THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

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

Discipline ou spécialité :

Sûreté Logiciel et Calcul Haute Performance

Présentée et soutenue par :
Nassima IZERROUKEN

le : mercredi 6 juillet 2011

Titre :

Développement prouvé de composants formels pour
un générateur de code embarqué critique pré-qualifié

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
IRIT

Directeur(s) de Thèse :
M. Patrick SALLE, Prof., INPT-ENSEEIHT, Directeur de thèse
M. Marc PANTEL, M.C., INPT-ENSEEIHT, Codirecteur de thèse

Rapporteurs :

Mme Sandrine BLAZY Prof., Université de Rennes
M. Jean-Pierre TALPIN, D.R., INRIA-Rennes

Membre(s) du jury :
M. Yamine AIT-AMEUR, Prof., Université de Poitiers, Examinateur
Mme Virginie WIELS, Ingénieur de recherche, ONERA-Toulouse, Examinateur
M. Xavier THIRIOUX, M.C., INPT-ENSEEIHT, Examinateur
M. Patrick SALLE, Prof., INPT-ENSEEIHT, Directeur de thèse
M. Marc PANTEL, M.C., INPT-ENSEEIHT, Codirecteur de thèse

Remerciements

Le travail présenté dans ce document a été réalisé à l'Institut de Recherche en Informatique de Toulouse, au sein de l'équipe Assistance à la certification d'Applications DIstribuées et Embarquées (ACADIE) et également à Continental Automotive dans le cadre du projet européen GeneAuto.

Je tiens à remercier en premier lieu mes chèrs encadrants Marc Pantel et Xavier Thirioux de m'avoir soutenue et guidée. Merci à Marc pour son humanité, sa sagesse, sa culture du domaine et la liberté qu'il m'a laissé pour guider ma recherche. Je le remercie d'avoir été disponible pour répondre aux nombreuses interrogations que je me posais de jour comme de nuit.

Merci à Xavier pour sa sympathie, son investissement pour m'initier au système Coq, sa gentillesse, sa disponibilité et son imaginaire débordant tant dans la recherche que dans la vie quotidienne.

Je remercie également Olivier Ssi Yan Kai, mon tuteur industriel à Continental, qui m'a soutenu dans les différentes réunions de GeneAuto. Merci à tous les membres du projet GeneAuto pour les échanges fructueux qu'on avaient eu.

Merci à Patrick Sallé qui a accepté d'être mon directeur de thèse.

Je remercie les membres du jury qui ont accepté d'évaluer mon travail :
- Virginie Wiels qui a présidé le jury ;
- Sandrine Blazy et Jean-Pierre Talpin, tous les deux rapporteurs de ma thèse, qui ont lu et évalué mon travail ;
- Jean-Louis Colaço et Yamine Ait-Ameur qui ont bien accepté d'examiner ma thèse ;

Ma thèse n'aurait jamais vu le jour sans le soutien et l'encouragement de plusieurs personnes.

Mes pensées affectueuses à mes amis et collègues : Hejer Rejeb qui m'est plus une soeur qu'une amie, Fatiha Bouabache qui est toujours là, ma cobureau Mounira Kezadri qui a supporté mes crises de stress, Philippe Queinnec (mon psy), fort sympathique, qui est toujours là pour m'écouter et me conseiller mais également pour partager avec moi un peu de chocolat, je n'oublierai pas les fous rires avec Sandrine Mouysset et ViTToria Moya qui ont égayé mes journées au labo ! Je remercie également Tanguy Le Berre et Nadège Pontisso pour les moments sympathiques que nous avons passés ensembles aux anciens bureaux. Merci à Andres Toom pour toutes les réponses qu'il m'a apporté lorsque Simulink se comporte "bizarrement" !

Un grand merci à Marcel Gandriau avec qui je me fait un plaisir d'enseigner le module «traduction des langages». Merci également à tous les membres ACADIE, notamment Philippe Mauran, Xavier Crégut, Christiane Massoutié et Aurélie Hurault pour leur sympathie.

Merci à l'équipe administrative et technique de l'IRIT-ENSEEIHT pour leur sympathie, je pense particulièrement à Sylvie Eichen et Sylvie Armengaud. Vous êtes les meilleures secrétaires !

Je remercie également André-Luc Beylot le directeur de l'IRIT-ENSEEIHT, Emmanuel Chaput et Riadh Dhaou pour leur sympathie.

Un remerciement particulier à mon amie Pascaline Parisot qui m'a soutenue du début de la thèse jusqu'à la fin, de m'avoir lu et corrigé le manuscrit et d'avoir été présente dans toutes les circonstances ...

Je souhaiterai remercier plusieurs personnes du coté de ma famille.

Je commence par tonton Moahand Tayeb Belarif qui sans lui je n'aurais rien fait de tout cela. You're the best ;)

Je ne remercierai jamais assez mes chers parents qui m'ont toujours fait confiance et ont cru en moi. Merci papa et merci maman pour tout ce que vous m'avez donné. Vous avez tout fais pour moi, je vous dois ce que je suis et je vous dois mon devenir.

Je remercie également mes frères Amine et Salah pour leur encouragement incessant et pour les blagues qui me distraient lorsque tout n'est pas rose.

Une grande pensée à la famille Izerrouken, Belarif et Kacimi.

Une immense dédicace à la plus belle ville du monde : Oran, la ville qui m'a bercé et qui m'a tant donné !

Enfin et non finalement, je remercie mon cher et tendre époux de m'avoir encouragé, soutenu et d'avoir supporté mon stress et surtout mon caractère ! Encore une fois merci.

Je dédie cette thèse à mon ange Louna qui a éclairé mon obscurité ...

Table des matières

Liste des figures

Liste des Algorithmes

Première partie

Introduction

Introduction

1

1.1 Introduction

De nos jours, les logiciels prennent une place de plus en plus importante dans la plupart des composants des systèmes complexes tels un avion, une voiture, un appareil de radiologie, etc. Ils sont désignés comme *embarqués* pour caractériser le lien essentiel entre le matériel et le logiciel. Dans ces systèmes, le logiciel n'est pas l'objectif principal du système mais il joue un rôle fondamental dans la réalisation de cet objectif. Les systèmes embarqués sont de plus en plus à l'origine de l'innovation dans le domaine des systèmes critiques et notamment dans l'aéronautique, le spatial et l'automobile. L'adjectif *critique* signifie qu'un dysfonctionnement peut entraîner des conséquences dramatiques pour des vies humaines, des dégâts matériels importants, des pertes économiques substantielles ou encore des conséquences graves pour l'environnement. L'évolution de la complexité intrinsèque des systèmes critiques est exponentielle depuis de nombreuses années. La loi de Moore [Moo98] en est la caractéristique la plus célèbre. Ces systèmes ont donc des coûts de développement et de maintenance de plus en plus élevés.

L'ingénierie dirigée par les modèles (*Model Driven Engineering*) permet de réduire leur coût de développement. Il s'agit de vérifier et de valider les exigences du système et les intermédiaires dans le développement en s'appuyant sur des modèles (abstractions) du système. Cela permet de détecter les anomalies le plus tôt possible, puis d'exploiter la génération automatique de code à partir de ces modèles afin de réduire le nombre d'anomalies introduites par le codage manuel et d'accélérer la production du système final. Les modèles deviennent ainsi l'élément central du développement.

Compilateur et générateur de code

Usuellement, le terme *générateur de code* est utilisé pour désigner un outil informatique qui traduit un modèle en programme d'un langage de haut niveau. Tandis que le terme *compilateur* est utilisé pour désigner un outil informatique qui traduit un programme, d'un langage de programmation de plus ou moins haut niveau, en un code exécutable en langage machine.

Étant donné la complexité et la criticité des systèmes embarqués, une assurance élevée est requise pour le code produit par les générateurs de code. Les méthodes de vérification les plus répandues dans le monde industriel consistent à relire et à tester rigoureusement les modèles d'entrée et/ou le code généré. Cela permet d'atteindre des objectifs de couverture du comportement du système par des tests basés sur les exigences du système et sur la structure du code source.

Dans le cadre de la génération automatique de code pour un modèle validé et vérifié par simulation, c'est-à-dire des tests au niveau des modèles, il est

indispensable que la vérification du code généré assure qu'aucune altération de la qualité du produit final ou, pire encore, un dysfonctionnement ne se produise. Or, les activités de vérification à base de tests sont coûteuses et non exhaustives.

Les modèles couramment manipulés pour la conception des systèmes embarqués critiques, dans le monde industriel, sont conçus avec des langages de modélisation et de simulation sophistiqués tels que Simulink/Stateflow [Sim], Scade [Dor08], Scicos [CCN09], etc.

Afin de réduire les coûts de tests du système final et les anomalies qui peuvent être introduites par la génération de code, il est indispensable d'exploiter des techniques plus rigoureuses, basées sur des formalismes mathématiques, pour développer les générateurs de code.

La vérification, telle qu'elle est définie par l'IEEE [IEE05], comprend toutes les techniques appropriées pour montrer qu'un système satisfait sa spécification. La spécification est l'ensemble des exigences concernant les besoins des utilisateurs et leurs attentes par rapport au système concerné.

L'activité de test assure, pour une situation particulière (un scénario de test), qu'un système a un comportement conforme à sa spécification. Un scénario de test est donc une forme de spécification du comportement attendu. Or, toute nouvelle entrée entraîne un changement du comportement du système informatique en question et requiert donc un nouveau scénario. Il est donc difficile d'assurer que la spécification du système sous la forme de scénarios et la couverture de ces scénarios par rapport au système final sont complètes.

De ce point de vue, les méthodes formelles sont plus avantageuses que ce soit pour la spécification ou la vérification. En effet, il est possible d'assurer formellement que tous les détails sur le fonctionnement d'un système sont couverts par les activités de spécification et de vérification, et donc que le système devrait fonctionner correctement pour toute entrée. Tandis qu'un oubli ou une ambiguïté de spécification et de vérification sont très courants lorsqu'il s'agit d'une argumentation sur papier.

Prouver la correction d'un programme ne correspond pas à la recherche et à la correction des erreurs qui est une action propre à la mise au point (debugging). Il s'agit plutôt de faire appel à des méthodes rigoureuses fondées sur les mathématiques, appelées *méthodes formelles*. Elles permettent de spécifier le programme qui doit être réalisé et de prouver que l'implantation fournie respecte toutes les propriétés décrites par sa spécification. Un tel programme est dit correct par construction : il n'est pas possible de construire un système incorrect.

Les méthodes formelles apportent une assurance très élevée aux systèmes informatiques. Toutefois, elles demandent un coût de mise en œuvre et de maintenance élevé. En conséquence, celles-ci ne sont pour l'instant préconisées que pour le développement des systèmes les plus critiques pour lesquels elles apportent une assurance très élevée et permettent de réduire le coût également très important des tests unitaires permettant d'obtenir les degrés de couverture exigé par certains standards. Des résultats relativement récents sur leur application pour des systèmes critiques ont montré à la fois la qualité des résultats et leur applicabilité pour des systèmes de grande taille. Nous citons ici les travaux de [BCC+03, CCF+05] qui appliquent la technique d'analyse statique par interprétation abstraite [CC77, CC92] sur des programmes C issues des applications de l'avionique. Ces travaux montrent bien l'intégration des méthodes formelles dans le processus de développement des systèmes embarqués critiques.

Les techniques de vérification sont dites statiques si elles n'ont pas besoin

d'exécuter le programme à analyser. Parmi ces techniques, les plus connues sont la vérification des modèles (*model-checking*) [CGP99], l'analyse statique par interprétation abstraite et la preuve de programmes. Le point commun à la vérification des modèles et à l'analyse statique par interprétation abstraite est qu'elles se basent sur une abstraction du programme à analyser. Soit l'utilisateur construit l'abstraction dans le cadre de la vérification de modèles, soit l'outil la construit automatiquement dans le cadre de l'analyse statique et plus précisément de l'interprétation abstraite. Comme le générateur de code est un programme complexe, il est difficile d'appliquer ces deux techniques directement au générateur de code, notamment si ce dernier ne dispose pas de spécification, formelle ou pas, de la sémantique d'exécution des langages source et cible.

Dans le cadre de preuves de programmes, le développement correct par construction en utilisant un assistant de preuve a remporté un succès significatif à travers le projet Compcert [BDL06, Ler06, Ler09]. Celui-ci consiste à développer un compilateur C, correct par construction, en utilisant l'assistant de preuve Coq [BC04].

Un assistant de preuve en informatique, selon la définition de Wikipedia, est un logiciel permettant l'écriture et la vérification de preuves mathématiques, soit pour des théorèmes au sens usuel des mathématiques, soit pour des assertions relatives à l'exécution de programmes informatiques. Les assistants de preuves trouvent leurs origines dans les projets LCF [Mil79] et AutoMath [dB70]. Ils s'appuient sur l'isomorphisme de Curry-Howard qui associe types et formules logiques, programmes et preuves. Ceux-ci permettent de formaliser des théories qui sont ensuite exploitées pour prouver des programmes, par exemple en s'appuyant sur la logique de Floyd-Hoare [Hoa69, Flo67] pour les programmes impératifs. Il existe plusieurs assistants de preuves qui ont été utilisés pour spécifier et vérifier des systèmes réels tels Coq [BC04], Isabelle [Pau89], Pvs [OSR95] ...

Dans le monde des systèmes critiques industriels, la confiance en ces systèmes s'exprime par des normes de certification ou de qualification qui contraignent leur développement et leur utilisation. Ces dernières dépendent du domaine d'application, par exemple, la norme DO-178B [RTC99] pour le développement logiciel dans le domaine de l'aéronautique, la norme ISO 26262 pour le développement logiciel dans le domaine de l'automobile qui est, elle-même, issue de la norme générique IEC 61508 [Gro01]. Les dernières versions des normes de certification prennent en compte l'utilisation des méthodes formelles, du développement à base de modèles et la nécessité de qualifier des outils, pour le développement de systèmes critiques. Par exemple, un supplément a été réservé pour chacune de ces technologies dans la future version C de la norme DO-178.

Le projet GeneAuto

Les travaux réalisés au cours de cette thèse ont été appliqués dans le cadre du projet GeneAuto [1]. Ce dernier consiste à développer un générateur de code qualifiable, selon la norme de certification DO-178B/ED-12B, qui transforme des circuits Simulink/Stateflow et Scicos en programmes C. Le générateur de code est constitué de plusieurs modules, appelés outils élémentaires, reliés

[1] http ://gforge.enseeiht.fr/projects/geneauto

en cascade comme le montre la figure 1.1. Cette figure montre également les parties qui seront étudiées dans le présent manuscrit.

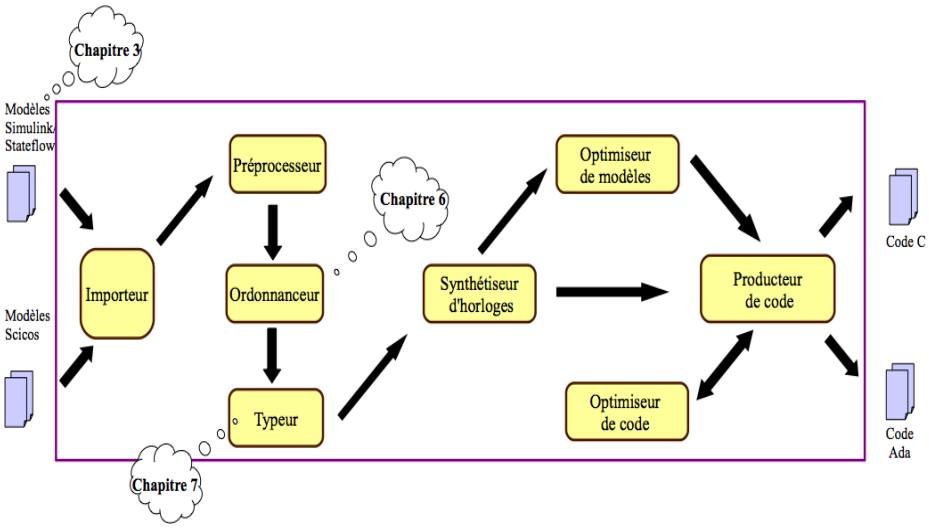


FIG. 1.1 – *Architecture du générateur de code* GENEAUTO

Nous nous plaçons dans le cadre du développement correct par construction d'outils élémentaires (ordonnanceur et typeur) du générateur de code. Notre objectif, dans le projet GENEAUTO, était d'expérimenter le développement correct par construction en prenant en compte les contraintes de qualification. Une étude des différentes approches possibles a conduit au choix de l'assistant de preuve COQ [BC04], dans le cadre de systèmes critiques de grande taille, en s'inspirant des travaux de [BDL06, Ler06, Ler09].

Plusieurs raisons motivent l'intérêt de notre travail. D'une part, la vérification utilisée par l'ensemble des générateurs de code existants dans le monde industriel repose sur le respect d'un processus de développement très précis mais qui n'est formalisé qu'en langage naturel, sur des activités de vérification par relecture indépendante et par tests, avec des contraintes particulières sur la couverture de tests par rapport aux exigences et aux sources des programmes. Or, aussi poussée que soit leur couverture, les tests ne garantissent pas l'absence d'erreurs [Mye79]. Par exemple, le processus de développement du générateur de code de MathWorks, Real Time Workshop Embedded Coder[2], propose une qualification compatible avec la norme ISO 26262, en s'appuyant sur une vérification à travers la génération de tests à partir du modèle qui assure que le code généré a le même comportement que le modèle simulé. Un autre exemple, le générateur de code qualifié SCADE KCG [Dor08] accepte en entrée des modèles en SCADE (langage de conception dérivé du formalisme Airbus SAO semblable à SIMULINK), avec une sémantique synchrone issue de LUSTRE [HCRP91] et génère du code C. La qualification de KCG exploite une vérification principalement basée sur le respect d'un processus compatible avec le standard DO-178B et sur une couverture des tests de type MC/DC (*Modified Condition/Decision Coverage*) adaptée au langage OCAML [PAC+08].

D'autre part, l'assistant de preuve COQ permet de spécifier formellement les exigences pour le générateur de code, puis d'écrire des programmes fonc-

[2]http ://www.mathworks.com/products/embedded-coder/

tionnels dans le langage de spécification (*Gallina*), et enfin de prouver directement dans ce langage que les fonctions satisfont les exigences. Le mécanisme d'extraction de Coq produit automatiquement du code OCaml, Haskell ou Scheme à partir des spécifications. La spécification et la vérification réalisées avec le même langage permettent de produire des programmes corrects par construction, qui couvrent toutes les exigences spécifiées sans risque d'oubli ou d'ambiguïté.

Pour autoriser l'élimination des tests unitaires sur le code produit, le standard DO-178B exige que tout outil utilisé dans le processus de développement d'un système, qui peut introduire des erreurs dans le produit final, soit qualifié avec la même rigueur que le système produit. Afin de supprimer le risque que l'extracteur Coq puisse introduire des erreurs dans le programme vérifié en Coq, des travaux sur la vérification de l'extracteur de Coq sont nécessaires, il devrait être possible de s'appuyer sur les travaux suivants [Glo09].

Contributions

De manière succincte, les contributions apportées par les travaux réalisés au cours de cette thèse sont résumées par les points suivants.

- Un processus de développement de générateurs de code qualifiés, combinant l'approche classique et l'approche formelle en utilisant l'assistant de preuve Coq, a été défini. Ce processus a été mis en pratique sur deux outils élémentaires de l'atelier GeneAuto en charge de l'ordonnancement et du typage des blocs dans les diagrammes fonctionnels. La spécification du générateur de code est écrite de manière semi-formelle en langage naturel par les utilisateurs partenaires du projet. Les exigences en langage naturel, pour les outils élémentaires développés dans le cadre de cette thèse, sont traduites en spécifications formelles exprimées dans le langage de spécification Gallina ;
- Un cadre formel générique a été défini pour le développement des outils élémentaires du générateur de code. Ce cadre représente la conception des outils élémentaires. Inspiré des principes de l'analyse statique par interprétation abstraite, il repose sur des ensembles partiellement ordonnés, des treillis de profondeur finie et sur le calcul du point fixe. Ce cadre générique a été appliqué aux deux cas d'étude pour le calcul des dépendances des blocs exploités pour l'ordonnancement du circuit et l'inférence de type des ports des blocs en avant et en arrière pour la vérification du typage du circuit. Chaque outil élémentaire est une instanciation du cadre formel et repose sur un algorithme dédié à l'analyse effectuée par chaque outil élémentaire ;
- Plusieurs formes d'algorithmes ont été proposés pour ordonnancer les blocs des circuits considérés. Une première version est dédiée aux circuits ne contenant que des flots de données. Elle repose sur un calcul de rangs qui impose un ordre préalable plus contraignant que les exigences initiales. Une seconde version, qui repose sur le calcul des dépendances des blocs, a été proposée pour respecter exactement les exigences dans le cas des circuits ne comportant que des flots de données. Elle a ensuite été étendue aux dépendances événementielles pour ordonnancer tout type de circuit combinant les flots de données et de contrôle ;
- Un algorithme de vérification des types des circuits considérés a été développé. Il s'appuie sur deux inférences de type : une inférence de type

depuis les entrées vers les sorties du circuit et une inférence dans le sens inverse. Ceci permet de calculer une borne supérieure et inférieure pour les types autorisés pour chaque port des blocs du circuit ;
- Le code source OCaml a été extrait pour les outils élémentaires développés, préservant les propriétés prouvées dans la spécification Coq. Le code extrait a ensuite été intégré dans la chaîne de développement du générateur de code à l'aide d'interfaces simplifiées qui réduisent les échanges au minimum nécessaire pour simplifier la vérification des entrées/sorties ;
- Contrairement aux générateurs de code industriels existants, GeneAuto comporte des modules (outils élémentaires) entièrement spécifiés et vérifiés formellement pour lesquels les tests unitaires ne sont plus nécessaires. Par contre, les tests fonctionnels, qui permettent de valider les exigences, et les tests d'intégration, qui permettent de valider l'architecture choisie, restent partiellement nécessaires car les spécifications formelles n'expriment pas toutes les propriétés considérées ;
- L'approche proposée a été présentée aux autorités de certification pour valider les choix technologiques effectués. Celles-ci ont donné un avis positif à l'égard du processus de développement et de l'intégration des méthodes formelles proposées dans GeneAuto. Celui-ci permet de poursuivre ces expérimentations dans le but de développer d'autres modules pour obtenir un outil entièrement spécifié et vérifié en exploitant des méthodes formelles.

Les enjeux de ces travaux ne se limitent pas au développement d'outils élémentaires dans GeneAuto. Nous avons montré à travers cette contribution la faisabilité de l'utilisation des méthodes formelles pour le développement d'outils qualifiés selon le standard DO-178B et leur intégration dans le processus de développement de systèmes embarqués critiques de grande taille.

De plus, la traduction des spécifications informelles dans le langage de spécification de Coq a mis en évidence de nombreuses ambiguïtés et plusieurs lacunes au niveau de la spécification écrite en langage naturel. Cette spécification est écrite par les partenaires industriels experts des outils de génération de code pour les systèmes critiques et est validée par une relecture indépendante effectuée par les autres partenaires industriels du projet. Ce constat s'appuie sur la formalisation des exigences et sur les tentatives de preuves établies pour vérifier la cohérences de ces exigences, d'où tout l'intérêt de l'utilisation d'un langage formel comme Gallina.

En particulier, le développement formel à l'aide de Coq a conduit à une clarification des mécanismes d'ordonnancement des blocs Simulink qui font partie de la sémantique d'exécution du langage, par rapport aux documents rédigés dans GeneAuto et aux ouvrages de référence sur le langage Simulink.

1.2 Travaux connexes

Maulik A. Dave répertorie dans une synthèse bibliographique [Dav03] les travaux sur la vérification formelle de compilateurs et d'étapes de compilation. Nous nous intéressons plus particulièrement à la vérification et à la qualification des générateurs de code.

Il existe de nombreux travaux sur la génération de code à partir de modèles et sur la vérification de générateurs de code pour des systèmes critiques. Une grande partie de ces travaux sont dédiés aux langages synchrones tels Scade/KCG/Lustre [BCHP08, CHP07, CHP06, CP04]. La sémantique des mo-

dèles supportés par GeneAuto est significativement différente, car elle combine de manière fine les flots de données et de contrôle. En Simulink, la sémantique découle de l'ordonnancement des blocs contrairement aux langages synchrones où l'ordonnancement des blocs découle de la sémantique des flots de données des circuits. Des travaux existent pour transformer les circuits Simulink en Lustre et exploiter ainsi le générateur de code Scade/KCG/Lustre qualifié [CCM+03, PAA+03]. Cependant, les travaux cités ne nous permettent pas de respecter facilement les contraintes de traçabilité imposées sur le circuit et/ou sur le code par les partenaires industriels de GeneAuto pour assurer la couverture structurelle MC/DC des tests unitaires du code source généré imposée par le DO-178B. La sémantique et les aspects de la génération de code sont traités dans [CCM+03, PAA+03, bGJ91, HRR91]. Outre la difficulté d'établir la traçabilité, il n'y a pas de vérification formelle du générateur de code lui-même. Une grande importance est portée à la vérification formelle du code généré dans [BBF+00]. Ces travaux ont été appliqués avec succès aux systèmes critiques au sein de Dassault Aviation. Néanmoins, la vérification formelle a été appliquée aux programmes sources générés et pas au générateur lui-même. [Ler06, BDL06] proposent une approche prometteuse dans le développement correct par construction de compilateurs. Nos travaux sont inspirés de ceux-ci mais notre approche est relativement différente. Techniquement, les méthodes formelles utilisées pour prouver la correction de compilateurs sont habituellement basées sur la spécification formelle des sémantiques des langages source et cible, de la traduction et de la preuve d'équivalence entre les sémantiques observées des codes source et cible. Dans le cadre de GeneAuto, nous avons choisi de ne pas nous éloigner du processus de qualification usuel afin de faciliter son acceptation par les autorités de certification, donc, nous ne nous sommes pas basés directement sur le niveau sémantique mais sur les exigences en langage naturel telles qu'elles sont écrites habituellement par les partenaires industriels du projet.

Visite guidée du manuscrit

Nous présentons ici un bref résumé de chaque chapitre de ce document. Nous nous focalisons dans ce manuscrit sur le processus de développement combinant méthodes formelles et classiques pour la construction de générateurs de code qualifiables, sur la démarche suivie pour la définition d'un cadre général pour la construction de modules d'analyse semblables, ainsi que sur les outils élémentaires développés pour l'ordonnancement et le typage des circuits Simulink. Nous ne présentons pas exhaustivement toutes les preuves réalisées dans le cadre de l'utilisation de Coq pour la spécification et la vérification formelle de ces outils. Nous avons choisi de présenter les différentes spécifications et de décrire les preuves mathématiquement afin d'expliquer le principe. Cependant, nous avons décrit la majeure partie des spécifications et certaines preuves en Coq lorsque cela nous a semblé pertinent. L'ensemble du développement est disponible à l'adresse `http://izerrouken.perso.enseeiht.fr/Code`.

Chapitre 2 : État de l'art

Ce chapitre présente un aperçu des méthodes de vérification existantes ainsi que quelques éléments issus des standards de certification des systèmes critiques. La méthode de vérification la plus répandue dans le milieu industriel

est le test. Les normes de certification, entre autres la norme DO-178B pour le logiciel en aéronautique, incluent explicitement les tests comme une exigence pour les processus de vérification. Néanmoins, une future norme est en cours d'étude, la norme DO-178C, qui permet l'exploitation des méthodes formelles dans le processus. Comme cité précédemment, la vérification par tests ne peut garantir l'absence d'erreurs dans le système vérifié. D'autres approches plus rigoureuses, fondées sur les mathématiques, sont indispensables pour les systèmes embarqués critiques. Nous présentons les principes de ces approches et les comparons par rapport à nos besoins afin d'expliquer les choix effectués.

Chapitre 3 : Éléments de mise en œuvre

Le générateur de code GeneAuto prend en entrée des circuits Simulink/Stateflow. Nous introduirons donc, dans ce chapitre, le langage de modélisation et de simulation Simulink ainsi que Stateflow. Il y sera également présenté le principe de l'ordonnancement et du typage des circuits en Simulink à travers des exemples. Nous présenterons les différentes contraintes pour l'ordonnancement des circuits Simulink. En effet, l'ordonnancement dépend de la nature des blocs et des signaux qui les relient. En ce qui concerne le typage, Simulink peut avoir plusieurs formes de typage en fonction de la configuration choisie. Cela implique des choix de typage, selon la forme retenue, pour les circuits Simulink lors de la spécification de l'outil élémentaire de typage.

Chapitre 4 : Processus de développement d'outils élémentaires

Ce chapitre décrit le processus de développement tel qu'il est préconisé par la norme de certification DO-178B [RTC99]. Le processus de développement comprend des plans détaillés de développement, de vérification et de qualification. Les outils élémentaires développés en Coq dans le cadre de cette thèse sont intégrés dans la chaîne de développement classique. Dès lors, il est important de montrer les deux facettes du processus de construction d'un générateur de code qualifié (développement, vérification et qualification) : avec des outils classiques comme Java et avec des technologies formelles comme Coq.

Chapitre 5 : Cadre formel de développement

Ce chapitre présente la conception des outils élémentaires en Coq. Le but est d'exploiter les mêmes exigences pour les deux types de développement : en Java et en Coq. La sémantique concrète n'apparaît donc pas explicitement dans les exigences des outils élémentaires de GeneAuto. Nous nous sommes donc limités à la sémantique abstraite déduite des exigences pour construire les composants d'analyse statique.

Nous nous sommes inspirés du principe de l'analyse statique par interprétation abstraite [CC77, CC79, CC92] pour définir un cadre formel commun aux outils d'ordonnancement et de typage. Il s'agit de s'appuyer sur les ensembles ordonnés, les treillis de profondeur finie et le calcul de point fixe en appliquant le théorème de Kleene [Col52] pour calculer une abstraction de toutes les exécutions possibles d'un circuit. Dans les travaux présentés dans cette thèse, nous définissons directement les domaines abstraits. Il serait certes plus satisfaisant de disposer de la sémantique concrète et des sémantiques abstraites des outils élémentaires afin de vérifier leur équivalence (correspondance de Galois).

Toutefois, nous ne disposons pas de la sémantique concrète. L'objectif était d'exploiter des exigences communes aux deux formes de développement.

Le cadre est composé d'un domaine de calcul, d'une relation d'ordre partiel, d'une famille de treillis, d'un environnement de calcul, d'une fonction de propagation et d'une implantation de l'algorithme de Kleene, paramétré par ce domaine et cette fonction, qui calcule le résultat attendu pour chaque instanciation du cadre dans un outil élémentaire. Nous spécifions donc chacun de ces éléments en illustrant leur implantation en Coq. À la fin de ce chapitre, nous présenterons la signature de la conception d'un outil élémentaire qui sera ensuite utilisée pour définir les outils élémentaires étudiés.

Chapitre 6 : Ordonnancement des circuits Simulink

Ce chapitre présente plusieurs versions de l'algorithme d'ordonnancement des blocs en Simulink ainsi que les preuves de leurs propriétés respectives. Le premier algorithme, une version préliminaire présentée dans [ITPS08], calcule un entier naturel, le rang, pour chaque bloc d'un circuit Simulink comprenant uniquement des flots de données. Plusieurs blocs peuvent posséder le même rang. Le concepteur du circuit peut définir des priorités pour les blocs. Un ordre total est alors calculé en triant les blocs possédant le même rang par ordre lexicographique de leurs priorités. Toutefois, cet algorithme calcule un rang entier qui forme un préordre total, et donc impose un ordre pour des blocs indépendants. En conséquence, l'ordonnancement construit est plus contraignant que les exigences ne le demandent. Celles-ci peuvent être satisfaites en construisant un ordre partiel issu de l'ensemble des dépendances de chaque bloc, c'est-à-dire, de l'ensemble des blocs qui doivent avoir été exécutés pour qu'il puisse l'être. Nous avons proposé, dans un premier temps, un algorithme qui calcule les dépendances de chaque bloc. Cette version reste adaptée uniquement aux circuits ne contenant que des flots de données. En effet, les flots de contrôle nécessitent un traitement particulier car les dépendances d'un bloc changent selon qu'il contrôle un autre bloc ou qu'il soit lui-même contrôlé, mais aussi selon son étape dans l'exécution (début ou fin). Un troisième algorithme est alors proposé pour prendre en compte les circuits combinant des flots de données et de contrôle. Cet algorithme calcule les dépendances en exploitant des événements de début et de fin d'exécution des blocs. Cela a permis de prendre en compte tout type de circuit Simulink. Ces algorithmes ont été validés sur plusieurs cas d'étude dont la détection du cliquetis dans le domaine automobile qui sera présenté en fin de ce chapitre.

Chapitre 7 : Typage des circuits Simulink

Il n'existe pas un unique mécanisme de typage pour un circuit en Simulink, mais plusieurs mécanismes sont possibles en fonction de la configuration de Simulink. De plus, il est très difficile, même pour une configuration donnée, de définir un système de type. En effet, certains blocs possèdent une forme de transtypage interne : les blocs arithmétiques peuvent accepter des entrées booléennes traduites en réels (faux par 0 et vrai par 1), ou encore des blocs logiques qui acceptent des entrées numériques. Afin de pallier certains de ces comportements, GeneAuto a mis en place un mécanisme de sous-typage plus contraignant. Par exemple, il n'est plus possible d'additionner un entier et un booléen ou de calculer la conjonction de deux valeurs numériques. Pour maîtriser les types d'un circuit, le concepteur de ce dernier doit définir explicitement

le type de chaque port des blocs. Dans la version classique de l'outil élémentaire de typage, chaque nouvelle initialisation des types requiert de ré-effectuer l'inférence de type. De plus, cet outil n'effectue que l'inférence des types en avant (depuis les entrées vers les sorties du circuit), et ne prend en compte que les circuits préalablement ordonnancés.

Nous proposons une solution plus générale qui permet de typer des circuits SIMULINK indépendamment de l'ordre choisi. L'approche laisse le libre choix au concepteur de spécifier ou non les types des ports internes au circuit et calcule pour chaque port la borne inférieure et supérieure des types que ce port peut prendre, pour que le port d'un bloc soit bien typé. Cet intervalle contient la ou les solutions pour que le circuit soit bien typé. L'approche repose sur deux inférences de type : *en avant* depuis les entrées vers les sorties, et *en arrière* depuis les sorties vers les entrées. L'intervalle des types calculés par cet algorithme n'est pas recalculé à chaque initialisation des ports autres que les ports d'entrée et de sortie d'un circuit, ou lorsque cette initialisation est comprise dans cet intervalle. Nous avons défini deux fonctions de typage pour chaque catégorie de blocs : une fonction pour l'inférence en avant et une pour l'inférence en arrière. Les deux fonctions doivent être cohérentes et satisfaire la propriété d'adjonction. Cet algorithme a été expérimenté sur des circuits comprenant une dizaine de blocs de base (blocs `Add`, `Divide`, `Logical Operator`, `In`, `Constant`, `Out` et `Unit Delay`).

Chapitre 8 : Conclusion & Perspectives

Ce chapitre clôt le manuscrit en récapitulant le bilan des travaux réalisés et des choix effectués au cours de cette thèse. Il décrit enfin les perspectives de recherche ouvertes.

Annexe A : Liste des blocs étudiés

Dans cette annexe, nous listons les blocs étudiés dans GENEAUTO, ainsi que leur description.

Annexe B : Code Coq

Cette annexe recense les fichiers coq décrivant le développement effectué pour les travaux présentés dans ce manuscrit.

État de l'art

2

2.1 Introduction

Les logiciels jouent actuellement un rôle prépondérant dans la plupart des systèmes complexes, notamment dans les domaines aéronautique, automobile, spatial et médical. Ils sont qualifiés de critiques si une défaillance peut entraîner des conséquences graves sur la vie humaine, des dégâts matériels, environnementaux, voire des pertes financières importantes. Des normes de certification ou de sécurité ont alors été définies pour encadrer, juridiquement et contractuellement, les systèmes et leur développement.

Dans ce chapitre, nous citons quelques normes de certification et énumérons les différentes techniques de vérification. Nous donnons dans un premier temps quelques définitions de base.

La vérification et la validation d'un système informatique sont des phases essentielles dans le cycle de vie du logiciel.
La **vérification** consiste à assurer qu'un système informatique respecte sa spécification. La **spécification** est l'activité qui décrit explicitement les besoins des utilisateurs. Alors que la **validation** consiste à assurer que le système informatique répond aux attentes des utilisateurs, autrement dit, aux besoins explicites et implicites de ces derniers. Il s'agit d'activités conduites en partie par l'utilisateur pour s'assurer que le système a le comportement attendu par rapport au domaine d'exploitation. Les activités de vérification sont également conduites partiellement par le développeur par rapport aux documents de spécifications des exigences.
Certification et qualification : L'objectif de la certification et de la qualification d'un système informatique est de s'assurer du respect des standards en vigueur. La certification concerne un système réel (matériel et logiciel) qui tourne dans son environnement opératoire. Tandis que la qualification est liée au processus de mise en place des outils qui servent à développer un système réel. Par abus de langage, en pratique, les deux termes sont souvent confondus. Dans le domaine de l'aéronautique régi par le standard DO-178B, la certification concerne les systèmes et la qualification les outils utilisés pour développer ces systèmes. Le terme certification est également utilisé dans les méthodes formelles pour faire référence à l'utilisation de certificat de confiance par rapport à un système et son développement ou pour exprimer la rigueur d'une technique.

2.2 Aperçu des normes de certification

Afin d'accorder de la confiance à un système informatique, les autorités publiques et les organisations industrielles, notamment celles qui produisent des applications critiques nécessitant une assurance très élevée, ont défini des normes de qualité du logiciel et de son développement. Ces normes se présentent sous forme de documents en langage naturel décrivant les exigences qui doivent être respectées par le logiciel lui-même et son processus de développement.

Une norme de certification détermine un ensemble de règles à appliquer pour un niveau de certification ou de qualification visé. Les normes de certification conduisant la conception des systèmes critiques se basent actuellement principalement sur deux concepts : le cycle de vie du logiciel (processus de développement) et son niveau de criticité. Ce dernier détermine l'importance de l'impact engendré par une défaillance du système.

La norme de certification automobile ISO 26262 [ISO] est issue de la norme IEC 61508 [Gro01]. Les prescriptions de la norme ISO 26262 couvrent les attentes des concepteurs des systèmes embarqués, avec des règles pour l'analyse de sûreté aussi bien pour le logiciel que pour le matériel. Cette norme est également applicable dans des secteurs qui n'ont pas encore de norme de certification.

Le domaine spatial s'est également doté de la norme de certification Spatial ECSS [ECS]. Celle-ci concerne principalement les relations entre les clients et les fournisseurs pour des systèmes n'engageant pas la vie humaine. Lorsque la vie humaine est engagée, comme lors des missions comportant des passagers, ou pouvant impacter la vie d'autres personnes, telle que le lancement d'une fusée ou les vaisseaux cargo automatiques vers la station spatiale internationale, les normes issues de l'aéronautique sont principalement appliquées.

La norme ED-12B [Eur] a été développée en commun avec la norme DO-178B. Elles fixent ensemble les conditions de sécurité applicables aux logiciels critiques aéronautiques et déterminent les règles nécessaires pour la sécurité du logiciel embarqué critique. Toutefois, son application dans d'autres domaines est de plus en plus courante car elle est la plus avancée, en termes de sûreté et de sécurité, et la plus contraignante. La norme DO-178 précise principalement les contraintes de développement liées à l'obtention de la certification d'un logiciel en aéronautique et la qualification des outils de développement de ce logiciel. Cette norme dispose de 5 niveaux d'assurance (DAL : Design Assurance Level). Ces niveaux varient de A à E, du niveau de sûreté fonctionnelle le plus élevé au moins élevé. Par exemple, le DAL A est le niveau qui a le plus d'objectifs à tenir dans la norme DO-178, car un dysfonctionnement peut provoquer la destruction de l'avion et la mort des passagers.

La norme DO-178B [RTC99] a vu le jour suite à de nombreux soucis d'interprétation des exigences de la DO-178A. Un plan de vérification complet et rigoureux a été significativement renforcé dans cette version. C'est la norme de certification DO-178B qui a été choisie par le projet GeneAuto.

Les deux normes DO-178B et ISO 26262 partagent beaucoup de caractéristiques.

Prochainement, la version DO-178C remplacera la DO-178B. Cette future norme considère dans le processus de développement l'utilisation des méthodes formelles et de l'approche dirigée par les modèles qui s'intègrent de plus en plus dans les systèmes critiques. Elle propose également un supplé-

ment décrivant précisément les activités de qualification des outils utilisés pour le développement de systèmes critiques. Elle définit un Tool Qualification Level compris entre 1 et 5 qui décrit les exigences nécessaires en fonction du DAL et du type d'outil (générateur de code, automatisation des activités de vérification avec suppression ou pas d'autres activités de vérification en fonction des résultats de l'activité automatisée).

2.3 Taxonomie des techniques de vérification et de validation

Plusieurs techniques de vérification sont applicables sur les systèmes informatiques. Elles peuvent être classifiées en deux grandes catégories. La première concerne les techniques semi-formelles, c'est-à-dire que, malgré une définition rigoureuse, il n'est pas possible de raisonner mathématiquement sur les propriétés de la spécification (consistance, correction, etc.), car ces méthodes reposent sur le langage naturel ou sur des formalismes graphiques avec une sémantique mal définie ou peu précise. La seconde regroupe les méthodes qui font appel à la logique et aux mathématiques pour effectuer cette vérification.

Nous nous intéressons, dans les travaux présentés dans cette thèse, aux méthodes formelles et particulièrement, celles qui sont utilisées pour la vérification de compilateurs.

2.3.1 Techniques de vérification et de validation semi-formelle

Usuellement, plusieurs techniques de vérification et de validation sont exploitées au sein de l'industrie du logiciel, notamment celles qui reposent sur les tests.

2.3.1.1 Vérification par tests

Un test est une technique de validation et de vérification par exécution d'une partie ou de la totalité du logiciel (se référer par exemple à [Mye04]). En pratique, un test est effectué en deux phases : la *construction des tests* et l'*exécution* de ces tests. La construction des tests produit, manuellement ou automatiquement, une suite de tests composée d'un ensemble de *cas de test*. Un cas de test est un ensemble d'interactions avec le système décrivant les entrées transmises au système et les sorties attendues. En fonction de l'objectif des tests, ils peuvent être classés selon [Bei90] en :

- *Tests de conformité* dont le but est de s'assurer de la conformité d'un système par rapport à son modèle (abstraction du système) vis-à-vis des exigences définies ;
- *Tests de robustesse* dont l'objectif est d'analyser le comportement du système dans un environnement dont les conditions ne sont pas prévues par la spécification ;
- *Tests de performance* dont l'objectif est de mesurer des paramètres de performance du système ;
- *Tests d'interopérabilité* dont le but est d'évaluer la capacité des différents composants du système à échanger des informations.

Un cas de test doit comporter les valeurs des paramètres et des résultats attendus ainsi que les critères pour décider si le résultat est conforme à la cible attendue. Ces tests doivent être les plus exhaustifs possibles pour couvrir toutes

les exigences du système. Or, il n'est évidemment pas possible, dans le cas d'applications complexes où le nombre de cas possibles est en général exponentiel par rapport à la taille du système, d'effectuer un test complet car le nombre de cas à explorer est généralement grand et potentiellement infini.

Les tests peuvent également être classés en tests unitaire, d'intégration, fonctionnel et de déploiement.

Test unitaire : il s'agit d'un test qui vérifie un module indépendamment du reste du système, afin de s'assurer qu'il répond à ses spécifications. Ces dernières sont écrites dans le cadre de sa conception par raffinement des exigences fonctionnelles du système. Le test unitaire est considéré comme essentiel notamment dans les applications critiques. Lors de la modification du code, les tests unitaires doivent être rejoués pour vérifier qu'il n'y a pas eu de dysfonctionnement introduit par les modifications (test de non régression).

Test d'intégration : il s'agit d'un test qui se déroule après les tests unitaires, c'est-à-dire lorsque chaque module a été vérifié indépendamment. Les tests d'intégration ont pour but de vérifier que toutes les parties développées indépendamment fonctionnent bien ensemble de façon cohérente vis-à-vis des exigences écrites dans le cadre de la conception.

Test fonctionnel : il s'agit d'un test qui vérifie l'adéquation du système aux contraintes définies dans les spécifications. Les tests fonctionnels évaluent la réaction du logiciel par rapport aux données d'entrée.

Test de déploiement : il s'agit d'un test qui analyse le comportement du logiciel une fois mis en place dans l'environnement réel d'exploitation.

2.3.1.2 Vérification par relecture

La vérification par relecture consiste à relire les éléments qui constituent le système à vérifier par rapport aux documents des exigences. La relecture concerne la vérification de la correspondance entre ce qui est requis dans les documents et ce qui est effectivement réalisé dans le système. La relecture doit généralement être réalisée de manière indépendante, c'est-à-dire que les personnes qui font la relecture ne sont pas celles qui ont développé le système ou qui ont écrit les exigences.

2.3.2 Techniques de vérification formelle

Les méthodes de vérification formelle désignent un ensemble de méthodes fondées sur les mathématiques et la logique pour assurer qu'un système informatique est conforme à ses spécifications. Ces dernières doivent être décrites dans un langage défini mathématiquement (syntaxe et sémantique), nous en citons ici quelques-unes.

La vérification formelle des compilateurs ou des transformations de modèles (ou de programmes) est un domaine à part entière depuis le milieu des années 60 [Pai67]. L'ensemble de ces travaux sont répertoriés dans [Dav03].

Les méthodes formelles ont connu une grande maturité cette dernière décade. Il est possible de nos jours de spécifier et vérifier complètement un logiciel par des techniques formelles et non sur une abstraction du système. En outre, les microprocesseurs sont également de plus en plus vérifiés formellement. Cependant, la vérification formelle du code source ne garantit pas à elle seule la correction de la compilation. Or, le compilateur est également un programme

qui peut introduire des erreurs lors de la compilation en générant un code cible erroné à partir d'un programme source correct.

La vérification d'un compilateur consiste à s'assurer que ce dernier ne produise pas de code cible erroné à partir d'un programme source correct.

Plusieurs techniques de vérification formelle existent. Nous distinguons parmi elles quelques familles telles que : la vérification de modèles (*model-checking*), l'analyse statique et le développement prouvé.

2.3.2.1 Génération automatique de tests et oracles

Il existe plusieurs méthodes de génération automatique de tests, citons par exemple [FJJV96]. Au lieu d'écrire lui-même manuellement les tests, l'utilisateur final fournit un générateur de tests qui exploite une spécification formelle du système et un oracle pour décider si le résultat de l'application des tests est correct par rapport au résultat attendu. Comme indiqué précédemment, les tests générés ne peuvent pas être exhaustifs sauf pour des systèmes très simples.

2.3.2.2 Vérification à l'exécution

La vérification à l'exécution [CM04] détermine lors de l'exécution d'un système si celui-ci satisfait un ensemble d'exigences. Ces dernières sont utilisées pour représenter les comportements désirés du système ou, au contraire, pour mettre en évidence ceux qui peuvent entraîner une défaillance logicielle. Ces exigences sont souvent exprimées par une spécification formelle. Une représentation abstraite est extraite de l'exécution du système pour être transmise à un oracle qui détermine si la propriété considérée est satisfaite. La détection d'erreurs par la technique ne doit pas influencer l'exécution du système vérifié (non intrusive).

2.3.2.3 Vérification de modèles

Pour déterminer si un système informatique satisfait une propriété, la vérification des modèles [CGP99], ou *model-checking* en anglais, décide, via un algorithme, si un modèle (une abstraction) du système satisfait la formule logique représentant la propriété souhaitée. Il s'agit d'abord de construire une représentation symbolique ou réelle de l'espace des états possibles du système, ensuite de vérifier si la formule est satisfaite pour chaque état.

Quand le modèle représentant le système à vérifier est fini, l'exploration de son espace d'états peut être exhaustive. Toutefois, ce modèle se retrouve rapidement de taille importante, lorsque le système à vérifier est réaliste. Des techniques d'approximation ont été développées pour réduire cette explosion combinatoire des états. Par exemple, [HP00] propose d'extraire une représentation réduite du modèle. Cependant, le passage à l'échelle reste problématique. De plus, la vérification de modèles repose sur une abstraction du programme à vérifier. Même si le modèle du programme est complètement vérifié, aucune garantie n'existe pour assurer que l'implantation dérivée du modèle soit également vérifiée. Il faut donc vérifier l'implantation par rapport au modèle.

2.3.2.4 Preuve de programmes

La preuve de programmes consiste à vérifier, automatiquement ou de manière assistée, que les formules logiques combinant les spécifications et les exécu-

tions possibles du programme sont des théorèmes. Par exemple, la vérification déductive est fondée sur la logique de Hoare [Hoa69] et l'annotation du programme par les pré-conditions (exigences sur les arguments de fonctions), les post-conditions (garanties sur les résultats des fonctions) ainsi que les variants et invariants de boucles. Cette méthode vérifie alors que, pour chaque fonction, les pré-conditions impliquent les post-conditions, en s'appuyant sur les variants et invariants. De plus, les post-conditions doivent être satisfaites pour chaque appel de fonction. Elle s'appuie sur des générateurs d'obligations de preuve (par exemple WHY [FM07]), puis des outils de démonstration automatique, ou des assistants de preuve. Notons que le programme à analyser doit être complètement accessible (boîte blanche). Parmi les outils utilisant la preuve de programmes, et particulièrement la vérification déductive, nous citons, par exemple, FramaC [Fra] qui analyse des programmes C. Cependant, cette méthode requiert généralement de nombreuses annotations manuelles, notamment, lorsque le programme est de taille conséquente.

2.3.2.5 Analyse statique

L'automatisation de l'étude des comportements possibles d'un programme, notamment dans le cadre de l'optimisation lors de la compilation ou de la vérification de propriétés, a fait naître des techniques particulières telles que l'analyse statique. L'analyse statique d'un programme permet de prédire les comportements ou les valeurs possibles à l'exécution, à partir du code source sans exécuter celui-ci, d'où le terme statique par rapport au test qui est de l'analyse dynamique. En effet, l'étude sémantique de la représentation syntaxique d'un programme permet d'inférer des propriétés survenant à l'exécution mais sans aucune exécution du programme.

L'étude des propriétés des programmes informatiques est soumise à deux limitations : d'une part, la sémantique d'un programme n'est en général pas calculable de manière finie et d'autre part, le théorème de Rice montre que : "*Toute propriété non triviale (ni toujours vraie, ni toujours fausse) sur la sémantique d'un langage de programmation est indécidable*". L'interprétation abstraite permet dans certains cas de pallier ces limitations [CC77, Cou81, CC92] grâce à la définition d'une méthode de calcul approchée par sur-approximation de la sémantique.

L'idée fondamentale de l'interprétation abstraite [CC77] est de s'appuyer sur les résultats des mathématiques discrètes pour décrire et calculer les propriétés de programmes issues de leur sémantique d'exécution. Les mathématiques interviennent d'une part dans les propriétés des structures de treillis complets ou d'ordre partiel, et leur préservation par une fonction monotone ou les connexions de Galois, et, d'autre part, dans les théorèmes de points fixes [Tar55] ainsi que leurs versions constructives par Kleene.

L'analyse statique par interprétation abstraite [CC79] calcule une abstraction des propriétés du système réel dans un domaine abstrait. La correction de cette technique assure qu'une propriété satisfaite dans le domaine abstrait le sera également dans le système réel.

Plusieurs travaux fondés sur l'analyse statique ont vu le jour dans le domaine des applications critiques. Par exemple [BCC+02, BCC+03, CCF+05] appliquent le principe de l'analyse statique par interprétation abstraite sur des programmes C pour des systèmes critiques de niveau A en aéronautique. Cette technique est appliquée directement au programme source pour détecter les erreurs à l'exécution. Ces travaux ont montré la faisabilité de l'utilisation des mé-

thodes formelles dans le processus de développement des systèmes critiques. Cependant, il reste nécessaire de tenir compte de l'environnement à partir duquel le programme a été généré [Cou05].

2.3.2.6 Validation de la traduction

Cette technique a été introduite par Pnueli, dans le cadre des projets européens SafeAir I et II sur le développement de systèmes sûrs en aéronautique, et exploitée en s'appuyant sur différentes techniques formelles : la vérification de modèles [PSS98, Nec00, ZPFG02], l'interprétation abstraite [Riv04], ou les algorithmes spécifiques prouvés dans un assistant de preuve [TL08]. Cette technique consiste à comparer l'origine et le résultat de la traduction en s'appuyant sur des méthodes adaptées à la propriété de correction souhaitée. Cette comparaison peut être syntaxique (par exemple, une classe Java est produite pour chaque classe Uml) ou sémantique (le résultat de l'exécution est le même pour la sémantique du modèle et du code). La validation de la traduction a été développée pour détecter les erreurs des générateurs de code. En pratique, elle complète le compilateur ou le générateur de code par un module qui vérifie les propriétés de correction par une analyse statique. Cette dernière s'assure de la correspondance du programme source et du code généré.

La validation de la traduction a été utilisée dans la vérification de plusieurs systèmes industriels mais a un coût de vérification élevée. Cependant, quand le module de validation échoue, l'utilisateur n'a généralement aucun diagnostic sur son système : l'erreur peut être liée au modèle, au compilateur, au générateur de code ou même à l'outil de vérification. À ceci s'ajoute la nécessité de vérifier le module de validation lui-même. Par contre, le générateur de code lui-même est une boîte noire qui n'est pas contrainte dans son implantation par la technique de vérification.

