text{VRAI}$$

La propriété 2 est donc satisfaite pour tous les états d'ACCU.

c. Vérification de la sémantique du SR-Dep modèle de TRSP

Preuve du théorème 1 pour TRSP Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 4, L = 4, Id = 4, Od = 1)$.

État	$m(s)$	$m(s) < Id$	**Transition requise**	**Transition observée**	**Satisfaite**
$\langle\rangle$	0	$0 < 4$	$\langle\rangle \xrightarrow{\bar{i},\bar{o}} \langle\rangle$	Boucle $(\bar{i},\bar{o})$	✓
$\langle 1\rangle$	1	$1 < 4$	$\langle 1\rangle \xrightarrow{\bar{i},\bar{o}} \langle 1\rangle$	Boucle $(\bar{i},\bar{o})$	✓
$\langle 2,1\rangle$	2	$2 < 4$	$\langle 2,1\rangle \xrightarrow{\bar{i},\bar{o}} \langle 2,1\rangle$	Boucle $(\bar{i},\bar{o})$	✓
$\langle 3,2,1\rangle$	3	$3 < 4$	$\langle 3,2,1\rangle \xrightarrow{\bar{i},\bar{o}} \langle 3,2,1\rangle$	Boucle $(\bar{i},\bar{o})$	✓
$\langle 4,3,2,1\rangle$	4	$4 \not< 4$	Condition fausse ⇒ propriété vraie		✓
$\langle 4,4,3,2\rangle$	4	$4 \not< 4$	Condition fausse ⇒ propriété vraie		✓
$\langle 4,4,4,3\rangle$	4	$4 \not< 4$	Condition fausse ⇒ propriété vraie		✓
$\langle 4,4,4,4\rangle$	4	$4 \not< 4$	Condition fausse ⇒ propriété vraie		✓

TABLE 4.5 – Vérification formelle de la Propriété 1 pour tous les états de TRSP

Conclusion :

$$\forall s \in S = \{\, \langle\rangle, \langle 1\rangle, \langle 2,1\rangle, \langle 3,2,1\rangle, \langle 4,3,2,1\rangle, \langle 4,4,3,2\rangle, \langle 4,4,4,3\rangle, \langle 4,4,4,4\rangle \,\},$$
$$[m(s) < Id \Rightarrow (s \xrightarrow{\bar{i},\bar{o}} s)] = \text{VRAI}$$

La propriété 1 est donc satisfaite pour tous les états de TRSP.

Preuve du théorème 2 pour TRSP Nous devons vérifier la propriété sur tous les états s de l'automate avec paramètres $(M = 4, L = 2, Id = 4, Od = 1)$. Rappelons que :

— T_O : représente $\exists s' \in s,\ s \xrightarrow{\bar{i}/o} s'$

— T_1 représente $nombre\_O(s) < Od{-}1$: dans le tableau $o(s)$ au lieu de $nombre\_O(s)$

État	$o(s)$	T_0	T_1	$T_0 \wedge T_1$	**Transition non permise**	**Transitions observées**	**Satisfaite**
$\langle\,\rangle$	0	False	$0 \not< 0$	False	/	$(i,\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 1\rangle$	0	False	$0 \not< 0$	False	/	$(i,\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 2,1\rangle$	0	False	$0 \not< 0$	False	/	$(i,\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 3,2,1\rangle$	0	False	$0 \not< 0$	False	/	$(i,\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 4,3,2,1\rangle$	0	False	$0 \not< 0$	False	/	$(\bar{i},\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 4,4,3,2\rangle$	0	False	$0 \not< 0$	False	/	$(\bar{i},\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 4,4,4,3\rangle$	0	False	$0 \not< 0$	False	/	$(\bar{i},\bar{o})$,$(\bar{i},\bar{o})$	✓
$\langle 4,4,4,4\rangle$	1	True	$1 \not< 0$	False	/	(i,o) ,$(\bar{i},o)$,$(\bar{i},\bar{o})$	✓

TABLE 4.6 – Vérification formelle de la Propriété 2 pour le SR-Dep modèle de TRSP

Conclusion :

$$\forall s \in S = \{\, \langle\rangle, \langle 1\rangle, \langle 2,1\rangle, \langle 3,2,1\rangle, \langle 4,3,2,1\rangle, \langle 4,4,3,2\rangle, \langle 4,4,4,3\rangle, \langle 4,4,4,4\rangle \,\},$$
$$[T_0 \wedge T_1 \Rightarrow \nexists \text{ transition } i/\alpha] = \text{VRAI}$$

La propriété 2 est donc satisfaite pour tous les états de TRSP.