Dans le cadre de GeneAuto, pour éviter ce problème, nous avons choisi de nous intéresser principalement à la vérification du générateur lui-même et non à des instances de génération.

2.3.2.7 Code auto-certifié

Le principe du code auto-certifié, ou *Proof Carrying Code* [NL97], consiste à produire en plus du code généré la preuve, appelée *certificat*, que ce code est correct. Celle-ci sera vérifiée indépendamment lors de l'utilisation de ce code. Cette technique a été appliquée aux systèmes nécessitant une garantie de sécurité, principalement dans le cas de code mobile (voir par exemple [BCG+07]). La preuve de correction peut être produite par l'utilisateur ou par le générateur. Cependant, la preuve fournie peut être incorrecte. Il s'agit ici, comme dans la validation de la traduction, d'une vérification lors de l'utilisation. De plus, il peut être plus coûteux de produire une preuve que de vérifier une propriété donnée. Le problème essentiel reste la possibilité d'un échec de vérification et de l'assistance à l'utilisateur dans ce cas. Dans notre cas, le générateur de code doit être une boîte blanche pour permettre la construction de la preuve conjointement à la génération du code.

2.3.2.8 Développement prouvé de compilateurs

La vérification est appliquée au compilateur lui-même. Il s'agit de spécifier formellement les exigences ainsi que l'implantation du compilateur et la preuve

que celle-ci satisfait ces exigences. Cette technique est ancienne et a donné de nombreux résultats concluants. Nous pouvons citer les travaux initiaux de Milner et Weyhrauch [MW72] avec LCF [Mil79] et la bibliographie relativement récente réalisée par Dave [Dav03].

Plusieurs travaux sur la vérification de compilateurs sont d'actualité. Nous citons par exemple, les travaux de [kN03, kN06] et de [Str02] sur la vérification d'un mini-compilateur JAVA en utilisant l'assistant de preuve Isabelle [Pau89]. De même, [Str05, LPP05] ont vérifié formellement un mini-compilateur C, en une seule passe en utilisant Isabelle.

Dans le cadre du développement d'un compilateur réaliste et applicable en industrie, nous citons les travaux récents de Leroy et al. [BDL06, Ler06, TL08]. Il s'agit d'un compilateur d'un sous-ensemble du langage C qui comporte les éléments nécessaires pour le développement d'applications embarquées critiques [BL09], et qui produit du binaire optimisé. Le compilateur a été développé directement avec l'assistant de preuve COQ. Pour simplifier la vérification du compilateur, celui-ci a été découpé en de nombreuses transformations élémentaires successives, chaque transformation est prouvée correcte en COQ. Le développement du compilateur C en COQ a permis d'assurer la préservation, dans le code machine, d'une certaine observation de la sémantique d'exécution (les événements d'entrée-sortie) et par conséquence des propriétés préalablement vérifiées sur le code source qui correspondent à cette observation. Cette approche est très prometteuse dans le domaine de la vérification formelle et particulièrement dans les applications critiques qui nécessitent plus d'assurance. Elle est en cours d'expérimentation au sein d'Airbus.

La plupart des approches de vérification formelle appliquées aux compilateurs s'appuient sur une définition de la sémantique des langages source et cible et sur la preuve de préservation de cette sémantique. Par exemple, Leroy et al. s'intéressent aux événements d'entrée/sortie avec l'environnement du programme et montrent que la cible en assembleur va effectuer les mêmes séquences d'entrée/sortie que le programme source en C.

2.4 APPROCHE CHOISIE DANS GENEAUTO

Nous nous sommes basés sur la technique de développement prouvé et de l'analyse statique pour développer certains modules du générateur de code GENEAUTO.

Cependant, nous ne nous sommes pas appuyés directement sur la sémantique des langages (d'entrée SIMULINK/STATEFLOW et SCICOS et de sortie C) mais sur des exigences écrites en langage naturel par les utilisateurs du générateur de code. Ceci permet d'avoir une approche cohérente pour tous les outils élémentaires du projet qu'ils soient développés avec ou sans méthodes formelles. Dans le développement formel, ces exigences ne sont pas exploitables dans leur état, mais doivent être raffinées avant de les transcrire et les utiliser. Notre approche consiste donc à considérer seulement la sémantique abstraite des modules développés.

Cette particularité est due au fait que :

– le générateur de code est composé de plusieurs modules, appelés outils élémentaires, dont certains sont développés et vérifiés de manière classique et doivent donc reposer sur ce type d'exigences. Ils sont développés

en Java et vérifiés par des tests. Il est impératif dans le cadre de GeneAuto de s'appuyer sur les mêmes exigences ;
- chaque utilisateur partenaire du projet a sa propre bibliothèque interne du langage d'entrée (Simulink/Stateflow et Scicos) car chacun a ses propres besoins ;
- la sémantique de Simulink est instable. D'une version à l'autre des outils et d'une configuration à l'autre de chaque version des outils, la sémantique change de manière plus ou moins documentée ou maîtrisée par les utilisateurs. La seule observation possible de la sémantique est le résultat de la simulation ou de la génération de code par les outils fournis par MathWorks le concepteur de Simulink ;
- la sémantique concrète des outils élémentaires n'est pas décrite dans les documents de spécification.
- nous sommes contraints par le standard de qualification DO-178B pour lequel tous les modules du générateur de code sont soumis aux mêmes exigences pour le processus de vérification.

2.5 Synthèse

Nous avons présenté dans ce chapitre un aperçu, non exhaustif, des normes de certification et des techniques de vérification habituelles. Nous nous intéressons particulièrement, dans le cadre des travaux de cette thèse, aux méthodes formelles, notamment celles qui sont utilisées pour vérifier les compilateurs.

Nos travaux reposent principalement sur la technique de développement prouvé en appliquant l'analyse statique des flots de données et/ou de contrôle. Cette analyse est fondée sur les ensembles partiellement ordonnés, les treillis de profondeur finie et sur le calcul de point fixe sur des domaines abstraits.

Avant d'exposer notre démarche, nous allons présenter, dans un premier temps, les éléments nécessaires pour sa mise en œuvre, et, dans un second temps, le processus que nous avons suivi pour développer, vérifier et intégrer les outils élémentaires dans la chaîne de développement du générateur de code.

Deuxième partie

Contexte & Formalisation

Éléments de mise en œuvre 3

3.1 Présentation

Matlab «MATrix LABoratory» est un logiciel de calcul scientifique conçu à la fin des années 70 et commercialisé par la société MathWorks en 1984. Celui-ci est centré autour d'un langage de modélisation mathématique hybride discret et continu et d'un environnement de calcul numérique offrant également des services de calcul symbolique. Le langage est enrichi par de nombreuses boîtes à outils (toolboxes) qui offrent des fonctions supplémentaires conçues pour des applications particulières : traitement du signal, contrôle de systèmes, optimisation, etc.

C'est un des logiciels les plus utilisés pour la modélisation et la simulation des systèmes. Le langage Matlab intègre nativement les matrices, les vecteurs, les résolveurs discrets d'équations continues. La simulation est une approche permettant de mieux comprendre et explorer les systèmes dynamiques pour concevoir des mécanismes de contrôle et de commande. Dans le contexte des systèmes critiques, Matlab est utilisé pour modéliser l'environnement d'exploitation des systèmes et les mécanismes de contrôle et commande qui interagissent avec cet environnement. La simulation permet alors de valider les modèles proposés pour l'environnement et le système. Matlab est ensuite utilisé pour discrétiser les systèmes continus et proposer des algorithmes discrets qui implantent les modèles continus dans un contexte d'informatique embarquée. La simulation est alors exploitée d'une part pour valider les modèles proposés et d'autre part pour vérifier que les algorithmes discrets sont conformes à leur spécification continue.

La figure 3.1 synthétise de façon simplifiée l'environnement Matlab, complété par Simulink et Stateflow, tel qu'il est utilisé pour la modélisation des systèmes embarqués. Ce dernier est constitué de l'interpréteur Matlab qui interprète et exécute soit les instructions d'un fichier (.m), soit les instructions données interactivement par l'utilisateur à travers la fenêtre de commandes. Les fichiers (.m) sont des programmes écrits en langage Matlab par l'usager ou alors générés automatiquement par Simulink à partir des modèles. Les boîtes à outils sont des collections de fichiers (.m) développés pour des domaines d'applications spécifiques : traitement du signal, commande et contrôle de systèmes, optimisation, réseaux de neurones, etc. L'interpréteur Matlab communique interactivement avec l'environnement Simulink. Ce dernier exploite des bibliothèques regroupant des blocs élémentaires développés pour des domaines particuliers. Les modèles conçus à l'aide de Simulink sont enregistrés comme des fichiers au format (.mdl). Stateflow quant à lui étend Simulink avec un environnement pour le développement de diagrammes d'états.

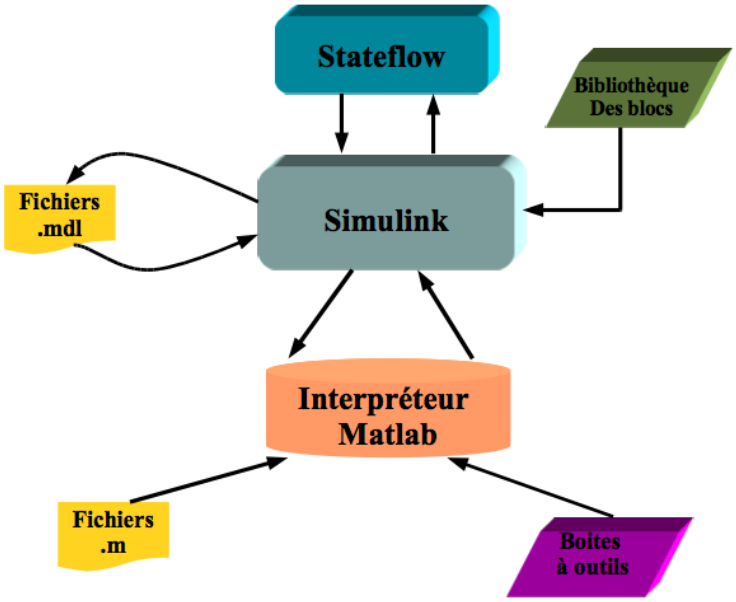


FIG. 3.1 – *La boîte à outils* MATLAB

3.2 LANGAGES SIMULINK ET STATEFLOW

SIMULINK [Sim] est un logiciel pour la conception et la simulation (d'où son nom), couplé avec le logiciel MATLAB. La simulation consiste à faire interpréter un modèle (une abstraction du système sous forme de diagramme de blocs) afin d'analyser ses propriétés sans exécuter l'implantation dérivée de ce modèle, et vérifier ainsi qu'il valide les exigences implicites et qu'il vérifie les exigences explicites.

L'exploitation d'un langage graphique permet de positionner SIMULINK comme un langage de modélisation de haut niveau, particulièrement utilisé dans les applications industrielles de différents domaines d'application critique, à savoir la médecine, l'aéronautique, le spatial, l'automobile, etc. Il a été conçu pour des besoins de spécification et de simulation d'applications complexes. Il a vite suscité l'intérêt de développement de générateurs de code commerciaux qui seront présentés dans la section 3.3. Nous décrivons dans ce qui suit les principaux concepts de SIMULINK et STATEFLOW. Pour davantage de détails, le lecteur peut se référer au site officiel de la société The MathWorks [1].

3.2.1 Simulink par l'exemple

L'utilisateur de SIMULINK manipule des blocs fonctionnels disponibles dans des bibliothèques spécialisées qu'il est possible de recopier et de relier entre eux, dans une fenêtre d'édition, pour construire le modèle souhaité.

Un bloc est une boîte représentant une fonction de calcul particulière. Un bloc comporte des ports d'entrée et/ou de sortie. Les blocs sont reliés par des signaux connectés sur les ports. Ces signaux sont représentés graphiquement par des segments orientés d'un port de sortie vers un port d'entrée. Les signaux reliant les blocs peuvent transporter des données ou du contrôle. Une

[1] www.mathworks.com

donnée correspond à une valeur concrète tandis qu'un contrôle est un événement qui déclenche l'exécution du bloc cible. Le bloc source d'un signal de contrôle est appelé bloc *contrôleur*, tandis que le bloc cible de ce signal est appelé bloc **contrôlé**.

La figure 3.2 illustre un circuit SIMULINK qui calcule la somme de deux entrées et la recopie sur une sortie. Le circuit est constitué de quatre blocs `In1`, `In2`, `Add` et `Out1`, reliés entre eux par les signaux `s1`, `s2` et `s3`. Un signal relie un port de sortie d'un bloc source à un port d'entrée d'un bloc cible, par exemple, le signal `s3` relie le port de sortie `o3` du bloc `Add` et le port d'entrée `i3` du bloc `Out1`. Notons que la direction d'un signal est importante, car elle indique le sens de propagation de l'information du bloc source vers le bloc cible.

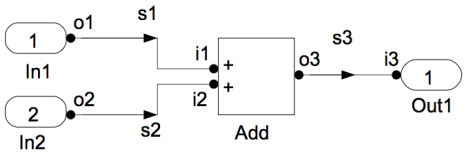
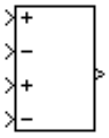
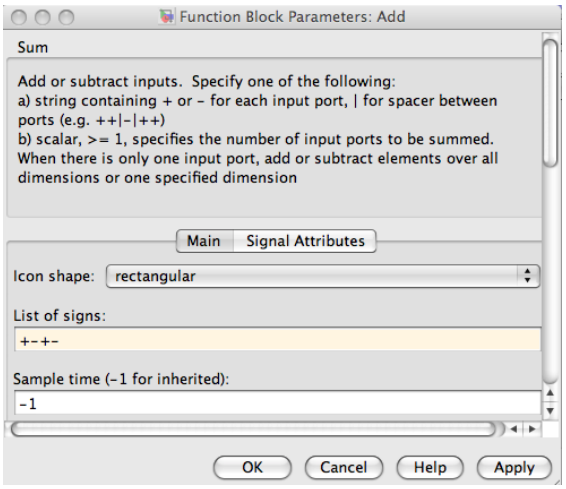


FIG. 3.2 – *Structure d'un circuit* SIMULINK

Les blocs SIMULINK peuvent être des fonctions génériques que l'utilisateur instanciera en configurant les paramètres du bloc. Prenons pour exemple, le bloc `Add` illustré dans la figure 3.3. Le bloc élémentaire `Add` en SIMULINK comporte deux ports d'entrée et un port de sortie qui recevra la somme des entrées. Cette construction est extensible en spécifiant dans la fenêtre de dialogue le nombre d'entrées et la suite d'opérations (qui est un mélange d'addition et de soustraction) à effectuer tel que cela est illustré dans la figure 3.3.



(a) *Bloc* SIMULINK `Add` *à plusieurs entrées*



(b) *Fenêtre de dialogue du bloc* SIMULINK `Add`

FIG. 3.3 – *Configuration du bloc* SIMULINK `Add`

Remarque Un bloc `Add` peut également avoir une seule entrée, auquel cas l'opération effectuée dépend du type de cette entrée (voir la section 3.2.4). Si l'entrée est un vecteur par exemple, le résultat correspond à la somme de ses

éléments. De plus, certains blocs SIMULINK peuvent réaliser la même fonction notamment en arithmétique, nous citons par exemple les blocs `Sum` et `Add` pour des opérations d'addition et de soustraction.

Blocs séquentiels : Outre les blocs fonctionnels, appelés également combinatoires qui exploitent les valeurs calculées dans un cycle donné, SIMULINK dispose également de blocs manipulant des valeurs calculées dans des cycles antérieurs. Ces blocs sont appelés des blocs séquentiels.

Définition 3.2.1. *Cycle*
Un cycle correspond à l'exécution des blocs d'un circuit. Une application embarquée exécute un circuit pendant un nombre quelconque de cycles tant que le système est actif. La périodicité (la fréquence) des cycles peut être configurée en SIMULINK.

Un exemple de circuit manipulant des blocs séquentiels est présenté dans la figure 3.4. Ce circuit calcule la moyenne de 5 valeurs différées dans le temps. Le bloc séquentiel manipulé dans le circuit est le bloc SIMULINK `Unit Delay`. Au cycle n, il mémorise la valeur de son entrée et produit sur sa sortie la valeur de son entrée mémorisée durant le cycle $n-1$. Il est nécessaire de lui fournir une valeur d'initialisation pour le premier cycle d'exécution. Celle-ci est donnée sous la forme d'un paramètre de chaque instanciation du bloc.

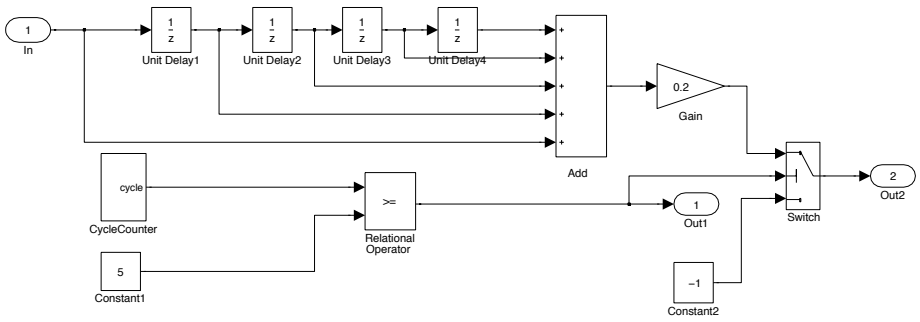


FIG. 3.4 – *Circuit moyenne 5 utilisant le bloc séquentiel* `Unit Delay`

Au cycle n, le bloc `Add` additionne 5 entrées dont une correspond à la sortie du bloc `In` au cycle courant et 4 autres correspondent aux 4 valeurs antérieures de la sortie de `In`. Par exemple, la sortie du bloc `Unit Delay1` est la valeur de `In` dans le cycle $n-1$ et la sortie de `Unit Delay4` est la valeur de la sortie de `In` dans le cycle $n-4$. Le bloc `Cycle Counter` est un simple compteur dont la structure est illustrée dans la figure 3.5. Il est utilisé pour calculer le nombre de cycles du circuit global qui ont déjà été exécutés.

La sortie du circuit (l'entrée du bloc `Out2`), correspond soit à la première entrée du bloc `Switch`, soit à sa troisième entrée selon la valeur de sa seconde entrée conformément à la sémantique de ce bloc. Autrement dit, à partir du cinquième cycle, la sortie du circuit correspond soit à la moyenne produite par le bloc `Gain`, soit à la sortie de la constante du bloc `Constant2`.

Sous-systèmes : Afin de faciliter la lecture des circuits complexes et la description modulaire des fonctions, il est possible de créer des sous-systèmes qui sont des blocs contenant eux-mêmes des circuits. SIMULINK dispose pour cela de la bibliothèque `Ports & Subsystems`. Par exemple, le bloc `Cycle Counter` de la figure 3.4 est un sous-système. Construire un sous-système en

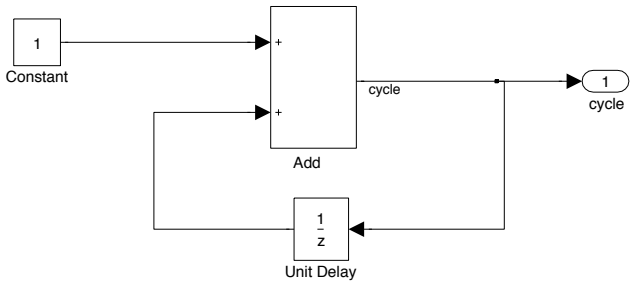


FIG. 3.5 – *Composition du bloc Cycle Counter*

regroupant des blocs pour offrir une fonctionnalité qui sera réutilisée plusieurs fois est similaire à l'écriture d'une fonction ou d'une procédure dans un programme.

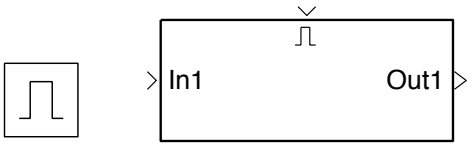
À l'opposé d'un sous-système, un bloc atomique est considéré comme une boîte noire. Il ne peut pas être décomposé et correspond à une seule fonction élémentaire. Ainsi, nous distinguons dans le reste du manuscrit les blocs atomiques et les sous-systèmes.

Un sous-système peut avoir différents comportements, chacun étant déterminé par une condition spécifique. Pour cela, SIMULINK dispose de deux blocs `Enable` et `Trigger`. Il est alors possible de construire deux types de sous-systèmes : conditionnel (activable) `Enabled Subsystem` et contrôlé (à déclenchement) `Triggered Sub-system`.

Enabled Subsystem : Il s'agit d'un sous-système (voir figure 3.6b) qui comporte un bloc `Enable` (voir figure 3.6a), appelé aussi port car il est utilisé comme un port d'activation du système en fonction du signal qu'il reçoit. `Enabled Subsystem` ne s'exécute que si le signal est positif et s'interrompt quand il est négatif. Tout bloc `Subsystem` contenant un bloc `Enable` est un `Enabled Subsystem`. Ce port permet donc une exécution conditionnelle des blocs en fonction de la valeur du signal de donnée connecté sur le port d'activation.

Deux configurations sont possibles pour la sortie d'un `Enabled Subsystem` lorsqu'il est désactivé. Il s'agit de configurer le bloc `Enable` sur :

- *Held* : la sortie est maintenue à sa valeur la plus récente, c'est-à-dire la valeur que le sous-système avait lors du dernier cycle où le sous-système était activé ;
- *Reset* : la sortie est remise à sa valeur initiale (zéro si celle-ci n'est pas définie).



(a) *Bloc* `Enable` (b) `Enabled Subsystem`

FIG. 3.6 – *Bloc Enable*

Triggered Subsystem : Il s'agit d'un sous-système (voir 3.7) comportant un bloc `Trigger` (voir 3.8), appelé aussi port car il est utilisé comme un port de déclenchement. Il s'exécute immédiatement lorsqu'un événement se produit sur le signal de contrôle connecté au port `Trigger`. Il est également possible d'obtenir un `Triggered Sub-system` en ajoutant un bloc `Trigger` à un bloc `Subsystem`

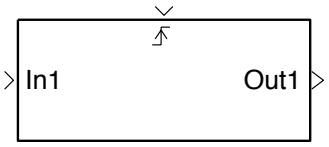
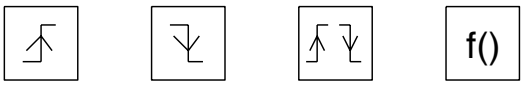


FIG. 3.7 – *Triggered Subsystem*

Un événement se produit quand la valeur du signal connecté sur le port `Triggered` change en fonction du paramètre choisi :

- *Rising* : pour déclencher l'exécution du sous-système lorsque le signal de contrôle qu'il reçoit est sur front montant (son changement est croissant) ;
- *Falling* : pour déclencher l'exécution du sous-système lorsque le signal de contrôle qu'il reçoit est sur front descendant (son changement est décroissant) ;
- *Either* : pour déclencher l'exécution du sous-système lorsque le signal de contrôle qu'il reçoit est sur les fronts montants ou descendants ;
- *Function-Call* : pour déclencher l'exécution du sous-système indépendamment de la valeur du signal, mais selon la logique interne de la fonction qui appelle ce sous-système. Dans ce cas, nous parlons de `Function-Call Subsystem` qui sera présenté par la suite.

Ces configurations sont illustrées dans la figure 3.8



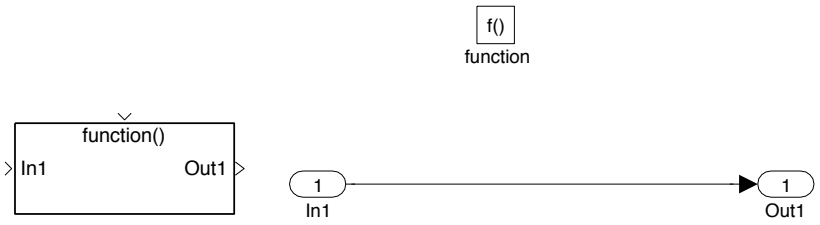
(a) *Rising* (b) *Falling* (c) *Either* (d) *Function-Call*

FIG. 3.8 – *Bloc* SIMULINK *Trigger*

Enfin, il est également possible qu'un sous-système soit à la fois activable et à déclenchement (contenant les ports `Enable` et `Trigger`). Dans ce cas, il agit comme un sous-système déclenché seulement lorsqu'il est actif, c'est-à-dire quand le signal reçu par `Enable` est positif. Dans le cas où il est inactif, la sortie du sous-système dépend de la configuration du port `Enable`.

Function-Call Subsystem : Un cas particulier de sous-système à déclenchement est le bloc `Function-Call Subsystem`. Il s'agit d'un bloc qui peut être appelé par un autre bloc de manière analogue à l'appel de fonctions dans un langage de programmation.

La figure 3.9b montre que `Function-Call Subsystem` est composé d'un bloc `function` qui agit comme un port de contrôle déclenchant l'exécution



(a) `Function−Call Subsystem` (b) *Composition de* `Function−Call Subsystem` *par défaut*

du sous-système. Le bloc `function` n'est en fait qu'un bloc `Trigger` configuré pour servir de port de contrôle du bloc `Function−Call Subsystem`. La figure 3.9b montre une composition par défaut d'un `Function−Call Subsystem` qui sera évidemment adaptée au besoin par l'utilisateur.

Soit le circuit de la figure 3.9. C'est un circuit SIMULINK à flots de contrôle dont le bloc `Chart S1` est un diagramme d'états STATEFLOW et les blocs `S2` et `S3` sont des blocs `Function-call Subsystem`. Cela signifie que le bloc `Chart S1` appelle, de la même manière qu'un appel de fonctions, le sous-système `S2` qui lui-même appelle le sous-système `S3`. Ainsi, la suite des blocs contrôleurs et blocs contrôlés se comporte comme si elle ne formait qu'un seul bloc que nous appelons *sous-système virtuel*. Notons qu'en SIMULINK, un sous-système défini comme virtuel indique à SIMULINK d'ignorer les frontières de ce sous-système, c'est-à-dire de remplacer le sous-système par les blocs et les signaux qu'il contient en effaçant les blocs d'entrée et de sortie du sous-système.

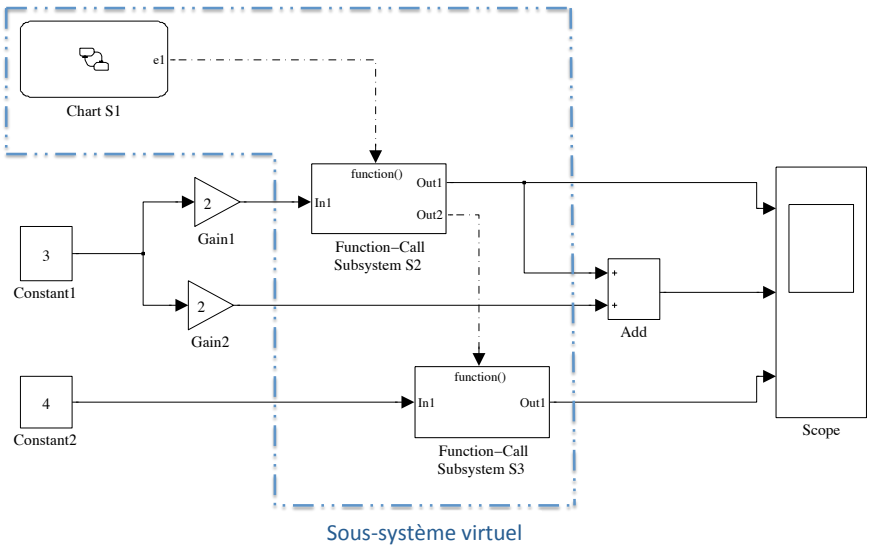


FIG. 3.9 – *Circuit* SIMULINK *à flots de contrôle*

Définition 3.2.2. *Bloc libre*
Un bloc `b` est dit libre si aucun bloc ne le contrôle, c'est-à-dire si aucun signal de contrôle n'est connecté à son port de déclenchement.

Par exemple le diagramme d'états `Chart S1` est un bloc libre.

Le terme flots de données, respectivement flots de contrôle, désigne le sens dans lequel les données circulent durant l'exécution d'un système (du producteur vers le consommateur), respectivement l'ordre dans lequel les différentes parties du système sont exécutées.

Dans les langages synchrones à flots de données [Ack82], le terme flot prend de plus une signification temporelle : il signifie une suite de valeurs étiquetées par un instant de production. Dans ces langages, chaque flot est caractérisé par une équation qui relie les entrées (consommateur) et les sorties (producteur), et qui peut être une définition ou une propriété. Les équations sont déduites des opérations réalisées par les blocs.

Dans les circuits SIMULINK, deux catégories de flots sont rencontrées : flots de données et flots de contrôle.

Définition 3.2.3. *Flot de données*
Un flot de données est la séquence des valeurs d'un signal de données reliant un port à un autre dans un circuit.

Définition 3.2.4. *Flot de contrôle*
Un flot de contrôle est la séquence des événements d'un signal de contrôle reliant un bloc contrôleur au port de déclenchement d'un bloc contrôlé dans un circuit.

3.2.2 Stateflow par l'exemple

STATEFLOW [Sta] est une extension de SIMULINK (boîte à outils de MATLAB) qui offre un outil de développement et de conception pour des systèmes complexes dont la loi de commande peut évoluer de manière discrète et discontinue en fonction de l'état du système (système possédant des modes de fonctionnement) ou pour des logiques de surveillance (protocoles par exemple). En STATEFLOW, il est possible de créer des tables de vérité grâce au bloc `Truth Table` ou encore des diagrammes d'états avec le bloc `Chart`. Une table de vérité permet de modéliser des conditions et des actions complexes tandis qu'un diagramme d'états simplifie la représentation des transitions complexes entre les états.

Inspirée des automates, la notation des diagrammes d'états (statechart) a été proposée par Harel [Har87] et a ensuite été adoptée par différents langages de modélisation (UML 2 [Sel06], STATEMATE [Val92], STATEFLOW, etc.). Un diagramme STATEFLOW est une machine à état décrite, entre autres, par des états, transitions, jonctions, événements, conditions et actions.

Étant donné que nos travaux concernent principalement SIMULINK, nous présentons brièvement les concepts de base des diagrammes STATEFLOW qui sont la source des événements de type *function-call*.

Considérons le diagramme d'états STATEFLOW présenté dans la figure 3.10. Il est utilisé comme référence pour expliquer les différents concepts d'un diagramme STATEFLOW.

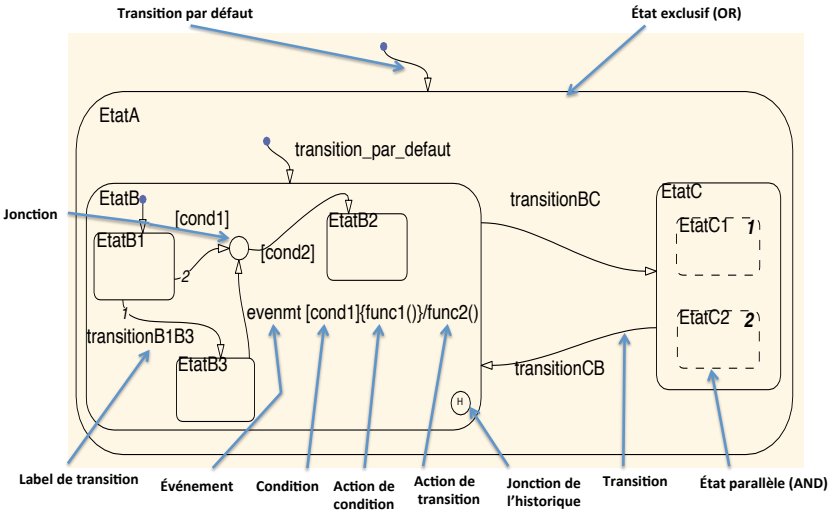


FIG. 3.10 – *Structure générale d'un diagramme d'états en* STATEFLOW

État : Il décrit une étape dans l'exécution d'un système piloté par les événements. Un état, peut contenir d'autres états, et être actif ou non selon les conditions et les événements qui se sont produits depuis le démarrage du système.

STATEFLOW dispose de deux états : exclusif (*OR*) et parallèle (*AND*). Dans un diagramme, ou dans un état exclusif (*OR*), un seul de ces états est actif lorsque l'état parent ou le diagramme est actif. Tandis qu'un état à sous-états parallèles peut avoir plusieurs états actifs simultanément quand l'état parent est actif.

Par exemple, lorsque `EtatA` est actif, `EtatB` et `EtatC` sont mutuellement exclusifs. `EtatB` et `EtatC` sont alors appelés des décompositions exclusives (*OR*). Elles sont représentées graphiquement par des rectangles continus.

De même, quand `EtatC` est actif, les états `EtatC1` et `EtatC2` sont actifs en même temps. Ils sont alors appelés états parallèles (*AND*) et sont représentés par des rectangles discontinus.

Événement : Ce sont les événements qui pilotent l'exécution d'un diagramme. Les événements n'ont pas de représentation graphique. Ils apparaissent dans un diagramme à travers leur libellé. Par exemple, `evnmt` est l'événement déclencheur de la transition entre `EtatB3` et la jonction.

Jonction : Une jonction est un point de décision pour la (les) transition(s). Elle est utilisée pour factoriser ou combiner les transitions afin de décrire plus simplement le comportement du système représenté. Elle est semblable à un état mais une transition ne peut s'arrêter dans une jonction, elle

doit être suivie immédiatement d'une autre transition ou elle doit être annulée.

La jonction d'historique (*History Junction*), quand elle est associée à un état, indique que ce dernier possède une mémoire. Cela signifie que lorsqu'un état à mémoire est réactivé, son sous-état actif lors de sa dernière désactivation est également réactivé.

Dans la figure 3.10, la jonction d'historique dans `EtatB` indique que lorsqu'une transition active `EtatB`, le sous-état qui sera activé est celui qui était actif parmi `EtatB1`, `EtatB2` et `EtatB3`, lorsque `EtatB` a été désactivé.

Une transition par défaut n'a pas d'état source, elle indique l'état qui sera activé lorsque son parent l'est également. Par exemple, les transitions qui déclenchent `EtatA`, `EtatB` et `EtatB1` sont des transitions par défaut.

Une transition peut être labellisée par l'événement déclencheur. Par exemple, `transitionCB`, le label de la transition qui relie `EtatC` à `EtatB`, est en fait un événement. Une transition peut également être labellisée par une condition ou par une action.

Condition : Une condition est une expression booléenne spécifiant qu'une transition se déroulera lorsque cette expression booléenne est évaluée à vrai. Dans la figure 3.10, `cond1` représente une expression booléen qui, évaluée à vrai, déclenche la transition entre `EtatB1` et la jonction.

Action : Les actions font partie de l'exécution d'un diagramme Stateflow. Une action peut être exécutée comme une partie d'une transition ou comme une action au sein d'un état. Une action peut être l'appel d'une fonction, la diffusion d'un événement vers Simulink par exemple, l'affectation d'une valeur à une variable, etc.

Une transition peut contenir une action conditionnelle (par exemple, {`func1()`} dans la figure 3.10) et/ou une action de transition (comme `func2()`).

Supposons, dans la figure 3.10, que le diagramme est inactif, l'état `EtatB3` est actif et l'événement `evnmt` survient. Le diagramme d'états est alors activé et déclenche une transition valide pour l'événement `evenmt`. Une transition valide de `EtatB3` vers la jonction est détectée. La condition `cond1` est vraie, alors l'action de conditionnelle {`func1()`} est activée et exécutée. `EtatB3` qui est encore actif termine ses actions (`exit_action()`) et devient inactif. Ensuite l'action de transition `func2()` est exécutée. La condition `cond2` est fausse. Donc la transition entre la jonction et `EtatB2` n'est pas valide. Par conséquent, une action après un délai (`dur_action()`) de l'état `EtatB3` est exécutée en gardant ce dernier actif. Le diagramme d'états redevient inactif en attendant d'être activé par un nouvel événement.

L'action liée à un état dépend des modalités d'activation. Ces modalités sont résumées dans la figure 3.11.

entry désigne que l'action *ent_action*() est exécutée une fois que l'état `EtatB` est devenu actif ;

during indique que l'action *dur_action*() est exécutée lorsque l'état `EtatB` est déjà actif et qu'un événement quelconque survient sans désactiver l'état `EtatB` ;

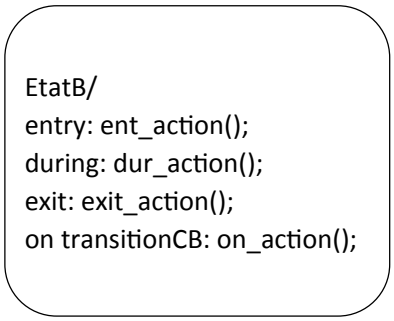


FIG. 3.11 – *Les actions associées à un état*

exit indique que l'action *exit_action*() s'exécute juste avant que l'état `EtatB` ne devienne inactif. Après exécution de l'action *exit_action*(), l'état `EtatB` devient inactif ;
on est semblable au mode **during** sauf que c'est la transition *transitionCB* qui déclenche l'exécution de l'action *on_action*().

Enfin les données consommées ou produites par un diagramme STATEFLOW sont explicitées dans la configuration de ce dernier. Cela consiste à préciser si une donnée est locale, constante, une sortie d'un bloc SIMULINK, une entrées d'un bloc SIMULINK, etc. De même, il est possible de définir les sorties du diagramme comme des événements déclencheurs de blocs SIMULINK (voir figure 3.9) ou inversement définir ses entrées comme des événements.

Ainsi, les diagrammes STATEFLOW agissent comme des blocs SIMULINK. Intégrés dans un circuit, les diagrammes d'états sont alimentés par des blocs SIMULINK et produisent des sorties consommées par d'autres blocs SIMULINK. En particulier, lorsqu'il s'agit de contrôler une partie du système dans le cadre des applications de commande, le bloc `Trigger` mis à l'intérieur d'un diagramme STATEFLOW sert à l'activer selon la condition du signal d'entrée comme le montre la figure 3.12.

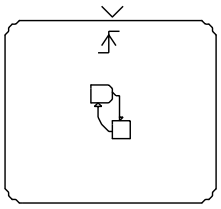


FIG. 3.12 – *Trigger un diagramme* STATEFLOW

Comme les diagrammes STATEFLOW sont intégrés dans les modèles SIMULINK sous la forme de blocs atomiques, nous ne parlons que de circuits SIMULINK de façon plus générale.

3.2.3 L'ordonnancement en Simulink

Nous présentons dans cette section plusieurs exemples afin d'illustrer l'ordonnancement des blocs qui constituent les circuits Simulink. Les circuits représentent des possibilités d'exécution concurrente, comme tout modèle flot de données. Cependant, Simulink a une sémantique d'exécution séquentielle ne permettant pas la concurrence.

Pour transformer un circuit en code séquentiel, il faut associer un ordre d'exécution unique pour chacun des blocs du circuit. Cet ordre dépend de la structure de ce dernier et des règles d'ordonnancement.

D'une part, l'ordonnancement des blocs Simulink consiste à respecter la causalité des données et du contrôle, c'est-à-dire suivre la structure du circuit selon les flots de données et de contrôle. Autrement dit, pour un bloc donné, les valeurs des signaux de données connectés sur ses port d'entrée doivent être calculées avant l'exécution de ce bloc, et les blocs contrôlés sont exécutés pendant l'exécution des blocs contrôleurs.

D'autre part, des contraintes additionnelles s'ajoutent pour assurer un ordre total des blocs, c'est-à-dire que chaque bloc a un ordre unique. Il s'agit ici des priorités que l'utilisateur peut associer aux blocs du circuit afin de forcer explicitement l'exécution d'une partie du circuit avant une autre. En pratique, un bloc peut posséder deux priorités : explicite qui est définie par l'utilisateur, et implicite qui est déduite de la position graphique du bloc. Le bloc le plus prioritaire selon la priorité implicite est celui se trouvant le plus en haut à gauche.

Afin d'introduire les contraintes d'ordonnancement, nous partons d'un cas simple de circuits aux flots de données sans blocs séquentiels (à mémoire), puis nous considérons les circuits aux flots de données avec blocs séquentiels (à mémoire) pour ensuite traiter le cas plus complexe contenant des flots de données avec des imbrications de flots de contrôle.

3.2.3.1 Circuits aux flots de données sans mémoire

L'ordonnancement des blocs dont les entrées ne sont reliées qu'à des signaux de données suit uniquement la propagation des données à travers le circuit. Il faut exécuter les blocs qui alimentent les entrées d'un bloc avant d'exécuter celui-ci. Un exemple de circuit à flots de données est illustré par la figure 3.13. Les blocs `Devide` et `Gain1`, qui produisent les données pour les entrées du bloc `Add`, doivent être exécutés avant le bloc `Add`. Un circuit à flots de données exprime une exécution concurrente potentielle entre les blocs.

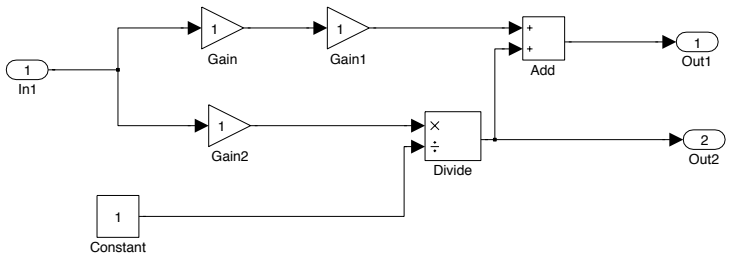


Fig. 3.13 – *Circuit* Simulink *à flot de données sans mémoire*

Considérons maintenant le circuit de la figure 3.14. Les entrées du bloc `Add` sont connectées sur la sortie du bloc `In1` et sur celle de `Gain1` dont les données

sont produites dans le même cycle. Or la sortie du bloc `Gain1` n'est produite que si le bloc `Add` est exécuté dans le même cycle. Ainsi, l'exécution du bloc `Add` nécessite d'exécuter le bloc `In1` et le bloc `Add` et ceci dans le même cycle. Il y a donc un problème de causalité de données. Ce problème de causalité est appelé *boucle algébrique* de données. Pour couper ce cycle qui existe autour du bloc `Add`, il est nécessaire d'insérer un bloc séquentiel (à mémoire) dans ce cycle. La section suivante présente comment ordonnancer des circuits qui comporte des blocs séquentiels (à mémoire).

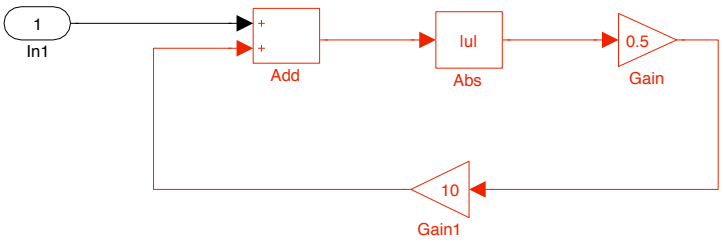


FIG. 3.14 – *Circuit à boucle algébrique*

3.2.3.2 Circuits aux flots de données avec mémoire

Les blocs séquentiels nécessitent un traitement particulier pour l'ordonnancement puisqu'ils dépendent des valeurs calculées dans certains cycles antérieurs. Considérons le circuit de la figure 3.15 présenté précédemment. Dans un cycle n, la sortie du bloc `Unit Delay` reçoit la valeur de son entrée durant le cycle $n-1$. Donc, le bloc `Add` ne peut être exécuté que si la sortie du bloc `Unit Delay` a été produite. Mais ce dernier prend en entrée la sortie du bloc `Add`. Il y aurait un problème de causalité des données si `Unit Delay` ne consomme pas une valeur en mémoire (son entrée lue au cycle $n-1$). Ainsi, nous considérons qu'un bloc séquentiel est constitué de deux sous-blocs : les sous-blocs de lecture noté R et d'écriture noté W.

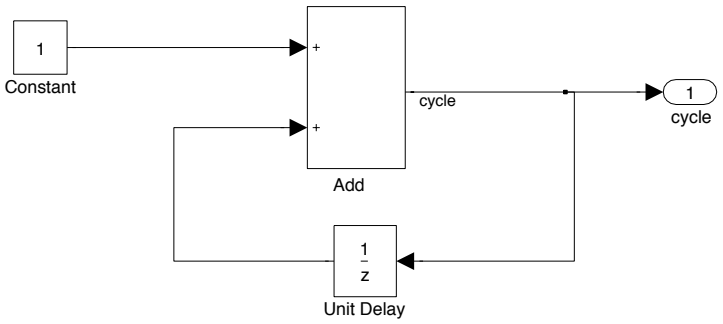


FIG. 3.15 – *Circuit* SIMULINK *à flots de données avec mémoire*

Pour résumer, l'ordonnancement des blocs séquentiels est effectué en deux étapes (voir figure 3.16) :

1. la lecture dans la mémoire d'une valeur calculée et mémorisée dans un cycle antérieur. Cette valeur est copiée sur les signaux de sortie du bloc séquentiel ;

2. puis l'écriture en mémoire de la nouvelle valeur qui sera exploitée dans les cycles suivants. Celle-ci est calculée à partir des entrées du bloc séquentiel dans ce cycle-ci.

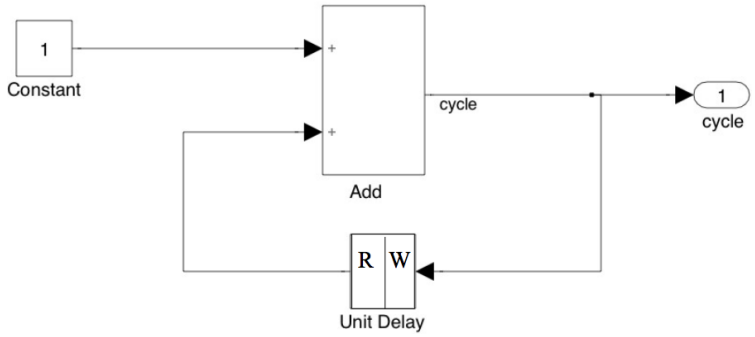


FIG. 3.16 – *Découpage du bloc séquentiel en deux sous-blocs*

Pour l'exécution du bloc `Add` de l'exemple ci-dessus, il faut d'abord exécuter dans l'ordre le bloc `Constant` puis la partie lecture (*R*) du bloc `Unit Delay`. La sortie produite après exécution du bloc `Add` sera sauvegardée en mémoire par l'exécution de la partie écriture (*W*) du bloc *Unit Delay*.

Ainsi, dans le reste du manuscrit, nous considérons une sémantique discrète des circuits traités. Nous distinguons, deux catégories de blocs SIMULINK :

- les blocs combinatoires produisant leurs sorties en fonction de leurs entrées et éventuellement des paramètres du bloc, durant le même cycle ;
- les blocs séquentiels ou à mémoire dont les sorties sont des valeurs calculées dans un cycle antérieur.

3.2.3.3 Circuits aux flots de contrôle

Les systèmes industriels utilisent très souvent des circuits à flots de contrôle. La difficulté dans l'ordonnancement de ce type de modèle est la présence des imbrications de flots de contrôle comme cela est illustré dans la figure 3.17 présentée précédemment. Les imbrications de flots de contrôle imposent la prise en compte de la fermeture transitive des flots de contrôle : les entrées des blocs appelés (ou contrôlés) doivent être disponibles au début de l'exécution du premier bloc appelant dans la chaîne des flots de contrôle. Notons que nous nous plaçons dans une restriction de SIMULINK telle que pour chaque sous-système, il y a un seul bloc ou sous-système qui puisse le contrôler.

La structure du code généré pour les sous-systèmes `Function-Call` est illustrée dans la figure 3.17. Par exemple, pour exécuter le sous-système *S*3, il est nécessaire d'exécuter tous les sous-systèmes qu'il contrôle ainsi que les blocs qui calculent leurs entrées respectives. Comme les sous-systèmes contrôlés sont le plus souvent imbriqués, il faut que les circuits soient dépliés par le module de pré-traitement *préprocesseur* (voir l'architecture de GENEAUTO illustrée dans la figure 3.22). Par conséquent, pour exécuter le sous-système *S*3, il faut d'abord exécuter le bloc *S*1 qui contrôle le bloc *S*2, les blocs qui calculent les entrées du bloc *S*2 ainsi que les entrées du bloc *S*3. Rappelons, ici, qu'un bloc contrôlé est exécuté à l'intérieur du sous-système virtuel qui l'appelle. Donc, il n'est exécuté ni avant ni après ces blocs contrôleurs mais pendant l'exécution de ces derniers. La figure 3.18 illustre la structure du code impératif généré pour le circuit de la figure 3.17.

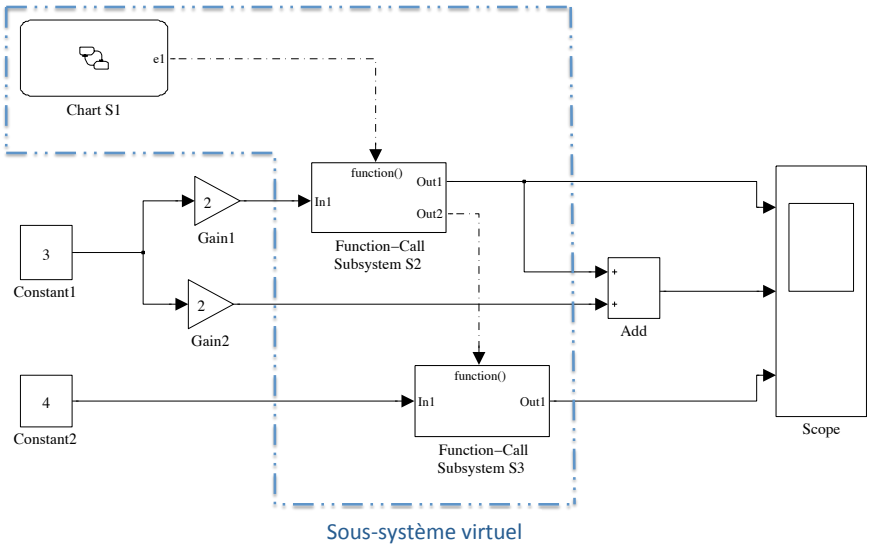


FIG. 3.17 – *Circuit* SIMULINK *mixant flots de données et de contrôle*

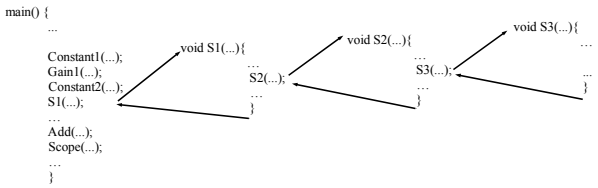


FIG. 3.18 – *Pseudo-code impératif généré pour l'exemple*

Donc, pour commencer l'exécution du bloc *S*3 il faut préalablement :
– avoir exécuté les entrées nécessaires pour l'exécution des contrôleurs successifs de *S*3 : `In1`, `Gain1` (la chaîne des entrées de *S*2) ;

- avoir exécuté `In2`, l'entrée de $S3$, qui doit être disponible lors de l'appel de $S3$;
- que les contrôleurs successifs de $S3$ (dans l'ordre le bloc $S1$ puis $S2$) aient commencé leur exécution.

Dans la figure 3.17, les blocs `Gain1` et `Gain2` sont indépendants l'un de l'autre. Il n'existe aucune contrainte issue des signaux permettant de les ordonnancer. SIMULINK a prévu ce cas en offrant aux concepteurs la possibilité d'associer aux blocs une priorité, notamment pour résoudre les dépendances cachées introduites par les effets de bords effectués par certains blocs de mémoire. Dans le cas où les deux blocs possèdent des priorités explicites égales, ou si l'utilisateur n'a pas précisé de priorité, une relation d'ordre totale est exploitée en utilisant la priorité implicite calculée à partir de la position graphique du bloc dans le plan où le circuit est modélisé. Dans le cas où les blocs `Gain1` et `Gain2` par exemple ont la même priorité explicite, le bloc `Gain1` sera exécuté avant le bloc `Gain2`, car `Gain1` se trouve au dessus de `Gain2`. Cette relation est le produit lexicographique total de l'ordre du haut vers le bas et de l'ordre de gauche vers la droite.

Remarque Dans le cadre du projet GENEAUTO, il est primordial que l'ordonnancement produise un ordre total de manière déterministe. De plus, à un instant t donné, un seul bloc doit être exécuté à la fois. Cet ordre doit donc respecter les contraintes d'exécution du langage SIMULINK.

Notons également, qu'il est fastidieux de vérifier par des tests exhaustifs la correction de l'ordonnancement des modèles SIMULINK/STATEFLOW quand il s'agit de développer et de vérifier un outil dédié. La couverture raisonnable d'un outil d'ordonnancement par des cas de tests est difficile à obtenir. En effet, l'ordonnancement est une propriété globale qui ne se déduit en général pas de l'ordonnancement des parties du circuit. Un nombre de tests important est alors nécessaire pour obtenir la confiance requise à la qualification. De plus, un des objectifs est de réduire les coûts des tests tout en augmentant l'assurance sur l'ordonnanceur.

En ce qui concerne les diagrammes STATEFLOW, l'ordre d'exécution des états est tributaire des conditions qui guident les transitions. En revanche, le problème d'ordonnancement ne se pose pas dans ce cadre car la sémantique d'exécution ne permet pas l'interprétation concurrente des diagrammes contrairement aux StateCharts de Harel. De plus, les diagrammes STATEFLOW sont intégrés dans un circuit SIMULINK comme des blocs atomiques.