4.3.2 Discussion

Les quatre composants d'IDCT respectent les deux théorèmes (1 et 2). Ce qui montre que les SR-Dep modèles de ces composants respectent la sémantique comportementale des composants en termes de dépendances d'entrées/sorties. Les transitions observées dans les automates générés respectent rigoureusement la sémantique énoncée dans la Définition 2, notamment les contraintes portant sur le nombre d'entrées reçues avant traitement, ainsi que la restriction sur les sorties conditionnelles. Ainsi, la formalisation proposée permet de garantir le bon ordre des événements et de représenter fidèlement les comportements temporels et causals attendus dans un système réactif.

Cette validation sur les composants d'IDCT constitue une première étape encourageante vers l'application de notre approche à des systèmes plus complexes.

Conclusion

Dans ce chapitre, nous avons évalué la fiabilité des SR-Dep modèles à travers une étude de cas appliquée au composant IDCT. Nous avons tout d'abord décrit les choix d'implémentation, notamment les structures de données, les bibliothèques utilisées ainsi que les fonctions auxiliaires permettant de construire le LTS associé au modèle.

Nous avons ensuite présenté le composant IDCT en détail, en décrivant son interface et son architecture interne, avant de procéder à la modélisation de ses différents sous-composants à l'aide des SR-Dep modèles. Cette modélisation a mis en évidence la capacité du formalisme à capturer à la fois les contraintes temporelles (latence et mémoire), mais surtout la dépendance de données (avec les paramètres Id et Od).

Enfin, nous avons réalisé une vérification formelle de la sémantique des modèles obtenus en démontrant la satisfiabilité de théorèmes par tous les composants d'IDCT. Offrant ainsi une première validation prometteuse de notre approche.

Conclusion générale

Ce mémoire s'inscrit dans le cadre de la recherche de méthodes formelles de modélisation fiables pour les systèmes embarqués réactifs. Face aux limites des SR-modèles existants à capturer la complexité des relations de dépendance entre les entrées et les sorties, nous avons proposé un enrichissement du modèle initial, que nous avons appelé SR-Dep modèle (Synchronous Reactive Dependency model).

Cette contribution repose sur l'introduction explicite de propriétés de dépendance d'entrées/sorties (I/O), tout en sauvegardant les propriétés déjà existantes et validées des SR-modèles. Ces propriétés permettent de spécifier les conditions qui définissent l'ordre des évènements d'entrées/sorties dans les Systèmes réactifs. Cette approche permet une représentation plus fidèle des comportements d'entrées/sorties observés dans les systèmes embarqués, offrant ainsi un cadre théorique plus adapté pour la validation des propriétés de sûreté et l'analyse formelle des différents scénarios d'exécution du système.

Nous avons implémenté les propriétés proposées sur un outil de génération automatique de SR-Dep modèles, dans le but de réaliser des expérimentations et d'évaluer notre approche. L'évaluation de notre proposition a été réalisée sur un cas d'étude réel, le composant IDCT, composant largement utilisé dans les chaînes de traitement du signal embarquées. Les résultats obtenus montrent que notre modèle permet une modélisation plus fidèle du comportement du système, une meilleure expressivité pour spécifier les dépendances d'entrées/sorties.En effet, nous avons confirmé que les SR-Dep modèles produits par notre outil satisfont les propriétés établies dans ce travail de recherche, constituant ainsi une première phase de validation de notre approche.

Perspectives

Le travail effectué offre de nombreuses perspectives, notre approche favorise une spécification plus précise des systèmes réactifs synchrones. Elle représente un point de départ vers la conception d'un nouveau modèle aux perspectives encourageantes. Nous pouvons par la suite prendre en compte les aspects suivants :

— La suite directe de ce travail de recherche portera sur la validation formelle des propriétés proposées en utilisant des techniques de preuves par théorèmes (Theorem Proving).

— Une première extension des SR-Dep modèle peut être l'intégration de nouveaux types de dépendance des entrées/sorties, afin d'élargir la classe de systèmes réactifs représentés.