3.2.4 Typage en Simulink

En SIMULINK, les types n'ont pas besoin d'être explicitement précisés. Cependant, le système de type de SIMULINK dispose de règles de typage qui doivent être respectées par les circuits.

Les types de base numériques en SIMULINK sont résumés dans la table 3.1. Avec `realmin` une fonction MATLAB qui renvoie la plus petite valeur positive à virgule flottante représentable sur la machine. `realmin('single')` est la plus petite valeur positive simple précision à virgule flottante représentable sur la machine concernée. Réciproquement, `realmax('single')` est la plus grande valeur réelle positive en simple précision à virgule flottante représentable sur la machine.

Dans la grammaire suivante, nous décrivons les types de base de SIMULINK qui sont acceptés par le système de type de GENEAUTO. Les types numériques

Type	Désignation	taille	plage
int8	Entier signé	8 bits	[-128 ,127]
uint8	Entier non signé	8 bits	[0 , 255]
int16	Entier signé	16 bits	[-32768 , 32767]
uint16	Entier non signé	16 bits	[0 , 65535]
int32	Entier signé	32 bits	[-2147483648, 2147483647]
uint32	Entier non signé	32 bits	[0, 4294967295]
single	Réel simple précision	32 bits	[realmin('single'), realmax('single')]
double	Réel double précision	64 bits	[realmin('double'), realmax('double')]

TAB. 3.1 – *Les types numériques de base en* SIMULINK

acceptés par ce dernier sont ceux décrits dans la table 3.1 hormis `int64` et `uint64`.

Les types de base en SIMULINK sont `boolean`, `single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32` et `uint32`. Le type `double` est le type des réels représentés sur 64 bits, les types `int8`, `int16` et `int32` sont des types entiers signés respectivement sur 8, 16 et 32 bits, contrairement aux entiers non signés `uint8`, `uint16` et `uint32`. La configuration par défaut de SIMULINK consiste à typer tous les signaux, initialement, par *double* sauf si le signal est connecté à un bloc nécessitant un autre type ou si l'utilisateur a précisé explicitement un autre type. Cette configuration est également différente dans le cas où un bloc *Data Type Converter* est utilisé. Ce bloc convertit une entrée de type t_1 en une sortie de type t_2. De plus, les blocs nécessitant un type différent tel que les blocs logiques changent également cette configuration. La spécification de type en SIMULINK se fait lors de la spécification des blocs en précisant le type des ports de sortie. Un signal de donnée, en SIMULINK, doit avoir le même type sur ses deux extrémités.

Si le type d'un port n'est pas défini par l'utilisateur, il doit être calculé par inférence de type. Si l'inférence de type ne permet pas de le calculer, il sera initialisé au type `double`. Une erreur de type peut se produire lorsque des blocs manipulent des signaux de types incompatibles. Les règles de compatibilité peuvent être définies par l'utilisateur dans l'outil SIMULINK. Par exemple, celui-ci a la possibilité de spécifier le type de sortie à travers une fenêtre de dialogue. Notons également la fenêtre de dialogue de la figure 3.19 qui offre la possibilité à l'utilisateur d'additionner des valeurs de types différents en cochant la case appropriée.

Si cette case n'est pas cochée, SIMULINK autorise l'application d'un opérateur arithmétique sur des entrées dont certaines sont de type `boolean`. Le circuit de la figure 3.20 additionne un entier (`int8`) et un booléen (`boolean`) sans pour autant signaler une erreur.

Remarquons que le résultat de l'addition obtenu est 3 alors que les valeurs additionnées sont 3 et *false*. L'explication de ce comportement en SIMULINK est simple : l'entrée booléenne `false` est représentée par 0, ensuite l'opération d'addition est effectuée normalement. Ceci est due à une conversion de type effectuée automatiquement par SIMULINK en interne du bloc `Add`.

Considérons maintenant le circuit de la figure 3.21. Il est spécifié dans le bloc `Constant1` que la valeur 4 est de type booléen. Là aussi, l'addition est effectuée sans rejeter le circuit car deux conversions sont effectuées : d'une part, la valeur 4 est convertie en *true*, et d'autre part, la valeur true est convertie en 1 pour effectuer l'addition. Donc, toute valeur numérique non nulle correspond à *true*, sinon à *false*.

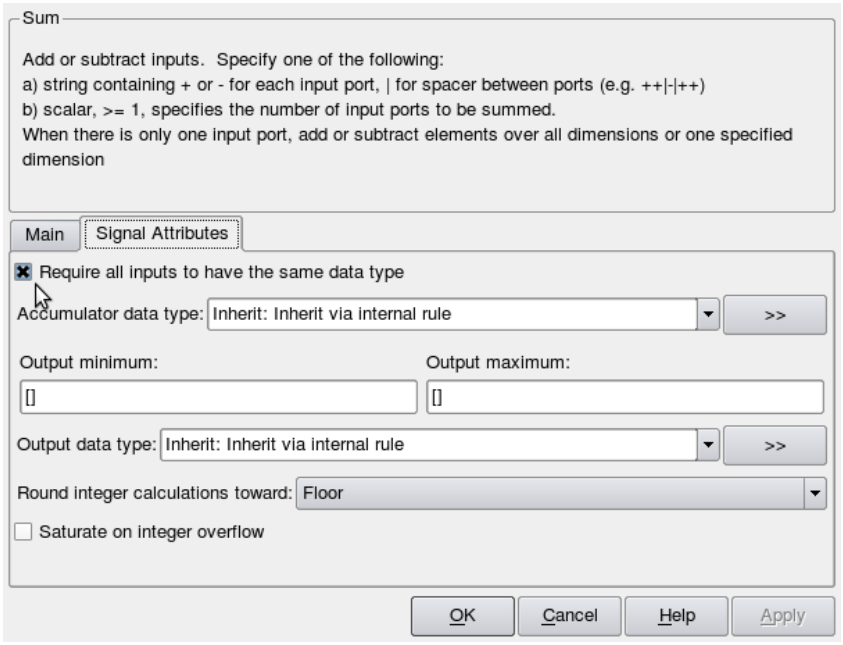


FIG. 3.19 – *Fenêtre de dialogue du bloc* SIMULINK `Add`

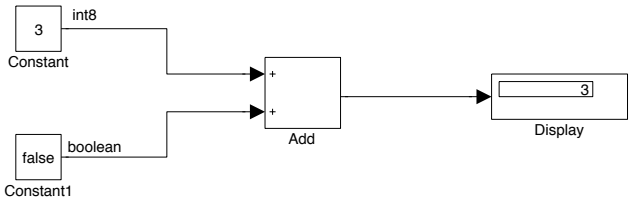


FIG. 3.20 – *La somme d'un entier et d'un booléen*

Les cas présentés sont issus d'une conversion de type effectuée automatiquement par SIMULINK si l'utilisateur ne précise pas ce qui est toléré et ce qui ne l'est pas. Il est possible de refuser la conversion de booléens en numérique, en configurant l'option correspondante sur les signaux logiques booléens.

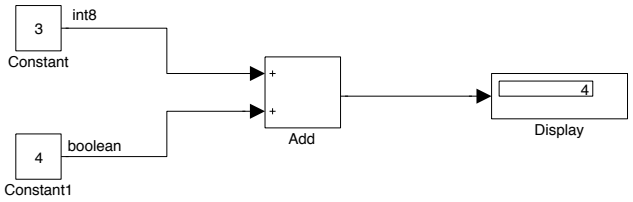


FIG. 3.21 – *Conversion implicite d'un entier en booléen*

Le même raisonnement reste valable pour les blocs logiques qui peuvent, en l'absence de configuration spécifique, manipuler des entrées numériques.

En SIMULINK, les nombres complexes sont représentés par des paires de nombres (entiers ou réels). Celui-ci offre également des types énumérés contenant des valeurs de type caractère qui ne sont pas considérées dans nos travaux. Le type d'un port peut être explicitement précisé par l'utilisateur ou déduit de-

puis la valeur des paramètres du bloc en question. Il peut également être hérité du type d'un autre port. Par exemple, le type d'un port de sortie d'un bloc peut être hérité (s'il n'est pas précisé) du type d'une constante, du type du port d'entrée ou encore du type de port du bloc cible auquel il est connecté. Simulink dispose également des types vecteur ou matrice dont les valeurs sont des types de base. Il supporte également les objets et les classes qui sont ignorés dans nos travaux. Les blocs Simulink ont généralement des types polymorphes. Il s'agit de polymorphisme ad-hoc où la sémantique du bloc dépend du type des paramètres, ou de polymorphisme de coercion induit par la conversion implicite entre certains types.

L'outil Simulink/Stateflow dispose de plusieurs sémantiques, qui varient suivant les versions ainsi que les options de configuration choisies par l'utilisateur et qui ne sont définies que de façon informelle, à travers les nombreux exemples et contre-exemples présentés dans la documentation de Simulink.

3.3 GeneAuto : Générateur de code Automatique

Dans le domaine des systèmes critiques, les générateurs de code sont utilisés, d'une part, pour réduire le temps de développement et de maintenance corrective et, d'autre part, pour faciliter la vérification que le code cible généré est correct par rapport au modèle source considéré. L'évolution de la complexité de tels systèmes conduit à une augmentation des exigences en terme de sûreté au niveau des normes concernées.

La part prépondérante qu'occupe actuellement le logiciel dans ces systèmes conduit à des coûts de développement élevés. La validation et la vérification sont principalement effectuées sur le système final et les anomalies détectées peuvent entraîner une remise en cause de nombreux outils intermédiaires dans le développement y compris les exigences initiales.

GeneAuto est un projet Européen ITEA, qui s'est déroulé de 2006 à 2008, dont l'objectif était le développement d'un générateur de code automatique pour les systèmes embarqués du domaine des transports aérien, spatial et automobile. Le générateur de code, appelé également GeneAuto, est un outil libre (*open source*) qui prend en entrée une description fonctionnelle d'une application spécifiée dans un langage de modélisation graphique (Simulink/Stateflow/Scicos) et qui produit en sortie un code source séquentiel en langage C. Une extension de GeneAuto génère également du code Ada. Actuellement, il n'existe pas de générateur de code pour le langage de modélisation graphique Simulink qui soit qualifié et qui intègre des approches formelles pour la spécification et la vérification dans son processus de développement. L'objectif majeur de GeneAuto était de proposer une alternative aux générateurs de code utilisés actuellement en industrie, et qui soit qualifié selon la norme 178B/ED-12B présentée dans le Chapitre 2. Les travaux amorcés dans GeneAuto se poursuivent dans les projets ITEA2 OPEES, ARTEMIS CESAR, et à partir d'octobre 2011 dans les projets FUI "*Projet P*" et EuroStarts "*HiMoCo*".

Les générateurs de code actuels sont usuellement qualifiés par l'application d'un processus de développement qui respecte les normes de qualification qui exigent généralement un grand nombre de tests pour atteindre une couverture structurelle de type MC/DC. Toutefois, à notre connaissance, aucun générateur de code n'applique des vérifications formelles sur le générateur lui-même ou sur ses composants. Or, les méthodes de vérification formelle sont souvent

appliquées sur le modèle source ou encore sur le code généré. En revanche, un point essentiel dans GeneAuto était l'expérimentation des méthodes formelles pour la vérification de l'implantation des outils élémentaires du générateur de code. Plus précisément, notre objectif est de montrer effectivement l'intégration de méthodes formelles dans le processus de développement de GeneAuto et de l'appliquer à des systèmes industriels de taille réelle dans les domaines des transports. De plus, outre les aspects de vérifications formelles apportés, GeneAuto consiste à définir un processus de développement certifié mixant les approches formelles et classiques.

3.3.1 Architecture de GeneAuto

Afin de permettre un maximum de flexibilité du générateur de code GeneAuto et de son processus de qualification, son architecture est modulaire. Il est constitué de plusieurs modules appelés outils élémentaires. Les spécifications de nature fonctionnelle des outils élémentaires permettent leur vérification et validation indépendamment les uns des autres. Ainsi, le comportement de chaque outil élémentaire dépend uniquement de ses entrées. Une succession de transformations est appliquée sur le système représenté par le circuit d'entrée jusqu'à la production du code. L'architecture du générateur de code est ainsi facilement extensible pour ajouter d'autres éléments, tels que d'autres langages source ou cible. Elle permet également le développement des outils élémentaires en utilisant différents processus, méthodes et outils. Une version simplifiée de cette architecture est présentée dans la figure 3.22.

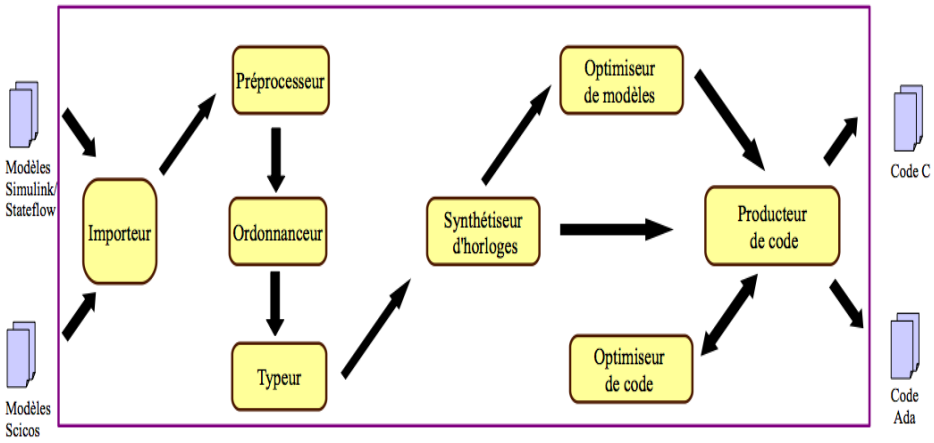


Fig. 3.22 – *Architecture du générateur de code* GeneAuto

Les outils élémentaires constituant le générateur de code GeneAuto sont décrits par les points suivants :

– Importeur : Les modèles Simulink/Stateflow (au format *mdl*) et Scicos (au format Xml) sont lus par ce composant qui se charge ensuite de vérifier l'existence des blocs dans la bibliothèque choisie par l'utilisateur. Cette bibliothèque contient un sous-ensemble sûr de ces langages commun aux besoins de tous les partenaires du projet. Les blocs acceptés par l'importeur des modèles sont les blocs usuels pour construire des systèmes de contrôle et de commande. L'ensemble des blocs Simulink sont cités dans l'annexe A ;

- Préprocesseur : Cet outil élémentaire transforme les modèles graphiques source correctement lus par l'importeur en langage de modélisation de GeneAuto, appelé GeneAuto System Model. Ce langage de modélisation est un langage intermédiaire entre le modèle graphique et le langage cible. Cet outil élimine les sous-systèmes non-atomiques en effectuant leur expansion. Cet outil résout également les blocs de branchement (`goto`) et normalise les modèles pour simplifier les outils suivants ;
- Ordonnanceur : Celui-ci vérifie que le modèle peut être ordonnancé et détermine un ordre d'exécution unique pour chaque bloc. Cet ordre doit respecter la causalité des données et du contrôle telle qu'elle est définie par Simulink/Stateflow ;
- Typeur : Il vérifie si un système ordonnancé est bien typé, c'est-à-dire, s'il est possible de calculer le type de chaque signal, puis si les différents types sont compatibles. Ce composant prend en compte les règles de typage de chaque catégorie de blocs issue de la définition des blocs. Les types considérés pour ce module sont les types communs à tous les utilisateurs industriels ;
- Synthétiseur d'horloges : Ce module est dédié aux systèmes qui présentent différentes fréquences d'exécution des blocs dans leurs modèles. Plus précisément, la fréquence de tous les blocs doit être un multiple d'une fréquence de base commune et la fréquence des blocs contenus dans un sous-système doit être plus élevée que la fréquence du sous-système qui les contient. Le module effectue une synthèse des horloges de chaque signal du système ordonnancé et correctement typé en suivant les flots de données et de contrôle ;
- Optimiseur : L'optimisation est réalisée en 2 phases, la première sur le modèle et la seconde sur le code généré. Plusieurs critères peuvent être considérés (actuellement, la mémoire ou le temps d'exécution) ;
- Producteur de code : La dernière phase de GeneAuto est la génération de code. Celle-ci est effectuée en deux étapes : la génération d'un modèle abstrait de code séquentiel, puis la traduction vers un langage spécifique, C ou Ada. Ce composant doit respecter la règle de génération de code séquentiel pour chaque bloc. La phase d'optimisation est effectuée sur le modèle de code.

Les travaux présentés dans cette thèse étudient les deux outils élémentaires d'ordonnancement et de typage des modèles d'entrée de GeneAuto.

Remarque Étant donné que le projet GeneAuto a sélectionné un sous-ensemble de Scicos qui se comporte comme Simulink, nous ne parlerons par la suite que de Simulink pour éviter de citer systématiquement Simulink et Scicos. Ainsi, dans les chapitres suivants, nous traitons l'ordonnancement et le typage des circuits Simulink. Ce traitement reste le même pour les blocs Scicos.

3.4 Langages et générateurs de code

Plusieurs langages dédiés aux systèmes embarqués existent ainsi que des générateurs de code qui exploitent les modèles exprimés dans ces langages. La plupart de ces générateurs de code sont commerciaux. Nous citons ici une liste non exhaustive de ces langages et des outils associés.

3.4.1 Real Time Workshop-Embedded Coder

Simulink dispose d'un générateur de code pour les modèles Simulink portant le nom de Real Time Workshop-Embedded Coder (RTW-EC) [Cod] qui est l'extension de Real Time Workshop (RTW) [Wor] pour les systèmes embarqués. Ces outils prennent en entrée des modèles Simulink et génèrent un code source en langage C ou C++.

GeneAuto se distingue de RTW-EC par les différences majeures suivantes :

- une architecture flexible ouverte : possibilité de modifier, supprimer et ajouter des transformations, notamment l'ajout de règles d'optimisation spécifiques à chaque utilisateur, ainsi que l'ajout de nouveaux blocs avec des règles de typage et de génération de code spécifiques à chaque utilisateur ;
- un outil libre, ce qui permet de rajouter des extensions pour considérer d'autres langages source ou cible ;
- qualifiable en intégrant les méthodes formelles : RTW-EC peut être qualifié selon les normes de qualification DO-178, IEC 61508 et ISO 26262, en s'appuyant sur des tests du code généré pour assurer qu'il se comporte comme le modèle considéré. Cette qualification s'appuie donc sur un processus de développement particulier et sur des tests dont la couverture est partielle. De plus, aucune vérification formelle n'est réalisée sur le générateur de code ou sur ses composants.

Principalement, le générateur de code GeneAuto représente un sous-ensemble des configurations possibles qu'offrent les générateurs de code Real Time Workshop (RTW) et Embedded Coder (RTW-EC). Plus particulièrement, GeneAuto se focalise sur la transformation des modèles d'entrée en C. D'autres restrictions viennent s'ajouter à la configuration de GeneAuto du fait qu'il considère un sous-ensemble sûr du langage de modélisation.

3.4.2 Target Link

Target Link de DSpace [DSp] est un autre générateur de code pour les modèles Simulink/Stateflow. Il est dédié à la modélisation et la génération de code pour l'industrie automobile. Dans un premier temps, les modèles Simulink/Stateflow importés par Target Link sont traduits dans le langage de modélisation propre à cet outil, puis les modèles obtenus sont raffinés et simulés avant de générer le code C dans un second temps. Le principal avantage de Target Link est le lien étroit avec le domaine des transports.

En terme de qualification Target Link n'est actuellement pas qualifié et n'exploite pas de méthodes formelles.

3.4.3 Scicos C

Scicos est un outil développé au sein du projet Metalau à l'INRIA. Il est utilisé pour la modélisation graphique et la simulation de systèmes dynamiques. Plusieurs domaines d'application sont actuellement supportés par Scicos, notamment le traitement de signal, le contrôle et commande, l'ordonnancement, ainsi que l'étude des systèmes biologiques. Le générateur de code Scicos C produit du code C pour tout modèle écrit en Scicos et Scicos Modelica. Ce générateur de code utilise le même ordonnancement que le simulateur de Scicos et fait appel aux mêmes procédures C utilisées pour les blocs. Ceci assure

la compatibilité entre le code C généré et les résultats obtenus par le simulateur de Scicos. De nouvelles extensions ont visé la génération de modèles basé composant pour des circuits électriques et hydrauliques en utilisant le langage Modelica [2]. Cependant, tous les blocs du modèle sont considérés comme des sous-systèmes `Function_Call Subsystem` ce qui limite les optimisations possibles du code généré. Il est également à noter que le générateur de code Scicos C n'est pas qualifié et ne s'appuie pas sur des méthodes formelles.

3.4.4 Langages et outils synchrones

L'approche synchrone vise à écrire des programmes concurrents déterministes. Les spécifications et les programmes sont écrits de façon à simplifier le processus de vérification de certaines propriétés. Plusieurs langages existent dont certains sont qualifiés de déclaratifs, reposant sur le principe des réseaux de Kahn [Kah74], comme Lustre [HCRP91], Lucid Synchrone [CP00] et Signal [bGJ91], et d'autres sont qualifiés de langages impératifs comme Esterel [BG92].

3.4.4.1 Lustre

Lustre [HCRP91] est un langage de flots de données synchrones où chaque variable du programme représente un flot qui est une suite infinie de valeurs. En Lustre, les objets de base sont le signal et le nœud. Un signal représente un flot de données infini. Un nœud représente l'unité de modularité et définit une fonction (le langage Lustre est fonctionnel) qui exprime une contrainte entre ses entrées et ses sorties. Il comporte, d'une part, une interface composée de signaux d'entrée et de sortie et, d'autre part, d'un ensemble d'équations et de variables locales. Un programme Lustre est donc un ensemble d'équations structurées sous la forme de nœuds et qui définissent les valeurs des variables à chaque instant. Lucid Synchrone est une extension fonctionnelle de Lustre [CP00].

Une version industrielle du langage Lustre est mise en œuvre dans l'outil Scade (Safety Critical Application Development Environment)[3]. Scade est un outil de modélisation, de simulation, de vérification de modèle et de génération de code pour les systèmes embarqués critiques. Outre l'aspect complet de l'outil, Scade 6/Kcg [Dor08] est le seul générateur de code commercial qualifié selon les normes DO-178B niveau A. Cette qualification s'appuie sur son processus de développement conforme au standard et sur des plans de tests.

La nature synchrone de Scade 6/Kcg permet de maîtriser la concurrence inhérente aux modèles flots de données et la synchronisation entre les flots. Toutefois, les outils actuellement disponibles ont certaines limites en terme d'ergonomie et de fonctionnalités et le langage est moins riche et souple que Simulink. En particulier, il ne permet pas de construire des modèles continus. Les utilisateurs ont tendance à préférer Simulink/Stateflow pour la modélisation de leurs applications. C'est pourquoi Esterel a offert une passerelle de Simulink/Stateflow vers Scade afin de bénéficier de la spécification formelle de Lustre et simplifier la tâche de vérification. Plusieurs études académiques sur la correspondance entre les deux langages ont été réalisées [CCM+03, PAA+03].

[2] https ://www.modelica.org/
[3] http ://www.esterel-technologies.com

Cependant, l'expérience industrielle montre qu'une modélisation avec un outil intermédiaire complique les activités de vérification. Des activités supplémentaires sur les modèles SIMULINK/STATEFLOW s'avèrent nécessaires une fois de plus. SCADE utilise des méthodes formelles pour la spécification et la vérification du langage d'entrée, mais n'inclut pas de vérification formelle garantissant que le générateur de code traduit correctement le modèle d'entrée.

3.4.4.2 Signal

SIGNAL [bGJ91] est un langage de modélisation pour les applications temps réel GALS (*Globally Asynchronous, Locally Synchronous*). Il offre l'opportunité de spécifier les circuits à différents niveaux d'abstraction. SIGNAL fait partie de la famille des langages synchrones. Il est proche de LUSTRE en considérant les signaux à la place des flots pour une spécification polychrone (horloges multiples). La différence la plus importante est que le calcul d'horloge sur les signaux de SIGNAL est relationnel alors que celui de LUSTRE sur les flots est fonctionnel. Des travaux ont été menés au sujet de l'utilisation de l'assistant de preuve COQ dans la spécification des programmes SIGNAL [KNT00]. Cela permet d'augmenter la confiance sur les systèmes spécifiés en SIGNAL par une preuve de correction des modèles vis-à-vis des exigences utilisateurs.

PLOYCHRONY [GTL02] est un outil basé sur le langage SIGNAL permettant de fournir un cadre formel pour la validation des transformations de modèles polychrones, et ceci de la spécification jusqu'à l'implantation. De plus, il se concentre sur la vérification formelle des modèles. Une version commerciale utilisant le même principe des horloges de SIGNAL est Sildex RT-Builder [4].

3.4.4.3 Esterel

ESTEREL [BG92] est un langage de programmation synchrone qui se focalise principalement sur la représentation de flots de contrôle dans les systèmes réactifs. Un programme ESTEREL consiste en plusieurs processus concurrents structurés dans des modules. Des versions récentes du compilateur ESTEREL, notamment depuis ESTEREL V5, génèrent du code C. Cependant, l'outil n'est pas qualifié. L'outil POLIS [5] est un environnement de conception et de simulation des systèmes embarqués où les représentations logicielle et matérielle sont combinées. Il exploite le principe des machines à états finis pour la spécification et accepte plusieurs langages de spécification dont ESTEREL pour les composants synchrones. Il se focalise sur la vérification de la conception.

3.4.4.4 Syndex

L'outil SYNDEX [6] fournit un environnement de développement pour les systèmes embarqués temps réel distribués, basé sur des formalismes de graphe avec une sémantique synchrone. Les aspects fonctionnels d'un système sont décrits de façon algorithmique, représentant les exécutions par un graphe de flots de données. La partie algorithmique décrit les langages synchrones (SIGNAL et LUSTRE) sous la forme d'un graphe. La méthodologie AAA permet de spécifier l'architecture physique sur laquelle l'algorithme doit être implanté. L'architecture est un graphe. Les nœuds représentent des supports de communica-

[4] http ://www.geensoft.com/
[5] http ://embedded.eecs.berkeley.edu/Respep/Research/hsc
[6] www-rocq.inria.fr/syndex/

tion ou encore des opérateurs, tandis que les arcs représentent les connexions entre les opérateurs et les supports. Les travaux autour de la génération de code [RBN+03] ne semble pas s'intéresser à la certification du générateur de code. De plus, le générateur de code utilise la vérification formelle plutôt dans le cycle de développement, au niveau de la spécification, ce qui ne garantit pas la correction du générateur de code lui-même.

3.4.4.5 Flots de Données Synchrones (Sdf)

Sdf [LM87] est un cas particulier des graphes flots de données. Il est à l'origine conçu pour programmer les applications de traitement de signal (Digital Signal Processing). Un Sdf est composé d'un ensemble de nœuds, représentant les exécutions, reliés par des arcs, représentant les dépendances de données. Les arcs représentent des flots infinis de valeurs, tout comme pour les langages synchrones, et les nœuds sont des exécutions infinies. Des outils ont été développés en exploitant les flots de données synchrones. Nous citons par exemple Ptolemy [7] [A.L01]. C'est un environnement de spécification, de simulation et de génération de code. Les spécification consistent en des blocs fonctionnels interconnectés entre eux. Le concepteur peut choisir un modèle d'exécution pour un bloc donné. Par exemple, l'évolution des blocs peut être exprimée sous forme d'automates ou de flots de données synchrones. Ptolemy exploite la logique temporelle pour vérifier que le code généré est conforme à la spécification.

3.4.5 Langages de description d'architecture

Les langages de description d'architecture (adl) ajoutent une couche de description de la structure du système (logique et physique) par rapport aux langages synchrones. Le langage de description d'architecture se focalise sur la modélisation des différentes interactions entre les composants haut-niveau du système modélisé. Il existe des adl qui s'appuient sur des spécifications formelles tels que Wright et Rapide, mais ils ne sont en général pas dédiés pour une intégration dans une démarche de génération de code.

3.4.5.1 Aadl

Aadl, Architecture Analysis and Design Language [FGH06], est un langage de description d'architecture (adl) destiné à la modélisation de l'architecture à la fois logicielle et matérielle d'un système embarqué. La spécification d'un système en Aadl consiste en un ensemble de composants qui interagissent via les ports de leurs interfaces. Les composants peuvent être des composants logiciels sous forme de *threads* ou matériels comme les processeurs. Les connexions peuvent quant à elles être immédiates ou retardées comme dans les outils de développement embarqué et temps réel. Les connexions peuvent décrire des flots de données ou de contrôle. Aadl dispose d'une sémantique précise pour l'ordonnancement, la communication et la synchronisation entre les activités. Aadl permet de générer un système fonctionnel à partir de sa description. Plusieurs outils exploitent cette sémantique pour vérifier formellement que l'architecture modélisée satisfait les exigences temporelles du système. Des travaux récents, notamment dans le projet Ocarina [8], visent à intégrer Simulink dans

[7] http ://ptolemy.eecs.berkeley.edu/
[8] http ://ocarina.enst.fr/

AADL afin de bénéficier du cadre de modélisation du premier et d'exploiter les aspects formels du second.

3.4.5.2 Uml

UML, le Langage de Modélisation Unifié [uml07], consiste en un ensemble de notations graphiques pour la création de modèles abstraits d'un système donné. La sémantique d'UML est pauvre, mais peut cependant être raffinée par la définition de profils spécifiques à un domaine. Il s'agit par exemple du profil UML MARTE [mar07] pour la modélisation et l'analyse des systèmes temps réel embarqués. Ce profil contient principalement le langage de spécification de contraintes d'horloges CCSL [And09] (*Clock Constraint Specification Language*) pour spécifier les propriétés temps réel. Plusieurs outils ont été développés pour générer du code à partir de UML. Par exemple, UMODEL [9] est utilisé pour la rétro-ingénierie mais n'est pas conçu pour décrire le contenu des fonctions et n'utilise pas de méthodes formelles. Des travaux récents de J-P. Talpin *et al.* concernent la génération de spécifications exécutables à partir de contraintes CCSL de MARTE [YTB+11].

3.5 FORMALISATION DES CIRCUITS ANALYSÉS

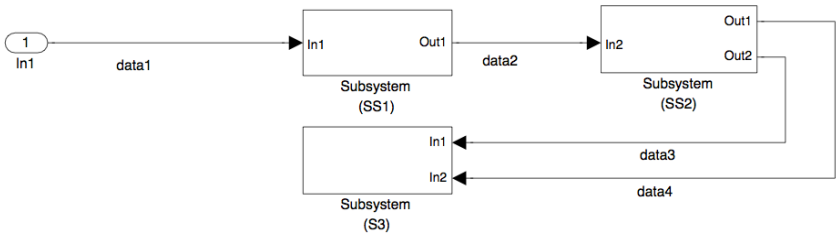
Nous nous focalisons sur des analyses basées sur la structure du modèle d'entrée du générateur de code et non sur la sémantique explicite des blocs. Cette dernière ne sera exploitée que lors de la phase de génération de code qui ne fait pas l'objet de nos cas d'études. Nous ne pourrons donc pour l'instant que prouver la correction de l'implantation par rapport aux exigences et non la correction sémantique des exigences.

Nous présentons donc une spécification formelle du langage d'entrée représentant les circuits SIMULINK acceptés par le générateur de code qui se limite à la structure des modèles et non à leur sémantique. La figure 3.23a est un exemple de circuit *complet*. Il représente une cascade de sous-systèmes $SS1$, $SS2$ et $SS3$. Afin de simplifier notre propos, nous présentons un exemple peu significatif en termes d'application industrielle réelle, mais qui couvre les constructions des circuits SIMULINK acceptés dans GENEAUTO. Un cas d'application industrielle sera présenté dans la partie expérimentale du chapitre 6.

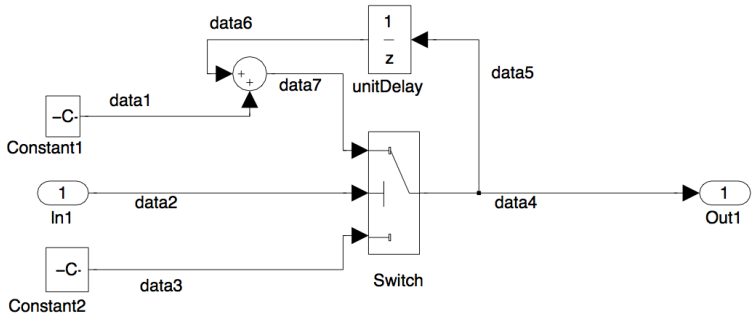
Comme présenté précédemment, un modèle ou circuit peut contenir des blocs élémentaires ou des diagrammes qui sont eux-mêmes des circuits (aspect hiérarchique du langage). Ceci est souvent le cas dans les applications industrielles afin de modulariser les différentes parties du système modélisé. L'entrée de nos outils élémentaires est sous forme de fichiers simples contenant le minimum d'éléments nécessaires sur la structure des circuits donnés en entrée du générateur de code. Ces fichiers simples sont produits par l'adaptateur JAVA et consommés par l'*adaptateur* OCAML (pour plus de détails, se référer à la section 4.3.2.2). Le langage d'entrée décrit dans cette section correspond alors à une représentation structurelle (simplifiée) des circuits donnés en entrée du générateur de code.

Par exemple, le sous-système $SS1$ est un circuit purement flots de données. Il est composé de blocs combinatoires et d'un bloc séquentiel $UnitDelay$. Le bloc combinatoire Sum peut être considéré comme une fonction $Sum((data6, data1), data7)$, où Sum est le nom de l'opérateur (c'est-à-dire

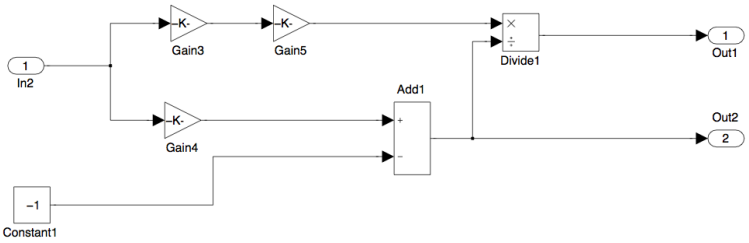
[9] http ://www.altova.com/umodel.html



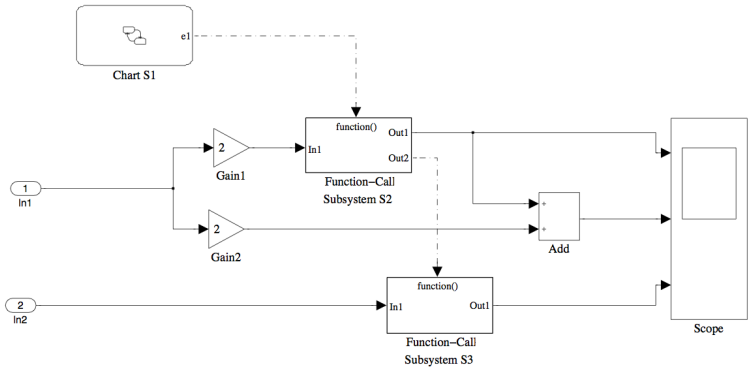
(a) *Système complet (S)*



(b) *Sous-système (SS1)*



(c) *Sous-système (SS2)*



(d) *Sous-système (SS3)*

FIG. 3.23 – *Exemple de circuit complet en* SIMULINK

l'addition dans ce cas) paramétré par la liste de ses entrées (*data*1 et *data*6) et de ses sorties (*data*7 dans cet exemple). Les signaux reliants les blocs sont tous des signaux de données, nous les avons dénotés par *data*xx.

Le sous-système *SS*2 est un circuit purement flots de données sans blocs séquentiels. Le sous-système *SS*3 est un circuit qui mélange les flots de don-

nées et de contrôle. Les signaux de contrôle sont représentés par des pointillés étiquetés par *control*xx.

Tout bloc de base est constitué de ports internes. Tel est le cas du bloc *Add*1 de la figure 3.23c par exemple. Il possède deux ports d'entrée et un port de sortie. Un sous-système est également composé de ports d'entrée et de sortie. Toutefois, ces ports sont représentés par des blocs `In` (pour les entrées) et `Out` (pour les sorties). Par exemple, le bloc *In*1 de la figure 3.23b est le port d'entrée du sous-système *SS*1.

Les connexions entre les blocs labellisées par *data*xx et *control*xx résultent des liaisons entre ports de sortie et ports d'entrée. En effet, par exemple, la connexion de données *data_*2 du système *S* relie le port de sortie *Out*1 du sous-système *SS*1 et le port d'entrée *In*2 du sous-système *SS*2. Un signal relie toujours un port de sortie d'un bloc source et un port d'entrée du bloc cible.

Les études focalisées sur l'analyse numérique de Simulink prêtent plus d'attention à la sémantique opérationnelle des blocs du langage, le lecteur peut se référer aux recherches autour de l'analyse statique de Simulink [Cha08]. Ce n'est cependant pas l'objet de nos travaux actuels.

3.5.1 Représentation des circuits

De par la description du langage Simulink présenté dans la section 3.2.1, un circuit d'entrée du générateur de code (ou des outils élémentaires) est défini par une structure de graphe particulière décrite par la grammaire simplifiée suivante :

$$
\begin{array}{lcl}
D & ::= & \langle G,\, E\rangle \\
G & ::= & \langle C_1, \ldots, C_n\rangle \\
E & ::= & \langle S_1, \ldots, S_m\rangle \\
C & ::= & B \mid D \\
B & ::= & op(p,i,o) \mid seq(p,i,o) \\
p & ::= & param(\overrightarrow{v}) \\
S & ::= & data(s,\, t) \mid control(s,\, t) \quad (s,\, t \in \mathbb{N} \times \mathbb{N})
\end{array}
$$

Un diagramme de blocs Simulink D est décrit par G qui est une suite de nœuds C_i et par E qui est une suite d'arcs S_i. La cardinalité d'un circuit représente le nombre de nœuds constituant le circuit d'entrée. Un nœud C peut être un bloc de base B ou encore un sous-diagramme D. Les blocs de base peuvent être des blocs combinatoires ou des blocs séquentiels. Un bloc combinatoire calcule la suite de sorties dont le nombre est de o en fonction de tous les paramètres p et les i ports d'entrée. Ce calcul dépend de l'opérateur du bloc $op(p,i,o)$. D'autre part, un bloc séquentiel, caractérisé par $seq(p,i,o)$, calcule les sorties en fonction des valeurs calculées dans des cycles antérieurs des entrées. Un bloc séquentiel ne peut pas intervenir dans le flot de contrôle. Un paramètre $param(\overrightarrow{v})$ est utilisé pour configurer un bloc. Par exemple, il est nécessaire de disposer d'une valeur initiale v pour chaque bloc *UnitDelay*. Les blocs sont reliés par des signaux S qui peuvent être des signaux de données $data(s,\, t)$ ou de contrôle $control(s,\, t)$, avec s et t les couples $(bloc\ source, port\ source)$ et $(bloc\ cible, port\ cible)$. *bloc source* et *bloc cible* représentent les indices des blocs dans l'intervalle $\lfloor 0, size-1\rfloor$, où *size* est le nombre de blocs dans le circuit. Les *port source* et *port cible* sont les indices des ports dans la séquence des ports du bloc concerné. Ils varient dans l'intervalle $\lfloor 0, size_b-1\rfloor$, où $size_b$ est la cardinalité des ports d'entrée ou de sortie d'un bloc b.

Nous choisissons dans le projet GeneAuto de traiter des circuits à un seul niveau. Autrement dit, il n'y a pas de systèmes imbriqués dans d'autres systèmes. Les niveaux hiérarchiques sont préalablement dépliés au niveau du préprocesseur avant qu'ils ne soient traités dans la chaîne d'outils du générateur de code.

La formalisation en Coq de la structure de données des circuits acceptés en entrées des outils élémentaires expérimentés dans le cadre de cette thèse est donnée dans la section suivante.

3.6 Assistant de preuve Coq

Il existe de nombreux assistants de preuve matures que nous pouvions utiliser pour l'expérience GeneAuto, nous citons par exemple : *Isabelle* [Pau93], *Hol* [M-J88], *PVS* [OSR95] et Coq [BC04]. Nous avons choisi d'utiliser l'assistant de preuve Coq pour expérimenter la spécification, l'implantation et la vérification des outils élémentaires étudiés dans cette thèse. Ce choix repose d'abord sur le fait que nous avions une connaissance de l'utilisation de Coq mais également que le noyau de Coq est très riche. Celui-ci dispose également de plusieurs langages cibles pour l'extraction de programmes que nous pourrons intégrer simplement dans la chaîne de développement de GeneAuto. Ce choix est motivé par les travaux autour de la certification du vérifieur de preuve et de l'extracteur Coq [Let04, Bar99]. Nous allons présenter dans cette section l'assistant de preuve Coq avec des exemples issus de notre implantation.

3.6.1 Introduction à Coq

Coq est un assistant de preuve dans lequel des spécifications sont exprimées dans le langage Gallina. Ce dernier est dérivé du calcul des constructions inductives [CH88], un λ-calcul richement typé. Coq n'est pas seulement un assistant de preuve, mais également un atelier de construction de programmes corrects par extraction depuis la spécification vérifiée. Les preuves sont construites impérativement à l'aide des tactiques, qui sont des outils préalablement existants dans l'assistant de preuve.

Coq est intéressant pour GeneAuto car il permet d'extraire un code fonctionnel certifié et relativement efficace en préservant les propriétés prouvées correctes dans le code source. Les langages fonctionnels cible du mécanisme d'extraction de Coq sont Objective Caml, Haskell et Scheme.

Une spécification en Coq représente un type. Par conséquent prouver une spécification revient à trouver un terme d'un tel type. Un programme typé peut être généré automatiquement grâce à l'extracteur de Coq. Les termes peuvent être construits en utilisant les quantificateurs, le pattern-matching, l'abstraction et l'application fonctionnelle ainsi que plusieurs autres constructions classiques préexistantes dans les langages de programmation fonctionnelle.

Nous allons aborder les deux facettes de l'assistant de preuve Coq : un langage de programmation avec lequel il est possible d'écrire des spécifications mais aussi des programmes et un outil qui permet d'écrire des théorèmes (décrivant des propriétés) et de les prouver de manière interactive. Nous nous focalisons dans ce manuscrit sur une description minimale du langage de spécification. Nous illustrons celle-ci avec quelques exemples, issus de notre implantation. Nous présentons ces concepts de base de la programmation et la démonstration avec cet assistant de preuve.

Techniquement, concernant la mise en œuvre en Coq [10], nous allons illustrer quelques concepts sur des exemples de spécification des circuits Simulink. Il y a trois méta-types en Coq. Premièrement, le type `Set` qui représente les spécifications des données, où ces spécifications sont elles-mêmes des types. Le type `Prop` est le type des propositions. Il est utilisé pour le raisonnement. Finalement, le type `Type` qui est le super-type des deux types précédents.

Dans le double but d'illustrer l'implantation du langage d'entrée et d'expliquer les concepts principaux de l'assistant de preuve, nous donnons dans la suite des extraits de notre spécification de la structure de données des circuits acceptés par GeneAuto.

3.6.1.1 Types inductifs

Nous avons vu, dans la section 3.2.1, qu'un bloc peut être combinatoire ou séquentiel. Nous utilisons les types inductifs pour spécifier les catégories des blocs.

```
Inductive BlockKindType : Set :=
| Sequential : nat -> nat -> BlockKindType
| Combinatorial : nat -> nat -> BlockKindType.
```

Les constantes `BlockKindType`, `Sequential` et `Combinatorial` sont rajoutées à l'environnement de spécification. Cet exemple montre que le type inductif `BlockKindType` est défini par la commande `Inductive` et par la liste de ses constructeurs et de leurs types respectifs. Le mot clé `Set` indique que le type proposé est une structure de données. Un bloc peut donc être soit séquentiel soit combinatoire. Par exemple, le constructeur `Sequential` est de type $(nat \rightarrow nat \rightarrow BlockKindType)$. Il prend deux arguments entiers naturels qui représentent respectivement le nombre de ports d'entrée et le nombre de ports de sortie du bloc. `nat` est un type inductif prédéfini en Coq représentant les entiers naturels de `Peano`. D'autres structures de données seront également illustrées sur le langage d'entrée représentant les circuits Simulink. La définition d'un type inductif engendre des tactiques de preuve par induction structurelle associées :

```
BlockKindType is defined
BlockKindType_rect is defined
BlockKindType_ind is defined
BlockKindType_rec is defined
```

Coq fournit un certain nombre de destructeurs pour `BlockKindType`. Ici, les destructeurs sont `BlockKindType_rect`, `BlockKindType_ind` et `BlockKindType_rec`. Ils correspondent respectivement aux principes d'élimination sur `Type`, `Prop` et `Set`. Le destructeur `BlockKindType_ind` est un principe d'élimination pour `BlockKindType`.

Le type de `BlockKindType_ind` est donné par la commande `Check BlockKindType_ind` dont le résultat est :

```
BlockKindType_ind
     : forall P : BlockKindType -> Prop,
       (forall n n0 : nat, P (Sequential n n0)) ->
       (forall n n0 : nat, P (Combinatorial n n0)) ->
       forall b : BlockKindType, P b
```

Cela permet de prouver la propriété $(\forall\, b : BlockKindType,\, P\, b)$ par induction sur b. Les types de `BlockKindType_rec` et de `BlockKindType_rect` sont

[10] Notre implantation ainsi que les exemples cités dans le manuscrit sont réalisés avec la version 8.2 de Coq. La version intégrée dans GeneAuto est en version 8.1

P : *BlockKindType* → *Set* et P : *BlockKindType* → *Type*. Ils correspondent aux principes d'induction sur `Set` et `Type`. En règle générale, pour un type inductif `ident`, la constante `ident_ind` est toujours fournie. Toutefois, `ident_rec` et `ident_rect` peuvent être impossibles à fournir, notamment lorsque `ident` est une proposition.

La commande `Check` est utilisée pour vérifier si une expression est bien formée et nous apprend de quel type est cette dernière. Par exemple, le type de `BlockKindType` est la sorte `Set`.

```
Check BlockKindType.
Blockkindtype
     : Set
```

Un type peut également être défini par le mot clé `Record` qui représente un enregistrement. Il correspond à une définition inductive à un seul constructeur.

Par exemple, un bloc de base est défini en utilisant cette construction.

```
Record BlockBaseType : Set :=
 makeBlockBase {
  blockKind : BlockKindType ;
  inDataPorts : nat ;
  outDataPorts : nat;
  controlPorts : nat;
  operator : BlockOperator;
  paramOp : list param;
  blockUserDefinedPriority : option nat ;
  blockAssignedPriority : nat ;
  blockIndex : nat
 }.
```

Les champs de cet enregistrement sont les éléments nécessaires à la définition d'un bloc. D'une part, cette structure doit contenir les données sur la sémantique du bloc :

- la nature du bloc déterminée par le champ `blockKind`
- l'opération effectuée par le bloc déterminé par le champ `operator` qui est de type `BlockOperator` :

```
Inductive BlockOperator : Set :=
|Seq : seqBlock -> BlockOperator
|Combin : combinBlock -> BlockOperator
with seqBlock : Set :=
|unitDelay : nat -> nat -> seqBlock
...
 with combinBlock : Set :=
|sum : nat -> nat -> combinBlock
...
```

et, d'autre part, les données sur la structure du bloc :

- les paramètres de configuration (*paramOp*) spécifiés par :

```
Inductive param : Set :=
|minus : param
|plus : param
...
```

  Il s'agit de décrire la suite des opérations à réaliser par le bloc (notamment lorsqu'il s'agit de bloc générique comme le bloc `Add`), de spécifier la valeur initiale, etc.
- le nombre de ports de données d'entrée (respectivement de sortie), qui sont distingués par leurs indices : *inDataPorts* (respectivement *outDataPorts*) ;
- le nombre de ports de contrôle est indiqué par `controlPorts`.

Outre les données relatives à la structure du bloc et sa connexion avec le circuit, d'autres informations déduites des exigences s'ajoutent lors de l'analyse

des circuits. Il s'agit des priorités des blocs qui sont explicitement spécifiées par le concepteur du circuit et la priorité implicite déduite de ce dernier (position graphique).

Nous manipulons un autre type inductif, il s'agit du type Coq $\{P\} + \{Q\}$ de type `Set`, avec `P` et `Q` des propriétés logiques (de type `Prop` en Coq). $\{P\} + \{Q\}$ est défini à l'aide des deux constructeurs `left : P` $\rightarrow \{P\} + \{Q\}$ et `right : Q` $\rightarrow \{P\} + \{Q\}$. Un terme de type $\{P\} + \{Q\}$ est soit de la forme `left h` avec `h` une preuve de `P`, ou de la forme `right h` avec `h` une preuve de `Q`.

Dans le cas où `Q` est égal à $\neg$`P`, le type $\{P\} + \{\neg P\}$ correspond à un type booléen complété par une preuve de `P` ou de sa négation.

3.6.1.2 Types dépendants

Nous pouvons également définir une structure de données en utilisant les types dépendants. Un type dépendant est un type dont la structure dépend d'une valeur. C'est le cas lorsque l'on définit un diagramme (`DiagramType`) paramétré par un type `ModelElementType`, sous forme d'un enregistrement dont le champ `blocks` est de type `ModelElementType`.

```
Record DiagramType (ModelElementType : Set) : Set :=
 makeDiagram {
  blocks : list ModelElementType ;
  dataSignals : list DataConnexionType ;
  controlSignals : list ControlConnexionType ;
  blocksNumber : nat
 }.
```

Un circuit `ModelElementType` est maintenant défini comme un type inductif dont les constructeurs sont soit un bloc de base soit un diagramme de blocs tels qu'ils sont présentés ci-dessus. Il faut noter ici que le constructeur `Diagram` est défini récursivement, il est de type `DiagramType ModelElementType`, ce qui revient à dire qu'un diagramme peut être composé de blocs de base et d'autres diagrammes.

```
Inductive ModelElementType : Set :=
  | Block : BlockBaseType -> ModelElementType
  | Diagram : DiagramType ModelElementType -> ModelElementType.
```

Un autre exemple de type dépendant est le type `I`. Il s'agit de l'ensemble des parties de l'ensemble `T.carrier`, avec `T` une implantation des ensembles finis dont les éléments sont de type `FiniteSet.t`. `I` est donc défini avec la condition que ces éléments soient inclus dans `carrier`. Autrement dit, $\forall\, p$: `FiniteSet.t`, $p \subseteq$ `carrier`.

```
Definition I :=
 { p : T.FiniteSet.t | T.FiniteSet.Subset p T.carrier }.
```

Le type de `I` est dépendant de la proposition (en forme mathématique) $p \subseteq carrier$.

La manipulation des types dépendants demande souvent de fournir des preuves de correction du typage outre la spécification. La programmation par la preuve permet justement de simplifier la tâche à l'utilisateur, comme nous le verrons dans la section 3.6.1.6.

Plus de détails sur l'utilisation de Coq, dans notre cas d'étude, seront donnés ultérieurement dans les chapitres 6 et 7.

3.6.1.3 Définitions et fonctions

Nous devons effectuer des calculs qui exploitent les données présentées avant. Soit la fonction `size` définie par :

```
Definition size (model : ModelType) :=
  length (blocks ModelElementType model).
```

La commande `Definition` permet de définir des types mais aussi des fonctions. Tel est le cas de l'exemple `size` où la taille d'un circuit est définie par la fonction `length` de la bibliothèque `List` de Coq paramétrée par la liste des blocs du circuit d'entrée `model`.

Une possibilité pour écrire une fonction récursive consiste à utiliser la commande `Fixpoint` qui est l'équivalent du `let rec` de OCaml. De plus, `Fixpoint` décrit une fonction structurellement récursive, dont l'un des arguments doit décroître à chaque appel récursif. Cela garantit la terminaison des fonctions qui sont les seules fonctions récursives acceptées dans Coq.

```
Fixpoint dataSourceFor (dataSignals : list DataConnexionType)
       (block : nat) { struct dataSignals } :=
  match dataSignals with
    | nil => nil
    | (DataSignal src dst) :: queue =>
      if (eq_nat_dec block (fst dst))
      then (fst src) :: (dataSourceFor queue block)
      else (dataSourceFor queue block)
  end.
```

La fonction `dataSourceFor` calcule la liste des blocs qui produisent les données du bloc transmis en paramètre `block`, en exploitant la liste des signaux (connexions) de données `dataSignals`. La notation {`struct dataSignals`} indique à Coq que la fonction est structurellement récursive vis-à-vis du type du paramètre `dataSignals`. Cette indication sert à assurer la terminaison de la fonction lors de sa construction.

Coq impose la définition de fonctions totales. Lors de la définition d'une fonction récursive, il est donc nécessaire de fournir les arguments pour la preuve de terminaison de cette fonction.

3.6.1.4 Modules

Coq dispose également d'une notion de module statique selon le même principe des modules OCaml tel qu'il est illustré dans l'analogie suivante. La partie gauche (respectivement droite) illustre un module Coq (respectivement OCaml).

```
Module Type T.
 Parameter S : Set.
 Parameter p : S.
End T.
```

```
module type T = sig
  type S
  val p : S
end
```

La signature `T` est définie par l'ensemble `S` et un paramètre `p` de type `S`. Cette signature sert d'interface qui est extensible lors de la création d'un module ayant la signature de `T`. Ce module est appelé foncteur. Un foncteur de module est une fonction prenant (ou non) un ou plusieurs modules en argument et produisant un autre module.

```
Module M : T.
  Definition S : Set := nat.
  Definition v : S := 0.
  Definition p := v.
End M.
```

```
module M = struct
  type S  = nat
  let v = 0
  let p = v
end
```

Ici, il ne s'agit pas seulement de créer le module `M` mais également de vérifier qu'il est de signature `T`.

3.6.1.5 Preuves de théorème

Abordons maintenant, le coté preuve en Coq. L'utilisateur peut énoncer un lemme ou un théorème puis construire sa preuve interactivement grâce au langage de tactiques et de commandes.

```
Lemma sym : forall x y : nat, x = y -> y = x.
  Proof.
    symmetry.
    apply H.
  Qed.
```

Les mots clés `Proof` et `Qed` sont des commandes qui délimitent les étapes du raisonnement. Les commandes comprises entre `Proof` et `Qed` sont appelées tactiques. Elles indiquent à Coq les étapes qui lui faut pour atteindre le but attendu. La tactique `symmetry` s'assure que les deux termes `x` et `y` sont égaux modulo les règles de conversion du calcul. Il s'agit de vérifier l'équivalence par symétrie. Le système de preuve de Coq indique donc un sous-but :

```
1 subgoal

  x : type
  y : type
  H : x = y
  ============================
   x = y
```

Le sous-but correspond à l'hypothèse H, il suffit d'appliquer cette hypothèse avec la tactique `apply`.

3.6.1.6 Programmation par la preuve

Il est également possible de mélanger preuve et programmation en utilisant la preuve interactive pour compléter la construction d'une définition dont la spécification est complexe. C'est l'objet de l'exemple suivant.

Soit une nouvelle définition de type T qui compte parmi ses paramètres le module `FiniteSet` de type `FSetInterface.S`. Ce type provient d'une bibliothèque de Coq qui contient une signature pour implanter les ensembles ordonnés finis. L'accès à l'un des champs de la signature se fait par ".". Par exemple `FiniteSet.t` permet d'accéder au champ `t` de `FiniteSet`, qui représente un type.

```
Module Type T.
  Declare Module FiniteSet : FSetInterface.S
  ...
  Variable carrier : FiniteSet.t.
End T.
```

La programmation par la preuve consiste à spécifier un programme qui doit avoir un certain type et de construire la preuve que ce programme existe (le type est habité) puis d'en extraire le programme. Un exemple est de définir la constante `bot` interactivement par une preuve qui lui associe l'ensemble vide.

```
Definition bot : I.
 Proof.
  exists T.FiniteSet.empty.
  unfold T.FiniteSet.Subset.
  intros.
  absurd (T.FiniteSet.In a T.FiniteSet.empty).
  firstorder.
  exact H.
 Defined.
```

La commande `Defined` est équivalente à la commande `Qed` utilisée précédemment, à la seule différence que les preuves terminées par `Defined` peuvent être utilisées dans les calculs. En terminant la ligne `Definition bot : type` par un point au lieu de `:=`, nous indiquons à Coq que la définition de `bot` est interactive. Coq nous demandera alors la preuve interactive de la propriété «le type `I` est habité ». Le terme construit comme habitant de `I` sera par conséquent la définition de `bot`.

```
1 subgoal

  ============================
   I
```

Nous allons maintenant suivre pas à pas la construction interactive de ce programme. La première étape consiste à fournir un sous-ensemble de `I` qui est, dans notre cas, l'ensemble vide `T.FiniteSet.empty` en utilisant la tactique `exists`.

```
1 subgoal

  ============================
   T.FiniteSet.Subset T.FiniteSet.empty T.carrier
```

La preuve consiste ensuite à remplacer le terme `subset` par sa définition avec la commande `unfold T.FiniteSet.Subset`.

```
1 subgoal

  ============================
   forall a : T.FiniteSet.elt,
   T.FiniteSet.In a T.FiniteSet.empty
     -> T.FiniteSet.In a T.carrier
```

L'étape suivante consiste à introduire les hypothèses `a` et `H` avec la tactique `intros`.

```
1 subgoal

  a : T.FiniteSet.elt
  H : T.FiniteSet.In a T.FiniteSet.empty
  ============================
   T.FiniteSet.In a T.carrier
```

De façon informelle, cela veut dire que nous avons comme hypothèse que `a` est un élément appartenant à l'ensemble vide, et que nous voulons prouver que `a` appartient à `carrier`. Nous procédons alors à une preuve par l'absurde en se basant sur l'hypothèse `H`. Nous avons dans `H`, la propriété $a \in \emptyset$, la tactique `absurd T.FiniteSet.In a T.FiniteSet.empty` génère deux sous-buts :

```
2 subgoals

  a : S.FiniteSet.elt
  H : S.FiniteSet.In a S.FiniteSet.empty
  ============================
   ~ S.FiniteSet.In a S.FiniteSet.empty
```

```
subgoal 2 is:
 S.FiniteSet.In a S.FiniteSet.empty
```

Le premier sous-but est résolu par la tactique *firstorder* qui raisonne sur les définitions inductives de la logique du premier ordre. Le symbole $\backsim$ est l'équivalent de la négation logique d'une proposition P ($\backsim P \equiv P \rightarrow False$). Par définition, `empty` ne contient aucun élément, ce qui correspond au but à prouver. Le second sous-but est exactement l'hypothèse `H` et est résolu par la commande `exact`.

3.6.2 Extraction de programmes

Le mécanisme d'extraction de Coq permet de transformer un programme Coq en un programme OCaml, Haskell ou Scheme. L'extraction est effectuée par la commande :

```
Extraction Language Ocaml.
```

Par défaut, le langage cible de l'extraction est OCaml. Voici un exemple de code OCaml extrait automatiquement de la spécification des diagrammes de blocs décrite précédemment. Le code OCaml extrait est une définition du type *diagramType*. Il s'agit d'une structure dépendant du type *modelElementType*.

```
Record DiagramType (ModelElementType :
    Set) : Set :=
 makeDiagram {
  blocks : list ModelElementType ;
  dataSignals : list DataConnexionType
      ;
  controlSignals : list
      ControlConnexionType ;
  blocksNumber : nat
```

```
type 'modelElementType diagramType =
        { blocks : 'modelElementType
            list;
        dataSignals : dataConnexionType
            list;
        controlSignals :
            controlConnexionType list;
        blocksNumber : nat }
```

Le code extrait de la définition `bot` correspond à :

```
let bot =
  S.FiniteSet.empty
```

où la partie purement logique constituant la preuve est supprimée.

Un autre exemple d'extraction concerne la fonction `dataSourceFor` précédemment présentée.

```
let rec dataSourceFor dataSignals block =
  match dataSignals with
    | Nil -> Nil
    | Cons (d, queue) ->
        let DataSignal (src, dst) = d in
        (match eq_nat_dec block dst with
           | Left -> Cons (src, (dataSourceFor queue block))
           | Right -> dataSourceFor queue block)
```

Le but de l'extracteur est de produire du code exécutable dans un langage de programmation usuel tout en préservant (par construction) les propriétés prouvées correctes. Ainsi, l'extracteur, pour produire un programme OCaml, supprime les parties logiques du programme Coq en question (propriétés et preuves). Les calculs occasionnés par la suppression des termes logiques dont le type est `Prop` sont alors réduits. Or, des exemples où les preuves supprimées sont nécessaires dans le code OCaml obtenu peuvent être trouvés dans la thèse de P. Letouzey [Let04]. L'un de ses exemples est la fonction de division entière `div` : $\mathbb{N} \rightarrow \{n \in \mathbb{N} \mid n \neq 0\} \rightarrow \mathbb{N}$, où la preuve que $n \neq 0$ sera supprimée

par l'extracteur. Des travaux autour de la certification du noyau de Coq et de son extracteur font l'objet de recherche depuis [Let04] et la thèse en cours de S. Glondu [Glo09]. Le processus de certification de nos travaux prend en considération les études menées autour de la certification de l'extracteur de Coq.

3.7 Conclusion

Nous avons présenté, dans ce chapitre, les principaux concepts du langage Simulink/Stateflow qui représente le langage d'entrée du générateur de code GeneAuto. Nous avons également décrit l'architecture de GeneAuto, ses objectifs ainsi que les outils similaires utilisés dans l'industrie. Le générateur est composé de plusieurs outils élémentaires indépendants les uns des autres en termes de spécification fonctionnelle. Ce choix d'architecture modulaire permet, outre la réduction du coût de la vérification et la maintenance du générateur de code, d'envisager différentes techniques de vérification et différents langages de programmation. Nous avons également montré, à travers les générateurs de code existants, que GeneAuto est qualifiable en intégrant des méthodes formelles dans son processus de développement et de vérification. Nous avons choisi d'expérimenter le développement et la vérification de certains outils élémentaires à l'aide de l'assistant de preuve Coq. GeneAuto est donc constitué d'outils élémentaires développés de manière classique en Java et d'autres outils élémentaires formellement développés en Coq.

4 Processus de développement d'outils élémentaires

4.1 Motivation

L'utilisation des méthodes formelles dans les applications industrielles critiques est une nécessité pour répondre à la complexité croissante des applications et des exigences de sûreté. Dans le cadre de GeneAuto, les outils élémentaires qui constituent le générateur de code pouvaient être développés selon deux approches, d'une part, un processus classique pour les outils élémentaires en Java et, d'autre part, un processus qui combine approche classique et méthodes formelles pour ceux réalisés à l'aide de l'assistant de preuves Coq. En outre, les outils sélectionnés pour un développement formel sont des modules dont la vérification à base de relecture et de tests est jugée difficile par les développeurs de ces outils.

L'intégration d'approches formelles dans un processus industriel classique pour le développement d'outils qualifiés est un des résultats importants de GeneAuto. Il s'agit d'une part de prendre en compte les contraintes industrielles sous leur forme habituelle, à savoir les exigences en langage naturel, les langages de modélisation utilisés, l'interopérabilité des composants et les normes de certification ; d'autre part, de proposer une approche fondée sur l'assistant de preuves Coq.

Cela nous a conduit à suivre des approches différentes des solutions habituelles pour l'exploitation des méthodes formelles. En effet, celles-ci s'appuient souvent sur une définition de la sémantique des langages source et cible, ainsi que sur la preuve de préservation de cette sémantique. Par exemple, Blazy et *al.* [BL09] s'intéressent aux événements d'entrée/sortie avec l'environnement du programme et montrent que la cible en assembleur effectue les mêmes séquences d'entrée/sortie que le code source en C. De même, les auteurs de [PSS98, Riv04] considèrent les modifications de la mémoire effectuées par les programmes.

Dans le cadre de GeneAuto, nous ne nous sommes pas appuyés directement sur la sémantique des langages mais sur des exigences écrites en langage naturel par les utilisateurs du générateur. Ces exigences sont dérivées implicitement de cette sémantique. Dans un développement formel, ces exigences ne sont en général pas exploitables telles quelles, mais doivent être raffinées, voire complétées par des exigences dérivées, avant de les transcrire en Coq et de poursuivre les développements. Le raffinement consiste à introduire de nouvelles exigences qui sont des conséquences logiques des précédentes à un niveau de détails plus précis. La dérivation consiste à introduire de nouvelles

exigences pour satisfaire des objectifs qui ne faisaient pas partie des exigences précédentes (par exemple, construire des composants qui seraient réutilisables dans d'autres outils). Cette phase de transcription nous a permis d'améliorer la qualité des exigences en langage naturel. Nous avons choisi de conserver le langage naturel comme référence, d'une part, pour simplifier les interactions avec les autorités de certification et, d'autre part, pour éviter d'introduire des contraintes supplémentaires pour les autres équipes assurant le développement des composants de GeneAuto avec les technologies classiques.

4.2 Processus de développement dans GeneAuto

Le cycle de vie du développement de GeneAuto suit les étapes présentées dans la figure 4.1.

Définition des exigences utilisateurs Il s'agit des besoins des utilisateurs (services attendus) par rapport aux outils. Ces exigences sont indépendantes des choix d'implantation et sont données par les utilisateurs ;

Planification Dans cette phase, tous les plans de développement, de vérification, d'intégration et de qualification sont définis dans le but de déterminer les méthodes à appliquer et les outils à utiliser. Dans un premier temps, il s'agit d'analyser, d'une part, les exigences des utilisateurs pour identifier l'ensemble des contraintes de développement et, d'autre part, les recommandations de qualification. Ensuite, la description des processus est formalisée dans les différents plans. Deux approches de processus sont distinguées dans GeneAuto : approche classique et approche formelle. Enfin, dans cette phase sont développés les standards qui guideront les processus de développement classique (spécification, conception et implantation) et ceux qui guideront le développement formel (spécification formelle et conception) ;

Conception architecturale Les exigences outils sont liées étroitement aux choix d'implantation et sont fournies par les développeurs du logiciel. Dans cette phase, les exigences globales (utilisateurs et outils) sont raffinées et l'architecture de l'outil y est définie. Les activités de cette étape consistent à définir les outils élémentaires et leurs adaptateurs (interfaces), à déterminer les exigences utilisateurs et outils de chaque outil élémentaire, et à définir les exigences fonctionnelles qui permettent d'effectuer la vérification de l'intégration. Cette phase détermine également les parties du logiciel sujettes à des activités de qualification ;

Activités d'un outil élémentaire Les outils élémentaires du générateur de code sont relativement indépendants. Ils peuvent donc être développés en utilisant différentes technologies. Dans GeneAuto, certains outils élémentaires ont été développés en langage Java et d'autres dans le langage de l'assistant de preuve Coq. Le développement et la vérification des outils élémentaires suivent donc soit le processus d'un développement classique soit le processus formel. Ces activités regroupent la spécification, la conception, l'implantation et la livraison de l'outil élémentaire. Nous décrivons particulièrement cette phase dans la section suivante ;

Intégration Elle a pour objectif d'intégrer tous les outils élémentaires qui composent GeneAuto afin de produire le logiciel GeneAuto complet ;

Livraison Cette phase consiste à produire l'outil GeneAuto final ainsi que la documentation associée.

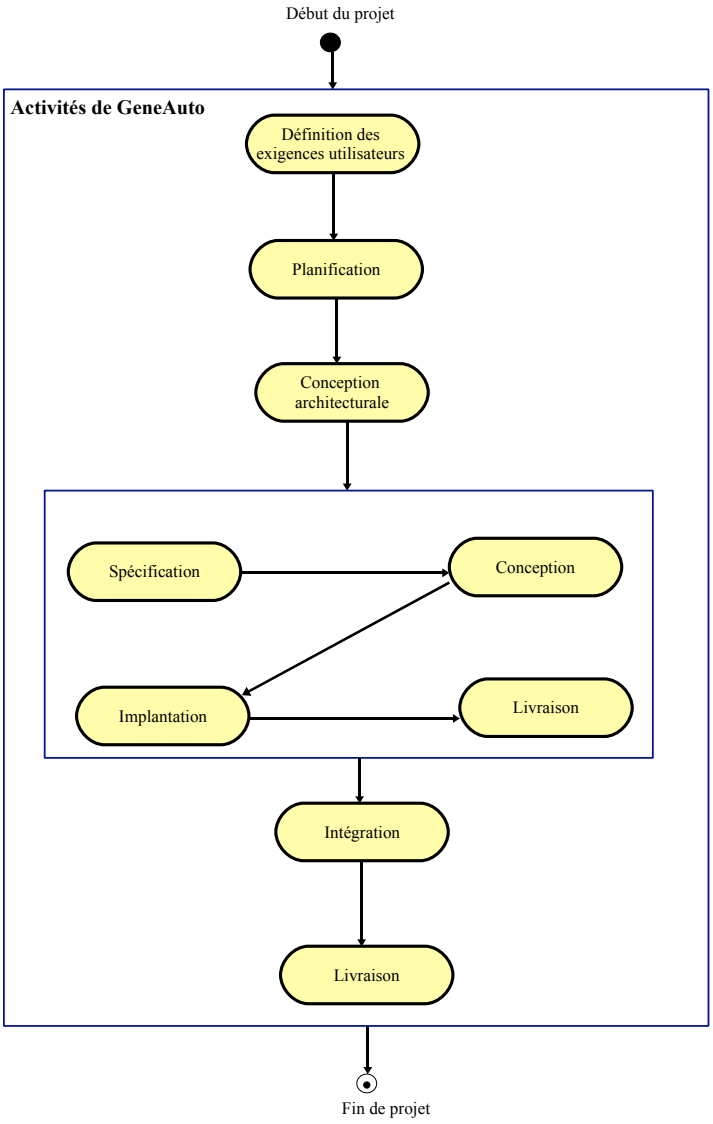


FIG. 4.1 – *Cycle de vie de développement de* GENEAUTO

4.3 PROCESSUS DE DÉVELOPPEMENT DES OUTILS ÉLÉMENTAIRES

Le cycle de vie d'un logiciel est défini par les étapes effectuées pour construire ce logiciel. Il existe plusieurs modèles de cycle de développement de logiciels [Som06]. Le cycle en V est le plus répandu dans l'industrie du développement de logiciels, même s'il s'agit d'une vision idéalisée. La phase descendante

représente les étapes de construction du logiciel, et la phase ascendante les étapes de validation et de vérification. Ce cycle de vie est celui adopté pour le développement de GeneAuto.

4.3.1 Plan de développement classique

Nous résumons dans cette section le processus sous forme de plan de développement des outils élémentaires avec des technologies classiques (Java dans GeneAuto). Les modèles d'entrée des outils élémentaires sont des fichiers Xml contenant l'intégralité des informations sur le système traité, y compris celles qui sont générées par les outils élémentaires appliqués auparavant. Ces informations ne sont pas forcément utiles à chaque outil élémentaire.

4.3.1.1 Spécification des exigences

Cette phase décrit les exigences applicables aux analyses qui seront implantées dans les outils élémentaires. Elle consiste à analyser les exigences des utilisateurs, l'architecture et les interfaces (entrées/sorties) des outils élémentaires. Ces exigences utilisateurs sont ensuite raffinées en exigences pour cet outil élémentaire. Le raffinement consiste à décrire le traitement à appliquer aux entrées de l'outil élémentaire pour produire ses sorties. Le domaine d'utilisation de l'outil est également défini dans cette phase. Le comportement de l'outil doit également être décrit dans le cas où les données sont hors du domaine. À l'issue de cette phase, la spécification des exigences de l'outil élémentaire est rédigée.

4.3.1.2 Phase de conception

Le premier niveau de raffinement de la fonction attendue est décrit en langage naturel et en notation Uml sous forme de composition modulaire à l'aide de diagramme de classes. Des scénarios de tests d'intégration des outils élémentaires sont définis. Pour chaque outil élémentaire, les données échangées et les services fournis sont décrits en langage naturel, en diagrammes Uml (d'états, de séquences ou d'activités selon le besoin) et par des scénarios de tests fonctionnels. L'objectif de cette phase est également de choisir les solutions techniques utilisées dans l'implantation des exigences. Il s'agit principalement :

- d'analyser les exigences de l'outil élémentaire afin de concevoir son architecture. Cette architecture est décrite en tant que modèle Uml ;
- de diffuser les exigences de l'outil élémentaire et de les prendre en compte dans l'architecture de l'outil. Ainsi, cette architecture identifie les interactions avec les différents composants des autres outils ;
- de compléter, si nécessaire, les exigences. Ces compléments peuvent être issus d'un raffinement d'exigences ou produits pour décrire des comportements non dérivables des exigences ;
- de fournir le document de conception de l'outil élémentaire dans le cas d'exigences complètes ;
- d'analyser les exigences complémentaires si les étapes précédentes en ont fait apparaître. Cela consiste à mettre à jour les exigences : celles qui sont raffinées et celles qui sont dérivées sont ajoutées à la spécification de l'outil élémentaire. D'autre part, les exigences raffinées restent dans le document de conception et sont liées aux exigences de l'outil élémentaire.

4.3.1.3 Phase d'implantation

La phase d'implantation consiste à produire, dans un premier temps, le squelette du code source en langage JAVA pour chaque composant (sous forme de classe) du modèle de conception. Ensuite, chaque composant est implanté à l'aide du squelette JAVA. Enfin, le code exécutable de l'outil élémentaire est produit en un fichier JAR qui sera ensuite analysé et corrigé en cas d'éventuelles erreurs.

4.3.1.4 Livraison

Elle a pour but d'autoriser la livraison de l'outil élémentaire pour l'intégration dans l'outil complet. Cette phase consiste à identifier le résultat produit durant le cycle de vie et de résumer les résultats de vérification de l'outil élémentaire.

Remarque Il est à noter que dans le processus de développement d'un outil élémentaire avec les technologies classiques, il est nécessaire d'avoir une traçabilité directe entre un composant identifié dans la conception et le fichier JAR correspondant. La traçabilité des exigences utilisateurs jusqu'aux fichiers JAR est préconisée dans la phase de qualification.

4.3.2 Plan de développement exploitant les technologies formelles

Le processus de développement formel que nous décrivons dans cette section concerne les outils élémentaires réalisés en utilisant un langage de spécification et de vérification formelles à travers l'assistant de preuve COQ. Seuls les outils élémentaires développés formellement sont considérés dans cette section.

D'une part, les exigences et les interfaces décrites en langage naturel d'un outil élémentaire sont traduites en langage formel. D'autre part, l'étape de conception implante les exigences formellement spécifiées. Ainsi, il est possible de vérifier que l'implantation de l'algorithme est correcte par rapport aux exigences. L'implantation est automatiquement extraite de la conception de l'outil élémentaire.

4.3.2.1 Phase de spécification de l'outil élémentaire

Cette activité est alimentée par le service souhaité, ses paramètres et ses résultats qui sont décrits en langage naturel et sous la forme de scénarios de tests fonctionnels. La phase de spécification de l'outil élémentaire est similaire à celle du développement classique. Elle est définie en langage naturel selon la structure décrite dans la section 4.3.1.

4.3.2.2 Développement d'adaptateurs Java et OCaml

Il s'agit dans cette étape de développer les adaptateurs nécessaires qui permettent à un outil élémentaire développé avec un assistant de preuve d'interagir avec les autres outils élémentaires. Les adaptateurs sont illustrés dans la figure 4.2.

La spécification des adaptateurs permet de définir le sous-ensemble minimal des informations contenues dans les modèles et qui sont nécessaires pour que l'outil rende le service attendu. Ces informations sont ensuite décrites sous la forme de grammaires régulières droites et de structures de données en COQ.

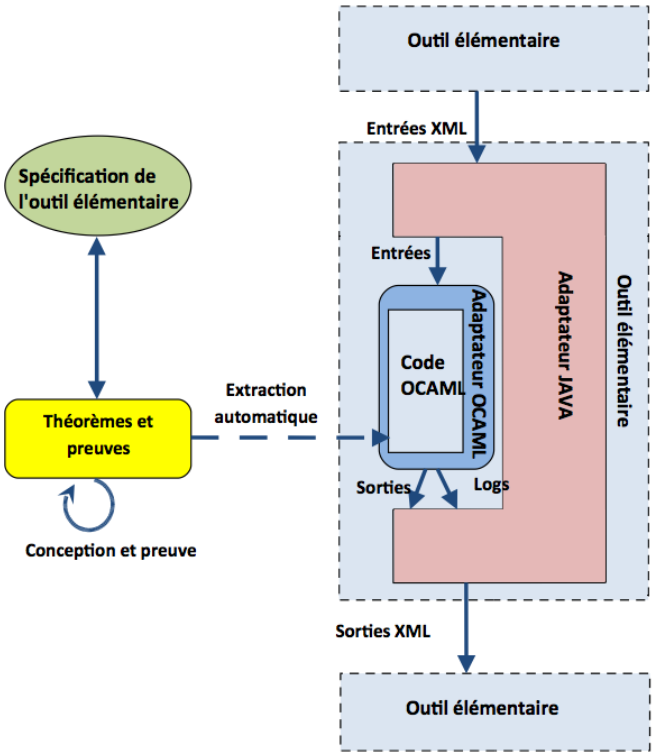


FIG. 4.2 – *Architecture d'un outil élémentaire formel*

Ces informations sont exploitées ou produites par la partie de l'outil implantée en COQ.

D'une part, l'adaptateur JAVA (lecteur/rédacteur) lit le fichier XML issu de la sortie de l'outil élémentaire précédent, et écrit le fichier XML pris en entrée de l'outil élémentaire suivant dans la chaîne d'outils élémentaires. Cet adaptateur exploite des bibliothèques qui ont été développées, vérifiées et qualifiées avec les approches classiques. Il n'était pas raisonnable, dans le cadre du projet, de développer un adaptateur similaire en COQ pour exploiter directement les fichiers XML. D'autre part, l'adaptateur JAVA communique avec l'adaptateur OCAML à l'aide de fichiers simples (sous format texte) dont le format est défini par les grammaires régulières.

L'adaptateur OCAML, également un lecteur/rédacteur, transmet au code extrait depuis COQ les données contenues dans les fichiers intermédiaires. Il permet de lire les données extraites des fichiers XML fournis par l'adaptateur JAVA, de lancer l'exécution de l'outil élémentaire, et d'écrire les données qui seront lues par l'adaptateur JAVA. Cet adaptateur produit un fichier de messages d'exécution conformément au standard de trace d'exécution de GENEAUTO.

4.3.2.3 Spécification formelle

La spécification décrite en langage naturel est traduite en langage formel. Les données en entrée (y compris les paramètres), les exigences ainsi que les propriétés associées à l'outil élémentaire sont traduites en langage Gallina (le langage de spécification en COQ). Il s'agit, ici, de spécifier en COQ la structure de données du modèle d'entrée, les fonctions ainsi que les propriétés de l'outil

élémentaire. Ces dernières sont décrites en langage naturel et sont extraites des documents de spécification et de conception de l'outil élémentaire et de l'outil complet. Chaque fonction de l'outil élémentaire doit être spécifiée sous forme de propriétés qui expriment l'impact des entrées sur les sorties de la fonction.

Exemple 4.3.1. S1/R2 *Respect de la causalité induite par les signaux de données : un bloc b ne peut être exécuté que si tous les blocs b.In qui alimentent les signaux de données connectés à ses entrées, ont déjà été exécutés dans le même cycle ;*
Cela se traduit en Coq *par :*

```
Lemma correctnessDataSignal :
 forall c B1 B2 E1 E2,
   isSequential B2 c = false
   -> isConnected B1 B2 c
   -> (B1 < (size c))
   -> (B2 < (size c))
   -> forwardInputDependencies c E1 E2 = E1
   -> forwardOutputDependencies c E1 E2 = E2
   -> DependencyLattice.PreOrder.le tt (E2 B1) (E2 B2).
Proof.
intros.
generalize (OutInBlock c B2 E1 E2).
generalize (OutInConnexion c B1 B2 E1 E2).
intros.
generalize (DependencyLattice.PreOrder.trans tt (E2 B1) (E1 B2) (E2 B2)).
intros.
intuition.
Qed.
```

La spécification formelle est elle-même caractérisée par des propriétés de complétude, de consistance ou de décidabilité. Par exemple, *les types sont comparables* est une propriété décidable (voir le fichier TypeLattice.v).

La traçabilité avec la spécification de l'outil élémentaire est réalisée par des commentaires intégrés dans la spécification formelle.

4.3.2.4 Description formelle de la conception

L'objectif est de produire une description formelle de la conception de l'outil élémentaire couvrant les différentes exigences. L'architecture de l'outil, les algorithmes ainsi que les propriétés associées sont décrites en langage Gallina. L'architecture d'un outil élémentaire est définie par un ensemble de fonctions décrivant le calcul devant être réalisés par l'outil selon la spécification formelle des exigences. Afin de vérifier certaines propriétés de l'outil élémentaire, il est souvent nécessaire de les décomposer en propriétés secondaires. Il est alors nécessaire d'exprimer les relations entre les propriétés raffinées et les exigences initiales. Les propriétés peuvent exprimer la complétude : une propriété est complète si elle prend en compte toutes les valeurs possibles de ses paramètres, autrement dit, la propriété exprime qu'une fonction est totale. Elles peuvent également décrire la terminaison d'un calcul.

En outre, la traçabilité entre la description formelle de la conception et la spécification formelle est clairement distinguée à l'aide de commentaires inclus dans le code Coq. Ils peuvent ensuite être exploités par les outils de couverture des exigences et de génération automatique des documents de certification.

Exemple 4.3.2. Tool/C1 *Le calcul effectué doit se terminer en un nombre fini d'itérations.*
Par exemple, cette exigence est traduite par :

```
Function tool_rec
  (data : E.EnvironmentLattice.PreOrder.type)
```

```
    (iterated : isIterate data)
    { wf (E.EnvironmentLattice.gt E.d) data }
    : E.EnvironmentLattice.PreOrder.type :=
    let nextdata := E.Forward E.diagram data in
       if (E.EnvironmentLattice.PreOrder.dec E.d
         nextdata
         data)
    then data
    else tool_rec nextdata (S_iter _ iterated).
Proof.
  ...
Qed.
```

Cette fonction construit, par la preuve, le point fixe de la fonction de calcul `Forward`*. Elle spécifie que chaque appel de* `Forward` *produit un nouveau résultat* `nextdata`*, jusqu'à ce que le point fixe soit atteint. La terminaison du calcul est assurée par la propriété* `wf (E.EnvironmentLattice.gt E.d) data` *qui indique à* Coq *le bon fondement de la relation* `E.EnvironmentLattice.gt`*. La preuve que le point fixe est atteint peut être consultée dans la section 5.4.2.*

4.3.2.5 Génération de code exécutable

Après avoir spécifié un outil élémentaire en Coq, son code source est automatiquement extrait en langage OCaml à l'aide de l'extracteur de Coq. Cette étape est nécessaire en vue d'intégrer l'outil développé formellement dans le reste de la chaîne du logiciel GeneAuto.

Il est important de préciser qu'avant toute extraction de code, la spécification et la conception formelles doivent être vérifiées. La partie vérification des chapitres 6 et 7 décrivent les principales preuves.

Afin de relier le code extrait à partir de Coq avec le reste de la chaîne, chaque outil élémentaire développé formellement est composé de deux adaptateurs présentés précédemment dans la section 4.3.2.2.

La phase de livraison est la même que celle décrite dans la section 4.3.1.4.

4.4 Processus de vérification

La phase de vérification concerne chaque outil élémentaire ainsi que le logiciel GeneAuto complet qui intègre l'ensemble des outils élémentaires. Ces derniers sont donc vérifiés indépendamment du reste avant d'être intégrés dans l'outil final. Les activités de vérification qui seront présentées dans ce qui suit sont en accord avec les objectifs de la norme de certification DO-178B. À l'issue de chaque phase de vérification, un document résumant ses résultats est fourni.

4.4.1 Processus de vérification d'un outil élémentaire développé avec des méthodes classiques

Ce processus est applicable pour chaque outil élémentaire développé avec les technologies classiques (Java dans GeneAuto) selon le processus de développement présenté dans la section 4.3.1.

4.4.1.1 Vérification de la spécification

Les activités de cette étape se résument par les point suivants. Il s'agit essentiellement de vérifier, par relecture, que :

- les données de spécifications sont complètes. Autrement dit, il s'agit de vérifier que les exigences de l'architecture de l'outil sont applicables à l'outil élémentaire ;
- les entrées et les sorties des outils élémentaires sont explicitées ainsi que le domaine d'application ;
- les exigences sont correctes et précises vis-à-vis des besoins identifiés. Elles doivent inclure la description fonctionnelle de l'outil élémentaire et les entrées/sorties associées ;
- les exigences sont vérifiables : leur description est basée sur les entrées/-sorties de l'outil élémentaire ;
- les algorithmes sont décrits correctement par rapport aux besoins attendus ;
- les informations de traçabilité sur l'architecture de GeneAuto sont incluses.
- les exigences comportent les détails d'utilisation (telle que l'installation, les messages d'erreurs, etc.) ;
- les exigences de robustesse sont fournies et couvrent le lancement de GeneAuto en dehors de son domaine d'utilisation. Les messages d'erreurs adéquats sont clairement indiqués.

La relecture indépendante des spécifications est appliquée pour accomplir cette vérification.

4.4.1.2 Vérification de la conception

Les activités de vérification appliquées à la conception concernent la vérification des données de conception, la conformité aux standards ainsi que l'analyse de traçabilité.

4.4.1.2.1 Vérification des données de conception

Cette vérification vise à assurer que :

- l'architecture de l'outil élémentaire définie en Uml couvre toutes les exigences définies dans la phase de spécification ;
- la conception des composants qui ne sont pas directement liés à la spécification est justifiée. Une explication sur leur présence est fournie puisque la conception ne découle pas seulement de la spécification mais d'autres contraintes ou choix arbitraires ;
- les composants extérieurs appelés par l'outil élémentaire sont identifiés ainsi que leurs comportements qui ne doivent pas aller à l'encontre des exigences fonctionnelles ;
- les descriptions textuelles relatives à la conception des composants sont correctes et consistantes (c'est-à-dire elles couvrent tous les comportements possibles des composants), et les algorithmes sont corrects et précis ;
- les exigences raffinées pour préciser des comportements non dérivables depuis les exigences sont correctes par rapport aux besoins définis.

La vérification des données de conception est accomplie par relecture.

4.4.1.2.2 Vérification de la conformité aux standards

Cette étape consiste à vérifier que les composants de la conception sont conformes aux standards de la conception. Les règles du standard sont défi-

nies de telle sorte que la description des composants de la conception de l'architecture soit correcte. La connexion des différents composants est également vérifiée afin d'assurer leur consistance.

Analyse de la traçabilité

La traçabilité consiste à assurer que toutes les exigences spécifiées sont couvertes par la phase de conception. De plus, les éléments de conception ne faisant pas référence directement à la spécification sont clairement identifiés et leur existence doit être justifiée.

La traçabilité entre les données de spécification et les données de conception est automatiquement vérifiée par un outil qui analyse les annotations données dans les commentaires des différents documents. L'outil utilisé dans le cadre de GeneAuto est TRAMWAY. Ce dernier identifie les exigences non couvertes par la conception d'un composant, ainsi que les éléments de conception ne faisant pas référence à la spécification.

4.4.1.3 Vérification de l'implantation

Les données devant être vérifiées par cette phase sont le code source (y compris le code généré par l'analyseur AntLr) et le fichier Jar généré à l'issue de l'outil élémentaire.

Trois points essentiels constituent les activités de cette phase :

Relecture de code source : Elle consiste à s'assurer, par relecture, que tous les éléments de conception sont implémentés en une seule fonction ou procédure et que toutes les données sont utilisées. De plus, il est également vérifié par relecture de code source que toutes les exigences identifiées dans la conception sont implémentées dans le code.

Vérification de conformité aux standards : Il s'agit de vérifier que le code source est conforme aux standards d'implantation. L'outil CheckStyle est utilisé pour cette activité.

Vérification de fichiers Jar : Le fichier Jar généré est vérifié par relecture afin de s'assurer qu'il est sans erreur.

La figure 4.3 résume les étapes de vérification de code ainsi que les techniques utilisées pour cette fin.

4.4.1.4 Vérification des outils élémentaires

L'objectif est de vérifier chaque outil élémentaire dans son format exécutable (fichier Jar). Cette phase se découpe en cinq étapes :

Tests fonctionnels des outils élémentaires : Cette phase a pour but de vérifier que l'exécution de l'outil est conforme aux exigences définies dans les données de spécification. Il faut définir, dans un premier temps, des cas de tests. Chaque cas de test fait référence à une exigence en déterminant les valeurs d'entrée et et les valeurs de sortie attendues. Ensuite, les cas de tests sont traduits en procédures de tests en utilisant l'outil JUnit de Java. Les procédures de tests sont exécutées pour comparer automatiquement les résultats avec ceux attendus ;

Tests unitaires : Cette étape complète l'étape précédente en prenant en compte les composants de conception qui raffinent les exigences ou les complètent. Les tests unitaires vérifient que l'exécution d'un élément de

Objectif	Activité	Procédure
Le code source correspond aux données de conception	Relecture de code source	Relecture
Le code source correspond à l'architecture	Relecture de source code	Relecture
Le code source est consistent	Relecture de code source	Relecture
Le code source est vérifiable	Vérification de correspondance aux standards	Check Style
Le code source correspond aux standards	Vérification de correspondance aux standards	Check Style
	Relecture de code source	Relecture
Chaque fonction/procédure du code source implante un seul composant de conception	Relecture de code source	Relecture
Les fichiers JAR sont correctement produits	Vérification de fichiers JAR	Vérification auto.

FIG. 4.3 – *Processus de vérification de codage*

conception est conforme aux exigences fonctionnelles définies dans la conception ;

Lecture de tests : Cette activité consiste à relire les résultats des tests unitaires. Il s'agit, dans un premier temps, d'effectuer un test unitaire sur chaque élément de la conception qui raffine ou contient des exigences dérivées. Une analyse complémentaire est conduite pour s'assurer que les cas de tests sont complets ;

Analyse de traçabilité : Elle a pour objectif de vérifier que toutes les exigences de la spécification et de la conception sont couvertes par les tests. La traçabilité est automatiquement réalisée par l'outil TRAMWAY. Ce dernier est appliqué entre les cas de tests et les exigences et entre les procédures de tests et les cas de tests ;

Analyse de couverture structurelle : La couverture structurelle se focalise sur la conception. Cette étape vérifie que les tests unitaires couvrent tout le code source de l'outil élémentaire. Elle préconise, également, d'identifier les parties du code source qui ne sont pas exécutées. Si tel est le cas, soit la couverture des exigences est incomplète et il faudra donc développer de nouveaux cas et procédures de tests, soit le code reste inatteignable par les tests et, par conséquent, il faut supprimer le code mort ou justifier les cas où le code est gardé.

La figure 4.4 récapitule les étapes de vérification concernées par les tests effectués sur l'outil élémentaire.

Objectif	Activité	Procédure
Le code exécutable est conforme à la spécification	Test d'outils élémentaires	Test
Le code exécutable est robuste par rapport à la spécification		
Le code exécutable est conforme avec les données de conception	Tests unitaires	
Le code exécutable est robuste par rapport à la conception		
Les résultats des tests sont corrects	Relecture de tests	Relecture
Les fonctions réalisées par les composants extérieurs sont correctes		
La spécification est couverte par les tests	Analyse de traçabilité	Outil de traçabilité
La conception est couverte par les tests		
L'architecture du logiciel est couverte par les tests	Analyse de couverture structurelle	Outil d'analyse de couverture structurelle

FIG. 4.4 – *Processus de test des outils élémentaires*

4.4.2 Processus de vérification d'un outil élémentaire formel

4.4.2.1 Vérification de la spécification

La spécification de l'outil élémentaire est écrite en langage naturel pour tout type de développement. Par conséquent, elle est la cible du processus de vérification décrit dans la section 4.4.1.1. Nous décrivons, dans la suite, le processus de vérification des spécifications formelles. Les activités de ce processus sont résumées dans la figure 4.5.

Objectif	Activité	Procédure
La spécification formelle est conforme à la spécification textuelle	Relecture de spécification formelle	Relecture
	Exécution de la spécification	Exécution de code extrait par Coq
La spécification formelle est correcte	Vérification de preuves	Vérifieur de preuves de Coq
La spécification formelle est vérifiable	Vérification de conformité aux standards	Vérification de syntaxe et type par Coq
La spécification formelle est conforme aux standards		
La spécification formelle est traçable avec la spécification textuelle	Analyse de traçabilité	Relecture

FIG. 4.5 – *Processus de vérification des spécifications formelles*

Relecture des spécifications formelles Les exigences textuelles sont transcrites en langage formel Gallina de l'atelier Coq. Il s'agit donc de vérifier, par relecture, que la spécification formelle est correcte par rapport à la spécification décrite en langage naturel ;

Exécution de la spécification L'extracteur de Coq est utilisé pour produire une spécification exécutable. Cette dernière est un programme qui doit se comporter exactement comme la spécification. Il est possible de vérifier la correspondance avec la spécification textuelle en exécutant des procédures de tests telles qu'elles sont décrites dans la section 4.4.1.4. L'exécution comprend la spécification des entrées/sorties (en format textuel) ainsi que l'adaptateur OCaml qui manipule ces fichiers en lecture/écriture ;

Vérification de preuves Le vérifieur de preuves Coq assure la correction de la spécification dont les preuves contenues dans cette dernière ;

Conformité aux standards Cette vérification est automatiquement effectuée par le vérifieur de Coq. Cela consiste à vérifier la correction de la syntaxe, le type des spécifications, la totalité et la terminaison des fonctions ;

Analyse de traçabilité Toutes les spécifications textuelles doivent être couvertes par la spécification formelle, c'est-à-dire, qu'elles ont été prises en compte et exprimées formellement. Les exigences ne faisant pas référence à des exigences textuelles doivent être justifiées. La traçabilité entre les spécifications textuelle et formelle est réalisée par l'outil TRAMWAY.

4.4.2.2 Vérification de la conception des adaptateurs Java et OCaml

Les adaptateurs Java et OCaml sont écrits en langage de programmation classique (respectivement Java et OCaml). Le processus de vérification de leur conception suit, par conséquent, le même principe que celui présenté dans la section 4.4.1.2. Les tests s'appuieront sur les fichiers Xml échangés entre les différents outils élémentaires du générateur de code.

4.4.2.3 Vérification de la description formelle de la conception

Les enjeux de vérification de la description formelle de la conception sont les mêmes que pour la conception classique. En revanche, les activités de vérification sont réalisées par des procédures différentes. La figure 4.6 récapitule ces activités.

Cela consiste, dans un premier temps, à relire la description formelle de la conception afin de vérifier la conformité avec l'architecture de l'outil élémentaire. La conception formelle est écrite sous la forme de propriétés et de fonctions (plus de détails sont fournis dans la section 5.4).

Les algorithmes développés dans la conception sont prouvés corrects en Coq. La vérification des scripts de preuves assure donc que les algorithmes développés sont corrects, que les propriétés de la spécification formelle sont garanties et que les propriétés de l'architecture sont vérifiées.

De plus, le standard d'implantation est assuré par Coq qui vérifie la syntaxe et le type des descriptions de la conception. Quant à l'analyse de traçabilité, elle est directe du moment que la spécification et la conception formelles sont écrites dans le même langage Gallina. Cependant, certaines propriétés requises dans la conception ne font référence à aucune exigence de la spécification formelle. L'impact de ces propriétés sur les sorties de l'outil est donc analysé afin de maintenir la conformité avec les exigences des utilisateurs.

Objectif	Activité	Procédure
La description formelle de la conception est conforme à la spécification formelle	Vérification par preuves	Coq
La description formelle de la conception est correcte		
Les algorithmes sont corrects		
L'architecture et la spécification formelle sont compatibles	Relecture de la description de la conception	Relecture
L'architecture est consistante		
La description formelle de la conception est vérifiable	Conformité aux standards	Analyseur Coq (syntaxe, type)
La description formelle de la conception et les standards sont conformes		
La description formelle de la conception est traçable vis-à-vis de la spécification formelle	Traçabilité directe	

FIG. 4.6 – *Processus de vérification de la conception formelle*

4.4.2.4 Vérification de l'implantation

L'objectif de cette phase est de vérifier le code source OCaml généré automatiquement depuis la spécification en Coq. Il s'agit d'accomplir les activités suivantes :

Relecture de code source : Cette activité consiste à relire le code source de l'outil élémentaire développé en Coq ainsi que son code extrait. Il s'agit, ici, de vérifier, par relecture, que le code implémente les exigences qui sont sous la forme de description formelle de la conception. De plus, il est pris en compte que le développement en Coq assure une similitude structurelle avec OCaml ;

Vérification de conformité aux standards : Il s'agit de vérifier automatiquement la conformité du code OCaml aux standards. Ces activités sont récapitulées dans la figure 4.7.

Objectif	Activité	Procédure
Le code source correspond aux exigences	Relecture de code source	Relecture
Le code source est consistent		
Le code source est vérifiable	Vérification de conformité aux standards	Vérifieur de code
Le code source est conforme aux standards	Vérification de conformité aux standards	Vérifieur de code
	Relecture de code source	Relecture

FIG. 4.7 – *Processus de vérification de l'implantation*

4.4.2.5 Vérification du code exécutable

Elle consiste à vérifier les résultats de l'exécutable Jar généré à la sortie de l'outil élémentaire ainsi que le code exécutable obtenu depuis le code OCaml

extrait (code binaire obtenu depuis OCaml). Cette phase permet de s'assurer que le code exécutable est conforme à la spécification formelle et à la description formelle de la conception de l'outil élémentaire. Les activités de cette vérification (tests de l'outil élémentaire, relecture des tests ainsi qu'analyse de traçabilité) sont similaires au processus décrit dans la section 4.4.1.4.

4.4.3 Vérification du logiciel GeneAuto complet

Les activités de cette phase considèrent l'outil complet. Elles vérifient que le processus respecte les standards.

Deux activités principales sont effectuées pour vérifier la spécification de l'architecture :

- d'une part, relire et analyser la traçabilité de la spécification de l'architecture avec les exigences de l'utilisateur ;
- d'autre part, réaliser des tests d'intégration sur l'outil pour assurer que l'intégration des outils élémentaires et l'exécution de GeneAuto sont conformes aux exigences globales définies dans l'architecture de GeneAuto. Cela consiste à développer des cas de tests vis-à-vis des exigences de l'architecture. Les cas de tests d'intégration doivent assurer une couverture des exigences.

4.4.4 Vérification de l'utilisateur

Le processus de cette vérification inclut les activités conduites par l'utilisateur, à savoir les exigences et les tests effectués par ce dernier. La vérification des exigences de l'utilisateur est réalisée à travers la relecture des spécifications de l'utilisateur. Quant aux tests de validation, chaque utilisateur identifie les données d'entrées représentatives, exécute l'outil et vérifie le code généré grâce à un jeu de tests. Il valide ainsi les exigences des utilisateurs.

4.5 Qualification de GeneAuto

La qualification de GeneAuto tient compte de la norme DO-178B. Pour cela, des documents décrivant les différents plans de développement, de vérification et de qualification doivent être définis dans le but de respecter la conformité avec la norme. En outre, certains outils doivent être qualifiés pour le développement de systèmes certifiés. GeneAuto exploite à la fois des outils de développement ou de vérification. Ces outils peuvent, soit être qualifiés, soit leur résultat sera vérifié.

Les outils associés à Java (comme Jar, compilateur Java, etc.) ne sont pas qualifiés. La qualification des outils élémentaires développés avec les techniques classiques demande des documentations détaillées sur le processus de développement. De plus, la vérification se fait modulo les tests et la relecture de code.

En revanche, l'extracteur de Coq ne demande pas une qualification en termes de DO-178B. Les échanges avec les autorités de certification, pour une pré-qualification, ont tenu compte des travaux sur la vérification du noyau de Coq [Bar99], ainsi que les travaux en cours sur la certification du mécanisme d'extraction de Coq [Let04, Glo09]. Ce qui aurait pour effet la suppression de la phase de vérification par relecture du code extrait, et apporte ainsi un niveau

de confiance suffisant pour les autorités de certification quant à l'utilisation de l'assistant de preuve Coq.

4.6 Synthèse

Nous avons présenté, dans ce chapitre, les principales contraintes de qualification pour le processus de développement de GeneAuto qui combine approche classique et formelle. Notre principale contribution dans le projet GeneAuto est l'intégration d'approches formelles dans le processus de développement industriel qualifié. Une partie importante du travail consiste à prendre en compte les contraintes industrielles (exigences liées aux modèles Simulink et aux autorités de certification) et leur mise en œuvre en Coq. Nous avons présenté dans ce chapitre les différentes activités de développement et de vérification, tant pour le langage Java que pour l'assistant de preuve Coq. Outre les différences entre les deux processus, la phase de vérification dans le développement classique repose principalement sur les tests et la relecture tandis que celle du développement formel s'appuie sur les preuves formelles. En revanche, dans tous les cas, des tests d'intégration ou fonctionnels sont requis dans le processus de développement. Nous présenterons dans le chapitre suivant le cadre formel pour le développement des outils élémentaires développés en Coq.

Cadre formel de développement

5

5.1 Motivation

Un générateur de code est un système informatique complexe dont la spécification et la vérification formelle sont nécessaires pour éliminer les risques induits par son utilisation. Comme indiqué dans le chapitre 3, GeneAuto est constitué de plusieurs outils élémentaires indépendants. En revanche, l'utilisation de méthodes formelles pour chacun de ces outils indépendamment les uns des autres risque d'être très coûteuse.

Pour réduire les coûts de développement, il est utile de partager certaines activités de spécification et de preuve qui se font en Coq. Nous proposons de nous appuyer sur un cadre formel générique pour différentes parties du générateur de code. Ce cadre général évitera de construire des preuves similaires en factorisant les parties communes du développement. Cela évitera également de construire chaque outil élémentaire de manière indépendante et de faire ainsi de nouvelles preuves pour chacun.

Ce chapitre décrit l'approche formelle générique que nous proposons pour le développement de quelques outils élémentaires du générateur de code. Il s'agit d'un cadre basé sur l'analyse statique par interprétation abstraite. Les éléments du cadre présenté dans ce chapitre relèvent clairement de la phase de conception des outils élémentaires.

5.2 Ensembles partiellement ordonnés

L'analyse statique par interprétation abstraite s'appuie sur l'utilisation de fonctions croissantes sur des ensembles ordonnés.

Avant de présenter les éléments du cadre formel, nous donnons quelques définitions utiles, qui sont largement inspirées de [Win93, Shr02].

Définition 5.2.1. ***Ensemble partiellement ordonné (poset : partially ordered set)*** *$\langle \mathcal{X}, \preceq \rangle$ est un ensemble partiellement ordonné si et seulement si $\mathcal{X}$ est un ensemble muni d'une relation d'ordre partiel $\preceq$ (réflexive, antisymétrique et transitive).*

$$\begin{array}{ll} \textit{réflexivité} : & \forall a \in \mathcal{X}, a \preceq a \\ \textit{antisymétrie} : & \forall a, b \in \mathcal{X}, a \preceq b \bigwedge b \preceq a \Rightarrow a = b \\ \textit{transitivité} : & \forall a, b, c \in \mathcal{X}, a \preceq b \bigwedge b \preceq c \Rightarrow a \preceq c \end{array}$$

Exemple 5.2.1. *L'ensemble des parties d'un ensemble $\mathcal{X}$, noté $\mathcal{P}(\mathcal{X})$, muni de la relation d'inclusion $\subseteq$, noté $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$, est un ensemble partiellement ordonné.*

Définition 5.2.2. ***Majorant (upper bound)***
Soit $\mathcal{A} \subseteq \mathcal{X}$, $b \in \mathcal{X}$ est un majorant de $\mathcal{A}$ si :

$$\forall a \in \mathcal{A}, a \preceq b$$

Définition 5.2.3. ***Borne supérieure (least upper bound)***
b est la borne supérieure de $\mathcal{A}$, si et seulement si b est un majorant de $\mathcal{A}$ et pour tous les majorants c de $\mathcal{A}$, $b \preceq c$.

Définition 5.2.4. ***Minorant (lower bound)***
Soit $\mathcal{A} \subseteq \mathcal{X}$, $b \in \mathcal{X}$ est un minorant de $\mathcal{A}$ si :

$$\forall a \in \mathcal{A}, b \preceq a$$

Définition 5.2.5. ***Borne inférieure (greatest lower bound)***
b est la borne inférieure de $\mathcal{A}$, si b est un minorant de $\mathcal{A}$ et si pour tous les minorants c de $\mathcal{A}$, $c \preceq b$

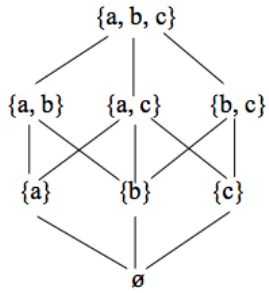
Exemple 5.2.2. *L'ensemble partiellement ordonné $\langle \mathbb{N}, \leq \rangle$ possède une borne inférieure 0 mais n'a pas de borne supérieure.*

Exemple 5.2.3. *L'ensemble partiellement ordonné $\langle \mathcal{P}(\mathcal{X}), \subseteq \rangle$ possède une borne inférieure $\emptyset$ et une borne supérieure $\mathcal{X}$.*

Définition 5.2.6. ***Treillis***
Un ensemble partiellement ordonné est un treillis noté par $\langle \mathcal{X}, \preceq, \sqcup, \sqcap \rangle$ si et seulement si : $\forall\, a,\, b \in \mathcal{X}$, il existe une borne supérieure et une borne inférieure de $\{a, b\}$, notées respectivement $(a \sqcup b)$ et $(a \sqcap b)$, avec :

1. $\forall a,\, b \in \mathcal{X}, a \preceq (a \sqcup b) \quad \wedge \quad b \preceq (a \sqcup b)$
2. $\forall a,\, b \in \mathcal{X}, a\ (a \sqcap b) \preceq a \quad \wedge \quad (a \sqcap b) \preceq b$

Exemple 5.2.4. *Soit $\mathcal{X} = \{a, b, c\}$. La représentation graphique, appelée diagramme de Hasse, du treillis $\langle \mathcal{P}(\mathcal{X}), \subseteq, \cup, \cap \rangle$, correspond au schéma suivant :*



La borne inférieure du treillis est $\emptyset$, et la borne supérieure est $\mathcal{X}$.

Définition 5.2.7. ***Treillis complet***
Un ensemble partiellement ordonné $\langle \mathcal{X}, \preceq \rangle$ est un treillis complet,
noté $\langle \mathcal{X}, \preceq, \bot, \top, \sqcup, \sqcap \rangle$ si et seulement si : $\forall\, \mathcal{A} \subseteq \mathcal{X}$, $\mathcal{A}$ a une borne supérieure et une borne inférieure dans $\mathcal{X}$.