— Un autre aspect important dans la modélisation des systèmes réactifs est la composition. Nous visons à introduire la composition des SR-Dep modèles, qui peut étendre la composition existante des SR-modèles en intégrant la dépendance des entrées/sorties afin de vérifier que les propriétés de sûreté soient préservées lors de la composition, et ainsi pouvoir représenter des systèmes plus complexes de manière modulaire.

— Le formalisme pourrait être adapté à d'autres types de systèmes, par l'intégration de dimensions supplémentaires telles que : le temps réel, le comportement probabiliste ou stochastique et ainsi introduire de l'asynchrone. Cela permettrait de modéliser une classe de systèmes plus large.

— Adapter notre approche afin de pouvoir l'intégrer aux chaînes d'outils existantes (SCADE, Simulink, etc.) pour faciliter l'adoption dans les contextes industriels de développement de systèmes critiques.

Bibliographie

[1] Eshrat Arjomandi, Michael Fischer et Nancy Lynch. “Efficiency of Synchronous Versus Asynchronous Distributed Systems”. In : *J. ACM* 30 (juill. 1983), p. 449-456. doi : 10.1145/2402.322387.

[2] A. Arnold. “Systemes De Transitions Finis Et Semantique”. In : Etudes et recherches en informatique (1992). url : https://books.google.dz/books?id=bBhnAAAACAAJ.

[3] Mehdi Assefi et al. “Experimental Evaluation of Apple Siri and Google Speech Recognition”. In : oct. 2015.

[4] Paul Attie et Nancy Lynch. *Dynamic Input/Output Automata : a Formal and Compositional Model for Dynamic Systems*. Avr. 2016. doi : 10.48550/arXiv.1604.06030.

[5] Manuel Baclet, Renaud Pacalet et Antoine Petit. *Register Transfer Level Simulation*. Research Report LSV-04-10. 15 pages. Laboratoire Spécification et Vérification, ENS Cachan, France, mai 2004. url : http://www.lsv.ens-cachan.fr/Publis/RAPPORTS_LSV/PS/rr-lsv-2004-10.rr.ps.

[6] Gilles Barthe et al. “Formal Certification of Code-Based Cryptographic Proofs”. In : *IACR Cryptology ePrint Archive* 2007 (jan. 2007), p. 314. doi : 10.1145/1480881.1480894.

[7] Françoise Baude et al. “GCM : A Grid Extension to Fractal for Autonomous Distributed Components”. In : *Annals of Telecommunications - annales des télécommunications* (2008). url : https://inria.hal.science/inria-00323919.

[8] Albert Benveniste, Paul Le Guernic et Christian Jacquemot. “Synchronous Programming with Events and Relations : the SIGNAL Language and Its Semantics”. In : *Sci. Comput. Program.* 16 (1991), p. 103-149. url : https://api.semanticscholar.org/CorpusID:205066459.

[9] Gérard Berry. *The Esterel v5 language primer : version v5_91*. Centre de mathématiques appliquées, Ecole des mines et INRIA, 2002.

[10] Gérard Berry et Georges Gonthier. “The Esterel synchronous programming language : design, semantics, implementation”. In : *Science of Computer Programming* 19.2 (1992), p. 87-152. issn : 0167-6423. doi : https://doi.org/10.1016/0167-6423(92)90005-V. url : https://www.sciencedirect.com/science/article/pii/016764239290005V.

[11] Gérard Berry et al. “ESTEREL : A formal method applied to avionic software development”. In : *Science of Computer Programming* 36.1 (déc. 2000), p. 5-25. doi : 10.1016/S0167-6423(99)00015-5. url : https://minesparis-psl.hal.science/hal-00579632.

[12] Yves Bertot et Pierre Castéran. *Interactive theorem proving and program development. Coq'Art : The Calculus of inductive constructions*. Jan. 2004. ISBN : 3540208542. DOI : 10.1007/978-3-662-07964-5.

[13] Michael Biesbrouck, Brad Calder et Lieven Eeckhout. "Efficient Sampling Startup for SimPoint". In : *Micro, IEEE* 26 (août 2006), p. 32-42. DOI : 10.1109/MM.2006.68.

[14] Mouad Bounouar et al. "Etat des lieux de la cobotique industrielle et de la conduite de projet associée". In : avr. 2019.

[15] Timothy Bourke et Marc Pouzet. "Zélus : A Synchronous Language with ODEs". In : avr. 2013, p. 113-118. DOI : 10.1145/2461328.2461348.

[16] Marius Bozga, Vasiliki Sfyrla et Joseph Sifakis. "Modeling Synchronous Systems in BIP". In : oct. 2009, p. 77-86. DOI : 10.1145/1629335.1629347.

[17] Marco Bozzano et al. "A Model Checker for AADL". In : juill. 2010. ISBN : 978-3-642-14294-9. DOI : 10.1007/978-3-642-14295-6_48.

[18] Ran Canetti et al. "Using Task-Structured Probabilistic I/O Automata to Analyze Cryptographic Protocols". In : *Crystal Growth and Design - CRYST GROWTH DES* (juill. 2006).

[19] Franck Cassez et Olivier Roux. "Traduction structurelle des Réseaux de Petri Temporels en Automates Temporisés". In : oct. 2003.

[20] Sarah Chabane, Rabea Ameur-Boulifa et Mohamed Mezghiche. "Formal framework for automated analysis and verification of distributed reactive applications". In : *2017 First International Conference on Embedded & Distributed Systems (EDiS)*. IEEE. 2017, p. 1-6.

[21] Sarah Chabane, Rabea Ameur-Boulifa et Mezghiche Mohamed. "Towards compositional verification of synchronous reactive systems". In : *International Journal of Critical Computer-Based Systems* 10.2 (2021), p. 120-142.

[22] Pierre Civit et Maria Potop-Butucaru. *Probabilistic Dynamic Input Output Automata (Extended Version)*. Cryptology ePrint Archive, Paper 2021/798. 2021. DOI : 10.4230/LIPIcs.DISC.2022.20. URL : https://eprint.iacr.org/2021/798.

[23] Thomas Constant et Guillaume Levieux. *Perception de la variabilité et facteurs de la curiosité dans les jeux vidéo*. Fév. 2022. DOI : 10.13140/RG.2.2.18564.60808.

[24] François Dormoy. "SCADE 6 A Model Based Solution For Safety Critical Software Development". In : (jan. 2008).

[25] Bradley Efron. "Bootstrap Methods : Another Look at the Jackknife". In : *Breakthroughs in Statistics : Methodology and Distribution*. Sous la dir. de Samuel Kotz et Norman L. Johnson. New York, NY : Springer New York, 1992, p. 569-593. ISBN : 978-1-4612-4380-9. DOI : 10.1007/978-1-4612-4380-9_41. URL : https://doi.org/10.1007/978-1-4612-4380-9_41.

[26] Johan Eker et al. "Taming Heterogeneity - The Ptolemy Approach". In : *Proceedings of the IEEE* 91 (jan. 2003), p. 127-144. DOI : 10.1109/JPROC.2002.805829.

[27] International Engineering. *INCOSE Systems Engineering Handbook : A Guide for System Life Cycle Processes and Activities*. Juin 2015. ISBN : 9781118999400.

[28] Fairouz Fakhfakh, Slim Kallel et Saoussen Cheikhrouhou. "Formal Verification of Cloud and Fog Systems :A Review and Research Challenges". In : t. 27. Avr. 2021, p. 341-363. DOI : 10.3897/jucs.66455.

[29] Peter Feiler, Bruce Lewis et Steve Vestal. “The SAE Architecture Analysis and Design Language (AADL) a standard for engineering performance critical systems”. In : nov. 2006, p. 1206-1211. doi : 10.1109/CACSD-CCA-ISIC.2006.4776814.

[30] Hubert Garavel et al. “CADP 2010 : A Toolbox for the Construction and Analysis of Distributed Processes”. In : t. 6605. Mars 2011. isbn : 978-3-642-19834-2. doi : 10.1007/978-3-642-19835-9_33.

[31] Stephen Garland et Nancy Lynch. “Using I/O automata for developing distributed systems”. In : jan. 2000, p. 285-312.

[32] Nicolas Halbwachs. *Synchronous programming of reactive systems*. T. 215. Springer Science et Business Media, 1992.

[33] Nicolas Halbwachs. “Synchronous Programming of Reactive Systems.” In : jan. 1998, p. 1-16.

[34] Nicolas Halbwachs et al. “The synchronous data flow programming language LUSTRE”. In : *Proceedings of the IEEE* 79 (oct. 1991), p. 1305-1320. doi : 10.1109/5.97300.