Exemple 5.2.5. *Reprenons l'exemple 5.2.4. $\langle \mathcal{P}(\mathcal{X}), \subseteq, \emptyset, \mathcal{X}, \cup, \cap \rangle$ est un treillis complet.*

Définition 5.2.8. ***Ordre total***
Soit un ensemble $\mathcal{X}$ muni d'une relation d'ordre $\preceq$. La relation d'ordre $\preceq$ est totale si et seulement : $\forall\, a, b \in \mathcal{X},\ a \preceq b \quad \bigvee \quad b \preceq a$.

Exemple 5.2.6. *La relation $\leq$ définie sur l'ensemble des entiers naturels $\mathbb{N}$ est un ordre total.*

Dans ce qui suit, nous rappelons quelques définitions utilisées pour manipuler des fonctions entre des ensembles partiellement ordonnés.

Définition 5.2.9. ***Chaîne***
Une chaîne d'un ensemble partiellement ordonné $\langle \mathcal{X}, \preceq \rangle$ est un sous-ensemble de $\mathcal{X}$ totalement ordonné.

Définition 5.2.10. ***Fonction continue***
Une fonction f entre deux ensembles partiellement ordonnés $\langle \mathcal{A}, \preceq_{\mathcal{A}} \rangle$ et $\langle \mathcal{B}, \preceq_{\mathcal{B}} \rangle$ est dite continue si et seulement si f préserve les bornes supérieures des chaînes croissantes de $\mathcal{A}$, si elles existent.

Définition 5.2.11. ***Fonction croissante (monotonous function)***
Soit $f : \mathcal{A} \rightarrow \mathcal{B}$ une fonction entre deux ensembles partiellement ordonnés $\langle \mathcal{A}, \preceq_{\mathcal{A}} \rangle$ et $\langle \mathcal{B}, \preceq_{\mathcal{B}} \rangle$. f est croissante (préserve l'ordre) si et seulement si :

$$\forall x, y \in \mathcal{A}, x \preceq_{\mathcal{A}} y \Rightarrow f(x) \preceq_{\mathcal{B}} f(y)$$

Définition 5.2.12. ***Point fixe***
Soit $f : \mathcal{X} \rightarrow \mathcal{X}$ une fonction continue sur l'ensemble partiellement ordonné $\langle \mathcal{X}, \preceq \rangle$. L'élément $p \in \mathcal{X}$ est un point fixe de f si et seulement si $f(p) = p$. L'ensemble des points fixes de f est :

$$fix(f) = \{p \in \mathcal{X} \mid f(p) = p\}$$

Théorème 5.2.1. ***Théorème de Kleene*** *[Col52]*
Soit $\langle \mathcal{X}, \preceq, \bot, \top, \sqcup, \sqcap \rangle$ un treillis complet. Pour toute fonction monotone $f : \mathcal{X} \rightarrow \mathcal{X}$, le plus petit point fixe de f, noté $lfp(f)$, existe et vaut $\bigsqcup_{i \geqq 0} f^i(\bot)$. Avec :
$f^0(x) = x$
$f^{i+1}(x) = f(f^i(x))$

Définition 5.2.13. ***Accessibilité*** *[Acz77, Nor88]*
Étant donnée une relation $\prec$ définie sur un ensemble $\mathcal{X}$, l'ensemble des éléments accessibles pour $\prec$, noté $Acc_{\prec}$, est défini inductivement par :

$$\frac{\forall y \in \mathcal{X},\ y \prec x \Rightarrow y \in Acc_{\prec}}{x \in Acc_{\prec}}$$

De manière informelle, un élément est dit accessible pour la relation $\prec$ s'il n'est pas le point de départ d'une chaîne infinie décroissante via $\prec$. Par exemple, un élément sans prédécesseur par la relation $\prec$ est un élément accessible.

Définition 5.2.14. ***Relation bien fondée***
Une relation $\prec$ sur un ensemble $\mathcal{X}$ est bien fondée si tous les éléments de $\mathcal{X}$ sont accessibles pour la relation $\prec$.

5.3 Cadre formel & représentation

Dans cette partie, nous décrivons les éléments du cadre générique utilisé pour le développement de plusieurs modules (outils élémentaires) du générateur de code GeneAuto. Les circuits sont des graphes qui peuvent contenir des cycles. Les analyses que nous devons réaliser reposent sur la fermeture transitive du graphe. Il est donc nécessaire de pouvoir calculer des points fixes.

Nous définissons un cadre formel générique instanciable à plusieurs outils élémentaires constituant le générateur de code GeneAuto. Ce cadre exploite un calcul de point fixe dans des ensembles partiellement ordonnés, soit le même principe de calcul que l'analyse statique par interprétation abstraite. Il s'agit de calculer, pour chaque élément d'un circuit, l'information requise par l'analyse concernée et ceci en fonction des informations associées aux autres éléments du circuit. Le cadre formel est construit à partir des éléments que nous allons présenter progressivement dans cette section.

Le but n'est pas de définir un cadre formel complet pour l'interprétation abstraite en utilisant l'assistant de preuve Coq tel que cela a été réalisé et approfondi par D. Pichardie dans [Pic05] et auparavant dans [Mon98]. Nous avons défini un cadre qui contient le minimum d'éléments nécessaires pour modulariser les analyses que nous effectuons dans les différents outils élémentaires du générateur de code.

Le cadre formel des calculs que nous proposons est caractérisé par l'ensemble des éléments suivants :

- Un domaine abstrait Δ qui représente l'ensemble des valeurs symboliques manipulées ;
- Une relation d'ordre partiel $\preceq$ définie dans le domaine Δ permettant de construire des chaînes finies ;
- Une famille de treillis paramétrée par un type $\mathfrak{D}$,
  notée $\{\theta_d\}_{d\in\mathfrak{D}} = \langle \Delta, \preceq, \bot, \top, \sqcup, \sqcap \rangle$. $\bot$ est le plus petit élément de la famille de treillis, $\top$ le plus grand élément de la famille de treillis. L'opérateur d'union $\sqcup$ calcule la borne supérieure et l'opérateur d'intersection $\sqcap$ calcule la borne inférieure. La structure de famille de treillis permettra l'utilisation du théorème de Kleene pour le calcul de point fixe dans le cadre des analyses effectuées dans cette thèse ;
- Un environnement Y qui consiste à associer à chaque composante du circuit donné (bloc, port, etc.) une valeur symbolique dans Δ. Un environnement est une famille de treillis qui ne considère à la fois que les mêmes composantes : soit les blocs, soit les ports, etc. Nous supposons que les composantes concernées sont distinguées par leurs identifiants (ou indices) entiers compris dans $[0 \dots s[$. Ainsi, Y est également une famille de treillis ;
- Une fonction de propagation monotone ρ qui propage l'information calculée à travers le circuit, lors des itérations pour calculer le point fixe. En fait, les éléments de l'environnement Y qui forment une famille de treillis sont calculés avec la fonction ρ.
- Le résultat de l'analyse souhaitée est obtenu en calculant successivement les itérés de la fonction $\rho : Y \rightarrow Y$. Ce résultat est le plus petit point fixe de ρ, noté $lfp(\rho)$, obtenu par application du théorème de Kleene. Ainsi, il est atteint en un nombre fini d'itérations car les circuits traités sont de tailles finies. De plus, pour que le point fixe soit atteint, il est nécessaire de disposer d'une relation stricte $\succ$ (relation inverse de $\preceq$) bien fondée.

Dans ce qui suit, nous détaillons chaque élément du cadre générique. Ce dernier est mis en œuvre en utilisant le langage de spécification de l'assistant de preuve Coq. L'implantation des éléments du cadre sera également présentée dans cette partie.

5.3.1 Domaine abstrait

Un domaine abstrait Δ représente l'ensemble des valeurs symboliques de l'information calculée. Les informations sont par exemple les rangs d'exécution des blocs, les dépendances des blocs, les types des ports des blocs, etc. Par conséquent, Δ dépend de la nature des informations calculées pour le circuit. Ce dernier est une structure finie. La terminaison des calculs effectués est garantie par la croissance de la fonction de propagation ρ dans une famille de treillis de profondeur finie.

Exemple 5.3.1. *Si l'information calculée est un entier, alors le domaine abstrait* $\Delta \subset \mathbb{N}$.

5.3.2 Relation d'ordre abstraite

Nous définissons, dans cette partie, les ordres que nous manipulons ainsi que leurs signatures Coq.

5.3.2.1 Relation d'ordre partiel

Cette relation guide l'analyse spécifique à l'outil en question et assure sa terminaison. Nous définissons dans la figure 5.1 la signature d'un préordre partiel. L'ensemble `type` représente le domaine Δ. La relation $\preceq$ (notée `le` dans le script Coq) est une relation de préordre partiel (réflexive et transitive, propriétés notées respectivement dans la signature Coq par `refl` et `trans`) complétée par la relation d'équivalence (`eq` dans la signature) dérivée de l'antisymétrie. C'est pourquoi nous parlons d'une relation ordre partiel au lieu de préordre partiel. Il aurait été possible de définir directement la propriété d'antisymétrie. Mais, la relation d'égalité qu'elle induit n'est pas exploitée dans certaines analyses. Par exemple, deux blocs peuvent avoir la même information calculée sans qu'il s'agisse du même bloc.

```
Module Type PreOrderType.

 Declare Module Data : DataType.
 Parameter type : Type.

 Parameter le : Data.type -> type -> type -> Prop.
 Definition eq : Data.type -> type -> type -> Prop :=
  fun d x y => le d x y /\ le d y x.

 Parameter refl : forall d x, le d x x.
 Parameter trans : forall d x y z, (le d x y) -> (le d y z) -> (le d x z).
 Parameter dec : forall d x y, {le d x y} + {~le d x y}.
 Parameter eq_dec : forall d x y, {eq d x y} + {~eq d x y}.

End PreOrderType.
```

FIG. 5.1 – *Signature d'un préordre partiel en* Coq

La relation d'équivalence `eq` est définie par complétion. Elle est donc plus forte que la définition usuelle d'égalité $x = y$, ce qui signifie qu'il existe plus

de paires (x, y) telles que *eq d x y* que de paires telles que $x = y$. De plus, le symbole $=$ est, au sens Coq, une égalité structurelle (égalité de Leibniz). Pour des raisons d'efficacité algorithmique, les librairies Coq n'assurent en général pas cette égalité structurelle. Par exemple, soient x et y deux entiers, il n'est pas possible de spécifier que $\frac{2\times x}{2\times y} = \frac{x}{y}$, car la formalisation du type quotient est problématique en théorie des types. Il est donc plus simple de raisonner avec la relation d'équivalence `eq` qu'avec l'égalité de Coq.

La définition de la relation `eq` est fixée dans la signature de l'ordre partiel `PreOrderType`.

Les propriétés `dec` et `eq_dec` spécifient respectivement la décidabilité de `le` et de `eq`. Cela signifie pour la propriété `dec`, par exemple, qu'il est toujours possible de démontrer soit `le d x y` soit sa négation, pour tout x et tout y dans `type`.

Par ailleurs, les calculs effectués par les outils élémentaires sont manipulés par des types dépendants que nous appelons familles de treillis. Cela nous a conduit à introduire le module `Data` utilisé pour définir le paramètre d'un type dépendant ou le paramètre d'une famille de treillis. Notons le type du paramètre, qui sera présenté dans la section 5.3.4, par $\mathfrak{D}$. Pour l'instant, nous nous limitons à sa signature.

La signature `DataType`, présentée dans la figure 5.2, comporte un ensemble `type`, et la décidabilité de la relation d'équivalence sur les éléments de cet ensemble. Seule l'équivalence est utile pour manipuler les éléments du type dépendant.

```
Module Type DataType.

  Parameter type : Type.
  Parameter eq : type -> type -> Prop.

  Parameter refl : forall x, eq x x.
  Parameter sym : forall x y, (eq x y) -> (eq y x).
  Parameter trans : forall x y z, (eq x y) -> (eq y z) -> (eq x z).
  Parameter dec : forall x y, {eq x y} + {~ eq x y}.

End DataType.
```

FIG. 5.2 – *Signature du module* `DataType`

5.3.2.2 Relation d'ordre partiel inverse

Nous définissons, dans cette section, la relation d'ordre qui construit l'ordre inverse, notée $\preceq^{-1}$, d'une relation d'ordre donnée, notée $\preceq$. Par exemple, si $x \preceq y$ alors $y \preceq^{-1} x$. Un fragment de la définition de la relation d'ordre partiel inverse $\preceq^{-1}$ est donné dans la figure 5.3.

```
Module ReversePreOrder (preOrder : PreOrderType)
  <: PreOrderType
    with Module Data := preOrder.Data
    with Definition type := preOrder.type.

  Module Data := preOrder.Data.

  Definition type := preOrder.type.

  Definition le d x y := preOrder.le d y x.
  ...
```

FIG. 5.3 – *Définition de l'ordre inverse d'un ordre partiel*

5.3.2.3 Relation d'ordre total

Certains outils élémentaires requièrent que les résultats obtenus par l'analyse effectuée soient triés pour les transmettre au reste des outils élémentaires de GENEAUTO. Il est alors nécessaire de définir un ordre total.

La signature d'un préordre total est décrite par la figure 5.4. Il s'agit de la même signature que l'ordre partiel avec en supplément la preuve qu'il est total.

```
Module Type TotalPreOrderType.

  Declare Module Data : DataType.
  Parameter type : Type.
  Parameter le : Data.type -> type -> type -> Prop.
  Definition eq : Data.type -> type -> type -> Prop :=
        fun d x y => le d x y /\ le d y x.

  Parameter refl : forall d x, le d x x.
  Parameter trans : forall d x y z, le d x y -> le d y z -> le d x z.
  Parameter dec : forall d x y, { le d x y } + { ~ (le d x y) }.
  Parameter total : forall d x y, { le d x y } + { le d y x }.
  Parameter eq_dec : forall d x y, { eq d x y } + { ~ (eq d x y) }.

End TotalPreOrderType.
```

FIG. 5.4 – *Signature d'un préordre total en* COQ

5.3.2.4 Produit cartésien d'ordres partiels

Nous notons à partir de maintenant $\times$ pour représenter un produit cartésien.

Définition 5.3.1. *Produit cartésien d'ordres partiels*
Soient $\theta \triangleq \langle \mathcal{X}, \preceq_\theta \rangle$ *et* $\theta' \triangleq \langle \mathcal{Y}, \preceq_{\theta'} \rangle$ *deux ordres partiels. La relation d'ordre partiel* $\preceq_{\theta\times\theta'}$ *est définie sur le produit cartésien* $\mathcal{X} \times \mathcal{Y}$ *par :*

$$\forall (a,b),\ (c,d) \in (\mathcal{X} \times \mathcal{Y}),\ (a,b) \preceq_{\theta\times\theta'} (c,d) \triangleq a \preceq_\theta c \wedge b \preceq_{\theta'} d$$

Nous utilisons le concept de foncteur pour définir le produit cartésien de relations d'ordre partiel. Le foncteur `PreOrderCartesianProduct` de la figure 5.5 implante le produit cartésien de deux relations d'ordre partiels respectivement définies par les modules `O1` et `O2`.

La notation `<:` signifie que le module `PreOrderCartesianProduct` est de type `PreOrderType` et que toutes les spécifications sont visibles à l'extérieur du module.
En outre, `O2 : PreOrderType with Module Data := O1.Data` est un

```
Module PreOrderCartesianProduct
  (O1 : PreOrderType)
  (O2 : PreOrderType with Module Data := O1.Data)
  <: PreOrderType with Module Data := O1.Data
    with Definition type := (O1.type * O2.type)%type.

  Module Data := O1.Data.
  Definition type := (O1.type * O2.type)%type.
  Definition le : Data.type -> type -> type -> Prop :=
    fun d x y =>
      match x, y with
        | (x1, x2), (y1, y2) => (O1.le d x1 y1) /\ (O2.le d x2 y2)
      end.
  Definition eq : Data.type -> type -> type -> Prop :=
   fun d x y => le d x y /\ le d y x.

  Definition eq_opt : Data.type -> type -> type -> Prop  :=
    fun d x y =>
      match x , y with
        | (x1, x2) , (y1, y2) => (O1.eq d x1 y1) /\ (O2.eq d x2 y2)
      end.

  Lemma eq_opt_is_eq : forall d x y, eq_opt d x y -> eq d x y.
    ...

  Lemma eq_is_eq_opt : forall d x y, eq d x y -> eq_opt d x y.
    ...

  Lemma eq_opt_dec : forall d x y, { (eq_opt d x y) } + { ~ (eq_opt d x y) }.
  ...

End PreOrderCartesianProduct.
```

FIG. 5.5 – *Produit cartésien d'ordres partiels*

foncteur déclarant `O2` de type `PreOrderType` ayant le même paramètre `Data` que celui du module `O1`.

La notation `% type` est utilisée pour interpréter `(O1.type*O2.type)` dans la sorte `Type`. Cette notation est utilisée en Coq pour traiter `(O1.type*O2.type)` comme un type et le symbole `*` comme un produit de types et non la multiplication.

La relation d'ordre $\preceq_{\theta\times\theta'}$ est spécifiée par `le` à partir des relations $\preceq_\theta$ et $\preceq_{\theta'}$ représentées respectivement par `O1.le` et `O2.le`.

Le module `Data`, qui permet de représenter les paramètres dans un type dépendant, est le même pour tous les ordres partiels manipulés. La définition `eq_opt` est une autre version optimisée de l'équivalence dans `type`. Les lemmes `eq_opt_is_eq` et `eq_is_eq_opt` démontrent conjointement l'équivalence des deux définitions `eq` et `eq_opt`.

Les preuves de réflexivité, de transitivité et de décidabilité réutilisent les preuves correspondantes des deux ordres `O1` et `O2`. Elles sont consultables dans le fichier Domain.v

5.3.2.5 Produit lexicographique d'ordres partiels

Nous avons également besoin dans certains calculs, notamment dans l'outil élémentaire d'ordonnancement, de manipuler un ordre lexicographique. Un cas d'utilisation est présenté dans la section 6.3.1.3. Notons $\times_{lex}$ pour représenter un produit lexicographique. Nous présentons dans un premier temps la définition usuelle de l'ordre lexicographique.

Définition 5.3.2. *Ordre lexicographique (tel qu'il est défini en littérature)*
Soient $\langle \mathcal{X}, \preceq_\theta \rangle$ et $\langle \mathcal{Y}, \preceq_{\theta'} \rangle$ deux ordres partiels. La relation d'ordre partiel $\preceq_{\theta\times_{lex}\theta'}$

sur le produit lexicographique de $\mathcal{X}$ et $\mathcal{Y}$ est définie par :

$$\forall (a,b),\ (c,d) \in (\mathcal{X} \times_{lex} \mathcal{Y}),\ (a,b) \preceq_{\theta \times_{lex} \theta'} (c,d) \triangleq a \prec_\theta c \vee (a =_\theta c \wedge b \preceq_{\theta'} d)$$

avec $=_\theta$ la relation d'égalité structurelle sur $\mathcal{X}$.

Cette définition ne peut pas être exploitée en l'état car, d'une part, la relation d'égalité structurelle ($=_\theta$) est difficilement manipulable et, d'autre part, nous ne disposons pas d'un ordre strict $\prec$ associé à la relation $\preceq$. Ainsi, nous définissons notre relation d'ordre lexicographique comme suit.

Définition 5.3.3. *Ordre lexicographique*
Soient $\langle \mathcal{X}, \preceq_\theta \rangle$ et $\langle \mathcal{Y}, \preceq_{\theta'} \rangle$ deux ordres partiels, la relation d'ordre partiel $\preceq_{\theta \times_{lex} \theta'}$ sur le produit lexicographique de $\mathcal{X}$ et $\mathcal{Y}$ est définie par :

$$\forall (a,b),\ (c,d) \in (\mathcal{X} \times_{lex} \mathcal{Y}),\ (a,b) \preceq_{\theta \times_{lex} \theta'} (c,d) \triangleq a \preceq_\theta c \wedge (c \preceq_\theta a \Rightarrow b \preceq_{\theta'} d)$$

Les deux définitions sont équivalentes.

Preuve. Nous disposons de $a \preceq_\theta c \wedge (c \preceq_\theta a \Rightarrow b \preceq_{\theta'} d)$. Cela est équivalent à $((a \preceq_\theta c) \wedge \neg(c \preceq_\theta a)) \vee ((a \preceq_\theta c) \wedge (b \preceq_{\theta'} d))$, en dépliant l'implication et en distribuant la conjonction.
Ce qui se simplifié en $((a \prec_\theta c) \vee ((a \preceq_\theta c) \wedge (b \preceq_{\theta'} d))$, quand la relation stricte $\prec_\theta$ est définie. Quand la relation $=_\theta$ est facilement manipulable, $a \preceq_\theta c$ est équivalente à $(a =_\theta c \vee a \prec_\theta c)$. Ainsi, nous obtenons $a \prec_\theta c \vee (a =_\theta c \wedge b \preceq_{\theta'} d)$.

La définition du produit lexicographique d'ordres partiels est décrite dans le foncteur `PreOrderLexicographicProduct` dont un extrait de code Coq est présenté dans la figure 5.6. La relation d'ordre partiel $\preceq_{lex}$ est décrite par `le`.

```
Module PreOrderLexicographicProduct (ord1 : PreOrderType)
  (ord2 : PreOrderType with Module Data := ord1.Data)
  <: PreOrderType
       with Module Data := ord1.Data
       with Definition type := (ord1.type * ord2.type)%type.

  Module Data := ord1.Data.

  Definition type := (ord1.type * ord2.type)%type.

  Definition le : Data.type -> type -> type -> Prop :=
    fun d x y =>
      match x , y with
        | (x1, x2), (y1,y2) => (ord1.le d x1 y1) /\
                               ((ord1.le d y1 x1) -> (ord2.le d x2 y2))
      end.

  Definition eq : Data.type -> type -> type -> Prop :=
        fun d x y => le d x y /\ le d y x.

  Definition eq_opt : Data.type -> type -> type -> Prop  :=
    fun d x y =>
      match x , y with
        | (x1,y1) , (x2, y2) => (ord1.eq d x1 x2) /\ (ord2.eq d y1 y2)
      end.

  Lemma eq_opt_is_eq : forall d x y, eq_opt d x y -> eq d x y.
  ...

End PreOrderLexicographicProduct.
```

FIG. 5.6 – *Produit lexicographique d'ordres partiels en* COQ

La suite des preuves sont également consultables dans le fichier Domain.v.

5.3.2.5.1 Ordre lexicographique sur des séquences

Nous définissons ici l'ordre lexicographique sur des séquences préalablement triées.

Définition 5.3.4. *Tri lexicographique des séquences* $\preceq_S$
Soient $\overrightarrow{s_1}$ *et* $\overrightarrow{s_2}$ *deux séquences dont les éléments sont triés par la relation* $\preceq$. *Les séquences représentent respectivement* $\langle b_1, \ldots, b_n \rangle$ *et* $\langle b'_1, \ldots, b'_m \rangle$.

$$\langle b_1, \ldots, b_n \rangle \preceq_S \langle b'_1, \ldots, b'_m \rangle \triangleq$$
$$n = 0 \vee (b_1 \preceq b'_1 \wedge (b'_1 \preceq b_1 \Rightarrow \langle b_2, \ldots b_n \rangle \preceq_S \langle b'_2, \ldots b'_m \rangle))$$

La figure 5.7 définit un fragment de l'ordre lexicographique de deux séquences.

```
Module PreOrderLexicographicSequence
  (ord : PreOrderType)
  <: PreOrderType
    with Module Data := ord.Data
    with Definition type := list ord.type.

  Module Data := ord.Data.

  Definition type := list ord.type.

  Fixpoint le (d : Data.type) (e1 e2 : type) { struct e1 } : Prop :=
    match e1 with
      | nil        => True
      | cons t1 q1 =>
        match e2 with
          | nil        => False
          | cons t2 q2 => (ord.le d t1 t2) /\
                          ((ord.le d t2 t1) -> (le d q1 q2))
        end
    end.

...
```

FIG. 5.7 – *Définition d'un ordre lexicographique sur des séquences*

5.3.3 Treillis et famille de treillis abstraits

La signature d'un ordre comporte le paramètre `Data`, il est donc possible de créer autant de treillis que de valeurs de `Data`. C'est pour cette raison que nous parlons de famille de treillis dans le cas général. Ainsi, un treillis est une famille de treillis paramétrée par d de type `Data` ($\mathfrak{D}$) instancié par le type `unit`. Le type Coq `unit` correspond au type unité qui contient un singleton (tt en Coq).

Nous construisons une famille de treillis $\{\theta_d\}_{d \in \mathfrak{D}}$ à partir du domaine abstrait Δ muni de la relation d'ordre partiel $\preceq$. Il s'agit d'une famille de treillis complets car les analyses effectuées ont un domaine Δ fini.

Éléments d'une famille de treillis : Les opérateurs de cette famille de treillis sont utilisés pour manipuler les informations calculées pour les différentes composantes du circuit d'entrée, c'est-à-dire les blocs, les ports, etc. La figure 5.8 présente la signature d'une famille de treillis en Coq.

Cette dernière, spécifiée par `LatticeType`, comporte le module `PreOrder` qui définit l'ordre partiel. Le paramètre d est de type `Data.type` (cf. Figure 5.2). L'opérateur $\sqcup$ (respectivement $\sqcap$) est défini par *join* (respectivement *meet*) tel qu'il est décrit dans la signature de la figure 5.8. Ces opérateurs calculent respectivement pour un couple d'éléments, la borne supérieure et la

```
Module Type LatticeType.

  Declare Module PreOrder : PreOrderType.

  Definition lt d x y := (PreOrder.le d x y) /\ ~ (PreOrder.le d y x).
  Definition gt d x y := lt d y x.
  Definition eq d x y := (PreOrder.le d x y) /\ (PreOrder.le d y x).
  Parameter gt_well_founded : forall d, well_founded (gt d).

  Parameter bot top : PreOrder.type.
  Parameter bot_least : forall d x, PreOrder.le d bot x.
  Parameter top_greatest : forall d x, PreOrder.le d x top.

  Parameter meet join : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
      PreOrder.type.

  Definition upper d a b x := (PreOrder.le d a x) /\ (PreOrder.le d b x).
  Parameter join_upper : forall d x y, upper d x y (join d x y) .
  Parameter join_supremum :
   forall d x y m, upper d x y m -> PreOrder.le d (join d x y) m.

  Definition lower d a b x := (PreOrder.le d x a) /\ (PreOrder.le d x b).
  Parameter meet_lower : forall d x y, lower d x y (meet d x y).
  Parameter meet_infimum :
   forall d x y m, lower d x y m -> (PreOrder.le d m (meet d x y)).

End LatticeType.
```

FIG. 5.8 – *Signature d'une famille de treillis en* COQ

borne inférieure. Les bornes inférieure $\bot$ et supérieure $\top$ de la famille de treillis sont représentées respectivement par les variables `bot` et `top` dans la signature. `upper` (respectivement `lower`) définit le majorant (respectivement le minorant) d'un couple d'éléments.

Par ailleurs, nous avons choisi de fixer une définition générique pour les relations `lt, gt` et `eq` en exploitant `le`. Cela permet de ne pas changer leurs spécifications dans les foncteurs.

Nous avons indiqué précédemment que le théorème de Kleene est appliqué sur un treillis complet pour la relation d'ordre $\preceq$ définie sur Δ. Or, pour calculer le plus petit point fixe d'une fonction monotone par itération, il faudra disposer d'hypothèses supplémentaires pour assurer la terminaison des calculs. Une condition nécessaire est que la relation $\preceq$ soit bien fondée.

La condition la plus simple est d'interdire l'existence de chaînes croissantes infinies. Ce critère ne dépend que de la famille de treillis manipulée. Ainsi, la preuve de terminaison du calcul ne dépend pas de l'analyse effectuée mais de l'environnement du calcul effectué.

Les informations que nous calculons croissent car la fonction de propagation doit être monotone. Il est donc nécessaire de vérifier que la relation inverse de $\preceq$ ne décroît pas à l'infini. Il s'agit donc de vérifier que la relation $\succ$ est bien fondée (voir définition 5.2.14).

Dans la signature d'une famille de treillis, la spécification `gt` ($\succ$) est la relation inverse de la relation `lt` ($\prec$). La propriété `gt_well_founded` assure que la relation `gt` est bien fondée.

La signature proposée d'un treillis (ou famille de treillis) n'est pas tout à fait similaire à la signature usuelle d'un treillis, car nous n'avons besoin que d'une relation de préordre partiel. En effet, deux blocs différents peuvent avoir la même information calculée, donc l'antisymétrie n'est pas utile dans notre cas.

La structure que nous manipulons est finie (car les circuits sont de taille finie) avec les propriétés :

- $\bot$ est la borne inférieure de la famille de treillis. Cette dernière correspond à la propriété `bot_least` ;
- $\top$ est la borne supérieure de la famille de treillis. Cette propriété est spécifiée par `top_greatest` ;

À ces propriétés s'ajoutent `join_upper` (respectivement `meet_lower`) qui indique que l'union (respectivement l'intersection) de deux éléments est leur majorant (respectivement minorant), `join_supremum` (respectivement `meet_infimum`) qui indique que le majorant (respectivement le minorant) de deux éléments est plus grand (respectivement plus petit) que leur union (respectivement leur intersection). Les propriétés `join_supremum` et `meet_infimum` découlent des définitions 5.2.3 et 5.2.5.

Lors de l'instanciation des foncteurs, les propriétés sont des lemmes à démontrer.

5.3.3.1 Produit cartésien de familles de treillis

Certaines analyses des outils élémentaires manipulent le produit cartésien de familles de treillis. Tel est le cas lorsque l'analyse calcule plusieurs informations pour chaque composante. Un cas d'utilisation sera présenté dans le chapitre 7. Pour le moment, nous nous contentons de définir la signature du produit cartésien de familles de treillis complets.

Définition 5.3.5. ***Produit cartésien de familles de treillis***
Soient $\theta \triangleq \langle \mathcal{X}, \preceq_\theta, \bot_\theta, \top_\theta, \sqcup_\theta, \sqcap_\theta \rangle$ *et* $\theta' \triangleq \langle \mathcal{Y}, \preceq_{\theta'}, \bot_{\theta'}, \top_{\theta'}, \sqcup_{\theta'}, \sqcap_{\theta'} \rangle$ *deux familles de treillis ayant le même paramètre* d.
Le produit cartésien $\theta \times \theta' \triangleq \langle \mathcal{X} \times \mathcal{Y}, \preceq_{\theta\times\theta'}, \bot_{\theta\times\theta'}, \top_{\theta\times\theta'}, \sqcup_{\theta\times\theta'}, \sqcap_{\theta\times\theta'} \rangle$ *est également une famille de treillis définie par :*

1. $\langle \mathcal{X} \times \mathcal{Y}, \preceq_{\theta\times\theta'} \rangle \triangleq \langle \mathcal{X}, \preceq_\theta \rangle \times \langle \mathcal{Y}, \preceq_{\theta'} \rangle$
2. *Soient* $\bot_\theta$ *et* $\bot_{\theta'}$ *les bornes inférieures de* θ *et* θ'. *La borne inférieure de* $\theta \times \theta'$, *notée* $\bot_{\theta\times\theta'}$ *est le couple* $(\bot_\theta, \bot_{\theta'})$
3. *Soient* $\top_\theta$ *et* $\top_{\theta'}$ *les bornes supérieures de* θ *et* θ'. *La borne supérieure de* $\theta \times \theta'$, *notée* $\top$ *est le couple* $(\top_\theta, \top_{\theta'})$
4. $\forall (a,b),\ (c,d) \in \mathcal{X} \times \mathcal{Y},\ (a,b) \sqcup (c,d) \triangleq (a \sqcup_\theta c\ ,\ b \sqcup_{\theta'} d)$
5. $\forall (a,b),\ (c,d) \in \mathcal{X} \times \mathcal{Y},\ (a,b) \sqcap (c,d) \triangleq (a \sqcap_\theta c\ ,\ b \sqcap_{\theta'} d)$

Le foncteur implantant le produit cartésien de familles de treillis est présenté dans la figure 5.9. Le produit cartésien de familles de treillis est une famille de treillis, par conséquent, il préserve la signature du module `LatticeType`.

5.3.3.2 Famille de treillis inverse

Dans certaines analyses, il est utile de considérer l'inverse d'une famille de treillis, comme dans l'étude du typage (voir Chapitre 7).

Nous définissons dans un premier temps la signature d'une famille de treillis ayant une relation d'ordre strict $\prec$ (spécifiée par `lt`) bien fondée. Cette propriété (de bon fondement) sera utilisée pour démontrer que la relation duale $\succ$ (spécifiée par `gt`) dans une famille de treillis inverse est également bien fondée.

```
Module LatticeCartesianProduct
 (L1 : LatticeType)
 (L2 : LatticeType with Module PreOrder.Data := L1.PreOrder.Data)
 <: LatticeType.

 Module PreOrder := PreOrderCartesianProduct (L1.PreOrder) (L2.PreOrder)

 Definition lt d e1 e2 := (PreOrder.le d e1 e2) /\ ~ (PreOrder.le d e2 e1).

 Definition gt d e1 e2 := lt d e2 e1.

 Definition eq d e1 e2 := (PreOrder.le d e1 e2) /\ (PreOrder.le d e2 e1).

 Definition bot := (L1.bot, L2.bot).

 Definition top := (L1.top, L2.top).

 Definition upper d a b x := (PreOrder.le d a x) /\ (PreOrder.le d b x).

 Definition join : PreOrder.Data.type -> O.type -> O.type -> O.type :=
   fun d x y =>
     match x, y with
       | (x1, x2), (y1, y2) => (L1.join d x1 y1, L2.join d x2 y2)
     end.

 Definition lower d a b x := (PreOrder.le d x a) /\ (PreOrder.le d x b).

 Definition meet : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
      PreOrder.type :=
   fun d x y =>
     match x, y with
       | (x1, x2), (y1, y2) => (L1.meet d x1 y1, L2.meet d x2 y2)
     end.
 ...

End LatticeCartesianProduct.
```

FIG. 5.9 – *Définition du produit cartésien de familles de treillis en* COQ

```
Module Type LatticeLtWfType.
  Include Type LatticeType.
  Parameter lt_well_founded : forall d, well_founded (lt d).
End LatticeLtWfType.
```

La signature de `LatticeLtWfType` importe celle d'une famille de treillis précédemment définie, par `Include Type LatticeType`, et est complétée par la propriété de bon fondement de $\prec$.

Ainsi, nous définissons le foncteur `LatticeInv` qui implante la famille de treillis inverse d'une famille de treillis donnée. `LatticeInv` est défini dans la figure 5.10.

5.3.4 Environnement abstrait

La sémantique abstraite consiste à associer à chaque circuit à analyser un environnement abstrait Y et une fonction de propagation ρ associée à l'analyse souhaitée. L'environnement associe aux composantes du circuit (soit bloc, soit port, etc.) une valeur du domaine abstrait Δ, formant une famille de treillis complets. Il s'agit de calculer, en utilisant ρ, l'information correspondant à une analyse donnée, par exemple, les rangs des blocs, les dépendances des blocs, les types des ports des blocs, etc. Nous étudions ces calculs en détail dans les chapitres suivants. À présent, nous nous focalisons sur la définition de l'environnement.

```
Module LatticeInv (L : LatticeLtWfType) <: LatticeLtWfType
  with Module PreOrder.Data := L.PreOrder.Data
  with Definition PreOrder.type := L.PreOrder.type.

  Module PreOrder := ReversePreOrder L.PreOrder.

  Definition lt d x y := (PreOrder.le d x y) /\ ~ (PreOrder.le d y x).

  Definition gt d x y := lt d y x.

  Definition eq d x y := (PreOrder.le d x y) /\ (PreOrder.le d y x).

  Lemma gt_well_founded : forall d, well_founded (gt d).
  ...

  Definition bot := L.top.
  Definition top := L.bot.

  Lemma bot_least : forall d x, PreOrder.le d bot x.
    ...

  Lemma top_greatest : forall d x, PreOrder.le d x top.
  ...

  Definition meet : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
       PreOrder.type :=
         fun d x y => L.join d x y.
  Definition join : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
       PreOrder.type :=
         fun d x y => L.meet d x y.

  Definition upper d a b x := (PreOrder.le d a x) /\ (PreOrder.le d b x).

  Lemma join_upper : forall d x y, upper d x y (join d x y).
  ...

  Lemma join_supremum : forall d x y m, upper d x y m ->
    PreOrder.le d (join d x y) m.
  ...

  Definition lower d a b x := (PreOrder.le d x a) /\ (PreOrder.le d x b).

  Lemma meet_lower : forall d x y, lower d x y (meet d x y).
    ...

  Lemma meet_infimum : forall d x y m, lower d x y m ->
    PreOrder.le d m (meet d x y).
  ...

  Lemma lt_well_founded : forall d, well_founded (lt d).
  ...

End LatticeInv.
```

FIG. 5.10 – *Implantation du foncteur* `LatticeInv`

Pourquoi manipuler des types dépendants ? Les environnements abstraits manipulés dans le cadre de nos travaux, exploitent, pour les calculs souhaités, des types dépendants (familles de treillis). Ces derniers dépendent d'une taille notée s : le nombre de blocs du circuit, le nombre de ports d'un bloc donné, etc. Cette taille est dynamique (change à chaque lecture de circuit), tandis que l'instanciation des modules est statique. Il est donc judicieux de définir un cadre général pour les données traitées et construire une famille de treillis dont la valeur du paramètre n'est pas imposée dans la signature. Autant de treillis existent dans une famille de treillis que de valeurs possibles du paramètre de cette famille de treillis. De plus, l'utilisateur peut modéliser son circuit sans être contraint par une taille fixée statiquement. Cette taille est finie puisque les cir-

cuits sont des structures finies. C'est pourquoi nous avons choisi de manipuler la structure de famille de treillis à la place de la structure de treillis.

Un environnement Y associe l'information qui doit être calculée à chaque composante du circuit (bloc ou port). Ce dernier est représenté par un identifiant entier naturel. En effet, il est plus simple de distinguer les différentes composantes du circuit par des entiers que par leurs structures. Une discussion sur le choix d'utiliser des entiers sera présentée dans le chapitre 8.

L'environnement est une famille de treillis paramétrée par le nombre de composantes à analyser (s), et par le type $\mathfrak{D}$ du paramètre de la famille de treillis $\{\theta_d\}_{d\in\mathfrak{D}}$.

Définition 5.3.6. ***Environnement abstrait*** Y
Un environnement de taille s est une famille de treillis définie par :

$$\mathrm{Y} \triangleq \{\{n \in \mathbb{N} | n < s\} \to \{\theta_{f(n)}\}\}_{s,f\in\mathbb{N}\to\mathfrak{D}}$$

L'environnement Y est paramétré par le couple (s, f) de type $\mathbb{N} \times (\mathbb{N} \to \mathfrak{D})$ qui sera défini dans ce qui suit.

5.3.4.1 Paramètre de l'environnement

Le paramètre (s, f) de l'environnement est un foncteur implantant le type `DataType`. La signature de ce type est spécifiée par `DataMap` illustré dans la figure 5.11.

```
Module DataMap (Data : DataType) <: DataType.

  Definition type := (nat * (nat -> Data.type))%type.
  Definition eq : type -> type -> Prop.
  Proof.
    destruct 1;
      destruct 1.
    exact (n=n0 /\ (forall p, p<n -> Data.eq (t p) (t0 p))).
  Defined.

  Lemma refl : forall x, eq x x.
  ...

  Lemma sym : forall x y, eq x y -> eq y x.
  ...

  Lemma trans : forall x y z, (eq x y) -> (eq y z) -> (eq x z).
  ...

  Lemma dec : forall x y, {eq x y}+{~eq x y}.
  ...
End DataMap
```

FIG. 5.11 – *Signature* Coq *du module* `DataMap`

L'équivalence dans `type` est définie par la preuve que deux éléments (n_0, t_0) et (n_1, t_1) de `type` sont équivalents par `eq` si $n_0 = n_1$ et $\forall\, p, p < n_0 \Rightarrow$ `Data.eq`$(t_0\ p)(t_1\ p)$. La condition $p < n_0$ assure, lors d'un calcul, que l'identifiant de la composante analysée (bloc ou port, etc.) est dans $[0..s[$. Le reste des preuves peut être consulté dans le fichier Domain.v.

5.3.4.2 Ordre partiel de l'environnement

Les éléments de l'environnement sont définis dans $\{n \in \mathbb{N} | n < s\} \to \{\theta_{f(n)}\}$. Étant donné $\{\theta_d\}_{d\in\mathfrak{D}}$ une famille de treillis, nous définissons un ordre partiel

pour l'environnement, noté $\preceq_Y$, sur $\{n \in \mathbb{N} | n < s\} \rightarrow \{\theta_{f(n)}\}$ préservant la relation d'ordre $\preceq$ définie dans la famille de treillis $\{\theta_d\}_{d \in \mathfrak{D}}$.
La relation d'ordre partiel $\preceq_Y$ est définie dans le module `DependentMapPreOrder` présenté dans la figure 5.12.

Définition 5.3.7. *Relation d'ordre partiel de l'environnement* $\preceq_Y$

$$\forall e_1, e_2 : Y,\ e_1 \preceq_Y e_2 \triangleq \forall n < s, e_1(n) \preceq e_2(n).$$

La relation d'ordre partiel $\preceq_Y$, spécifiée par `le`, assure que pour deux environnements e_1 et e_2, e_1 est plus petit que e_2 si pour toute composante, l'ordre est préservé. L'identifiant des composantes est un élément de $[0..s[$, avec s le nombre des composantes (spécifié par `size`).

```
Module DependentMapPreOrder (PreOrder : PreOrderType) <: PreOrderType.

  Module Data := DataMap (PreOrder.Data).
  Definition type := nat -> PreOrder.type.
  Definition le (d : Data.type) (e1 e2 : type) :=
    let (size, Dn) := d
        in
      forall n, n < size -> PreOrder.le (Dn n) (e1 n) (e2 n).

  Definition eq : Data.type -> type -> type -> Prop :=
   fun d x y => (le d x y) /\ (le d y x).
  Definition eq_opt (d : Data.type) (e1 e2 : type) :=
    let (size, Dn) := d
        in
      forall n, n < size -> PreOrder.eq (Dn n) (e1 n) (e2 n).
  Lemma eq_opt_is_eq : forall d x y, (eq_opt d x y) -> (eq d x y).
  ...
  Lemma eq_is_eq_opt : forall d x y, eq d x y -> eq_opt d x y.
  ...
  Lemma eq_opt_dec : forall d x y, { (eq_opt d x y) } + { ~ (eq_opt d x y) }.
  ...
  Lemma eq_dec : forall d x y, { eq d x y } + { ~ (eq d x y) }.
  ...
  Lemma refl : forall d e, le d e e.
  ...
  Lemma trans : forall d e1 e2 e3, le d e1 e2 -> le d e2 e3 -> le d e1 e3.
  ...
  Lemma dec :  forall d e1 e2, { (le d e1 e2) } + { ~ (le d e1 e2) }.
  ...

End DependentMapPreOrder.
```

FIG. 5.12 – *Définition d'un ordre partiel pour les types dépendants* `DependentMapPreOrder`

Les preuves des propriétés de l'ordre partiel `DependentMapPreOrder` exploitent principalement les propriétés prouvées du module `PreOrder`. Elles sont consultables dans le fichier Domain.v.

5.3.4.3 Implantation de l'environnement

Maintenant, nous construisons, la famille de treillis Y en utilisant la relation d'ordre $\preceq_Y$. L'environnement Y est implanté par le foncteur de `MakeDependentEnvironment` présenté dans la figure 5.13.
La borne inférieure de Y est la fonction $\bot_Y = n \in \{n \in \mathbb{N} | n < s\} \mapsto \bot$, et sa plus grande borne supérieure est la fonction $\top_Y = n \in \{n \in \mathbb{N} | n < s\} \mapsto \top$.

La terminaison des calculs est assurée par la propriété de bon fondement de la relation gt définie sur les environnements, notée $\succ_Y$.

Théorème 5.3.1. *La relation* `gt` *est bien fondée*

La preuve peut être consultée dans le fichier EnvironmentLattice.v.

```
Module MakeDependentEnvironment (Lattice : LatticeType) <: LatticeType.

  Module PreOrder := DependentMapPreOrder (Lattice.PreOrder).
  Definition bot : PreOrder.type := fun _ => Lattice.bot.
  Lemma bot_least : forall d x, PreOrder.le d bot x.
  ...
  Definition top : PreOrder.type := fun _ => Lattice.top.
  Theorem top_greatest : forall d x, PreOrder.le d x top.
  ...
  Definition lt d e1 e2 := (PreOrder.le d e1 e2) /\ ~ (PreOrder.le d e2 e1).
  Definition gt d e1 e2 := lt d e2 e1.
  Definition eq d e1 e2 := (PreOrder.le d e1 e2) /\ (PreOrder.le d e2 e1).
  Definition upper d a b x := PreOrder.le d a x /\ PreOrder.le d b x.
  Definition join (d : PreOrder.Data.type) (e1 e2 : O.type) : PreOrder.type :=
    let (size, Dn) := d in
    fun n => Lattice.join (Dn n) (e1 n) (e2 n).

  Lemma join_upper : forall d x y, upper d x y (join d x y).
    ...
  Lemma join_supremum : forall d x y m,
   upper d x y m -> PreOrder.le d (join d x y) m.
   ...
  Definition lower d a b x := PreOrder.le d x a /\ PreOrder.le d x b.
  Definition meet (d : PreOrder.Data.type) (e1 e2 : PreOrder.type) :=
    let (size, Dn) := d in
    fun n => Lattice.meet (Dn n) (e1 n) (e2 n).
  Lemma meet_lower : forall d x y, lower d x y (meet d x y).
  ...
  Lemma meet_infimum : forall d x y m,
   lower d x y m -> PreOrder.le d m (meet d x y).
   ...
  Lemma gt_well_founded : forall d, well_founded (gt d).
    ...

End MakeDependentEnvironment.
```

FIG. 5.13 – *Définition d'un environnement*

À l'utilisation, l'environnement est instancié par :

```
Module EnvironmentLattice := MakeDependentEnvironment(dataLattice).
```

avec `dataLattice` la spécification en Coq de la famille de treillis $\{\theta_d\}_{d\in\mathfrak{D}}$ propre à une analyse particulière.

5.3.4.4 Exemples

Le paramètre $\mathfrak{D}$ d'une famille de treillis varie en fonction de l'analyse souhaitée. Nous présentons, ici, deux exemples qui requièrent différents paramètres et donc différentes familles de treillis pour l'environnement Y.

Exemple 5.3.2. *L'analyse souhaitée correspond au calcul du rang entier des blocs par rapport aux premiers blocs sources dans un circuit. Un entier sera, donc, associé à chaque bloc du circuit.*

L'environnement associe donc une valeur entière de $\Delta \subset \mathbb{N}$ à chaque identifiant de bloc d'un circuit contenant s blocs. Le calcul des rangs ne dépend que de la structure du circuit. Ainsi, le type $\mathfrak{D}$ du paramètre de l'environnement Y est le type unité (`unit` en Coq et en anglais). Il s'agit d'un type à un seul élément `tt`.

Le domaine de calcul Δ muni de la relation $\leq$ est un ordre partiel. Soit $\{\theta_d\}_{d\in\mathfrak{D}}$ la famille de treillis obtenue à partir de cet ordre partiel dont le type de dépendance $\mathfrak{D}$ est l'identité.

L'environnement permettant d'associer pour chaque bloc son rang est défini par :

$$\mathrm{Y} \triangleq \{\{n \in \mathbb{N} | n < s\} \to \{\theta_{f(n)}\}\}_{s\in\mathbb{N}, f\in\mathbb{N}\to \mathit{unit}}$$

avec $f(n)$ égal à `tt` pour tout n.

Exemple 5.3.3. *Si l'analyse souhaitée est le calcul des types des ports constituant les blocs d'un circuit donné, l'environnement associe alors un type à chaque port de chaque bloc de ce circuit.*

Le domaine Δ correspond à l'ensemble des types acceptés par l'analyse. Δ muni de la relation d'ordre partiel $\preceq$ (définie sur les types) est un ordre partiel. Soit $\{\theta_d\}_{d \in unit}$ la famille de treillis obtenue à partir de cet ordre partiel.

L'environnement, noté Υ', qui permet d'associer à chaque port un type pour un bloc donné, est défini par :

$$\Upsilon' \triangleq \{\{n \in \mathbb{N} | n < p\} \rightarrow \{\theta_{f'(n)}\}\}_{p \in \mathbb{N}, f' \in \mathbb{N} \rightarrow unit}$$

avec p le nombre de ports du bloc et $f(n)$ égal à `tt` pour tout n.

Cet environnement servira pour définir un environnement, noté Υ, qui associe un type pour chaque port de chaque bloc du circuit. Notons $\mathfrak{D}$ le type $\mathbb{N} \times (\mathbb{N} \rightarrow unit)$.

Le type d'un port est lié à un bloc particulier. Donc, le type du paramètre de l'environnement Υ est $\mathbb{N} \times (\mathbb{N} \rightarrow \mathfrak{D})$. L'environnement Υ est alors défini par :

$$\Upsilon \triangleq \{\{n \in \mathbb{N} | n < s\} \rightarrow \Upsilon'_{f(n)}\}_{s \in \mathbb{N}, f \in \mathbb{N} \rightarrow \mathfrak{D}}$$

5.3.5 Fonction de propagation

Un outil élémentaire du générateur de code est dédié à une analyse donnée. La fonction de propagation dépend donc du calcul souhaité. Le calcul est effectué par la fonction $\rho : \Upsilon \rightarrow \Upsilon$ qui doit satisfaire la propriété de croissance. Cette propriété dépend de l'analyse effectuée et fera l'objet de preuves décrites dans la partie vérification des chapitres 6 et 7.

La fonction ρ calcule à partir d'un état courant i de l'environnement Υ l'état suivant de ce dernier tel que $(\Upsilon)_{i+1} = \rho((\Upsilon)_i)$. Initialement, le calcul démarre avec un environnement initial, noté $(\Upsilon)_0$. ρ exploite la structure du circuit et son résultat doit être conforme avec la relation d'ordre $\preceq_\Upsilon$ préalablement définie. La fonction de propagation a la signature :

```
Parameter Forward : ModelType -> EnvironmentLattice.PreOrder.type ->
    EnvironmentLattice.PreOrder.type.
```

5.3.6 Calcul de point fixe

Les itérations de la fonction ρ depuis l'environnement initial $(\Upsilon)_0$ permettent de propager les informations calculées dans le circuit.

Selon le théorème de Kleene, toute chaîne croissante
$e_1 \preceq_\Upsilon \ldots \preceq_\Upsilon e_n \preceq_\Upsilon \ldots$ de la famille de treillis Υ est stationnaire après un nombre fini d'itérations de ρ.
Autrement dit, $\exists\, p \in \Delta, \exists\, i, \forall\, j,\ i < j \Rightarrow e_i = e_j = p$, avec p égal à $lfp(\rho)$, c'est-à-dire le point fixe de la fonction ρ, et le symbole $=$ fait référence à l'équivalence `eq` définie pour l'environnement Υ.

Nous définissons, dans un premier temps, le type inductif `isIterate` qui spécifie l'ensemble des itérations d'une fonction ρ.

```
Inductive isIterate : E.EnvironmentLattice.PreOrder.type -> Prop :=
 | O_iter : isIterate E.initialEnv
 | S_iter : forall currentStep,
     isIterate currentStep
   -> isIterate (E.Forward E.diagram currentStep).
```

Soit `initialEnv` la spécification Coq de l'environnement initial Y_0. Ce dernier doit être un pré-point fixe, c'est-à-dire, $Y_0 \preceq_Y \rho(Y_0)$. La preuve de cette propriété dépend de l'environnement initial de chaque analyse (ordonnancement, typage, etc.). `O_iter` désigne l'itération initiale (à partir de Y_0), tandis que `S_iter` désigne l'itération suivante de l'état courant spécifié par `currentStep`.

Soit $\star$ l'opérateur d'itération qui appelle récursivement ρ. Notons par $\rho^\star$ la fonction de construction de point fixe.

La terminaison de la fonction de $\rho^\star$ est assurée en démontrant qu'à partir d'un environnement initial, le calcul converge vers un point fixe. Nous utilisons le principe de la programmation par la preuve pour exprimer que la construction du point fixe se termine. Cette preuve est commune à tous les outils élémentaires. Soit la fonction `tool_rec` suivante qui définie $\rho^\star$.

```
Function tool_rec
   (data : E.EnvironmentLattice.PreOrder.type)
   (iterated : isIterate data)
   { wf (E.EnvironmentLattice.gt E.d) data }
   : E.EnvironmentLattice.PreOrder.type :=
   let nextdata := E.Forward E.diagram data in
      if (E.EnvironmentLattice.PreOrder.dec E.d
        nextdata
        data)
  then data
  else tool_rec nextdata (S_iter _ iterated).
```

`E` est le module contenant les différents éléments manipulés pour définir la fonction `tool_rec`. Cette dernière est structurellement récursive vis-à-vis du bon fondement de la relation `gt` de l'environnement de calcul, noté $\succ_Y$.

Il s'agit de prouver deux sous-buts :

```
2 subgoals

============================
 forall data : E.EnvironmentLattice.PreOrder.type,
 isIterate data ->
 forall
   anonymous : ~
               E.EnvironmentLattice.PreOrder.le E.d
                 (E.Forward E.diagram data) data,
 E.EnvironmentLattice.PreOrder.dec E.d (E.Forward E.diagram data) data =
 Utils.in_right ->
 E.EnvironmentLattice.gt E.d (E.Forward E.diagram data) data

subgoal 2 is:
 well_founded (E.EnvironmentLattice.gt E.d)
```

Le premier sous-but correspond à la propriété `nextData` $\preceq_Y$ `data`, ce qui signifie qu'une nouvelle itération de la fonction `tool_rec` doit avoir lieu, car le calcul ne s'est pas terminé. Tandis que le second sous-but correspond à la propriété de bon fondement de la relation $\succ_Y$.

Nous appliquons ensuite des simplifications liées à la définition de $\succ_Y$.

```
intros.
unfold E.EnvironmentLattice.gt.
unfold E.EnvironmentLattice.lt.
```

Cela génère les sous-buts suivants :

```
============================
 E.EnvironmentLattice.PreOrder.le E.d data (E.Forward E.diagram data) /\
 ~ E.EnvironmentLattice.PreOrder.le E.d (E.Forward E.diagram data) data

subgoal 2 is:
 well_founded (E.EnvironmentLattice.gt E.d)
```

La propriété `isPreFixpointForward` assure qu'un appel de ρ depuis un environnement $(Y)_i$ permet d'obtenir un environnement $(Y)_{i+1}$ plus grand. La première partie de la conjonction est résolue en appliquant la propriété `isPreFixpointForward` ainsi que la propriété `iterated`, tandis que la seconde est vraie par hypothèse.

```
split.
apply isPreFixpointForward.
exact iterated.
exact anonymous.
```

Le dernier sous-but, concernant le bon fondement de $\succ_Y$, est satisfait car l'environnement Y comporte la preuve `gt_well_founded`.

Le script de preuve de terminaison de la fonction $\rho^\star$ correspond à `tool_rec` de la figure 5.14.

La définition de la fonction $\rho^\star$ sera utilisée pour démontrer que le point fixe est atteint.

Théorème 5.3.2. *Un point fixe de ρ est atteint si et seulement si :*

$$\rho(\rho^\star(Y_0)) \ =_Y \ \rho^\star(Y_0)$$

Le point fixe calculé par $\rho^\star$ correspond au plus petit point fixe.

Théorème 5.3.3. $\rho^\star(Y_0)$ *calcule le plus petit point fixe à partir de* Y_0

$$\forall X,\ Y_0 \preceq_Y X \Rightarrow X = \rho(X) \Rightarrow \rho^\star(Y_0) \preceq_Y X$$

Ce théorème correspond à `isLessThanFixPointTool_rec` (voir figure 5.15) appliqué à l'environnement initial Y_0. Les preuves relatives au point fixe sont également données dans la figure 5.15.

5.4 Outils élémentaires

Le choix d'une technique de développement correct par construction est, d'une part, motivé par la réduction du coût des tests et, d'autre part, par la rigueur apportée par l'utilisation des méthodes formelles dans le cycle de développement d'applications industrielles critiques. La rigueur des méthodes formelles augmente la qualité des outils.

5.4.1 Conception d'un outil élémentaire

La signature de la conception d'un outil élémentaire est illustrée par la figure 5.14. Elle comporte les éléments nécessaires pour l'analyse attendue par l'outil élémentaire.

La signature de la conception d'un outil élémentaire comporte les éléments suivants :

Circuit : le circuit lu en entrée du générateur de code. Il est déclaré comme variable globale et sera lu par les différents outils élémentaires ;

Famille de treillis : la famille de treillis $\{\theta_d\}_{d \in unit}$ sur laquelle repose le calcul de l'outil élémentaire ;

Paramètre du type dépendant : le type $\mathfrak{D}$ du paramètre de l'environnement ;

```
Module Type Input.
 Parameter diagram : ModelType.
End Input.

Declare Module I : Input.

Module Type ElementaryToolDesign.

  Definition diagram := I.diagram.
  Declare Module dataLattice : LatticeType.
  Parameter d : EnvironmentLattice.PreOrder.Data.type.
  Module EnvironmentLattice := MakeDependentEnvironment(dataLattice).

  Parameter Forward : ModelType -> EnvironmentLattice.PreOrder.type ->
      EnvironmentLattice.PreOrder.type.

  Parameter initialEnv : EnvironmentLattice.PreOrder.type.

  Parameter Forward_monotonous : forall c : ModelType, forall env0 env1 :
    EnvironmentLattice.PreOrder.type, EnvironmentLattice.PreOrder.le d env0 env1
        ->
    (EnvironmentLattice.PreOrder.le d (Forward diagram env0)
        (Forward diagram env1)).

  Parameter initialEnvIsPreFixpoint :
    EnvironmentLattice.PreOrder.le d initialEnv (Forward diagram initialEnv).

End ElementaryToolDesign.
```

FIG. 5.14 – *Signature de la conception d'un outil élémentaire*

Environnement de calcul : l'environnement Y, construit en utilisant la famille de treillis $\{\theta_d\}_{d \in unit}$;

Fonction de propagation : la fonction de calcul ρ ;

Environnement initial : l'environnement Y_0 à partir duquel démarre le calcul du point fixe de l'analyse souhaitée ;

Croissance de l'itération à partir d'un environnement initial : cette propriété spécifie que l'environnement Y_0 est un pré-point fixe ;

Croissance de la fonction de propagation : la propriété de croissance assurant que les calculs effectués à partir de Y_0 croissent et convergent.

5.4.2 Définition d'un outil élémentaire

Les outils élémentaires sont implantés comme des foncteurs à partir de la signature de `ElementaryToolDesign`.

Le calcul souhaité est alors effectué par le foncteur `ElementaryTool` présenté dans la figure 5.15. Il est paramétré par la conception de l'outil élémentaire en question, et calcule le point fixe de la fonction ρ avec la fonction `computeData`.

Le module `ElementaryTool` comporte :

Définition d'une itération : une itération est définie par un type inductif. Une itération peut être définie à partir d'un état initial (donc une première itération), ou alors par une itération $i+1$ définie en appliquant ρ à l'itération i. Cette définition inductive de l'itération est exploitée pour raisonner sur la terminaison du calcul itératif $\rho^\star$ et sur les propriétés de point fixe ;

Propriété de point fixe : la propriété assure que l'environnement de calcul obtenu par application de ρ croît d'un état i à un état $i+1$ d'une itération de $\rho^\star$. L'état i est donc un pré-point fixe ;

Terminaison de la construction de point fixe : elle assure que le calcul itératif $\rho^\star$ se termine ;

Calcul depuis l'environnement initial : il s'agit d'appliquer $\rho^\star$ pour l'environnement initial Y_0 ;

Propriétés assurant que le résultat est un point fixe : Cette preuve est découpée en plusieurs pour assurer que $\rho^\star$ calcule le pré-point fixe qui est aussi le post-point fixe. $\rho^\star$ calcule donc le plus petit point fixe.

```
Module ElementaryTool (E : ElementaryToolDesign).

  Definition l := E.diagram.
  Require Import Recdef.

  Inductive isIterate : E.EnvironmentLattice.PreOrder.type -> Prop :=
  | O_iter : isIterate E.initialEnv
  | S_iter : forall currentStep,
      isIterate currentStep
    -> isIterate (E.Forward E.diagram currentStep).

  Definition isPreFixpoint (data : E.EnvironmentLattice.PreOrder.type) :=
    E.EnvironmentLattice.PreOrder.le E.d data (E.Forward E.diagram data).

  Lemma isPreFixpointForward :
    forall
      (data : E.EnvironmentLattice.PreOrder.type),
      isIterate data
      -> isPreFixpoint data.
  Proof.
    intros.
    induction H.
    exact E.initialEnvIsPreFixpoint.
    unfold isPreFixpoint in |- *.
    apply E.Forward_monotonous.
    exact E.diagram.
    exact IHisIterate.
  Qed.

  Function tool_rec
    (data : E.EnvironmentLattice.PreOrder.type)
    (iterated : isIterate data)
    { wf (E.EnvironmentLattice.gt E.d) data }
    : E.EnvironmentLattice.PreOrder.type :=
    let nextdata := E.Forward E.diagram data in
       if (E.EnvironmentLattice.PreOrder.dec E.d
         nextdata
         data)
   then data
   else tool_rec nextdata (S_iter _ iterated).
  Proof.
    intros.
    unfold E.EnvironmentLattice.gt in |- *.
    unfold E.EnvironmentLattice.lt in |- *.
    split.
    apply isPreFixpointForward.
    exact iterated.
    exact anonymous.
    apply (E.EnvironmentLattice.gt_well_founded E.d).
  Defined.

Definition computeData :=
  tool_rec E.initialEnv O_iter.

Lemma iterateLessThanFixpoint :
    forall
      (data : E.EnvironmentLattice.PreOrder.type)
      (iterated : isIterate data)
      (fixe : E.EnvironmentLattice.PreOrder.type),
      E.EnvironmentLattice.PreOrder.le E.d E.initialEnv fixe
   -> E.EnvironmentLattice.eq E.d fixe (E.Forward E.diagram fixe)
   -> E.EnvironmentLattice.PreOrder.le E.d data fixe.
Proof.
  intros.
  induction iterated.
```

```
  exact H.
```

```
  apply
    (E.EnvironmentLattice.PreOrder.trans E.d (E.Forward E.diagram currentStep)
      (E.Forward E.diagram fixe) fixe).
  apply E.Forward_monotonous.
  exact E.diagram.
  apply IHiterated.
  unfold E.EnvironmentLattice.eq in H0.
  intuition.
Qed.

Lemma isIterateTool_rec :
    forall
      (data : E.EnvironmentLattice.PreOrder.type)
      (iterated : isIterate data),
      isIterate (tool_rec data iterated).
Proof.
  intros data.
  elim (E.EnvironmentLattice.gt_well_founded E.d data).
  intros.
  clear H.
  rewrite tool_rec_equation in |- *.
  simpl in |- *.
  destruct E.EnvironmentLattice.PreOrder.dec.
  exact iterated.
  apply H0.
  unfold E.EnvironmentLattice.gt in |- *.
  unfold E.EnvironmentLattice.lt in |- *.
  split.
  apply isPreFixpointForward.
  exact iterated.
  exact n.
Qed.

Lemma isLessThanFixpointTool_rec :
    forall
      (data : E.EnvironmentLattice.PreOrder.type)
      (iterated : isIterate data)
      (fixe : E.EnvironmentLattice.PreOrder.type),
      E.EnvironmentLattice.PreOrder.le E.d E.initialEnv fixe
   -> E.EnvironmentLattice.eq E.d fixe (E.Forward E.diagram fixe)
   -> E.EnvironmentLattice.PreOrder.le E.d (tool_rec data iterated) fixe.
Proof.
  intros.
  eapply iterateLessThanFixpoint.
  apply isIterateTool_rec.
  exact H.
  exact H0.
Qed.

Lemma isPostFixpointTool_rec :
    forall
      (data : E.EnvironmentLattice.PreOrder.type)
      (iterated : isIterate data),
      E.EnvironmentLattice.PreOrder.le E.d (E.Forward E.diagram (tool_rec data
          iterated))
                                                    (tool_rec data iterated).
Proof.
  intros data.
  elim (E.EnvironmentLattice.gt_well_founded E.d data).
  intros.
  clear H.
  rewrite tool_rec_equation in |- *.
  simpl in |- *.
  destruct E.EnvironmentLattice.PreOrder.dec.
  exact l0.
  apply H0.
  unfold E.EnvironmentLattice.gt, E.EnvironmentLattice.lt in |- *.
  intuition.
  apply isPreFixpointForward.
  exact iterated.
Qed.
```

```
Lemma isLeastFixpointGreaterThanInitialEnvComputeData :
      E.EnvironmentLattice.eq E.d (E.Forward E.diagram computeData) computeData
   /\ forall fixe,   E.EnvironmentLattice.PreOrder.le E.d E.initialEnv fixe
                  -> E.EnvironmentLattice.eq E.d fixe (E.Forward E.diagram fixe)
                  -> E.EnvironmentLattice.PreOrder.le E.d computeData fixe.
Proof.
  intuition.
  unfold E.EnvironmentLattice.eq in |- *.
  split. unfold computeData.
  eapply isPostFixpointTool_rec.
  unfold computeData.
  apply isPreFixpointForward.
  apply isIterateTool_rec.
  unfold computeData.
  apply isLessThanFixpointTool_rec.
  exact H.
  exact H0.
Qed.

End ElementaryTool.
```

FIG. 5.15 – *Définition d'un outil élémentaire en* COQ

5.5 CONCLUSION ET DISCUSSION

Nous avons défini, dans ce chapitre, un cadre formel général qui a pour but de guider le développement des outils élémentaires du générateur de code qui exploitent l'analyse statique. Mais il peut, plus généralement, être utilisé dans d'autres analyses statiques manipulant des types dépendants (familles de treillis) dans leurs calculs. Ce cadre formel est inspiré de celui de l'analyse statique par interprétation abstraite. Il définit le minimum d'éléments qui répond à nos objectifs. Notons qu'un point important lors de la spécification du cadre général est de bien représenter des familles de treillis de sorte que les preuves effectuées ne compliquent pas le développement des outils élémentaires. Par exemple, l'analyse à effectuer par les outils élémentaires est tributaire de la taille du circuit lu en entrée du générateur de code, le typage d'un circuit doit être paramétré par le nombre de ports de chacun des blocs. Autrement dit, l'initialisation des familles de treillis doit se faire à la lecture des circuits dans le générateur de code et ces valeurs ne peuvent pas être fixées préalablement.

La thèse de D. Pichardie [Pic05] représente une formalisation complète en COQ de l'interprétation abstraite basée sur les travaux de P. Cousot [CC77, Cou81, CC92]. Dans [Pic05], les analyses qui reposent sur des types dépendants doivent être paramétrées par une borne fixée au départ avant d'analyser tout programme. Cette solution est justifiée car les adresses manipulées sont fixées à 32 bits. La différence avec notre cadre d'étude est que les circuits analysés peuvent aller du très simple au très complexe, et, par conséquent, aucune borne sur leur taille n'est spécifiée à l'avance.

Nous n'avons pas étudié les différentes variantes possibles de spécification et de preuve en COQ, afin d'économiser le temps nécessaire au développement. Cela n'était pas l'objet des travaux présentés. Notre but est, d'une part, de montrer l'intégration d'un outil de développement formel comme COQ dans le cadre d'applications industrielles critiques et, d'autre part, d'étudier la qualification d'un outil développé formellement.

Dans ce qui suit, nous détaillerons les outils élémentaires, développés dans le cadre de cette thèse, qui ont été intégrés dans GENEAUTO. Ainsi, nous mettrons en évidence l'instanciation du cadre et le gain obtenu par la factorisation qu'il permet.

Troisième partie

Étude de cas

6 Ordonnancement des circuits Simulink

6.1 Motivation

Ce chapitre présente l'ordonnancement des blocs des circuits Simulink. Les circuits traités par le générateur de code présentent des possibilités d'exécution concurrente tandis que le code généré doit être séquentiel. Il est donc nécessaire de calculer un ordre d'exécution unique pour chaque bloc du circuit d'entrée. Cet ordre correspond à la position du code associé au bloc dans la séquence d'instructions générée par GeneAuto. Par exemple, la figure 6.2 contient le code généré pour le circuit Simulink présenté dans la figure 6.1. Rappelons que le circuit *moyenne 5*, présenté précédemment dans la section 3.2.1, calcule la moyenne, sur une fenêtre de taille 5, des données transmises en séquence sur une entrée unique. Le circuit exploite pour cela 4 blocs `Unit Delay` pour mémoriser les valeurs précédentes et constituer la fenêtre.

Par ailleurs, les partenaires industriels du projet GeneAuto ont indiqué que la vérification de l'outil élémentaire d'ordonnancement par des tests classiques est complexe, car cela nécessite plusieurs jeux de tests pour couvrir toutes les exigences. En effet, la nature de certains blocs influe sur l'ordonnancement, et il y a un grand nombre de combinaisons de blocs possibles à tester. Ce qui implique un coût élevé en termes d'effort investi. De plus, l'ordonnancement est souvent une propriété globale non compositionnelle. Il est donc difficile de couvrir des scénarios complexes comme l'assemblage des scénarios élémentaires.

Afin de produire un ordre d'exécution pour chaque bloc, l'ordonnancement proposé doit respecter les exigences de l'outil élémentaire «ordonnanceur» qui seront présentées dans ce chapitre.

Nous proposons, dans un premier temps, un algorithme *classique* qui calcule un rang entier pour les blocs des circuits purement flots de données et, dans un second temps, un algorithme qui traite des circuits complets, combinant des flots de données et de contrôle, et qui est basé sur un calcul des dépendances entre les blocs.

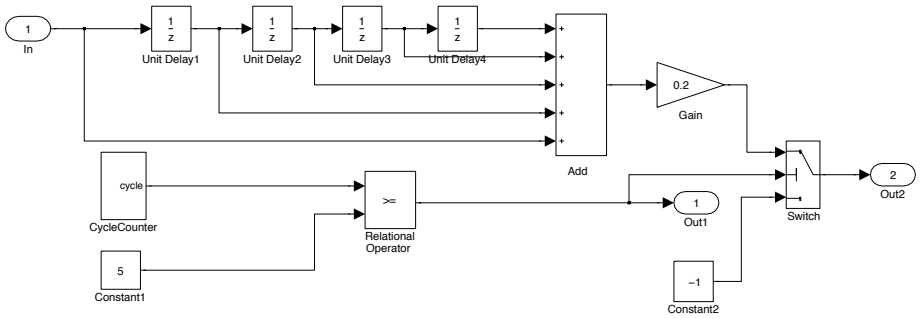


FIG. 6.1 – *Circuit* SIMULINK *représentant la moyenne 5*

```
#include "mmoyenne5.h"

void Moyenne5_init(_C_Moyenne5_ *_C_)
{
        /* Code of block CycleCounter (Subsystem) */
        CycleCounter_init(&(_C_->_CycleCounter_));
        _C_->_Unit_Delay1.Memory = 3.14;
        _C_->_Unit_Delay2.Memory = 0;
        ...
}

void Moyenne5_compute(_C_Moyenne5_ *_C_)
{
        REAL V_unnamed_1;
        BOOL V_unnamed_2;
        ...
        /* Code of block Input (Inport) */
        V_unnamed_1 = _C_->I1;
        /* Code of block Unit Delay1_get (UnitDelay) */
        V_unnamed_2 = _C_->_Unit_Delay1.Memory;
        /* Code of block Unit Delay2_get (UnitDelay) */
        V_unnamed_3 = _C_->_Unit_Delay2.Memory;
        V_unnamed_4 = _C_->_Unit_Delay3.Memory;
        /* Code of block Unit Delay4_get (UnitDelay) */
        V_unnamed_5 = _C_->_Unit_Delay4.Memory;
        /* Code of block Sum1 (Sum) */
                V_unnamed_6 =  V_unnamed_5
                             + V_unnamed_4
                             + V_unnamed_3
                             + V_unnamed_2
                             + V_unnamed_1;
        ...
}
```

FIG. 6.2 – *Code C généré pour le circuit moyenne 5*

6.2 RAPPEL DES EXIGENCES SUR L'ORDONNANCEMENT DES CIRCUITS

Plusieurs contraintes doivent être prises en compte pour le séquencement des blocs d'un circuit donné. D'une part, certaines contraintes peuvent être liées à la structure du circuit (dites contraintes structurelles) : la nature des signaux (c'est-à-dire signaux de données ou de contrôle) et des blocs (c'est-à-dire blocs combinatoires ou séquentiels). D'autre part, des contraintes peuvent être imposées par l'utilisateur au moment de la conception du circuit. Tel est le cas des priorités indiquées explicitement par l'utilisateur ou déduites implicitement à partir de la représentation graphique du circuit comme la position des blocs.

En effet, s'il n'est pas possible de déduire un ordre unique pour chaque bloc en s'appuyant uniquement sur les contraintes structurelles du circuit, les prio-

rités des blocs interviennent pour déterminer celui-ci. Les priorités (explicite et implicite) sont exploitées par les utilisateurs pour contrôler l'ordre d'exécution et les effets de bord des blocs SIMULINK en lecture/écriture mémoire. Notons que les blocs atomiques d'un circuit aux flots de données ne devraient pas faire d'effets de bord. Mais, certains utilisateurs exploitent quand même des blocs atomiques d'écriture en mémoire ou sur des périphériques externes. L'ordonnanceur doit donc respecter strictement les exigences pour produire le même ordonnancement que l'outil utilisé pour la simulation.

Pour un bloc b donné, nous noterons dans le reste du manuscrit $b.In$ l'ensemble des blocs qui produisent les entrées du bloc b, et $b.Out$ l'ensemble des blocs dont le bloc b produit les entrées. Les blocs $b.In$ et $b.Out$ sont reliés par des flots de données avec le bloc b. En ce qui concerne les flots de contrôle, nous noterons $b.Callers$ l'ensemble des blocs qui contrôlent le bloc b et $b.Callees$ l'ensemble des blocs que le bloc b contrôle. `dataSources`, `controlSources`, `dataTargets` et `controlTargets` sont les fonctions qui calculent respectivement $b.In$, $b.Callers$, $b.Out$ et $b.Callees$.

```
Definition dataSources (diagram :ModelType) (blockIndex : nat):=
  dataSourceFor (dataSignals ModelElementType diagram) blockIndex.

Definition controlSources (diagram : ModelType) (blockIndex : nat) :=
  controlSourceFor (controlSignals ModelElementType diagram) blockIndex.

Definition dataTargets (diagram : ModelType) (blockIndex : nat) :=
  dataTargetFor (controlSignals ModelElementType diagram) blockIndex.

Definition controlTargets (diagram : ModelType) (blockIndex : nat) :=
  controlTargetFor (controlSignals ModelElementType diagram) blockIndex.
```

Dans ce script, la fonction `dataSourceFor` a été décrite dans la section 3.6.1.3. Les autres fonctions sont semblables.

Les exigences sont résumées par les points suivants :

S1/R1 Ordonnancement des blocs libres (qui ne sont pas contrôlés) : seuls les blocs qui ne sont pas contrôlés doivent être ordonnancés ;

S1/R2 Respect de la causalité induite par les signaux de données : un bloc b ne peut être exécuté que si tous les blocs $b.In$, qui alimentent les signaux de données connectés à ses entrées, ont déjà été exécutés dans le même cycle. Autrement dit, les ordres d'exécution des blocs de $b.In$ sont strictement inférieurs à celui de b ;

S1/R3 Traitement des blocs séquentiels : la partie écriture des blocs séquentiels peut toujours être exécutée en dernier, l'ordre d'exécution considéré pour un bloc séquentiel est celui de la partie lecture ;

S2/R4 Respect de la causalité induite par les signaux de contrôle : un bloc contrôlé est exécuté pendant l'exécution du bloc qui le contrôle. Cette exigence combinée avec la causalité des données, il en résulte une exigence dérivée qu'un bloc contrôlant b ne peut être exécuté que si les blocs qui alimentent les entrées des blocs qu'il contrôle $b.Callees$ ont déjà été exécutés dans le même cycle. Cette exigence s'applique de manière transitive le long d'une chaîne de signaux de contrôle ;

S1/R5 Prise en compte de la priorité explicite des blocs : si les exigences précédentes ne permettent pas d'ordonnancer deux blocs, ceux-ci doivent être ordonnés selon leurs priorités explicites respectives si celles-ci ont été renseignées par l'utilisateur dans le circuit. L'absence de priorité explicite correspond à la priorité la plus faible ;

S1/R6 Prise en compte de la priorité implicite (position graphique) des blocs : si les exigences précédentes ne permettent pas d'ordonner deux blocs, ceux-ci doivent être ordonnés selon leur priorité implicite dans le circuit. Il s'agit d'une relation d'ordre total qui combine lexicographiquement l'abscisse et l'ordonnée des blocs (de gauche à droite puis de haut en bas) ;

S3/R7 Indépendance des autres critères : l'ordonnancement réalisé doit être indépendant de toutes autres caractéristiques des éléments du diagramme. Par exemple, il ne doit pas dépendre de l'ordre de rangement des éléments dans les fichiers qui contiennent les circuits. Cette exigence utilisateur se raffine en l'exigence d'outils suivante : la relation d'ordre qui permet d'ordonnancer les blocs en s'appuyant sur les exigences précédentes doit être un ordre total.

La spécification de l'ordonnancement des blocs d'un circuit donné a été une tâche complexe. Elle a nécessité le raffinement du cahier des charges de l'ordonnanceur défini dans le cadre du projet, afin de préciser les exigences relatives à l'ordonnancement. Ce raffinement s'est avéré nécessaire, car les exigences de l'ordonnanceur étaient incomplètes et imprécises. Il aurait été difficile de le détecter sans l'usage des méthodes formelles lors de l'analyse des besoins. Ces exigences avaient été validées par la relecture indépendante croisée des différents partenaires industriels du projet GENEAUTO. Par exemple, il était complexe de formaliser les exigences concernant l'ordonnancement d'une succession en cascade de sous-systèmes contrôlés. De plus, la sémantique du langage SIMULINK n'est pas définie de manière complète, notamment en ce qui concerne les flots de contrôle, et il n'est pas précisé, exactement et sans ambiguïtés dans la documentation associée à SIMULINK, comment l'outil procède à l'ordonnancement des blocs. Ainsi, aucune règle n'existe décrivant les combinaisons illégales. Pour mettre en place des règles, il n'est possible que de procéder par généralisation (ou déduction) à travers la liste des exemples erronés présentés dans l'aide de l'outil SIMULINK.

L'ordonnancement des circuits est plus complexe que ce que la documentation de SIMULINK ne le laisse penser. Il nécessite de prendre en compte la structure globale du circuit en suivant les flots de données mais aussi les flots de contrôle. L'ordonnancement exploite des informations provenant des sous-systèmes contrôlés et de leurs entrées. Il dépend donc de la nature des blocs (combinatoire ou séquentiel) et de la nature des flots les reliant (flots de données ou de contrôle).

6.3 ORDONNANCEMENT DES CIRCUITS AUX FLOTS DE DONNÉES

Dans cette partie, nous allons présenter un algorithme simple qui ordonnance les blocs d'un circuit purement flots de données, c'est-à-dire un circuit constitué de blocs reliés entre eux uniquement par des signaux de données. Il s'agit de calculer pour chaque bloc du circuit aux flots de données son ordre d'exécution à un cycle donné. L'ordonnancement dans ce cas reste simple dans la mesure où le calcul n'est guidé que par les signaux de données reliant les entrées des blocs du circuit en question.

6.3.1 Ordonnancement guidé par les rangs

Une solution intuitive pour calculer l'ordre d'exécution consiste à construire les rangs des blocs constituant le circuit d'entrée du générateur de code. Le rang d'un bloc représente la profondeur de celui-ci dans la séquence des blocs produisant ses entrées, en partant des blocs sources du circuit et des parties lecture des blocs séquentiels. Cette solution produit un rang d'exécution, de type entier, pour chaque bloc du circuit que l'on souhaite ordonnancer. Il s'agit d'une approche classique dans les langages de modélisation à base de flots de données.

6.3.1.1 Calcul des rangs

Nous avons proposé dans [ITPS08] une version préliminaire de l'ordonnanceur basée sur le calcul des rangs pour les circuits aux flots de données. Le principe d'ordonnancement des blocs est donné par l'algorithme 1.

Algorithme 1: Algorithme de calcul des rangs pour les circuits flots de données

Entrées: Un circuit c comptant s blocs atomiques.
Sorties: Chaque bloc atomique b du circuit c doit avoir un rang, noté $rank(b)$.

```
pour tout bloc b du circuit c ayant un identifiant i ∈ [0..s[ faire
    Le rang de chaque bloc b est initialement indéfini (⊥)
    rank(i) ← ⊥ ;
fin pour
tantque le point fixe n'est pas atteint faire
    pour tout bloc b du circuit c ayant un identifiant i ∈ [0..s[ faire
        si (le bloc b est un bloc séquentiel) ∨ (b.In = ∅) alors
            rank(b) ← 0 ;
        sinon
            rank(b) ← succ(max({0} ∪ {rank(b') | b' ∈ b.In}))
        finsi
    fin pour
fin tantque
```

6.3.1.2 Calcul des rangs

Nous appliquons le cadre général, que nous avons présenté précédemment dans le chapitre 6, à l'ordonnanceur. Ce dernier calcule les rangs des blocs composant les circuits d'entrée du générateur de code.

6.3.1.2.1 Domaine de calcul

L'information calculée et propagée à travers le circuit est de type entier. Elle est spécifiée par le type inductif `rank`.

```
Inductive rank : Set :=
  | Top : rank
  | Val : nat -> rank
  | Bot : rank.
```

Le domaine abstrait Δ correspond au type `rank`.

```
Definition type := rank.
```

Un bloc peut avoir un rang défini, représenté par le constructeur `Val` paramétré par un entier ou ne pas avoir de rang (constructeurs `Top` et `Bot`). Le constructeur `Bot` correspond à un rang qui n'a pas été calculé ou qui n'a pas pu être calculé. Le constructeur `Top` est défini disposer d'une structure de treillis.

La relation d'ordre partiel $\preceq$ définie sur l'ensemble Δ est spécifiée par la définition `le` :

```
Definition le : Data.type -> type -> type -> Prop :=
 fun _ r1 r2 =>
  match r1, r2 with
   | Bot   , _      => True
   | Val n1, Val n2 => n1 = n2
   | _     , Top    => True
   | _     , _      => False
end.
```

Le domaine abstrait Δ muni de la relation d'ordre partiel $\preceq$ est un ensemble partiellement ordonné, noté $\langle \Delta, \preceq \rangle$. Les preuves que $\preceq$ est une relation d'ordre partiel s'appuient sur les propriétés Coq de la librairie des entiers *nat*. Nous n'allons pas développer ici ces preuves. En revanche, elles peuvent être consultées dans le fichier RankLattice.v.

6.3.1.2.2 Famille de treillis des rangs

L'ensemble partiellement ordonné $\langle \Delta, \preceq \rangle$ muni des éléments suivants est une famille de treillis, notée $\{\theta_d\}_{d \in unit} = \langle \Delta, \preceq, \bot, \top, \sqcup, \sqcap \rangle$. Le type du paramètre d est `unit`, donc d vaut `tt`, ce qui signifie que cette famille de treillis comporte un seul treillis. Cette famille de treillis est représentée par le diagramme de Hasse de la figure 6.3 :

- $\bot$, la borne inférieure de la famille de treillis $\{\theta_d\}_{d \in unit}$, qui vaut `Bot` ;
- $\top$, la borne supérieure de la famille de treillis $\{\theta_d\}_{d \in unit}$, qui vaut `Top` ;
- $\sqcup$, l'opérateur qui calcule la borne supérieure d'un couple d'éléments défini par :

```
Definition join  : PreOrder.Data.type -> PreOrder.type ->  PreOrder.type ->
      PreOrder.type :=
 fun _ r1 r2 =>
  match r1, r2 with
   | Bot   , _      => r2
   | _     , Bot    => r1
   | Val n1, Val n2 => if eq_nat_dec n1 n2 then r1 else Top
   | _     , _      => Top
end.
```

- $\sqcap$, l'opérateur qui calcule la borne inférieure d'un couple d'éléments défini par :

```
Definition meet : PreOrder.Data.type -> PreOrder.type ->  PreOrder.type ->
     PreOrder.type :=
 fun _ r1 r2 =>
  match r1, r2 with
   | Top   , _      => r2
   | _     , Top    => r1
   | Val n1, Val n2 => if eq_nat_dec n1 n2 then r1 else Bot
   | _     , _      => Bot
end.
```

L'implantation Coq de la famille de treillis des rangs est définie dans le fichier RankLattice.v.

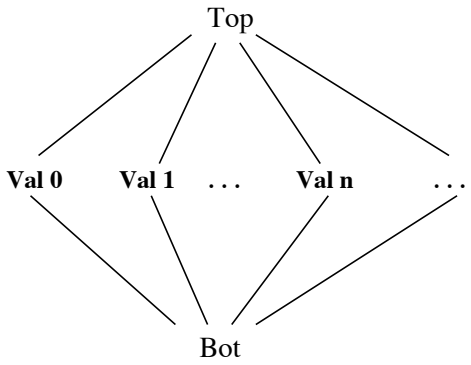


FIG. 6.3 – *Diagramme de Hasse de la famillle de treillis des rangs*

6.3.1.2.3 Environnement de calcul des rangs

L'environnement Y associe à chaque bloc un rang entier dans Δ. Le calcul des rangs ne dépend que de la validité des identifiants des blocs (appartenant à $[0..s[$), de la nature des blocs ainsi que de la taille du circuit analysé (nombre de blocs). La nature des blocs (séquentiels ou combinatoires) est exploitée dans la fonction de calcul des rangs. Par conséquent, l'environnement, tel qu'il est défini dans la section 5.3.4, est une famille de treillis paramétrée par un couple de type $\mathbb{N} \times (\mathbb{N} \rightarrow \texttt{unit})$. Le premier élément représente la taille du circuit, notée s, et le second est une fonction qui associe pour chaque identifiant de bloc la valeur `tt`. Ainsi, $\{\theta_{f(n)}\}_{f(n)\in\texttt{unit}}$ correspond à la famille de treillis des rangs définie dans la section 6.3.1.2.2.

Définition 6.3.1. *Environnement de calcul des rangs*

$$Y \triangleq \{\{n \in \mathbb{N} | n < s\} \rightarrow \{\theta_{f(n)}\}\}_{s\in\mathbb{N}, f\in\mathbb{N}\rightarrow\ \texttt{unit}}$$

où le couple (s, f) est le paramètre de l'environnement.

L'environnement des rangs Y est créé avec le script :

```
Module RankEnvironment := MakeDependentEnvironment(RankLattice).
```

avec `RankLattice` la définition en Coq de $\{\theta_d\}_{d\in\texttt{unit}}$, la famille de treillis (ici un seul treillis) des rangs .

Initialement, l'environnement $(Y)_0$ associe le rang $\perp$ par défaut à chaque bloc du circuit d'entrée.

6.3.1.2.4 Fonction de calcul des rangs

Elle consiste, dans un premier temps, à récupérer pour chaque bloc b les blocs produisant ses entrées ($b.In$). Un bloc peut disposer de plusieurs entrées, son rang doit donc être le successeur du plus grand rang de ses blocs d'entrée $b.In$. La figure 6.4 illustre un tel cas.

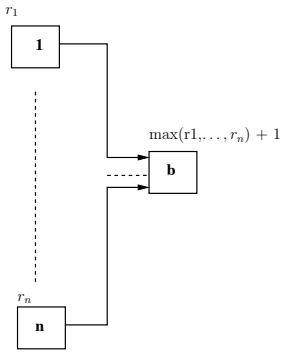


FIG. 6.4 – *Rang d'un bloc à entrées multiples*

L'étape suivante consiste à calculer le maximum des rangs de *b.In* à l'aide de la fonction `max_rank`. Elle utilise la fonction `max` de la bibliothèque `Max`.

```
Definition max_rank t p :=
 match t, p with
  |Bot   , _
  |_   , Bot      => Bot
  |Top   , _
  |_   , Top      => Top
  |Val n, Val m  => Val (max n m)
 end.
```

Le rang du bloc *b* sera donc le successeur du `max_rank` calculé avec la fonction `succ_rank` définie par :

```
Definition succ_rank (r : rank) :=
 match r with
  | Bot    => Bot
  | Val m => Val (S m)
  | Top    => Top
 end.
```

Le rang d'un bloc *b* est égal au successeur du maximum des rangs des blocs produisant ses entrées (*b.In*) selon l'exigence S1/R2. Dans le cas où le bloc en question n'a pas d'entrée, son rang vaut `Val 0`. Un cas particulier concerne les blocs séquentiels : comme mentionné précédemment (exigence S1/R3), la partie lecture d'un bloc séquentiel doit être exécutée avant tous les blocs reliés à sa sortie. Ainsi le plus petit rang (c'est-à-dire `Val 0`) lui est associé. En effet, dans l'ordonnancement final, c'est l'ordre de la partie lecture du bloc séquentiel qui est considéré.

```
Definition out_rank (E : RankEnvironment.PreOrder.type) (d : ModelType) (m : nat)
     :=
match lookForBlock m d with
|Some b => if dec_block_op b Delay then Val 0 else E m
|_       => Bot
end.

Definition max_out_rank (E : RankEnvironment.PreOrder.type) (d : ModelType) (l :
     list nat) :=
 fold_right (fun t q => max_rank (out_rank E d t) q) (Val 0) l.
```

La fonction `lookForBlock` vérifie si l'identifiant du bloc donné est valide.

La fonction `max_out_rank` calcule, pour un bloc *b*, le maximum des rangs des blocs *b.In*.

Les rangs calculés sont propagés à travers la structure du circuit avec la fonction de propagation ρ décrite par `Forward`. Le code suivant représente son implantation en COQ.

```
Definition Forward (d : ModelType) (E : RankEnvironment.PreOrder.type) :
    RankEnvironment.PreOrder.type :=
fun n     =>
  if le_lt_dec n ((size d)-1)
  then
    match lookForBlock n d with
      | Some (bl)  => succ_rank (max_out_rank E d (dataSourceFor d n))
      | error      => Bot
      end
  else Bot.
```

Notons dans la suite $(\rho(e))(b)$ le rang du bloc b obtenu par la fonction ρ à partir d'un environnement e. Le rang de tous les blocs du circuit est obtenu par le calcul du point fixe $lfp(\rho)$. Ce dernier est calculé par la fonction récursive `tool_rec` présentée dans la section 5.3.6.

6.3.1.3 Calcul d'un ordre total

Le rang des blocs, obtenu par le calcul de point fixe, est un préordre total : réflexif, transitif et total. Cela est dû au fait que les rangs sont des entiers qui peuvent toujours être comparés. La relation issue des rangs n'est pas antisymétrique car deux blocs indépendants peuvent avoir le même rang. Cette relation est différente de celle qui est utilisée dans la famille de treillis des rangs.

Les blocs ayant le même rang sont concurrents et peuvent être exécutés en parallèle. Or, le code généré doit être un code séquentiel où les blocs du circuit considéré doivent être transformés en une suite séquentielle d'instructions. Donc, chaque bloc doit avoir un ordre d'exécution unique et, par conséquent, l'ordre calculé doit être total.

Par conséquent, des exigences additionnelles s'ajoutent à l'ordonnancement proposé afin d'obtenir un ordre unique pour tous les blocs du circuit. Il s'agit des exigences liées aux priorités des blocs (voir les exigences S1/R5 et S1/R6). Si les rangs calculés par l'algorithme ne permettent pas de déduire un ordre total, l'ordre lexicographique des priorités (explicites et implicites) des blocs permet de fournir un ordre d'exécution unique aux blocs du circuit (donc un ordre total).

Les blocs du circuit sont alors triés comme suit :

1. Les blocs sont d'abord triés suivant les rangs calculés ;
2. S'il y a des blocs de rangs conflictuels (c'est-à-dire deux blocs avec le même rang calculé), l'ordre lexicographique des priorités (explicite et implicite) est considéré.

   Rappelons qu'une priorité explicite est définie dans `option nat` car certains blocs peuvent ne pas en posséder une. Nous définissons donc un ordre total sur le type `option nat` (voir le module `PriorityTotalPreOrder` dans Domain.v).

   La priorité implicite est représentée par un entier naturel. Un ordre total sur les priorités implicites est donc défini sur $\mathbb{N}$ (voir le module `NaturalTotalPreOrder` dans Domain.v.

   Un ordre lexicographique total des priorités est ensuite construit à l'aide des deux ordres totaux (voir section 5.3.2.5 et le module `TotalPreOrderLexicographicProduct` dans Domain.v).

   Nous avons choisi de représenter la plus haute priorité (explicite ou implicite) d'un bloc par la plus petite valeur du domaine de priorité : `Some 0` (respectivement l'entier 0) est la priorité explicite (respectivement implicite) la plus élevée. Ainsi, l'ordre lexicographique total des priorités

permet d'ordonner ces dernières par ordre décroissant, du plus prioritaire au moins prioritaire.

Ces étapes correspondent à l'ordre lexicographique construit à partir des rangs et de l'ordre lexicographique des priorités. Ainsi, l'ordre obtenu correspond à l'ordre d'exécution des blocs.

6.3.1.4 Exemple

Reprenons l'exemple «moyenne 5» auquel nous appliquons l'algorithme 1, les rangs obtenus sont indiqués au dessus des blocs dans la figure 6.5.

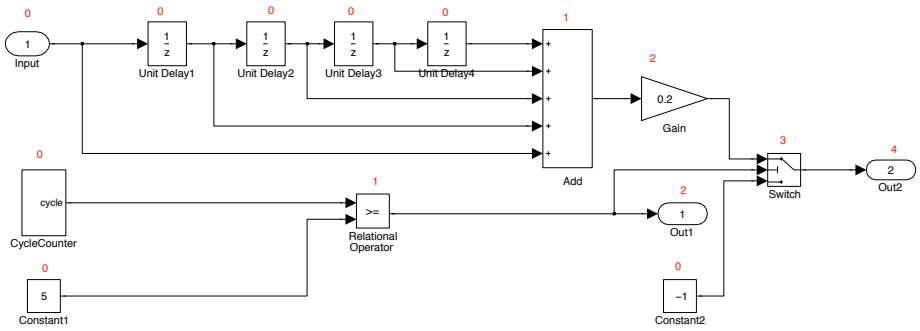


FIG. 6.5 – *Rangs des blocs du circuit «moyenne 5»*

Notons que tous les blocs sans entrées ainsi que les blocs séquentiels ont le rang `Val 0`. Supposons que tous les blocs ont la même priorité explicite et seule la priorité implicite est déduite du circuit (la position graphique des blocs). Ainsi, la composition lexicographique de ce préordre partiel avec les priorités permet d'obtenir un ordre total pour le circuit «moyenne 5» tel qu'il est indiqué dans la figure 6.6. Comme tous les blocs de cet exemple possèdent la même priorité explicite, la priorité implicite est celle qui permet d'ordonnancer les blocs ayant le même rang. Les blocs les plus en haut à gauche dans le circuit sont les plus prioritaires (voir exigence S1/R6).

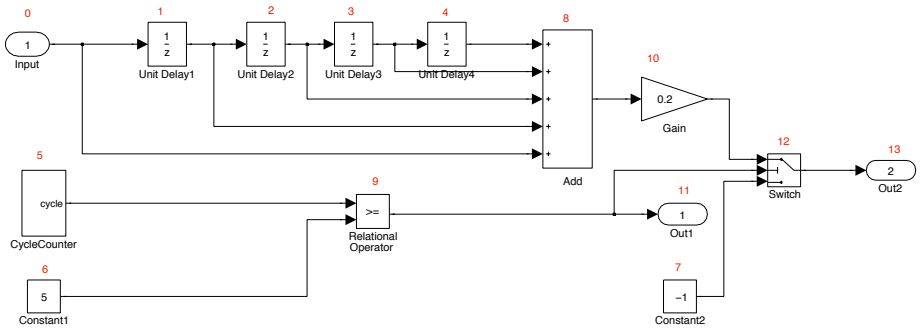


FIG. 6.6 – *Ordre d'exécution des blocs du circuit moyenne 5*

6.3.1.5 Correction de l'algorithme d'ordonnancement

Dans un premier temps, nous vérifions que l'environnement initial Y_0 est un pré-point fixe.

Propriété 6.3.1. *L'environnement initial est un pré-point fixe*

$$\bot_{\mathrm{Y}} \preceq_{\mathrm{Y}} \rho(\bot_{\mathrm{Y}})$$

Il s'agit de vérifier la propriété `initialEnvIsPreFixpoint` présentée dans la figure 5.14.

Preuve. La preuve est immédiate en exploitant la propriété `bot_least` de l'environnement Y.

Parmi les preuves qui doivent être réalisées pour assurer la correction de l'ordonnanceur, nous citons celles qui sont liées à l'algorithme à savoir la croissance de la fonction de propagation ρ ainsi que les propriétés liées à la correction du calcul des rangs.

6.3.1.5.1 Croissance de la fonction de propagation

La croissance assure que chaque itération de la fonction de propagation ρ, à partir de l'environnement initial $(\mathrm{Y})_0$, calcule un nouvel environnement plus grand, jusqu'à ce que le point fixe $lfp(\rho)$ soit atteint. Afin de simplifier la preuve, nous prouvons des propriétés auxiliaires. Il s'agit de démontrer que les fonctions impliquées dans le calcul induit par ρ sont croissantes.

Théorème 6.3.2. *Croissance de la fonction* `succ_rank` *définie sur les entiers*
$\forall\, r_1\, r_2 : \Delta, r_1 \preceq r_2 \Rightarrow$ `succ_rank`$(r_1) \preceq$ `succ_rank`(r_2)

Preuve. La preuve se fait par induction sur les valeurs de r_1 et de r_2.
Cas 1 : $r_1 = \bot$. Cela signifie que `succ_rank`$(\bot) \preceq$ `succ_rank`(r_2). Ceci est vrai $\forall\, r_2 \in \Delta$.
Cas 2 : $r_1 = \top$. Supposons que $r_1 \preceq r_2$. Cela signifie que $r_2 = \top$. Ainsi, `succ_rank`$(\top) \preceq$ `succ_rank`$(\top)$.
Cas 3 : $r_1 = Val\ n_1$. Supposons que $r_1 \preceq r_2$. Cela signifie, selon la définition de $\preceq$, que $(r_2 = Val\ n_1 \vee r_2 = \top)$. Nous avons donc `succ_rank`$(r_2) =$ `succ_rank`$(r_1) \vee$ `succ_rank`$(r_2) = \top$. Ainsi, `succ_rank`$(r_1) \preceq$ `succ_rank`(r_2).

Théorème 6.3.3. *Croissance de la fonction* `max_rank`
$\forall\, r_1\, r_2\, r_3\, r_4 : \Delta,\ r_1 \preceq r_2 \Rightarrow\ r_3 \preceq r_4 \Rightarrow$ `max_rank`$(r_1, r_3) \preceq$ `max_rank`(r_2, r_4)

Preuve. La preuve se fait également par induction sur les valeurs des paramètres. Pour davantage de détails, se reporter au fichier RankSequencer.v.

Considérons dans ce qui suit un circuit c. Afin de simplifier la présentation des preuves, nous notons par $rank_e(b)$, pour un environnement e et un bloc b, le rang du bloc b calculé par (`out_rank e d b`). Notons également par $max_e(l)$ le rang maximum de la liste des blocs l, calculé par (`max_out_rank e d l`).

Théorème 6.3.4. *Croissance de la fonction* `out_rank`

$$\forall b \in [0..s[,\ \forall e_1\, e_2 : \mathrm{Y},\ e_1 \preceq_{\mathrm{Y}} e_2 \Rightarrow rank_{e_1}(b) \preceq rank_{e_2}(b)$$

Preuve. La preuve ne pose aucune difficulté. Elle peut être consultée dans le fichier RankSequencer.v.

Théorème 6.3.5. *Croissance de la fonction* `max_out_rank`
Soit l une liste de blocs dont les identifiants sont dans $[0..s[$.

$$\forall e_1\, e_2 : \mathrm{Y},\ e_1 \preceq_{\mathrm{Y}} e_2 \ \Rightarrow max_{e_1}(l) \preceq max_{e_2}(l)$$

Preuve. Nous raisonnons par induction sur la liste l.
Cas 1 : La liste l est vide. Il en résulte $max_{e_1}(l) = max_{e_2}(l) =$ `Val 0`. Par conséquent, $max_{e_1}(l) \preceq max_{e_2}(l)$.
Cas 2 : La liste $l = t :: q$ avec t un entier. Supposons $e_1 \preceq_{\mathrm{Y}} e_2$. Nous avons alors :

$$\begin{cases} max_{e_1}(l) = \texttt{max\_rank}(rank_{e_1}(t), max_{e_1}(q)) \\ max_{e_2}(l) = \texttt{max\_rank}(rank_{e_2}(t), max_{e_2}(q)) \end{cases}$$

Nous disposons, selon le théorème 6.3.4, de $rank_{e_1}(t) \preceq rank_{e_2}(t)$. Par hypothèse d'induction, $max_{e_1}(q) \preceq max_{e_2}(q)$. Par conséquent, $\texttt{max\_rank}(rank_{e_1}(t), max_{e_1}(q)) \preceq \texttt{max\_rank}(rank_{e_2}(t), max_{e_2}(q))$ en appliquant la croissance de `max_rank`. Ainsi, $max_{e_1}(l) \preceq max_{e_2}(l)$.

Théorème 6.3.6. *Croissance de la fonction* ρ

$$\forall b \in [0..s[,\ \forall\, e_1\, e_2 : \mathrm{Y},\ e_1 \preceq_{\mathrm{Y}} e_2 \ \Rightarrow \rho(e_1) \preceq_{\mathrm{Y}} \rho(e_2)$$

Preuve. Soient e_1 et e_2 deux environnements tels que $e_1 \preceq_{\mathrm{Y}} e_2$. En appliquant la définition de la relation d'ordre $\preceq_{\mathrm{Y}}$ et de la fonction ρ, il faut démontrer la propriété :

$$\forall b \in [0..s[,\ \texttt{succ\_rank}(max_{e_1}(b.In)) \preceq \texttt{succ\_rank}(max_{e_2}(b.In))$$

Cas 1 : L'identifiant du bloc b n'est pas valide ($b \notin [0..s[$). Donc, $\texttt{succ\_rank}(max_{e_1}(b.In)) = \texttt{succ\_rank}(max_{e_2}(b.In)) = \bot$.
Nous avons par réflexivité de la relation $\preceq$,
$\texttt{succ\_rank}(max_{e_1}(b.In)) \preceq \texttt{succ\_rank}(max_{e_2}(b.In))$.
Cas 2 : $b < s$. Supposons que pour tout bloc b, les identifiants des blocs d'entrée $b.In$, quand ils existent, sont valides. Nous savons que les fonctions `succ_rank` et `max_out_rank` sont monotones (voir les théorèmes 6.3.2 et 6.3.5).
Ainsi, en appliquant le théorème 6.3.5, nous avons $max_{e_1}(b.In) \preceq max_{e_2}(b.In)$.
En appliquant le théorème 6.3.2, il en résulte
$\texttt{succ\_rank}(max_{e_1}(b.In)) \preceq \texttt{succ\_rank}(max_{e_2}(b.In))$.

6.3.1.5.2 Correction du calcul des rangs par rapport à la structure du circuit

Nous devons démontrer que l'algorithme proposé fournit des ordres d'exécution corrects par rapport au flot de donnée. Autrement dit, si un bloc A est relié à l'entrée du bloc B par un signal de donnée, le rang calculé pour B est inférieur au rang de A.

Théorème 6.3.7. *Correction du rang calculé*
$\forall\, b_1\, b_2 \in [0..s[,$

$$b_1 \in b_2.In \ \wedge\ \neg(isSequential(b_2)) \ \Rightarrow\ (lfp(\rho))(b_1) \preceq (lfp(\rho))(b_2)$$

Preuve. Soit le bloc b_1 tel que $b_1 \in b_2.In$ et $A = \{b_2.In\} \setminus \{b_1\}$. Par définition, nous avons $(lfp(\rho))(b_2) = (\rho(lfp(\rho)))(b_2)$. En appliquant la définition de la fonction ρ, il en résulte $(lfp(\rho))(b_2) = \texttt{succ\_rank}(max_{lfp(\rho)}(b_2.In))$. De plus, $max_{lfp(\rho)}(b_2.In) = \texttt{max\_rank}(((lfp(\rho)))(b_1), max_{lfp(\rho)}(A))$.
Cas 1 : $max_{lfp(\rho)}(b_2.In) = (lfp(\rho))(b_1)$.
Ainsi, $(lfp(\rho))(b_2) = \texttt{succ\_rank}((lfp(\rho))(b_1))$. Par conséquent, il en résulte $(lfp(\rho))(b_1) \preceq (lfp(\rho))(b_2)$.
Cas 2 : $max_{lfp(\rho)}(b_2.In) = max_{lfp(\rho)}(A)$.
Ce qui signifie $(lfp(\rho))(b_1) \preceq max_{lfp(\rho)}(A)$. Nous en déduisons alors $(lfp(\rho))(b_1) \preceq (lfp(\rho))(b_2)$.

6.3.1.5.3 Boucles algébriques

Les circuits SIMULINK présentant des boucles algébriques sont rejetés par SIMULINK. En effet, celles-ci représentent une erreur de conception du circuit qui ne peut pas être exécuté en temps discret. Cette erreur (problème de causalité des données) est actuellement détectée lors de la phase de pré-traitement et le processus de génération de code s'arrête à ce niveau. Nous avons choisi, dans le développement de l'ordonnanceur, de traiter tous les circuits et de proposer un algorithme général permettant de détecter les boucles algébriques.

Dans le cadre du calcul des rangs, nous illustrons dans ce qui suit comment le calcul des rangs permet de détecter les boucles algébriques dans des circuits aux flots de données. Soit l'exemple illustré dans la figure 6.7 présentant une boucle algébrique.