[35] D. Harel et A. Pnueli. “On the Development of Reactive Systems”. In : *Logics and Models of Concurrent Systems*. Sous la dir. de Krzysztof R. Apt. Berlin, Heidelberg : Springer Berlin Heidelberg, 1985, p. 477-498. isbn : 978-3-642-82453-1.

[36] Ludovic Henrio, Florian Kammüller et Muhammad Uzair Khan. “A Framework for Reasoning on Component Composition”. In : *FMCO* (nov. 2009). doi : 10.1007/978-3-642-17071-3_1.

[37] Gerard Holzmann. “The Model Checker SPIN”. In : *IEEE Transactions on Software Engineering* 23 (mai 1997), p. 279-295.

[38] Mohamad Houssein. “Model-based IDS design pour ICS”. Thèse de doct. Nov. 2020.

[39] John Hudak et Peter Feiler. “Developing AADL Models for Control Systems : A Practitioner’s Guide”. In : (nov. 2011).

[40] Chung-Wen Hung, Cheng-Tsung Lin et Chih-Wen Liu. “An Efficient Simulation Technique for the Variable Sampling Effect of BLDC Motor Applications”. In : déc. 2007, p. 1175-1179. doi : 10.1109/IECON.2007.4460272.

[41] Jeff Jensen, Edward Lee et Sanjit Seshia. *An Introductory Lab in Embedded and Cyber-Physical Systems*. Jan. 2014. doi : 10.13140/2.1.3291.4242.

[42] Hamoudi Kalla. “Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes”. In : (jan. 2004).

[43] Dilsun Kaynar et al. “Timed I/O Automata : A Mathematical Framework for Modeling and Analyzing Real-Time Systems.” In : jan. 2003, p. 166-177. doi : 10.1109/REAL.2003.1253264.

[44] Shahidul Khan, Pritam Singh et Shuvo Gupta. “Controlling Home Appliances Using Computer Application”. In : oct. 2012.

[45] Jin Kim, Min-Gu Kim et Sung Pan. “A study on implementation of real-time intelligent video surveillance system based on embedded module”. In : *EURASIP Journal on Image and Video Processing* 2021 (nov. 2021). doi : 10.1186/s13640-021-00576-0.

[46] Mikkel Kjaergaard et Jonathan Bunde-Pedersen. “Towards a formal model of context awareness”. In : (jan. 2006).

[47] Phillip Laplante et Seppo Ovaska. “Real-Time Systems Design and Analysis : Tools for the Practitioner”. In : *Real-Time Systems Design and Analysis : Tools for the Practitioner* (nov. 2011). doi : 10.1002/9781118136607.

[48] Kim Larsen, Ulrik Nyman et Andrzej Wasowski. “Interface Input/Output Automata”. In : août 2006, p. 82-97. isbn : 978-3-540-37215-8. doi : 10.1007/11813040_7.

[49] Gilles Lasnier et al. “Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications”. In : juin 2009, p. 237-250. isbn : 978-3-642-01923-4. doi : 10.1007/978-3-642-01924-1_17.

[50] Q. Leclère. “Caractérisation expérimentale de sources vibratoires et acoustiques”. In : (déc. 2012).

[51] Jie Liu et al. “Actor-Oriented Control System Design : A Responsible Framework Perspective”. In : *Control Systems Technology, IEEE Transactions on* 12 (avr. 2004), p. 250-262. doi : 10.1109/TCST.2004.824310.

[52] Aymen Louati. “Contribution à la formalisation et à la vérification des diagrammes dynamiques UML2 à base des réseaux de Petri”. Thèse de doct. Déc. 2015.

[53] Nancy Lynch, Roberto Segala et Frits Vaandrager. “Hybrid I/O automata”. In : *Information and Computation* 185 (août 2003), p. 105-157. doi : 10.1016/S0890-5401(03)00067-1.

[54] Nancy Lynch et Mark Tuttle. “Hierarchical Correctness Proofs for Distributed Algorithms”. In : (jan. 2001). doi : 10.1145/41840.41852.

[55] Nancy A. Lynch et Mark R. Tuttle. “An introduction to input/output automata”. In : *CWI quarterly* 2 (1989), p. 219-246. url : https://api.semanticscholar.org/CorpusID:10292247.

[56] Radu Mateescu et Damien Thivolle. “A Model Checking Language for Concurrent Value-Passing Systems”. In : mai 2008. isbn : 978-3-540-68235-6. doi : 10.1007/978-3-540-68237-0_12.