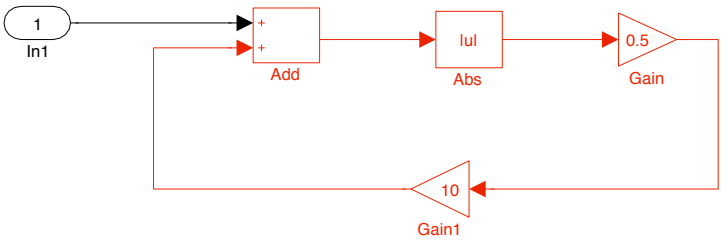


FIG. 6.7 – *Circuit à boucle algébrique*

Dans ce circuit, une boucle algébrique est présente et est composée des blocs `Add`, `Abs`, `Gain` et `Gain1`. En appliquant l'algorithme 1, le bloc `In1` n'ayant pas d'entrées, est exécuté en premier avec le rang `Val 0`. L'exécution du bloc `Add` nécessite l'exécution préalable du bloc `In1` et du bloc `Gain1` qui requiert à son tour l'exécution préalable du bloc `Add`. Ainsi, l'exécution du bloc `Add` demande qu'il soit lui-même préalablement exécuté. Le calcul du point fixe de ρ s'arrête aussitôt, et au bout d'une seule itération, pour donner les rangs suivants :

$$Y = \begin{cases} In1 & \mapsto & 0 \\ Add & \mapsto & \bot \\ Abs & \mapsto & \bot \\ Gain & \mapsto & \bot \\ Gain1 & \mapsto & \bot \end{cases}$$

L'association de la valeur $\bot$ à un bloc signifie que celui-ci fait partie d'une boucle algébrique.

6.3.1.6 Discussion

L'algorithme de calcul des rangs est simple à mettre en oeuvre. Il a été conçu pour les circuits ne comportant que des flots de données. Cependant, il présente quelques limitations illustrées par le circuit de la figure 6.1. Le calcul des rangs des blocs de ce circuit permet de déduire la relation de préordre total suivante entre les blocs du circuit. Nous nous contentons de citer les contraintes non triviales (qui ne sont pas déductibles directement de la structure du circuit) :

$$\begin{cases} \bullet\ \Upsilon(Add) \preceq \Upsilon(Out1) \\ \bullet\ \Upsilon(Input) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Unit\ Delay1) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Unit\ Delay2) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Unit\ Delay3) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Unit\ Delay4) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Cycle\ Counter) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Constant1) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Constant2) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Constant2) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Relational\ Operator) \preceq \Upsilon(Gain) \end{cases}$$

Le calcul obtenu exige, par exemple, que le bloc `Relational Operator` soit exécuté avant le bloc `Gain` alors qu'il n'y a aucune liaison entre ces blocs. De même, si le bloc `Relational Operator` était plus prioritaire que le bloc `Unit Delay1`, il devrait être exécuté en premier, ce qui n'est pas le cas selon l'ordre calculé. Cette limitation est due à l'utilisation d'entiers pour représenter les rangs. Ceci impose un préordre initial total sur les blocs du circuit qui est plus fort que l'ordre partiel dérivé des flots de données.

6.3.2 Ordonnancement guidé par les dépendances

L'utilisation de rangs entiers introduit un préordre total ne permettant pas une prise en compte satisfaisante des contraintes additionnelles telles que les priorités. Seuls les blocs ayant exactement le même rang peuvent être considérés comme concurrents, et donc ordonnancés selon leurs priorités respectives.

Nous proposons, pour pallier cette limitation, de calculer pour chaque bloc ses dépendances. Les dépendances expriment naturellement ce qui doit être exécuté pour qu'un bloc produise ses sorties. Les dépendances sont formalisées par la définition suivante.

Définition 6.3.2. *Dépendances d'un bloc b*

$$\mathcal{D}(b) \triangleq \bigcup_{x \in b.In} \mathcal{D}(x) \cup \{b\} \quad \textit{si b est combinatoire} \tag{6.1}$$

$$\mathcal{D}(b) \triangleq \{b\} \quad \textit{si b est séquentiel (il s'agit de la partie lecture de b)} \tag{6.2}$$

L'ensemble des dépendances d'un bloc combinatoire correspond à l'ensemble des blocs produisant ses entrées, ainsi que le bloc lui-même qui doit s'exécuter pour produire sa sortie (cf. équation 6.1). Tandis que l'exécution d'un bloc séquentiel ne dépend de rien d'autre que la lecture en mémoire et donc l'exécution de la partie lecture du bloc (équation 6.2). En effet, il faut rappeler que l'ordre final d'un bloc séquentiel considère la partie lecture (exigence S1/R3).

L'algorithme 2 présente le calcul des ensembles de dépendances des blocs d'un circuit donné. La mise en œuvre de cet algorithme sera détaillée par la suite.

Algorithme 2: Algorithme de calcul des dépendances pour les circuits aux flots de données

Entrées: Un circuit c comptant s blocs atomiques.
Sorties: Chaque bloc atomique b du circuit c doit avoir un ensemble de dépendances.
pour tout bloc b du circuit c ayant un identifiant $i \in [0..s[$ **faire**
  L'ensemble des dépendances de chaque bloc b est vide
  $\mathcal{D}(i) \leftarrow \emptyset$;
fin pour
tantque le point fixe n'est pas atteint **faire**
  pour tout bloc b du circuit c ayant un identifiant $i \in [0..s[$ **faire**
    si le bloc b est un bloc séquentiel **alors**
      $\mathcal{D}(i) \leftarrow \mathcal{D}(i) \cup \{i\}$;
    sinon
      pour tout b' un bloc connecté à l'entrée du bloc b ($b' \in b.In$) **faire**
        $\mathcal{D}(i) \leftarrow \mathcal{D}(i) \cup \mathcal{D}(b') \cup \{i\}$;
      fin pour
    finsi
  fin pour
fin tantque

6.3.2.1 Calcul des dépendances

Nous appliquons le cadre général pour définir la nouvelle sémantique de l'ordonnanceur en se basant sur le calcul des dépendances des blocs.

6.3.2.1.1 Domaine du calcul des dépendances

Les valeurs calculées sont des ensembles de blocs (représentés par leurs identifiants entiers). Le domaine Δ est donc l'ensemble fini des parties de l'ensemble des blocs. Nous le représentons en exploitant la bibliothèque `FSetInterface` de Coq. Le domaine est spécifié en construisant un type dépendant de l'ensemble de tous les blocs du circuit (le support du domaine que nous appelons `carrier`). Le domaine est décrit par `type`, présenté précédemment dans la section 3.6.1.2.

```
Definition type := { value : S.FiniteSet.t | S.FiniteSet.Subset value S.carrier }.
```

Les éléments de `type` sont de la forme `exist x (S.FiniteSet.Subset x S.carrier)` en Coq (mathématiquement équivalente à $\exists\, x,\ x \subseteq S.carrier$).

```
Module Type FiniteSetType.

  Declare Module FiniteSet : FSetInterface.S
    with Module E := OrderedNatural.
  Parameter carrier : FiniteSet.t.

End FiniteSetType.
```

`OrderedNatural` définit un ordre total sur les entiers selon la signature du module Coq prédéfini `OrderedType` (voir fichier Domain.v). L'utilisation de `OrderedType` est requise dans la manipulation de `FSetInterface`.

Nous définissons ensuite la relation $\preceq$ sur le domaine abstrait Δ. Elle est décrite par le code Coq suivant :

```
Definition le : Data.type -> type -> type -> Prop  :=
      fun d x y =>
        match x, y with
          | exist sx _, exist sy _ => S.FiniteSet.Subset sx sy
        end.
```

La relation `le` est définie par la relation d'inclusion d'ensembles prédéfinie en Coq qui correspond à l'opérateur `Subset` de la bibliothèque `FSetInterface`.

$\preceq$ est une relation d'ordre partiel. Les preuves que cette relation est un ordre partiel ne seront pas détaillées dans ce manuscrit, en revanche elles peuvent être consultées dans le fichier DependencyLattice.v.

Le domaine Δ muni de la relation d'ordre partiel $\preceq$, que nous venons de définir, est un ensemble partiellement ordonné, noté $\langle \Delta, \preceq \rangle$. Sa borne inférieure notée $\bot$, correspond à $\emptyset$, tandis que sa borne supérieure correspond à l'ensemble des identifiants entiers (ou indices) de tous les blocs (`carrier`).

6.3.2.1.2 Famille de treillis des dépendances

L'ensemble partiellement ordonné $\langle \Delta, \preceq \rangle$ muni des éléments suivants est une famille de treillis notée $\{\theta_d\}_{d \in \texttt{unit}}$ (voir diagramme de Hasse de la figure 6.8) :

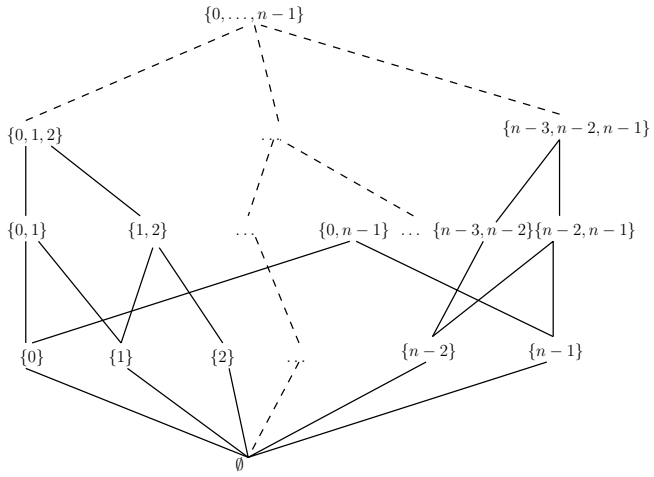


FIG. 6.8 – *Diagramme de Hasse de la famille de treillis des ensembles de dépendances*

- $\bot$, la borne inférieure de la famille de treillis $\{\theta_d\}_{d \in \texttt{unit}}$, qui vaut $\emptyset$;
- $\top$, la borne supérieure de la famille de treillis $\{\theta_d\}_{d \in \texttt{unit}}$, qui vaut `carrier` ;

– $\sqcup$, l'opérateur qui calcule la borne supérieure d'un couple d'éléments de Δ, est défini par la preuve comme suit :

```
Definition join : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
    PreOrder.type.
  Proof.
    intros.
    destruct H0 as (x0);
    destruct H1 as (x1).
    exists (S.FiniteSet.union x0 x1).
    unfold S.FiniteSet.Subset.
    intros.
    elim (S.FiniteSet.union_1 H0).
    auto.
    auto.
  Defined.
```

L'idée est de définir l'opérateur `join` sur deux ensembles x_0 et x_1, qui correspond à $x_0 \cup x_1$, complétée par la preuve que $(x_0 \cup x_1) \subseteq carrier$;

– $\sqcap$, l'opérateur qui calcule la borne inférieure d'un couple d'éléments de Δ, est défini par la preuve comme suit :

```
Definition meet : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
    PreOrder.type.
  Proof.
    intros.
    destruct H0 as (x0);
    destruct H1 as (x1).
    exists (S.FiniteSet.inter x0 x1).
    unfold S.FiniteSet.Subset in |- *.
    intros.
    apply s.
    eapply S.FiniteSet.inter_1.
    apply H0.
  Defined.
```

L'opérateur `meet` appliqué à deux ensembles x_0 et x_1 est défini par $x_0 \cap x_1$, complété par la preuve que $(x_0 \cap x_1) \subseteq carrier$.

Les éléments de la famille de treillis $\{\theta_d\}_{d \in \texttt{unit}}$ sont décrits par le foncteur `FinitePowerSetLattice` consultable dans le fichier DependencyLattice.v. Il s'agit d'un foncteur qui définit le treillis des ensembles finis.

Un ensemble d'entiers fini est ensuite défini avec le foncteur `DependencySet`.

```
Module DependencySet <: FiniteSetType.

 Module FiniteSet := FSetList.Make(OrderedNatural).
 Fixpoint makeSetFromList (l : list nat) {struct l} : FiniteSet.t :=
    match l with
      |nil          => FiniteSet.empty
      |head::queue => (FiniteSet.add head (makeSetFromList queue))
    end.
 Definition carrier :=
    makeSetFromList (blockIndexes (blocks ModelElementType model)).

End DependencySet.
```

`makeSetFromList` est la fonction qui construit un ensemble à partir d'une liste. Elle sera utilisée pour définir la borne `carrier`.

Le treillis des dépendances est alors défini par :

```
Module DependencyLattice := FinitePowerSetLattice (DependencySet).
```

6.3.2.1.3 Environnement de calcul des dépendances

L'environnement Y associe à chaque bloc un ensemble de dépendances dans Δ. Le calcul des dépendances ne dépend que de la taille du circuit à analyser ainsi

que de la nature des blocs. Cette dernière est exploitée par la fonction ρ qui calcule les dépendances de chaque bloc. Ainsi, l'environnement est une famille de treillis paramétrée par un couple de type $\mathbb{N} \times (\mathbb{N} \rightarrow \texttt{unit})$. L'environnement Y reste donc défini de la même manière que pour la version des rangs.

Définition 6.3.3. *Environnement de calcul des ensembles de dépendances*

$$Y \triangleq \{\{n \in \mathbb{N} | n < s\} \rightarrow \{\theta_{f(n)}\}\}_{s \in \mathbb{N}, f \in \mathbb{N} \rightarrow \texttt{unit}}$$

où le couple (s, f) est le paramètre de l'environnement.

L'environnement Y est créé avec le script :

```
Module DependencyEnvironment := MakeDependentEnvironment(DependencyLattice).
```

avec `DependencyLattice` la définition en Coq de $\{\theta_d\}_{d \in \texttt{unit}}$, la famille de treillis (ici un seul treillis) des ensembles de dépendances.

Initialement, l'environnement $(Y)_0$ associe pour chaque bloc du circuit à ordonnancer un ensemble de dépendances vide ($\perp$).

6.3.2.1.4 Fonction de propagation des dépendances

Les dépendances sont propagées à travers la structure du circuit avec la fonction ρ spécifiée en Coq par `Forward`. Il s'agit d'appliquer les équations données par la définition 6.3.2 sur l'environnement courant pour en calculer un nouveau (qui converge vers le point fixe).

```
Definition Forward
  (diagram : ModelType)
  (Env : DependencyEnvironment.PreOrder.type)
  : DependencyEnvironment.PreOrder.type :=
    fun blockIndex =>
      if (leb (size diagram) blockIndex)
      then DependencyLattice.bot
      else
        match (lookForBlock blockIndex diagram) with
          | Some( blockKind ) =>
              DependencyLattice.join
                tt
                (DependencyLattice.singleton blockIndex)
                (match blockKind with
                  | Sequential _ _ => DependencyLattice.bot
                  | _ => (inputEnv blockIndex)
                end))
          | error => DependencyLattice.bot
        end.
```

La propagation des dépendances des blocs est réitérée jusqu'à ce que le point fixe $lfp(\rho)$ soit atteint. Le résultat de ce dernier correspond à l'ensemble des dépendances des blocs du circuit analysé.

Le calcul de l'ordre d'exécution des blocs du circuit est basé ensuite sur le tri des dépendances. En effet, l'ensemble des dépendances des blocs correspond au résultat de calcul du plus petit point fixe des itérations de la fonction de propagation ρ. Il s'agit de réitérer le calcul des dépendances des blocs en utilisant la fonction de propagation ρ.

```
Definition Scheduler_rec
    (diagram : ModelType)
    (dependencies nextDependencies : EnvironmentPair.PreOrder.type)
    (iterated : isIterateBot diagram dependencies)
    (nextIsForward :
      forwardDependencies diagram dependencies = nextDependencies)
    : EnvironmentPair.PreOrder.type.
```

Le calcul des dépendances des blocs permet, en premier lieu, de déterminer les blocs qui sont indépendants les uns des autres et ceux qui sont reliés par les flots de données (directement ou indirectement). Ainsi, un ordre partiel peut être obtenu par simple inclusion des ensembles de dépendances. De plus, la présence de blocs indépendants est caractérisée par l'impossibilité de comparer leurs dépendances selon la relation d'inclusion. Le calcul de l'ordre d'exécution des blocs d'un circuit donné s'appuie donc sur le tri de leurs dépendances et sur la prise en compte des priorités.

6.3.2.2 Exemple

Si nous reprenons l'exemple «moyenne 5» présenté dans la figure 6.1, les dépendances de chaque blocs sont données dans la figure 6.9.

Bloc	**Ensemble de dépendances**
Input	{*Input*}
Cycle Counter	{*Cycle Counter*}
*Constant*1	{*Constant*1}
*Consant*2	{*Constant*2}
*Unit Delay*1(*R*)	{*Unit Delay*1}
*Unit Delay*2(*R*)	{*Unit Delay*2}
*Unit Delay*3(*R*)	{*UnitDelay*3}
*Unit Delay*4(*R*)	{*Unit Delay*4}
Relational Operator	{*Cycle Counter*, *Constant*1, *Relational Operator*}
Add	{*Input*, *Unit Delay*1, *Unit Delay*2, *Unit Delay*3, *Unit Delay*4, *Add*}
*Out*1	{*Cycle Counter*, *Constant*1, *Relational Operator*, *Out*1}
Gain	{*Input*, *Unit Delay*1, *Unit Delay*2, *Unit Delay*3, *Unit Delay*4, *Gain*}
Switch	{*Input*, *Unit Delay*1, *UnitDelay*2, *Unit Delay*3, *Unit Delay*4, *Gain*, *Cycle Counter*, *Constant*1, *Relational Operator*, *Constant*2, *Switch*}
*Out*2	{*Input*, *Unit Delay*1, *Unit Delay*2, *Unit Delay*3, *Unit Delay*4, *Gain*, *Cycle Counter*, *Constant*1, *Relational Operator*, *Constant*2, *Switch*, *Out*2}

FIG. 6.9 – *Les dépendances des blocs de «moyenne 5»*

Il est ainsi aisé de déduire les contraintes suivantes qui guideront l'ordonnancement des blocs. Ces contraintes sont déduites directement de l'inclusion

de dépendances des blocs :

$$\begin{cases} \bullet\ \Upsilon(Input) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Unit\ Delay1) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Unit\ Delay2) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Unit\ Delay3) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Unit\ Delay4) \preceq \Upsilon(Add) \\ \bullet\ \Upsilon(Add) \preceq \Upsilon(Gain) \preceq \Upsilon(Switch) \preceq \Upsilon(Out2) \\ \bullet\ \Upsilon(Cycle\ Counter) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Constant1) \preceq \Upsilon(Relational\ Operator) \\ \bullet\ \Upsilon(Relational\ Operator) \preceq \Upsilon(Out1) \\ \bullet\ \Upsilon(Relational\ Operator) \preceq \Upsilon(Switch) \\ \bullet\ \Upsilon(Constant2) \preceq \Upsilon(Switch) \end{cases}$$

Le calcul des dépendances permet d'obtenir un ordre partiel des blocs. En effet, la relation $\preceq$ qui correspond à l'inclusion d'ensembles de dépendances est réflexive et transitive, en revanche elle n'est pas antisymétrique. Il en est de même pour la relation $\preceq_\Upsilon$.

Afin d'associer un ordre d'exécution pour chaque bloc, un ordre total doit être construit à partir de l'ordre sur les dépendances. Celui-ci est présenté dans la section suivante.

6.3.2.3 Construction d'un ordre total

L'ordonnancement des blocs ne peut avoir comme seul guide leurs priorités, car le tri des blocs doit prendre en compte les dépendances des blocs les uns vis-à-vis des autres. Par conséquent, nous proposons de construire l'ordre total des blocs en triant les ensembles de dépendances obtenus par $lfp(\rho)$. Dans ce but, et comme cela a été évoqué précédemment, la relation d'ordre que nous pouvons prendre en compte naturellement est l'inclusion d'ensembles ($\preceq$). Toutefois, les ensembles de dépendances ne sont pas toujours comparables avec la relation d'ordre partiel $\preceq$. L'inclusion d'ensembles ne peut donc pas être une relation d'ordre totale. Une approche intuitive, pour trier ces dépendances incomparables, consiste à ordonner les blocs par ordre lexicographique de leurs priorités. De manière informelle, la relation d'ordre pourrait être obtenue de la manière suivante (similaire au principe de l'ordre total des rangs présenté en section 6.3.1.3) :

1. trier les blocs en fonction de leurs dépendances par la relation d'inclusion $\preceq$;
2. trier les blocs dont les dépendances sont incomparables par $\preceq$ selon l'ordre lexicographique de leurs priorités ;

Appliquons ce principe au circuit de la figure 6.10. α représente la priorité explicite des blocs. Le bloc C est plus prioritaire que le bloc A, mais aussi moins prioritaire que le bloc B. Le bloc le plus prioritaire correspond à celui qui a la plus petite valeur de priorité.

Notons par $\sqsubset$ la relation «est exécuté avant» qui résulte de la construction de l'ordre total. Nous avons $A \sqsubset B$ car $\mathcal{D}(A) \preceq \mathcal{D}(B)$, et le bloc C est indépendant des blocs A et B, ainsi il est incomparable avec ces derniers par $\preceq$. Par conséquent, la relation $B \sqsubset C \sqsubset A$ découle du tri lexicographique des priorités des blocs du circuit. Cette relation est contradictoire avec la relation $A \sqsubset B$.

Ainsi, le fait de trier les blocs uniquement en se basant sur leurs priorités ne va pas prendre en compte leurs dépendances en termes de flots de données,

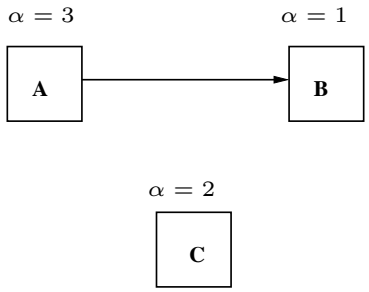


FIG. 6.10 – *Circuit* SIMULINK *à conflit d'exécution*

ce qui peut induire des incompatibilités avec le tri obtenu par $\preceq$. Il faut donc prendre en compte les dépendances lors du tri basé sur les priorités. Cette incohérence des exigences issues de la documentation de SIMULINK a pu être détectée grâce à la formalisation réalisée.

Une solution possible est de trier les éléments d'un ensemble de dépendances pour former une liste triée que nous dénommons séquence de dépendances. Les séquences obtenues pour chaque bloc seront ensuite elles-mêmes triées pour ordonnancer les blocs. Ainsi, l'affectation d'un ordre d'exécution pour chaque bloc est résumée par les étapes suivantes :

1. Les éléments de chaque ensemble de dépendances sont d'abord triés par ordre total lexicographique de leurs priorités (explicite et implicite) selon la définition 5.3.3. Cela produit une séquence de dépendances dont les éléments sont triés du plus prioritaire au moins prioritaire (tri décroissant selon les priorités).

   Considérons la séquence inverse des dépendances (voir section 5.3.2.2), c'est-à-dire, la séquence dont les éléments sont triés du moins prioritaire au plus prioritaire. Cette séquence inverse de dépendances est construite avec le foncteur `ReverseTotalPreOrder` défini dans le fichier Domain.v. Notons les séquences obtenues par $\overleftarrow{s}$;

2. Les éléments des séquences obtenues sont triés par ordre croissant de priorités. Ce qui permet de mettre en évidence les séquences dont les éléments sont les plus prioritaires. Dès lors, ces séquences sont triées par ordre total lexicographique (voir la définition de $\preceq_S$ dans la section 5.3.2.5.1). Le tri des séquences est construit avec le foncteur `TotalPreOrderLexicographicSequence` du fichier Domain.v.

   Ainsi, l'ordre des séquences fournit l'ordre d'exécution des blocs selon la définition suivante.

   Définition 6.3.4. *Ordre total des blocs*
   Soient $\overleftarrow{s_1}$ *et* $\overleftarrow{s_2}$ *deux séquences de dépendances respectives des blocs* b_1 *et* b_2*.*
   $b_1 \sqsubset b_2 \triangleq \overleftarrow{s_1} \preceq_S \overleftarrow{s_2}$

En appliquant l'algorithme d'ordonnancement avec calcul d'ordre total au circuit « moyenne 5 », nous obtenons le circuit annoté de la figure 6.11.

Cette solution répond bien au problème posé par l'algorithme 1. En effet, l'utilisateur a le choix de définir ses propres priorités sans que l'algorithme d'ordonnancement n'impose d'ordre particulier outre l'ordre impliqué par les contraintes sur la structure du circuit (les dépendances) et les contraintes de l'utilisateur (les priorités des blocs).

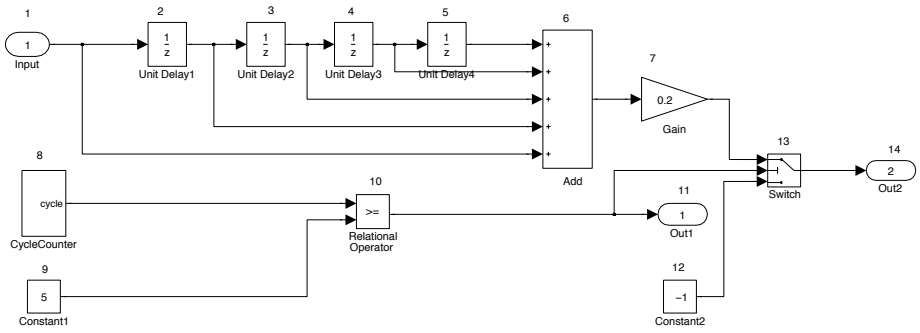


FIG. 6.11 – *Circuit moyenne 5 ordonnancé suivant les dépendances*

Les ordres d'exécution affectés aux blocs du circuit «moyenne 5» sont différents de ceux obtenus avec le calcul des rangs. En effet, l'ordonnancement basé sur les dépendances des blocs détermine bien les blocs qui sont comparables et ceux qui ne le sont pas tout en respectant les priorités de l'utilisateur ce qui n'était pas le cas pour l'ordonnancement basé sur le calcul des rangs.

6.3.2.3.1 Correction de l'ordre obtenu par rapport aux dépendances calculées

Une propriété importante de l'ordonnanceur est d'assurer que l'ordre calculé par l'algorithme d'ordonnancement est correct par rapport à la structure du circuit donné en entrée. Il s'agit principalement de vérifier que les séquences obtenues pour le calcul de l'ordre total des blocs respecte bien les ensembles de dépendances calculés par l'algorithme 2.

Théorème 6.3.8. *Correction des séquences construites par rapport à l'inclusion des ensembles de dépendances*
$\forall\, A\, B : \Delta,\ A \preceq B \Rightarrow \overleftarrow{A} \preceq_S \overleftarrow{B}$

Preuve. La preuve se fait par induction sur l'ensemble A.
Cas 1 : $A = \emptyset$. La séquence $\overleftarrow{A} = \langle\,\rangle \Rightarrow \langle\,\rangle \preceq_S \overleftarrow{B}$, pour tout ensemble $B \in \Delta$.
Cas 2 : $A = \{a\} \cup A'$, avec $\overleftarrow{A'} \preceq_S \langle a \rangle$ (c'est-à-dire l'élément a est moins prioritaire que les éléments de A'). Nous avons, donc, $\overleftarrow{A} = a :: \overleftarrow{A'}$.
Soit $B \in \Delta$ tel que $A \preceq B$, alors $a \in B$.
Considérons $\forall B' : \Delta,\ (A' \preceq B' \Rightarrow \overleftarrow{A'} \preceq_S \overleftarrow{B'})$ l'hypothèse d'induction, notée HI.
Cas 2-a : $\overleftarrow{B} \preceq_S \langle a \rangle$. Nous avons donc $\overleftarrow{B} = a :: \overleftarrow{B'}$, avec $B' \in \Delta$ et a l'élément le moins prioritaire de B. Étant donné $A \preceq B$, il en résulte
$\{a\} \cup A' \preceq \{a\} \cup B'$ et par conséquent $A' \preceq B'$. En appliquant l'hypothèse d'induction, il en résulte $\overleftarrow{A'} \preceq_S \overleftarrow{B'}$, et donc $a :: \overleftarrow{A'} \preceq_S a :: \overleftarrow{B'}$. Ainsi, la propriété $\overleftarrow{A} \preceq_S \overleftarrow{B}$ est satisfaite.
Cas 2-b : $\neg(\overleftarrow{B} \preceq_S \langle a \rangle)$. L'élément b en tête de $\overleftarrow{B}$ est moins prioritaire que l'élément a. Soit $\overleftarrow{B} = b :: \overleftarrow{B'}$, tel que $\langle a \rangle \preceq_S \langle b \rangle$. Ainsi, l'introduction de l'élément a implique $a :: \overleftarrow{A'} \preceq_S b :: \overleftarrow{B'}$ (car a est plus prioritaire que b). Par conséquent, la propriété $\overleftarrow{A} \preceq_S \overleftarrow{B}$ est vérifiée.

6.3.2.4 Discussion

Le calcul des dépendances répond exactement aux exigences de l'ordonnancement des circuits Simulink. Mais, cette approche reste inadaptée aux circuits contenant des flots de contrôle. En effet, outre le rajout des éléments indiquant les flots de contrôle, cette solution nécessite la prise en compte du fait que les sous-systèmes contrôlés ne s'exécutent ni avant ni après les blocs contrôleurs mais durant leur exécution. [IPTK11] présente une version préliminaire de l'algorithme dédié aux circuits contenant des flots mixtes. Toutefois, cet algorithme ne traite qu'un sous-ensemble des circuits combinant les flots de données et de contrôle. De plus, les blocs contrôleurs dépendent des entrées des sous-systèmes qu'ils contrôlent (à cause de la causalité des données), de même que les blocs contrôlés dépendent de la fermeture transitive des blocs qui les contrôlent. L'extension de l'algorithme d'ordonnancement a été présentée dans [IPT09]. Dans ce qui suit, nous présentons une solution qui répond au problème et qui ordonne les circuits comportant des flots de données et de contrôle tout en respectant l'ensemble des exigences de l'ordonnancement.

Notons, néanmoins, que nous avons déjà utilisé deux fois le cadre général défini précédemment.

6.4 Ordonnancement des circuits aux flots mixtes

Les signaux de contrôle sont fréquemment utilisés dans les systèmes industriels. Ils permettent d'imposer des contraintes d'exécution qui ne peuvent pas être exprimées simplement dans le langage de modélisation. Cela permet d'ordonner explicitement une partie du circuit de manière plus simple et compositionnelle qu'à travers les priorités (implicites et explicites). Il est donc nécessaire d'étendre les équations présentées précédemment qui ne traitent que les circuits aux flots de données. L'adaptation de l'algorithme 2 aux circuits avec des flots de contrôle n'est pas une tâche simple. En effet, les flots de contrôle nécessitent un traitement différent de celui des flots de données. L'ordonnancement des circuits ne contenant que des flots de données est guidé par les blocs sources dans une propagation en avant, car les valeurs des paramètres doivent être connues avant d'exécuter une fonction dans le code généré. Tandis que l'ordonnancement des circuits aux flots de contrôle dépend des début et fin d'exécution des blocs impliqués dans les flots de contrôle.

6.4.1 Équations de dépendances événementielles

Nous étendons les équations précédentes afin de prendre en compte des circuits combinant les deux types de flots. Considérons le circuit de la figure 6.12.

Le début d'exécution du bloc A demande l'exécution préalable de ses entrées (le bloc W). Tandis que lorsque le bloc A produit ses sorties, il doit avoir fini d'exécuter les blocs qu'il contrôle (le bloc B). Pour effectuer l'exécution de B, son entrée X doit avoir été exécutée. Donc, pour exécuter A, il faut que W et X aient été exécutés. Notons également que le bloc B produit sa sortie avant que le bloc A ne produise la sienne. Il est donc important de :

- distinguer les événements de début et de fin d'exécution d'un bloc ;
- disposer de deux équations afin de propager d'une part les dépendances pour le début d'exécution d'un bloc donné, et d'autre part les dépendances pour la fin de son exécution.

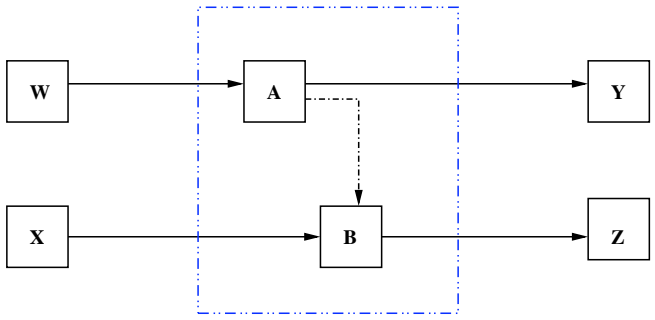


FIG. 6.12 – *Circuit combinant les flots de données et de contrôle*

Notons par $Call(b)$ l'événement marquant l'appel d'un bloc b (ou le début d'exécution du bloc b) et $Return(b)$ l'événement marquant le retour d'appel du bloc b (ou la fin de son exécution). Notons par $b.Callees$ l'ensemble des blocs contrôlés par b et par $b.Callers$ l'ensemble des blocs qui contrôlent le bloc b.

Nous distinguons les dépendances en deux catégories : dépendances d'entrée et dépendances de sortie.

Définition 6.4.1. ***dépendances d'entrée***
L'ensemble des dépendances d'entrée d'un bloc b, noté $\mathcal{D}_{in}(b)$, est défini comme l'ensemble des événements qui doivent s'être produits avant le début de l'exécution du bloc b.

Définition 6.4.2. ***dépendances de sortie***
L'ensemble des dépendances de sortie d'un bloc b, noté $\mathcal{D}_{out}(b)$, est défini comme l'ensemble des événements qui se sont produits avant la fin de l'exécution du bloc b et la production de ses sorties.

6.4.1.1 Contraintes de calcul des dépendances

Le calcul des dépendances exploite la structure du circuit en termes de flots de données et de contrôle ainsi que la nature des blocs (combinatoires ou séquentiels). Des contraintes additionnelles relatives à la nature du code généré s'y rajoutent également. Nous détaillons par la suite ces différentes contraintes. Nous nous appuyons sur la figure 6.12 pour illustrer certaines contraintes.

6.4.1.1.1 Contraintes structurelles

L'ordonnancement des blocs est d'abord tributaire de leur nature combinatoire ou séquentielle.

A Contraintes des blocs combinatoires

Les blocs connectés par des flots de données en entrée d'un bloc doivent avoir été exécutés pour produire ses entrées (exigence S1/R2). La figure 6.13 illustre l'enchaînement dans le temps de deux blocs W et A reliés par un flot de données tel que $W \in A.In$.

Pour commencer l'exécution du bloc A, ses entrées doivent être prêtes. Par conséquent, les événements $Call(W)$ et $Return(W)$ doivent avoir été préalablement produits afin de fournir les entrées du bloc A. Cela signifie que le bloc W produit ses sorties avant de commencer l'exécution du bloc A.

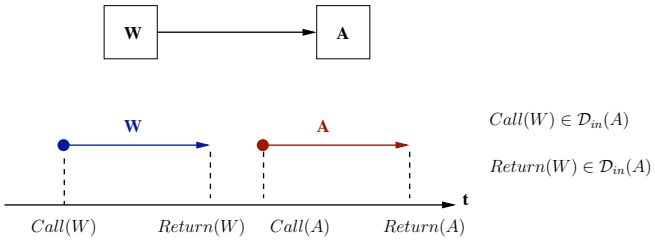


FIG. 6.13 – *Exécution de blocs reliés par flot de données*

Ainsi, les contraintes des blocs combinatoires se résument en :

$$\begin{array}{l} \forall x \in b.In,\ \mathcal{D}_{out}(x) \subseteq \mathcal{D}_{in}(b) \\ \forall x \in b.In,\ Call(x) \in \mathcal{D}_{in}(b) \\ \forall x \in b.In,\ Return(x) \in \mathcal{D}_{in}(b) \end{array} \qquad \forall\, b \text{ bloc combinatoire} \tag{6.3}$$

Remarque Afin qu'un bloc combinatoire b produise des sorties, les blocs produisant ses entrées doivent avoir été préalablement exécutés. Ainsi, nous en déduisons :

$$\mathcal{D}_{in}(b) \subseteq \mathcal{D}_{out}(b) \text{ avec b bloc combinatoire} \tag{6.4}$$

B Contraintes des blocs séquentiels

Les blocs séquentiels imposent l'ordonnancement des événements associés aux sous-blocs de lecture et d'écriture selon l'exigence S1/R3 (lecture avant écriture). La figure 6.14 illustre un circuit simple contenant un bloc `Unit Delay`.

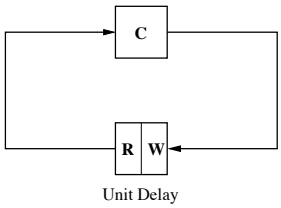


FIG. 6.14 – *Circuit à bloc séquentiel*

Nous devons interpréter les événements de manière légèrement différente. L'événement $Call(b)$ est interprété comme la lecture des données des cycles précédents nécessaire à l'exécution de ce bloc dans ce cycle et à la production de ses sorties. L'événement $Return(b)$ est interprété comme l'écriture des données du cycle courant.

D'une part, la sortie du bloc séquentiel `Unit Delay` correspond donc à une lecture de sa mémoire. Ceci se traduit par l'appel du bloc séquentiel (partie lecture R). Par conséquent, nous obtenons :

$$\forall\, b \text{ bloc séquentiel}, \quad Call(b) \in \mathcal{D}_{out}(b) \tag{6.5}$$

D'autre part, l'entrée du bloc séquentiel `Unit Delay` est une écriture dans la mémoire du résultat fourni par l'exécution du bloc C. Ceci se traduit donc par la fin d'exécution du bloc séquentiel (partie écriture W). De plus, un bloc séquentiel ne peut être contrôlé (ne peut être cible d'un flot de contrôle). Nous

avons ainsi :

$$\begin{array}{c} Return(b) \in \mathcal{D}_{in}(b) \\ \forall x \in b.In,\ \mathcal{D}_{out}(x) \subseteq \mathcal{D}_{in}(b) \\ \forall x \in b.In,\ Return(x) \in \mathcal{D}_{in}(b) \end{array} \qquad \forall\, b \text{ bloc séquentiel} \qquad (6.6)$$

Remarque Pour qu'un bloc b quelconque (séquentiel ou combinatoire) produise des sorties, il faut avoir commencé à l'exécuter.

$$\text{Pour tout bloc } b,\ Call(b) \in \mathcal{D}_{out}(b)$$

C Contraintes des flots de contrôle

Les blocs contrôlés s'exécutent durant l'exécution des blocs contrôleurs. Néanmoins, ils ne sont pas ordonnancés séparément car ils forment avec les blocs qui les contrôlent un seul bloc virtuel tel que nous l'avons expliqué dans la section 3.2.1. Pour finir l'exécution du bloc A, il faut avoir commencé et fini d'exécuter le bloc B. La figure 6.15 illustre ce flot d'exécution.

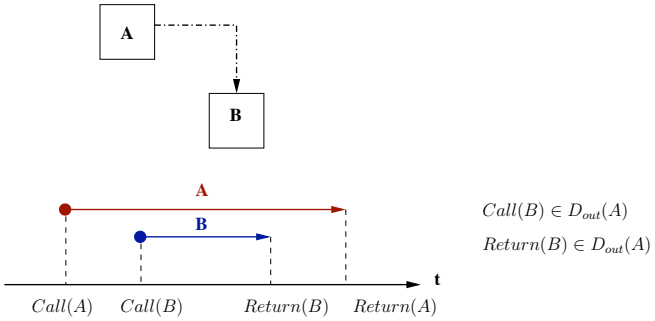


FIG. 6.15 – *Exécution du flot de contrôle entre A et B*

Ainsi, pour tout bloc b, nous avons les contraintes :

$$\begin{array}{c} \forall x \in b.Callees,\ \mathcal{D}_{out}(x) \subseteq \mathcal{D}_{out}(b) \\ \forall x \in b.Callees,\ Return(x) \in \mathcal{D}_{out}(b) \end{array} \qquad (6.7)$$

De même, commencer l'exécution de B demande le début d'exécution du bloc A qui le contrôle ainsi que les exécutions des entrées de A. Ainsi, pour tout bloc b, les contraintes de flots de contrôle sont complétées par :

$$\begin{array}{c} \forall x \in b.Callers,\ \mathcal{D}_{in}(x) \subseteq \mathcal{D}_{in}(b) \\ \forall x \in b.Callers,\ Call(x) \in \mathcal{D}_{in}(b) \end{array} \qquad (6.8)$$

6.4.1.1.2 Contraintes du code séquentiel

Par ailleurs, des contraintes additionnelles liées à la nature du code généré interviennent dans les équations de dépendances. Comme évoqué précédemment, [IPT09] introduit les dépendances événementielles pour ordonnancer les circuits aux flots mixtes. Cependant, une anomalie a été détectée lorsque les sorties d'un sous-système virtuel sont plus prioritaires que les contrôleurs de ce dernier. Cela est dû à l'absence d'exigences sur la nature du code généré.

GeneAuto génère du code séquentiel en C, ainsi, aucune exécution parallèle n'est tolérée en termes de *threads* par exemple. Par conséquent, les blocs dont les exécutions se chevauchent doivent être ordonnancés en séquence. Ainsi, toutes les entrées d'un sous-système virtuel doivent s'exécuter en premier, et toutes les sorties d'un sous-système virtuel doivent s'exécuter après celui-ci.

A Les entrées d'un sous-système virtuel

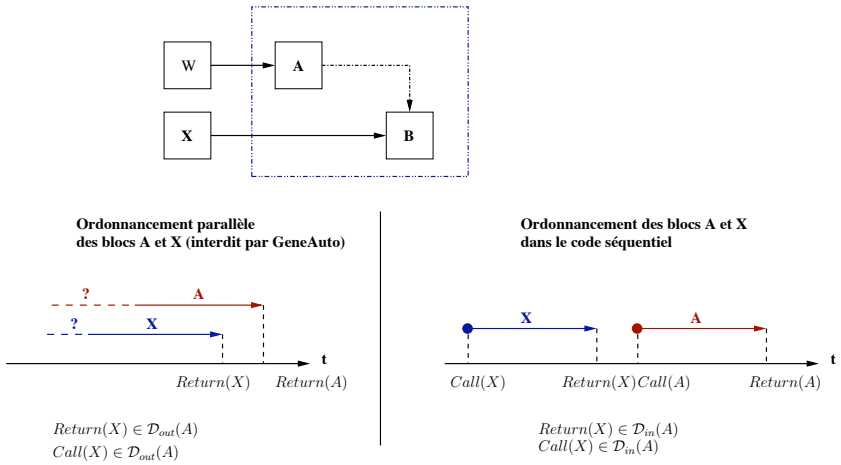


Fig. 6.16 – *Contraintes d'ordonnancement pour code séquentiel*

La partie gauche de la figure 6.16 montre que le bloc X doit terminer son exécution avant la fin du bloc A. Cela signifie que le bloc X s'exécute avant le bloc A dans le code séquentiel. Cela s'explique par le fait que le bloc X est exécuté avant le bloc B selon la contrainte de flots de données. Ainsi, comme les blocs A et B sont considérés comme un seul bloc virtuel (une fonction en C), le bloc X s'exécute avant le bloc A.

Rappelons que, selon l'exigence S1/R1, les blocs contrôlés ne sont pas ordonnancés. Néanmoins, ils sont utilisés par notre algorithme, et particulièrement manipulés par les équations de dépendances, pour transmettre les événements aux blocs non contrôlés.

Pour que le bloc A produise une sortie, il faut avoir exécuté, en plus de son entrée (le bloc W), les entrées des blocs qu'il contrôle, en l'occurrence le bloc X.

Ainsi, de manière plus générale, dans une cascade de flots de contrôle, toutes les entrées libres des blocs contrôlés doivent avoir été exécutées. En effet, ces dernières sont nécessaires pour produire les sorties de la chaîne des blocs contrôleurs. Ces informations sont présentes dans les dépendances de sortie du bloc A. Ainsi, nous utilisons les dépendances de sortie de A pour les extraire.

Notons par $\widehat{\mathcal{D}_{out}}(b)$ l'ensemble des dépendances de sortie de b qui ne sont pas contrôlées. Autrement dit, l'ensemble des événements des blocs libres nécessaires pour produire la sortie de b. Par exemple, dans la figure 6.12, pour le bloc A nous avons :

$$\widehat{\mathcal{D}_{out}}(A) = \{Call(W), Return(W), Call(X), Return(X)\}$$

Définition 6.4.3.

$$\widehat{\mathcal{D}_{out}}(b) \triangleq \mathcal{D}_{out}(b) \cap \{Call(x),\ Return(x) | \neg isControlled(x) \wedge x \neq b\}$$

avec $isControlled(x)$ est la fonction qui vérifie si un bloc x est contrôlé (c'est-à-dire appelé par un quelconque bloc).

Il en résulte la contrainte que tous les blocs libres, ayant fini leur exécution avant la fin du bloc A, devraient avoir été exécutés avant le bloc A :

$$\widehat{\mathcal{D}_{out}}(b) \subseteq \mathcal{D}_{in}(b) \tag{6.9}$$

B Les sorties d'un sous-système virtuel

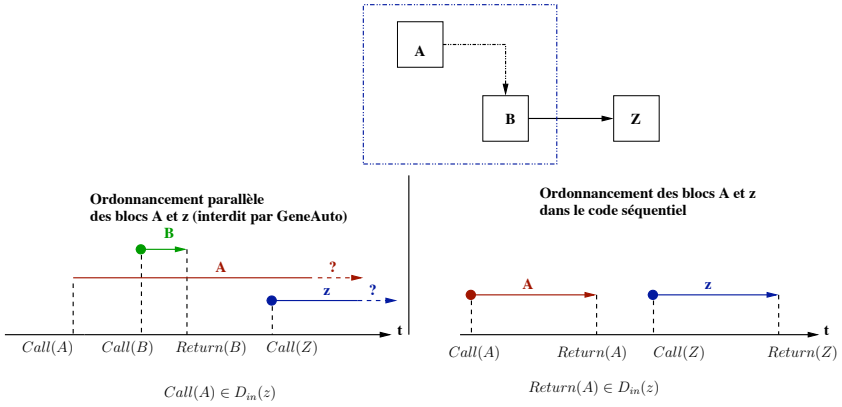


FIG. 6.17 – *Contraintes d'ordonnancement pour code séquentiel*

Dans la partie gauche de la figure 6.17, le bloc Z débute son exécution après que le bloc A ait commencé à être exécuté. Comme le code généré doit être séquentiel, le premier bloc qui débute son exécution doit finir avant qu'un autre bloc s'exécute. Ainsi, le bloc Z est ordonnancé après le bloc A. Cela s'explique, informellement, par le fait que B produisant l'entrée du bloc Z est exécuté dans le même bloc virtuel qui englobe les blocs A et B.

Par conséquent, toutes les sorties d'une succession de flots de contrôle sont exécutées après avoir fini l'exécution des blocs contrôleurs libres de cette chaîne de contrôle.

Notons par $roots(b)$ l'ensemble des événements de retour d'appel des blocs libres nécessaires pour produire les entrées du bloc b. Soit $isCalled(b, b')$ la fonction qui vérifie si le bloc b appelle directement ou indirectement le bloc b'.

$$roots(b) \triangleq \{Return(x) | Call(x) \in \mathcal{D}_{in}(b) \wedge \neg(isCalled(x, b) \vee isControlled(x))\} \tag{6.10}$$

Ainsi nous disposons de la contrainte :

$$roots(b) \subseteq \mathcal{D}_{in}(b) \tag{6.11}$$

6.4.1.2 Équations de dépendances

Nous avons montré précédemment que différentes contraintes interviennent pour le calcul des dépendances d'entrée et de sortie d'un bloc b donné. Nous résumons, dans les points suivants, ces contraintes.

1. les dépendances d'entrée regroupent tous les événements nécessaires pour produire les entrées du bloc b. Deux cas sont possibles :

- b est séquentiel : il s'agit alors de tous les événements nécessaires pour produire les entrées du bloc b : c'est-à-dire $\bigcup_{x \in b.In} \mathcal{D}_{out}(x) \cup \bigcup_{x \in b.In} \{Return(x)\}$). L'exécution des entrées d'un bloc séquentiel demande l'écriture des entrées dans la mémoire ($Return(b)$).
- b est combinatoire : il s'agit alors des événements produisant les entrées du bloc b ($\bigcup_{x \in b.In} \mathcal{D}_{out}(x)$), des événements produisant les entrées des blocs contrôlant b ($\bigcup_{x \in b.callers} \mathcal{D}_{in}(x)$), de l'appel des blocs contrôlant b ($\bigcup_{x \in b.callers} \{Call(x)\}$), du retour d'appel des entrées de b ($\bigcup_{x \in b.In} \{Return(x)\}$), des événements des blocs libres produisant les sorties du bloc b ($\widehat{\mathcal{D}_{out}}(b)$), ainsi que du retour d'appel des premiers blocs contrôleurs des entrées de b ($roots(b)$).

2. les dépendances de sortie représentent tous les événements devant avoir eu lieu pour que le bloc b produise ses sorties. Deux cas se présentent :
   - b est séquentiel : il s'agit alors d'une lecture de la mémoire ($\{Call(b)\}$) ;
   - b est combinatoire : il s'agit alors des événements produisant les entrées du bloc b ($\mathcal{D}_{in}(b)$), des événements produisant les sorties des blocs appelés par b ($\bigcup_{x \in b.callees} \mathcal{D}_{out}(x)$), du retour d'exécution des blocs appelés par b ($\bigcup_{x \in b.callees} \{Return(x)\}$), ainsi que de l'appel du bloc b ($\{Call(b)\}$). Tous ces événements sont nécessaires pour produire les sorties de b.

Enfin, les équations de dépendances sont calculées comme suit :

Définition 6.4.4. *dépendances d'entrée*

- *Si b est un bloc combinatoire :*

$$\mathcal{D}_{in}(b) \triangleq \bigcup_{x \in b.In} (\mathcal{D}_{out}(x) \cup \{Return(x)\}) \cup \bigcup_{x \in b.callers} (\mathcal{D}_{in}(x) \cup \{Call(x)\}) \cup \widehat{\mathcal{D}_{out}}(b) \cup roots(b) \tag{6.12}$$

- *Si b est un bloc séquentiel :*

$$\mathcal{D}_{in}(b) \triangleq \bigcup_{x \in b.In} (\mathcal{D}_{out}(x) \cup \{Return(x)\}) \cup \{Return(b)\} \tag{6.13}$$

Définition 6.4.5. *dépendances de sortie*

- *Si b est un bloc combinatoire :*

$$\mathcal{D}_{out}(b) \triangleq \mathcal{D}_{in}(b) \cup \bigcup_{x \in b.callees} (\mathcal{D}_{out}(x) \cup \{Return(x)\}) \cup \{Call(b)\} \tag{6.14}$$

- *Si b est un bloc séquentiel :*

$$\mathcal{D}_{out}(b) \triangleq \{Call(b)\} \tag{6.15}$$

L'algorithme 3 explicite le calcul des dépendances des blocs d'un circuit. Pour simplifier la représentation, nous dénotons par $\mathcal{R}$ l'événement *Return* et par $\mathcal{C}$ l'événement *Call*.

Algorithme 3: Algorithme de calcul des dépendances d'entrée et de sortie des circuits aux flots mixtes

Entrées: Un circuit c possédant s blocs.

Sorties: Chaque bloc atomique b du circuit c doit avoir un ensemble de dépendances d'entrée qui peut être éventuellement vide et de sortie qui doit au moins contenir l'événement $Call(b)$.

pour tout bloc b du circuit c **faire**

Initialisation de l'ensemble des dépendances d'entrée de chaque bloc b à vide

$\mathcal{D}_{in}(b) \leftarrow \emptyset$

Initialisation de l'ensemble des dépendances de sortie de chaque bloc b à vide

$\mathcal{D}_{out}(b) \leftarrow \emptyset$

fin pour

tantque le point fixe n'est pas atteint **faire**

pour tout bloc b du circuit c **faire**

si le bloc b est un bloc séquentiel **alors**

$$\mathcal{D}_{in}(b) \leftarrow \mathcal{D}_{in}(b) \cup \bigcup_{x \in b.In} (\mathcal{D}_{out}(x) \cup \{\mathcal{R}(x)\}) \cup \{\mathcal{R}(b)\}$$

$$\mathcal{D}_{out}(b) \leftarrow \mathcal{D}_{out}(b) \cup \{\mathcal{C}(b)\}$$

sinon

$$\mathcal{D}_{in}(b) \leftarrow \mathcal{D}_{in}(b) \cup \bigcup_{x \in b.In} (\mathcal{D}_{out}(x) \cup \{\mathcal{R}(x)\}) \cup \bigcup_{x \in b.callers} (\mathcal{D}_{in}(x) \cup \{\mathcal{C}(x)\}) \cup \widehat{\mathcal{D}_{out}}(b) \cup roots(b)$$

$$\mathcal{D}_{out}(b) \leftarrow \mathcal{D}_{out}(b) \cup \mathcal{D}_{in}(b) \cup \bigcup_{x \in b.callees} (\mathcal{D}_{out}(x) \cup \{\mathcal{R}(x)\}) \cup \mathcal{C}(b)$$

finsi

fin pour

fin tantque

6.4.2 Calcul des dépendances événementielles

Pour les circuits combinant les flots de données et de contrôle, le calcul d'ordre d'exécution des blocs est guidé par deux ensembles de dépendances : d'entrée et de sortie.

6.4.2.1 Domaine des dépendances événementielles

Le domaine Δ est un ensemble d'événements représentant l'appel et le retour des blocs. Les événements sont représentés par les identifiants des blocs (plus précisément, par leur indice, un entier naturel), marqués avec *Call* ou *Return* pour indiquer respectivement l'appel et le retour d'appel de ces blocs.

Définissons, dans un premier temps, les événements d'appel et de retour comme un type inductif. Ce type est défini à l'intérieur d'un module appelé `TaggedType` :

```
Module TaggedType.

  Inductive taggedNumbers : Set :=
  |Call : nat -> taggedNumbers
  |Return : nat -> taggedNumbers
  .
  Definition t := taggedNumbers.

End TaggedType.
```

Les dépendances que nous représentons en Coq sont des ensembles. Nous utilisons donc la librairie `FSetInterface` précédemment exploitée dans la section 6.3.2 pour manipuler ces ensembles.