[57] B.R. Mehta et Y.J. Reddy. “SCADA systems”. In : jan. 2015, p. 237-300. isbn : 9780128009390. doi : 10.1016/B978-0-12-800939-0.00007-3.

[58] Mohamed Mezghiche, Rabéa Ameur-Boulifa et Sarah Chabane. “Towards Compositional Verification of Synchronous Reactive System”. In : *International Journal of Critical Computer-Based Systems* 10 (jan. 2021), p. 1. doi : 10.1504/IJCCBS.2021.10040408.

[59] Leonardo de Moura et Nikolaj Bjørner. “Z3 : an efficient SMT solver”. In : t. 4963. Avr. 2008, p. 337-340. isbn : 978-3-540-78799-0. doi : 10.1007/978-3-540-78800-3_24.

[60] François Muzard. “BIM, du logiciel à la plateforme. Concept, enjeux, faisabilité d’une plateforme BIM en ligne dans la conception de projet.” Thèse de doct. Juin 2016. doi : 10.13140/RG.2.2.19948.77443.

[61] Andres Pace et Marcelo Campo. “An Empirical Study About Separation of Concerns Approaches”. In : (déc. 2002).

[62] Frantisek Plasil et Stanislav Visnovsky. “Behavior Protocols for Software Component”. In : *Software Engineering, IEEE Transactions on* 28 (déc. 2002), p. 1056-1076. doi : 10.1109/TSE.2002.1049404.

[63] Alexandr Plyasovskikh et. “SCIENCE AND INNOVATION EXPERIMENT ON MEASURING THE OBSERVED RATE OF A MOVING CLOCK”. In : 2 (avr. 2023), p. 169-188. DOI : 10.5281/zenodo.7767737.

[64] J. Queille et Joseph Sifakis. “Specification and verification of concurrent systems in CESAR”. In : t. 137. Jan. 2006, p. 337-351. ISBN : 978-3-540-11494-9. DOI : 10.1007/3-540-11494-7_22.

[65] K. R. Rao et P. Yip. *Discrete cosine transform : algorithms, advantages, applications*. USA : Academic Press Professional, Inc., 1990. ISBN : 012580203X.

[66] Roberto Segala. “Modeling and Verification of Randomized Distributed Real - Time Systems”. In : (mars 2007).

[67] Roger Smith et al. “Component-Based Development ? Refining the Blueprint.” In : jan. 2000, p. 563-. DOI : 10.1109/TOOLS.2000.10079.

[68] Ganesh Sundaram et Saicharan Sathyam. *Modelling and Simulation of a Vehicle Powertrain and Anti-Lock Braking System*. Nov. 2017.

[69] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Jan. 2002. ISBN : 0-201-745572-0.

[70] Mohamed Tabaa. “Conception d’un système de transmission ultra-large bande par impulsions orthogonales”. Thèse de doct. Nov. 2014. DOI : 10.13140/RG.2.2.26283.23848.

[71] Ferucio Tiplea et Aurora Ţiplea. “Petri Net Reactive Modules”. In : *Theoretical Computer Science* 359 (août 2006). DOI : 10.1016/j.tcs.2006.02.001.

[72] Cherif Tolba, Philippe Thomas et Abdellah Moudni. “Approche multi-modèles pour la commande des feux de trafic”. In : *Sciences et Technologie de l’Automatique* 2 (avr. 2005). DOI : 10.3845/e-sta.2005.n2.

[73] Jiacun Wang et William Tepfenhart. “Petri Nets”. In : juin 2019, p. 201-243. ISBN : 9780429184185. DOI : 10.1201/9780429184185-8.

[74] J.V. Woods et al. “AMULET1 : an asynchronous ARM microprocessor”. In : *Computers, IEEE Transactions on* 46 (mai 1997), p. 385-398. DOI : 10.1109/12.588033.

[75] You Zhou et Chuan He. “A Review on Reliability of Integrated Electricity-Gas System”. In : *Energies* 15 (sept. 2022), p. 6815. DOI : 10.3390/en15186815.

[76] W.M. Zuberek. “Timed Petri nets definitions, properties, and applications”. In : *Microelectronics Reliability* 31.4 (1991), p. 627-644. ISSN : 0026-2714. DOI : https://doi.org/10.1016/0026-2714(91)90007-T. URL : https://www.sciencedirect.com/science/article/pii/002627149190007T.