Afin d'utiliser la structure d'ensemble fini du module `FSetInterface` de la librairie Coq `FSets`, il est nécessaire que le type des éléments des ensembles soit ordonné. Nous définissons, ainsi, le foncteur `OrderedTaggedNumbers` qui implante la signature `OrderedType` et définit l'ordre sur les événements d'exécution de type `taggedNumbers`.

```
Module OrderedTaggedNumbers : OrderedType with Definition t := TaggedType.t.

  Definition t := TaggedType.t.
  Definition eq : t -> t -> Prop :=
    fun x y =>
      match x, y  with
        |TaggedType.Call n , TaggedType.Call m     => n = m
        |TaggedType.Return n , TaggedType.Return m => n = m
        |_ , _                  => False
       end.
  Definition lt : t -> t -> Prop :=
    fun x y =>
      match x , y with
        |TaggedType.Call n, TaggedType.Call m      => n < m
        |TaggedType.Return n , TaggedType.Return m => n < m
        |TaggedType.Call n , TaggedType.Return m   => True
        |TaggedType.Return n , TaggedType.Call m   => False
      end.
  Lemma eq_refl : forall x : t, (eq x x).
  ...
  Lemma eq_sym : forall x y : t, (eq x y) -> (eq y x).
  ...
  Lemma eq_trans : forall x y z : t, (eq x y) -> (eq y z) -> (eq x z).
  ...
  Lemma lt_trans : forall x y z : t, (lt x y) -> (lt y z) -> (lt x z).
  ...
  Lemma lt_not_eq: forall x y : t, (lt x y) -> ~ (eq x y).
  ...
  Lemma compare : forall x y : t, Compare lt eq x y.
  ...

End OrderedTaggedNumbers.
```

La relation d'ordre sur ces événements est telle que l'appel d'un bloc est toujours plus petit que le retour d'appel d'un bloc quelconque. Pour les événements d'un bloc séquentiel, cela signifie que la lecture précède l'écriture en mémoire. Quant aux autres événements, ils sont ordonnés de la même façon que les entiers naturels, selon la définition `lt`. Un ensemble de dépendances est défini comme un ensemble fini d'événements de type `OrderedTaggedNumbers`. Nous réutilisons le principe présenté dans la section 6.3.2.1 en changeant seulement le type des éléments de l'ensemble.

```
Module Type FiniteSetTaggedType.

  Declare Module FiniteSet : FSetInterface.S
    with Module E := OrderedTaggedNumbers.
  Variable carrier : FiniteSet.t.

End FiniteSetTaggedType.
```

Les valeurs du domaine Δ sont alors implantées en utilisant la signature `FiniteSetTaggedType` par le foncteur `DependencySetTagged` de la figure 6.18. L'ensemble `carrier`, qui est la borne supérieure des ensembles de dépendances, contient tous les événements possibles d'un circuit : les événements d'appel et de retour de tous les blocs de ce circuit.

```
Module DependencySetTagged <: FiniteSetTaggedType.

  Module FiniteSet := TaggedFiniteSet.
  Fixpoint makeSetFromListTagged (l : list TaggedType.t ) {struct l} : FiniteSet.t
       :=
    match l with
      |nil           => FiniteSet.empty
      |head::queue   => (FiniteSet.add head (makeSetFromListTagged queue))
    end.
  Fixpoint callList (l : list nat) {struct l} : list TaggedType.t:=
    match l with
      |nil => nil
      |head::queue => (TaggedType.Call head)::(callList queue)
    end.
  Fixpoint returnList (l : list nat) {struct l} : list TaggedType.t :=
    match l with
      |nil         => nil
      |head::queue => (TaggedType.Return head)::(returnList queue)
    end.
  Definition carrier :=
    let cIndex := callList (blockIndexes (blocks ModelElementType model))
    in
    let rIndex := returnList (blockIndexes (blocks ModelElementType model))
    in
    makeSetFromListTagged (cIndex++rIndex).

End DependencySetTagged.
```

FIG. 6.18 – *Implantation du domaine des événements*

Les fonctions `callList` et `returnList` permettent de construire respectivement la liste de tous les appels et celle des retours d'appels des blocs passés en paramètre.

La relation d'ordre partiel sur les ensembles finis de dépendances, notée $\preceq$, correspond à la relation d'inclusion que nous avons définie dans la section 6.3.2.1.

6.4.2.2 Famille de treillis des dépendances événementielles

La famille de treillis (ici un seul treillis) $\{\theta_d\}_{d\in\texttt{unit}} \triangleq \langle \Delta, \preceq, \bot, \top, \sqcup, \sqcap \rangle$ est munie des mêmes opérateurs d'union et d'intersection définis précédemment.

La borne inférieure $\bot$ reste $\emptyset$, en revanche la borne supérieure $\top$ correspond à l'ensemble des événements d'appel et de retour de tous les blocs du circuit en question (`carrier`).

Le treillis des dépendances événementielles est défini par :

```
Module DependencyEventsLattice := FinitePowerSetLattice (DependencySetTagged).
```

6.4.2.3 Environnement de calcul des dépendances événementielles

L'ordonnancement des blocs d'un circuit exploite les informations calculées à travers les ensembles de dépendances d'entrée et de sortie. Par conséquent, l'environnement Υ associe pour chaque bloc b une paire d'ensembles de dépendances $(\mathcal{D}_{in}(b), \mathcal{D}_{out}(b))$. Υ est défini par $\Upsilon \triangleq \Upsilon_{in} \times \Upsilon_{out}$ avec :

- Υ_{in} l'environnement qui associe pour chaque bloc du circuit ses dépendances d'entrée ;
- Υ_{out} l'environnement qui associe pour chaque bloc du circuit ses dépendances de sortie.

Ainsi, les environnements Υ_{in} et Υ_{out} sont des familles de treillis définies par :

Définition 6.4.6. *Environnement de dépendances d'entrée* Υ_{in}

$$\Upsilon_{in} \triangleq \{\{n \in \mathbb{N} | n < s\} \to \{\theta_{f(n)}\}\}_{s \in \mathbb{N}, f \in \mathbb{N} \to \texttt{unit}}$$

où le couple (s, f) *est le paramètre de l'environnement.*

L'environnement des dépendances de sortie Υ_{out} a la même définition.

En Coq, les environnements Υ_{in} et Υ_{out} sont définis par le foncteur `MakeEnvironment` paramétré par le treillis des ensembles d'événements `DependencyEventsLattice`.

```
Module DependencyEventsEnvironment := MakeEnvironment(DependencyEventsLattice).
```

Ainsi, l'environnement Υ est spécifié par :

```
Module EnvironmentPair :=
  LatticeCartesianProduct (DependencyEventsEnvironment)
                          (DependencyEventsEnvironment).
```

L'environnement Υ est doté de la relation d'ordre partiel définie dans la section 5.3.3.1. Pour chaque bloc d'un circuit donné en entrée, les environnements Υ_{in} et Υ_{out} calculent respectivement les dépendances d'entrée et de sortie en partant d'ensembles vides comme valeurs initiales.

Initialement, l'environnement Υ associe, pour chaque bloc du circuit à ordonnancer, un ensemble vide ($\bot$) de dépendances d'entrée et un ensemble vide de dépendances de sortie.

6.4.2.4 Fonction de propagation des dépendances événementielles

En fonction de la nature des blocs (combinatoires ou séquentiels), pour chaque bloc, les équations des définitions 6.4.4 et 6.4.5 sont itérées jusqu'à ce que le point fixe soit atteint (voir l'algorithme 3).

Ainsi, les dépendances calculées pour chaque bloc sont propagées par la fonction de propagation $\rho \triangleq \rho_{in} \times \rho_{out}$, avec ρ_{in} et ρ_{out} les fonctions de propagation respectives des dépendances d'entrée et de sortie.

Les fonctions de propagation des dépendances ρ_{in} et ρ_{out} calculent les dépendances d'entrée et de sortie en fonction des contraintes décrites dans la section 6.4.1.1. Nous découpons, ainsi, ρ_{in} et ρ_{out} en fonctions élémentaires par type d'exigence.

Nous décrivons, d'une part, la spécification des contraintes de calcul des dépendances d'entrée :

- Les contraintes des blocs combinatoires (*cf.* équations 6.3) décrites par la fonction `CombinatorialContraintsIn`.

```
Definition CombinatorialConstraintsIn (diagram : ModelType)
  (outputEnv : DependencyEventsEnvironment.PreOrder.type) (blockIndex : nat)
       (dataSources : list nat)
  : DependencyEventsLattice.PreOrder.type :=
  DependencyEventsLattice.join
  tt
  (mergeDependencies
    outputEnv
    diagram
    dataSources)
    (makeSetFromList (DependencySetTagged.returnList dataSources))
```

- Les contraintes des blocs séquentiels (*cf.* équations 6.6) décrites par la spécification `SequentialConstraintsIn`.

```
Definition SequentialContraintsIn (diagram :  ModelType) (blockIndex : nat)
  : DependencyEventsLattice.PreOrder.type :=
  DependencyEventsLattice.singleton (TaggedType.Return blockIndex).
```

- Les contraintes des flots de contrôle décrites par la fonction `ControlFlowConstraintsIn` (*cf.* équations 6.8).

```
Definition ControlFlowConstraintsIn (diagram : ModelType) (inputEnv :
     DependencyEventsEnvironment.PreOrder.type)
  (blockIndex : nat) (controlSources : list nat)
  : DependencyEventsLattice.PreOrder.type :=
  DependencyEventsLattice.join
  tt
  (mergeDependencies inputEnv diagram controlSources)
  (makeSetFromList (DependencySetTagged.callList controlSources)).
```

- Les contraintes relatives au code séquentiel (*cf.* équations 6.9 et 6.10) décrites par la fonction `SequentialCodeConstraintsIn`.

```
Definition mergeSequentialC (diagram : ModelType) (blockIndex : nat) (
     inputDependencies outputDependencies : DependencyEventsEnvironment.
     PreOrder.type) :=
  DependencyEventsLattice.join tt (dependencyNotControlled diagram
       outputDependencies blockIndex) (returnRoots diagram inputDependencies
        blockIndex).

Definition SequentialCodeConstraintsIn (diagram : ModelType) (inputEnv
     outputEnv : DependencyEventsEnvironment.PreOrder.type)
 (blockIndex : nat)
  : DependencyEventsLattice.PreOrder.type :=
  (mergeSequentialC diagram blockIndex currentInputDependencies
       currentOutputDependencies)
```

D'autre part, la spécification des contraintes de calcul des dépendances de sortie est résumée par :

- La contrainte de bloc séquentiel se traduit par la fonction `SequentialConstraintsOut` dont la spécification est décrite dans l'équation 6.5.

```
Definition sequentialConstraintsOut (diagram :  ModelType) (blockIndex : nat
     )
  : DependencyEventsLattice.PreOrder.type :=
  DependencyEventsLattice.singleton (TaggedType.Call blockIndex).
```

– Les contraintes des flots de contrôle (*cf.* équations 6.7) sont décrites dans la fonction suivante.

```
Definition ControlFlowConstraintsOut (diagram : ModelType) (outputEnv :
     DependencyEventsEnvironment.PreOrder.type) (blockIndex : nat) (
     controlTargets : list nat)
  : DependencyEventsLattice.PreOrder.type :=
  (DependencyEventsLattice.join tt
    (makeSetFromList (DependencySetTagged.returnList controlTargets))
      (mergeDependencies outputEnv diagram controlTargets)).
```

La propagation des dépendances d'entrée est effectuée par la fonction $\rho_{in} : Y_{in} \times Y_{out} \rightarrow Y_{in}$. Il s'agit de calculer les dépendances d'entrée $\mathcal{D}_{in}$ des blocs, selon l'état courant de l'environnement des dépendances d'entrée Y_{in} et de sortie Y_{out}. La fonction ρ_{in} est décrite par la fonction `forwardInputDependencies`.

```
Definition forwardInputDependencies
  (diagram : ModelType)
  (inputEnv outputEnv : DependencyEventsEnvironment.PreOrder.type)
  : DependencyEventsEnvironment.PreOrder.type :=
  fun blockIndex =>
    if (leb (size diagram) blockIndex)
      then DependencyEventsLattice.bot
      else
        match (lookForBlock blockIndex diagram) with
          | (Some blockKind) =>
            let controlSources := controlSources diagram blockIndex
              in
              let dataSources := dataSources diagram blockIndex
                in
                    let controlTargets := controlTargets diagram blockIndex
                      in
                      DependencyEventsLattice.join
                      tt
                      (CombinatorialFlowConstraintsIn diagram outputEnv blockIndex
                           dataSources)
                      (match blockKind with
                         | (Sequential _ _) => DependencyEventsLattice.singleton (
                             TaggedType.Return



                         | _ =>
                           (DependencyEventsLattice.join
                             tt
                             (CombinatorialFlowConstraintsIn diagram outputEnv
                                  blockIndex dataSources)
                              (match blockKind with
                                 | (Sequential _ _) => SequentialConstraintsIn
                                     diagram blockIndex
                                 |_ =>
                                   (DependencyEventsLattice.join
                                     tt
                                     (ControlFlowConstraintsIn diagram inputEnv
                                          blockIndex controlSources)
                                     (SequentialCodeConstraintsIn diagram inputEnv
                                          outputEnv blockIndex)
                                     )end))
                    end)
          |_ => DependencyEventsLattice.bot
        end.
```

De même, les dépendances de sortie sont propagées par la fonction $\rho_{out} : Y_{in} \times Y_{out} \rightarrow Y_{out}$. Elle calcule pour chaque bloc les nouvelles dépendances de sortie en fonction de l'état courant des environnements Y_{in} et Y_{out}. La fonction ρ_{out} est décrite par la fonction `forwardOutputDependencies`.

```
Definition forwardOutputDependencies
  (diagram : ModelType)
  (inputEnv : DependencyEventsEnvironment.PreOrder.type)
  (outputEnv : DependencyEventsEnvironment.PreOrder.type)
  : DependencyEventsEnvironment.PreOrder.type :=
  fun blockIndex =>
    if (leb (size diagram) blockIndex)
      then DependencyEventsLattice.bot
      else
        match (lookForBlock blockIndex diagram) with
          | Some( blockKind ) =>
            let controlTargets := controlTargetFor (controlSignals
                 ModelElementType diagram)

                                  blockIndex
              in
              match blockKind with
                | Sequential _ _ => SequentialConstraintsOut diagram blockIndex
                | _ =>
                  DependencyEventsLattice.join
                  tt
                  (inputEnv blockIndex)
                  (DependencyEventsLattice.join
                    tt
                    (DependencyEventsLattice.singleton (TaggedType.Call blockIndex
                         ))
                    (ControlFlowConstraintsOut diagram outputEnv blockIndex
                         controlTargets))
              end
          | error => DependencyEventsLattice.bot
        end
```

Enfin, la fonction ρ, qui est la combinaison des fonctions ρ_{in} et ρ_{out}, est décrite dans le fichier SequencerDependencyEvents.v. À présent, nous ne détaillons pas l'implantation de ρ, car elle fait appel à un mécanisme de cache pour optimiser l'exécution du code extrait. Plus de détails sont donnés dans la section 8.1.7.

6.4.3 Ordre total

L'ordonnancement des blocs repose sur le tri des ensembles de dépendances présentés précédemment. Les dépendances d'entrée $\mathcal{D}_{in}$ servent à calculer les dépendances de sortie et sont contenues dans celles-ci. En outre, pour produire une sortie, un bloc b doit être appelé, cela correspond à l'événement $Call(b)$ compris dans les dépendances de sortie $\mathcal{D}_{out}(b)$ du bloc b. Ainsi, plusieurs blocs peuvent avoir les mêmes dépendances d'entrée alors que les dépendances de sortie de tous les blocs sont différentes car elles contiennent l'appel du bloc.

Nous ne considérons dans le tri que les dépendances de sortie des blocs. Celles-ci contiennent tous les événements qui doivent s'être exécutés pour produire les sorties des blocs d'un circuit donné. Or, l'ordre d'exécution des blocs correspond à leur ordre d'appel, et, par conséquent, seuls les événements d'appel de blocs sont considérés par le tri.

Les événements de retour d'appel sont donc supprimés des dépendances avant le tri des blocs. Toutefois, ils seront utilisés lors de la phase de détection de boucles algébriques dans la section 6.4.5.3. Enfin, les blocs appelés, déterminés par les événements d'appels, sont triés en utilisant l'approche décrite dans la section 6.3.2.1.

6.4.4 Exemple

Soit le circuit de la figure 7.17 présenté précédemment.

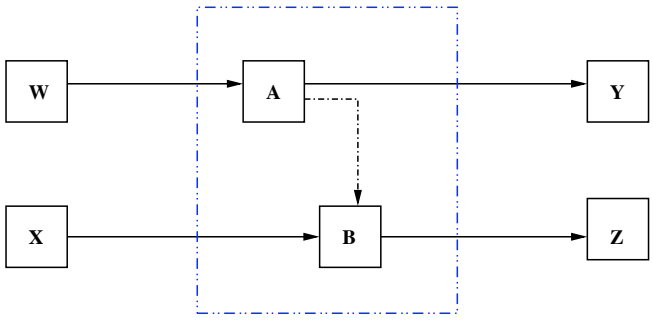


FIG. 6.19 – *Circuit combinant des flots de données et de contrôle*

Les figures 6.20 et 6.21 donnent respectivement les dépendances d'entrée et de sortie des blocs du circuit.

Bloc	**Dépendances d'entrée**
W	$\emptyset$
X	$\emptyset$
A	$\{C(W), R(W), C(X), R(X)\}$
B	$\{C(W), R(W), C(X), R(X), C(A)\}$
Y	$\{C(W), R(W), C(X), R(X), C(A), R(A), C(B), R(B)\}$
Z	$\{C(W), R(W), C(X), R(X), C(A), R(A), C(B), R(B)\}$

FIG. 6.20 – *Ensembles de dépendances d'entrée*

Nous donnons quelques exemples de calcul de *roots* et de $\widehat{\mathcal{D}_{out}}$.

$$\begin{aligned}
\widehat{\mathcal{D}_{out}}(A) &= \{C(W), R(W), C(X), R(X)\} \\
roots(A) &= \{R(W), R(X)\} \\
\widehat{\mathcal{D}_{out}}(Z) &= \{C(W), R(W), C(X), R(X), C(A), R(A)\} \\
roots(Z) &= \{R(W), R(X), R(A)\}
\end{aligned}$$

Bloc	**Dépendances de sortie**
W	$\{C(W)\}$
X	$\{C(X)\}$
A	$\{C(W), R(W), C(X), R(X), C(A), C(B), R(B)\}$
B	$\{C(W), R(W), C(X), R(X), C(A), C(B)\}$
Y	$\{C(W), R(W), C(X), R(X), C(A), R(A), C(B), R(B), C(Y)\}$
Z	$\{C(W), R(W), C(X), R(X), C(A), R(A), C(B), R(B), C(Z)\}$

FIG. 6.21 – *Ensembles de dépendances de sortie*

Supposons que tous les blocs possèdent les mêmes priorités explicites. Considérons les événements d'appel des blocs libres dans les ensembles de dépendances de sortie. Les éléments de ces derniers sont d'abord triés pour construire des séquences de blocs du moins prioritaire au plus prioritaire (composition lexicographique des priorités). La figure 6.22 contient les séquences triées des blocs du circuit de l'exemple.

Bloc	Séquences triées par ordre lexicographique
W	$\langle W \rangle$
X	$\langle X \rangle$
A	$\langle X, A, W \rangle$
Y	$\langle X, Y, A, W \rangle$
Z	$\langle Z, X, Y, A, W \rangle$

FIG. 6.22 – *Séquences de dépendances triées*

Enfin, les séquences sont ordonnées (voir section 6.3.2.3) pour associer aux blocs l'ordre d'exécution indiqué dans la figure 6.23.

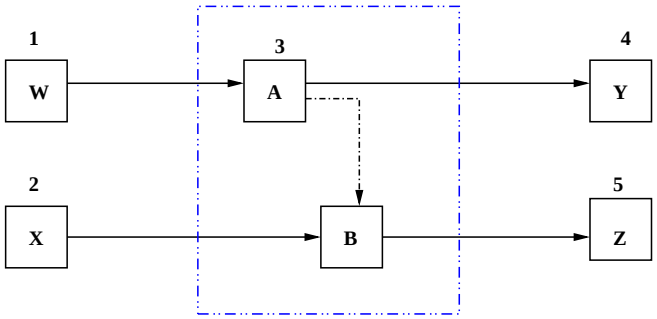


FIG. 6.23 – *Circuit combinant des flots de données et de contrôle*

6.4.5 Vérification de l'ordonnanceur

Nous avons présenté précédemment l'algorithme d'ordonnancement qui s'appuie sur les dépendances événementielles. La phase de vérification de l'ordonnanceur constitue une partie importante en termes de coût de développement. Le coût de la vérification peut être significativement supérieur au coût de la mise au point à base de tests du programme correspondant. Par contre, le composant obtenu ne contient aucune erreur cachée qui n'aurait pas été détectée par les tests. Dans ce qui suit, nous allons illustrer les principales preuves qui ont été réalisées.

Afin d'exploiter les propriétés du calcul de point fixe $lfp(\rho)$, présentées dans la section 5.3.6, l'environnement initial Y_0 doit être un pré-point fixe. Y_0 est égal à l'ensemble vide ($\perp_Y$).

Propriété 6.4.1. *L'environnement initial est un pré-point fixe*

$$\perp_Y \preceq_Y \rho(\perp_Y)$$

Il s'agit de vérifier la propriété `initialEnvIsPreFixpoint` de la signature de la conception d'un outil élémentaire (voir figure 5.14).

Preuve. La preuve est immédiate en exploitant la propriété `bot_least` de l'environnement Y.

6.4.5.1 Croissance des fonctions de propagation

Cette propriété assure que chaque appel récursif de la fonction d'itération ρ pour le calcul des dépendances fait progresser le calcul qui doit alors converger si le treillis est de profondeur finie. La fonction de propagation des dépendances événementielles est définie par : $\rho \triangleq \rho_{in} \times \rho_{out}$. La preuve que ρ est monotone se décompose en deux preuves : que ρ_{in} et ρ_{out} sont croissantes.

La plupart des preuves sont fondées sur la croissance de l'opérateur $\sqcup$ (union d'ensembles spécifié par `join`) de la famille de treillis $\{\theta_d\}_{d\in\texttt{unit}}$. Par définition de l'opérateur $\sqcup$, nous avons la propriété :

Propriété 6.4.2. *Croissance de l'opérateur* $\sqcup$
$\forall x_1\, x_2\, y_1\, y_2 : \Delta,\ x_1 \preceq x_2 \Rightarrow\ y_1 \preceq y_2 \Rightarrow\ (x_1 \sqcup y_1) \preceq (x_2 \sqcup y_2)$

6.4.5.1.1 Croissance de la fonction ρ_{in}

Dans un premier temps, nous démontrons que la fonction de propagation des dépendances d'entrée est monotone.

Théorème 6.4.3. *Croissance de la fonction de propagation* ρ_{in}

$$\forall e_1, e_3 : \mathrm{Y}_{in}, \forall e_2, e_4 : \mathrm{Y}_{out}, e_1 \preceq_{\mathrm{Y}_{in}} e_3 \Rightarrow e_2 \preceq_{\mathrm{Y}_{out}} e_4 \Rightarrow \rho_{in}(e_1, e_2) \preceq_{\mathrm{Y}_{in}} \rho_{in}(e_3, e_4)$$

Preuve. Soient les environnements e_1, e_2, e_3 et e_4 tels que
$e_1 \preceq_{\mathrm{Y}_{in}} e_3$ et $e_2 \preceq_{\mathrm{Y}_{out}} e_4$. Par définition, $\rho_{in}(e_1, e_2) \triangleq b \mapsto \mathcal{D}_{in}(b)$. Soit c un circuit, deux cas sont possibles selon la nature de chaque bloc b.
Cas 1 : b est un bloc séquentiel. L'ensemble des dépendances d'entrée de b correspond à $(\rho_{in}(e_1, e_2))(b) = e_1(b) \cup \bigcup_{x\in b.In}(e_2(x) \cup \{\mathcal{R}(x)\}) \cup \{\mathcal{R}(b)\}$, tel que e_1 et e_2 sont respectivement les environnements d'entrée et de sortie. De même, $(\rho_{in}(e_3, e_4))(b) = e_3(b) \cup \bigcup_{x\in b.In}(e_4(x) \cup \{\mathcal{R}(x)\}) \cup \{\mathcal{R}(b)\}$, tel que e_3 et e_4 sont respectivement les environnements d'entrée et de sortie.

Par hypothèse, nous avons $e_1 \preceq_{\mathrm{Y}_{in}} e_3$ et $e_2 \preceq_{\mathrm{Y}_{out}} e_4$. En utilisant la croissance de l'opérateur $\sqcup$, il en résulte $e_1(x) \preceq e_3(x)$ et $e_2(x) \preceq e_4(x)$ pour tout bloc x. Par conséquent, $(\rho_{in}(e_1, e_2))(b) \preceq (\rho_{in}(e_3, e_4))(b)$ pour tout bloc b séquentiel du circuit c.

Cas 2 : b est un bloc combinatoire. D'une part, nous avons :

$$\begin{aligned}(\rho_{in}(e_1, e_2))(b) = \quad & e_1(b) \cup \bigcup_{x\in b.In}(e_2(x) \cup \{\mathcal{R}(x)\}) \cup \\ & \bigcup_{x\in b.callers}(e_1(x) \cup \{\mathcal{C}(x)\}) \cup \widehat{e_2}(b) \cup roots(b)\end{aligned}$$

tel que e_1 et e_2 sont respectivement les environnements d'entrée et de sortie. D'autre part, nous disposons de

$$\begin{aligned}(\rho_{in}(e_3, e_4))(b) = \quad & e_3(b) \cup \bigcup_{x\in b.In}(e_4(x) \cup \{\mathcal{R}(x)\}) \cup \\ & \bigcup_{x\in b.callers}(e_3(x) \cup \{\mathcal{C}(x)\}) \cup \widehat{e_4}(b) \cup roots(b)\end{aligned}$$

tel que e_3 et e_4 sont respectivement les environnements d'entrée et de sortie.

Par hypothèse, nous avons $e_1 \preceq_{\mathrm{Y}_{in}} e_3$ et $e_2 \preceq_{\mathrm{Y}_{out}} e_4$. En utilisant la monotonie de l'opérateur $\sqcup$, il en résulte $e_1(x) \preceq e_3(x)$ et $e_2(x) \preceq e_4(x)$ pour tout bloc x. De plus, $\widehat{e_2}(b) \preceq \widehat{e_4}(b)$ car $e_2(b) \preceq e_4(b)$, et l'intersection de $e_2(b)$ et de $e_4(b)$ respectivement avec une constante préserve l'ordre (voir définition 6.4.3).

Par conséquent, $(\rho_{in}(e_1, e_2))(b) \preceq (\rho_{in}(e_3, e_4))(b)$ et ceci pour tout bloc b combinatoire.

Ainsi, il en résulte $\rho_{in}(e_1, e_2) \preceq_{\mathrm{Y}_{in}} \rho_{in}(e_3, e_4)$.

6.4.5.1.2 Croissance de la fonction ρ_{out}

Nous prouvons ensuite la croissance de la fonction qui propage les dépendances de sortie (ρ_{out}).

Théorème 6.4.4. *Croissance de la fonction de propagation ρ_{out}*

$$\forall e_1, e_3 : \mathrm{Y}_{in}, \forall e_2, e_4 : \mathrm{Y}_{out}, e_1 \preceq_{\mathrm{Y}_{in}} e_3 \Rightarrow e_2 \preceq_{\mathrm{Y}_{out}} e_4 \Rightarrow \rho_{out}(e_1, e_2) \preceq_{\mathrm{Y}_{out}} \rho_{out}(e_3, e_4)$$

Preuve. La preuve est similaire à celle du théorème 6.4.3. Elle consiste à remplacer ρ_{out} par sa définition qui dépend de la nature des blocs du circuit. L'équation est une suite d'unions $\sqcup$. Comme cette dernière est une opération monotone, la propriété $\forall b, (\rho_{out}(e_1, e_2))(b) \preceq (\rho_{out}(e_3, e_4))(b)$ est vraie.

Rappelons que l'environnement des dépendances est défini par $\mathrm{Y} \triangleq \mathrm{Y}_{in} \times \mathrm{Y}_{out}$.

Théorème 6.4.5. *Croissance de la fonction de propagation ρ*
$$\forall\, e_1\, e_2 : \mathrm{Y},\ e_1 \preceq_{\mathrm{Y}} e_2 \Rightarrow \rho(e_1) \preceq_{\mathrm{Y}} \rho(e_2)$$

Preuve. Par définition, la fonction ρ est représentée par les fonctions ρ_{in} et ρ_{out}. Les fonctions de propagation de dépendances d'entrée ρ_{in} et de sortie ρ_{out} sont croissantes selon les théorèmes 6.4.3 et 6.4.4. Nous en déduisons que la fonction ρ est également croissante.

6.4.5.2 Correction des équations de dépendances

Les équations de dépendances d'entrée et de sortie doivent respecter la structure du circuit. Il est donc important de démontrer deux propriétés : la correction des équations par rapport aux flots de données et aux flots de contrôle.

6.4.5.2.1 Correction des équations par rapport aux flots de données

Soit l'exemple élémentaire de flots de données illustré par la figure 6.24. Il s'agit de démontrer que les dépendances de sortie du bloc A sont incluses dans celles de B (selon l'exigence S1/R2).

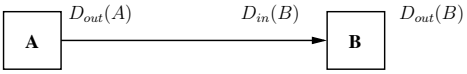


FIG. 6.24 – *Transitivité de flot de données*

Théorème 6.4.6. *Correction des équations par rapport aux flots de données*
$$\forall\, A,\, B,\, \neg isSequential(B) \Rightarrow A \in B.In \Rightarrow \mathcal{D}_{out}(A) \subseteq \mathcal{D}_{out}(B)$$

Preuve. Soient A, B deux blocs avec B combinatoire et $A \in B.In$. En dépliant la définition de $\mathcal{D}_{in}(B)$ (respectivement $\mathcal{D}_{out}(B)$), nous prouvons que $\mathcal{D}_{out}(A) \subseteq \mathcal{D}_{in}(B)$ (respectivement $\mathcal{D}_{in}(B) \subseteq \mathcal{D}_{out}(B)$). Enfin, la correction d'ordre est atteinte par transitivité.

6.4.5.2.2 Correction des équations par rapport aux flots de contrôle

Par ailleurs, la correction de l'ordre concerne également les flots de contrôle. Puisque les blocs contrôlés s'exécutent à l'intérieur (dans le code généré) des blocs contrôleurs, ceci implique qu'ils doivent être exécutés au moment de l'évaluation des blocs contrôleurs. Il faudra prouver que les blocs contrôleurs dépendent des blocs qu'ils contrôlent et vice-versa.

Soit le circuit illustré dans la figure 6.25 qui combine flots de données et de contrôle. Comme l'ordre final ne considère pas les blocs contrôlés, nous nous

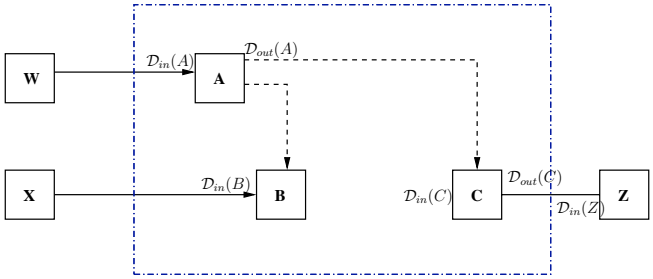


FIG. 6.25 – *Circuit mixant les flots de données et de contrôle*

intéressons particulièrement aux propriétés relatives aux blocs libres.

Une propriété pertinente consiste à vérifier que les blocs libres dont dépend le début d'exécution d'un bloc contrôleur (le bloc A dans l'exemple) sont également nécessaires pour produire la sortie des blocs qu'il contrôle (les blocs B et C dans l'exemple).

Notons par $\widehat{\mathcal{D}_{in}}(b)$ (respectivement par $\widehat{\mathcal{D}_{out}}(b)$) l'ensemble des blocs libres (autres que b) dont les événements figurent dans $\mathcal{D}_{in}(b)$ (respectivement dans $\mathcal{D}_{out}(b)$).

Théorème 6.4.7. $\forall A\ B,\ isCalled(A,B) \Rightarrow \widehat{\mathcal{D}_{in}}(A) \preceq \widehat{\mathcal{D}_{out}}(B)$

Preuve. Soit le bloc A qui contrôle le bloc B. En remplaçant $\mathcal{D}_{out}$ par sa définition, nous obtenons $\mathcal{D}_{in}(A) \preceq \mathcal{D}_{in}(B)$. Nous avons également $\mathcal{D}_{in}(B) \preceq \mathcal{D}_{out}(B)$. Il en résulte alors $\mathcal{D}_{in}(A) \preceq \mathcal{D}_{out}(B)$ par transitivité de la relation $\preceq$. D'autre part, la fonction de filtrage des blocs libres préserve la relation d'ordre. Ainsi, nous obtenons $\widehat{\mathcal{D}_{in}}(A) \preceq \widehat{\mathcal{D}_{out}}(B)$ qui montre que les blocs libres nécessaires pour débuter l'exécution de A sont également nécessaires pour produire la sortie de B.

Ainsi, dans la figure 6.25, le bloc W est nécessaire pour produire les sorties des blocs B et C.

Une autre propriété importante consiste à montrer que les blocs libres nécessaires pour exécuter un bloc contrôlé sont également nécessaires pour produire la sortie des blocs qui le contrôlent.

Théorème 6.4.8. $\forall A\ B,\ isCalled(A,B) \Rightarrow \widehat{\mathcal{D}_{in}}(B) \preceq \widehat{\mathcal{D}_{out}}(A)$

Preuve. Soit A un bloc qui contrôle le bloc B. Nous avons par définition, d'une part, $\mathcal{D}_{in}(B) \preceq \mathcal{D}_{out}(B)$, et d'autre part, $\mathcal{D}_{out}(B) \preceq \mathcal{D}_{out}(A)$. Par transitivité, il en résulte $\mathcal{D}_{in}(B) \preceq \mathcal{D}_{out}(A)$. Ainsi, nous déduisons $\widehat{\mathcal{D}_{in}}(B) \preceq \widehat{\mathcal{D}_{out}}(A)$.

Corollaire 6.4.9. *Toutes les entrées libres d'un sous-système virtuel sont exécutées avant celui-ci.*

Ceci est expliqué par la transitivité suivante :

$$\begin{cases} \widehat{\mathcal{D}_{in}}(B) \preceq \widehat{\mathcal{D}_{out}}(A) \\ \widehat{\mathcal{D}_{out}}(A) \preceq \mathcal{D}_{in}(A) \end{cases} \Rightarrow \widehat{\mathcal{D}_{in}}(B) \preceq \mathcal{D}_{in}(A)$$

avec A et B deux blocs tels que $isCalled(A, B)$. Ce qui signifie que tous les événements qui doivent avoir lieu avant l'appel de B, doivent se produire avant l'appel du bloc A, notamment les entrées libres du sous-système.

Ainsi, dans le circuit de la figure 6.25, le bloc X est exécuté avant le bloc A.

Par ailleurs, les sorties d'un sous-système virtuel sont exécutées après celui-ci.

Théorème 6.4.10. $\forall A\ B,\ isCalled(A, B) \Rightarrow \widehat{\mathcal{D}_{out}}(A) \preceq \mathcal{D}_{out}(B)$.

Preuve. La preuve est simple. Elle résulte de la transitivité suivante :

$$\begin{cases} \widehat{\mathcal{D}_{out}}(A) \preceq \mathcal{D}_{in}(A) \\ \mathcal{D}_{in}(A) \preceq \mathcal{D}_{in}(B) \\ \mathcal{D}_{in}(B) \preceq \mathcal{D}_{out}(B) \end{cases} \Rightarrow \widehat{\mathcal{D}_{out}}(A) \preceq \mathcal{D}_{out}(B)$$

Ainsi, toutes les sorties des blocs contrôlés doivent avoir lieu après le sous-système virtuel. Par exemple, dans la figure 6.25, le bloc Z est exécuté après le bloc A, c'est-à-dire après le sous-système contrôlé par A.

Outre les théorèmes correspondant aux principales propriétés montrées précédemment, des propriétés auxiliaires ont été nécessaires pour découper et accomplir les démonstrations complexes. Notons que plusieurs preuves sont indépendantes de l'ordonnanceur et peuvent être réutilisées dans d'autres composants du générateur de code tels que des preuves sur les types manipulés (listes, ensembles, etc).

6.4.5.3 Détection des boucles algébriques

Une propriété importante dans la modélisation de systèmes discrets à base de diagrammes de blocs en SIMULINK est l'absence de boucles algébriques, autrement dit, l'absence de problèmes de causalité liés à des cycles entre les signaux. Nous distinguons trois types de boucles algébriques : les boucles de données, les boucles de contrôle et les boucles mixtes qui combinent des signaux de données et de contrôle. Les boucles de données de base sont illustrées dans la figure 6.26(a). Ces boucles apparaissent lorsqu'un signal de données relie une sortie et une entrée d'un même bloc de base ou d'un même système et ceci dans le même cycle. D'un point de vue sémantique, cela signifie que, dans un même cycle, une donnée en sortie d'un signal est nécessaire pour calculer la donnée en entrée du signal, c'est-à-dire qu'une donnée est nécessaire pour se calculer elle-même, elle est donc définie par une récursivité mal fondée.

Les boucles de contrôle correspondent quant à elles à un bloc qui s'appelle récursivement tel que cela est illustré dans la figure 6.26(b). Dans une telle situation, le bloc peut être exécuté dans le même cycle un nombre de fois qui n'est pas borné statiquement. Ceci est interdit en temps discret en SIMULINK.

Ces deux types de boucles sont interdits sémantiquement car ils n'ont pas de sens. Par conséquent, les circuits représentant des cas similaires doivent également être rejetés en tant que circuits erronés.

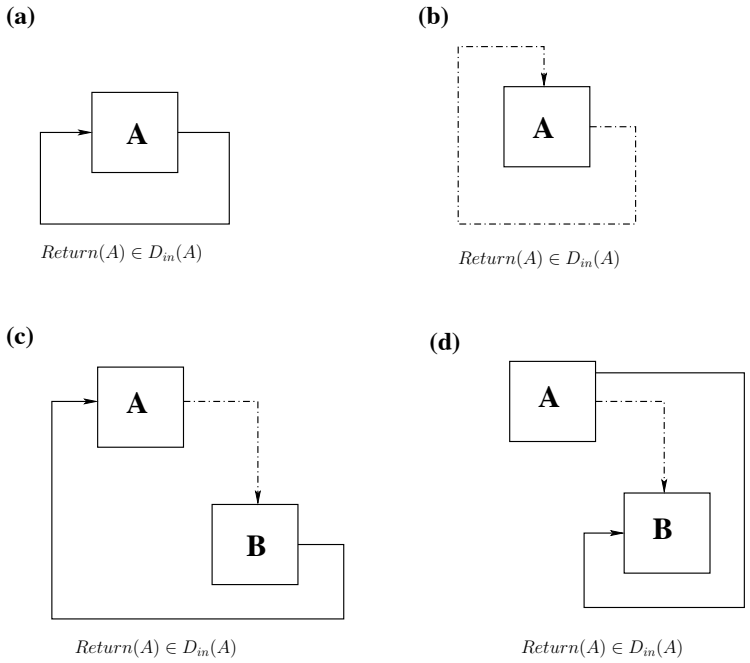


FIG. 6.26 – *Boucles de base en* SIMULINK

Les boucles de données et de contrôle peuvent être combinées en une boucle mixte, des exemples sont donnés dans les figures 6.26 (c) et 6.26 (d). Ces dernières sont considérées comme des erreurs potentielles dans notre analyse. Toutefois, SIMULINK accepte les circuits présentant ce genre de boucles, car il associe une mémoire à chaque signal de données et modifie celle-ci chaque fois que le signal est mis à jour par le calcul du bloc source. Son comportement est donc plus proche de la mise à jour d'une variable à partir de sa valeur précédente plutôt qu'à une équation récursive mal fondée définissant la valeur de la variable. L'utilisation d'un retard `Unit Delay` permet de briser la boucle algébrique à travers l'introduction des parties lecture et écriture de ce bloc séquentiel. Lorsqu'une boucle est signalée à l'utilisateur, celui-ci peut donc introduire explicitement un retard et retrouver le comportement habituel de SIMULINK, en explicitant celui-ci au lieu de s'appuyer sur un comportement implicite qui peut évoluer avec les versions de l'outil.

Pour obtenir le même comportement que SIMULINK en termes de détection de boucles, la distinction entre les erreurs certaines et les erreurs potentielles demande une analyse plus complexe pour détecter les boucles de données et de contrôle qui correspondent à des erreurs et les boucles mixtes qui correspondent à des avertissements. Dans notre analyse, les boucles de données ou de contrôle sont considérées comme des erreurs.

Les circuits lus par GENEAUTO sont rejetés par l'outil élémentaire de prétraitement *Preprocessor* s'ils présentent des problèmes de causalité sur les données. Par contre, cet outil élémentaire ne détecte ni les boucles de contrôle, ni les boucles mixtes. Toutefois, l'analyse que nous proposons par calcul de dépendances d'événements permet de détecter les problèmes de causalité des circuits traités indépendamment du traitement réalisé par l'outil élémentaire *Preprocessor*. Cette étude a permis de valider à travers des cas de tests la correction et la complétude des équations de dépendances événementielles proposées.

Le théorème suivant permet de détecter les boucles algébriques de la figure 6.26.

Théorème 6.4.11. $\forall\, b$ *bloc libre,*
il existe une boucle autour du bloc $b \Rightarrow Return(b) \in D_{in}(b)$

Actuellement, la preuve de ce théorème n'est pas réalisée. Elle peut être effectuée en parcourant la structure du circuit pour chercher l'existence d'un chemin reliant un bloc libre à lui-même sans passer par un bloc séquentiel comme `Unit Delay`. Ceci constitue la définition d'une boucle en général. Ainsi, $\mathcal{D}_{in}(b)$ comportera l'événement $Return(b)$ par application des équations qui définissent les dépendances événementielles.

Les expériences réalisées durant le développement de l'ordonnanceur nous permettent de supposer que la présence de l'événement $Return(b)$ dans $\mathcal{D}_{in}(b)$ est une condition de présence d'une boucle contenant le bloc b.

Conjecture 6.5. $\forall\, b$ *bloc libre,*
$Return(b) \in D_{in}(b) \Rightarrow$ *il existe une boucle autour du bloc* b

6.6 CAS D'ÉTUDE

Nous avons expérimenté l'ordonnanceur présenté sur plusieurs circuits de taille réelle dépassant les 5000 blocs. L'outil élémentaire associé est l'ordonnanceur officiel distribué avec la version libre du générateur de code GENEAUTO. Nous présentons ici une application automobile étudiée parmi plusieurs.

6.6.1 Cas d'étude : Contrôleur automobile

Il s'agit d'un système de contrôle automobile issu du système de régulation moteur de Continental Automobile. Ce cas d'étude est un extrait d'un système développé au sein de Continental. Il décrit la fonction de détection du cliquetis (KNOCK en anglais) dans un calculateur de contrôle moteur (gasoil pour ce système). Le cliquetis est décrit dans la figure 6.27. Le circuit de ce cas d'étude est trop volumineux pour présenter son ordonnancement ici. Nous nous contentons de décrire ce système.

Le cliquetis est une inflammation spontanée et incontrôlée du mélange air-carburant. L'automobile dispose d'éléments pour détecter ce phénomène, il s'agit de capteurs de cliquetis associés à un logiciel de détection. Le capteur de cliquetis «KNOCK » détermine lorsque le carburant brûle dans le moteur de façon inégale en causant des vibrations irrégulières dans le moteur. Il se compose d'une bobine électrique qui est enroulée autour de deux tiges en céramique avec un aimant dans le centre. Les vibrations du moteur provoquent la vibration des tiges. Cela perturbe le champ magnétique de la bobine et modifie le courant traversant la bobine. Cette perturbation se traduit par un signal particulier qui peut être analysé pour déterminer si les vibrations sont caractéristiques de cliquetis du moteur.

Le circuit correspondant au cliquetis est illustré dans la figure 6.28. Il combine des flots de données et de contrôle. Les signaux de contrôle sont représentés par des fils discontinus. Le circuit SIMULINK entier est composé de 9 couches hiérarchiques. Ce circuit contient plus de 5790 blocs (en comptant les sous-systèmes, les blocs atomiques ainsi que les ports enable/trigger). L'ordonnancement de la totalité du circuit automobile dure une seconde. Tandis

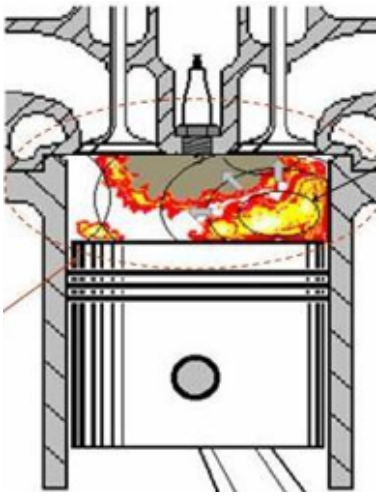


FIG. 6.27 – *Dégâts du moteur causés par le cliquetis*

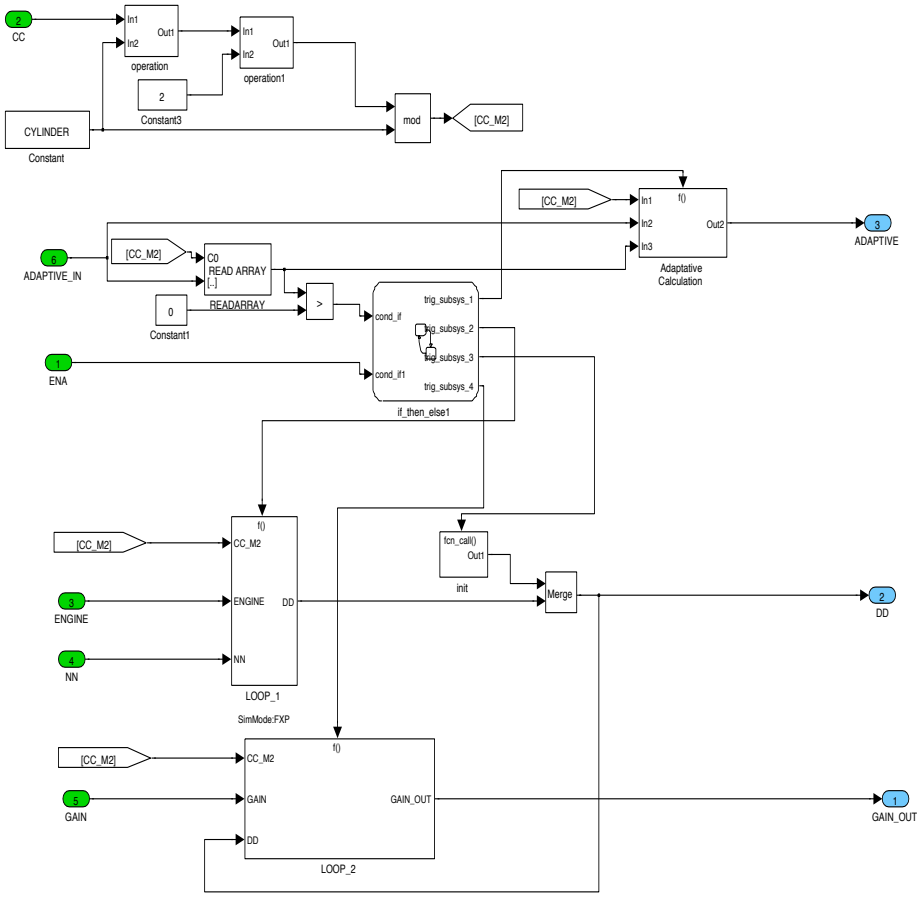


FIG. 6.28 – *Contrôleur d'un moteur automobile*

que la génération de code globale qui utilise les outils élémentaire développés formellement dure moins de 18 secondes.

6.6.2 Autres cas d'études

Nos outils élémentaires ont été expérimentés avec différentes applications industrielles de taille réelle [TIN+, RD]. Le générateur de code a été appliqué avec succès en incluant l'outil élémentaire ordonnanceur sur des cas d'études industriels critiques. Quelques exemples de cas d'études sont présentés dans la figure 6.29.

Cas d'étude	Blocs du circuit	Profondeur
Satellite AOCS (Attitude and Orbit Control System)	1085	8
Automotive power train software controler	5793	9
Airplane Flight Control System	2800	7
Satellite Agile Control System	1931	6
Sensor Networks	1108	7

FIG. 6.29 – *Récapitulatif des tailles d'autres études de cas industriels*

Blocs du circuit représente la taille du circuit étudié en termes de nombre de blocs considérés atomiques. Le nombre de niveaux hiérarchiques est représenté par l'entité *Profondeur*. Le coûts de l'outil élémentaire Ordonnanceur est comparable au reste des outils élémentaires du générateur de code. Nous avons également appliqué l'ordonnanceur à un circuit particulier qui constitue un «pire cas» : un «pipeline» de 5000 blocs dans un seul niveau hiérarchique. Ce pire cas exécuté sur un processeur Core 2 Duo à 2.6 GHz de fréquence avec 2Go de RAM a duré 45 minutes. Nous avons vérifié avec nos partenaires industriels qu'il n'existait aucun circuit d'une telle longueur (le nombre de bloc le plus grand entre une entrée et une sortie dans un circuit déplié). La plupart des circuits ont une longueur inférieure à 100 avec une moyenne comprise entre 10 et 20 selon le type de circuits.

6.7 SYNTHÈSE

La spécification de l'ordonnanceur, le raffinement des exigences et la formalisation des dépendances n'étaient pas des tâches simples. En effet, les exigences étaient ambiguës, incomplètes voire incohérentes.

Ceci est dû à la complexité du langage SIMULINK et à la forme de sa documentation composée d'exemples et de contre exemples.

Nous avons spécifié formellement l'ordonnanceur à partir des exigences et avons implanté l'ordonnanceur en nous appuyant sur les dépendances événementielles d'entrée et de sortie. Cette implantation permet de prendre en compte tout type de circuit combinant les flots de données et de contrôle. Les équations de dépendances proposées expriment naturellement les événements liés aux blocs qui doivent se produire avant d'exécuter un bloc. La construction de cet algorithme a été prouvée en démontrant les propriétés de monotonie des fonctions de propagation et de terminaison du calcul du point fixe. Le résultat obtenu par l'algorithme est également validé en vérifiant la correction de l'ordre obtenu par rapport aux exigences sur les flots de données et de contrôle. En outre, la propriété de détection de boucles algébriques permet de valider également que les équations événementielles proposées sont complètes et que les fonctions de propagation transmettent bien les informations.

Typage des circuits Simulink 7

7.1 Motivation

Le typage est le second cas d'étude qui exploite le cadre général dans GeneAuto. Il s'agit d'un algorithme pour associer un type à chaque port des blocs d'un circuit Simulink donné en entrée du générateur de code.

Nous parlerons dans ce chapitre d'inférence de type [Mil78, DM82] au lieu de propagation de type. L'inférence de type, dans un circuit, permet de rechercher automatiquement un type pour les ports des blocs qui le composent. Ceci est effectué à partir des informations données explicitement dans le circuit, des règles de typage des blocs ainsi que des règles de propagation des types dans le circuit.

Notre objectif à travers ce cas d'étude est, d'une part, d'assurer que les types calculés sont corrects par rapport à la nature des blocs et, d'autre part, de valider les types donnés par le concepteur du circuit.

L'inférence de type en Simulink est un mécanisme relativement simple à réaliser, notamment quand tous les types des signaux du circuit sont explicités par l'utilisateur. Tel est le cas des circuits communs à plusieurs partenaires industriels car leurs calculateurs n'utilisent pas tous les types prédéfinis par Simulink, mais exploitent des types différents définis par les utilisateurs. Le concepteur du circuit fixe au préalable les types de tous les signaux pour mieux contrôler les types par lui-même. Dans ce cas, il ne s'agit plus d'inférence de type mais de vérification de types qui est réalisée en une seule étape sans propagation.

Soit le circuit illustré dans la figure 7.1. La sortie du bloc `Constant` est définie par l'utilisateur de type `int32 [2]` (tableau de deux entiers à 32 bits). L'inférence de type depuis l'entrée vers la sortie du circuit permet de déduire les types mentionnés dans la figure 7.1. Par exemple, le bloc `Add` a une seule entrée de type `int32 [2]`, sa sortie est donc un scalaire de type `int32`. Cela signifie que les types des ports définis par l'utilisateur doivent être au minimum, dans le système de types considéré, les types produits par inférence en avant.

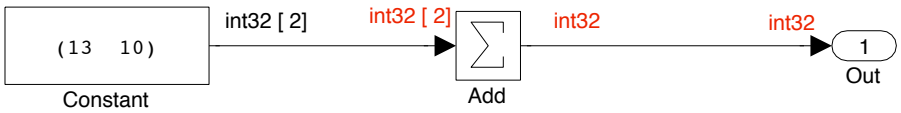


Fig. 7.1 – *Exemple d'inférence de type depuis les entrées vers les sorties*

Inversement, si l'entrée du bloc `Out` est spécifiée par l'utilisateur de type `double` (réel de 64 bits), alors l'inférence de type depuis la sortie du circuit vers son entrée produit les types annotés dans la figure 7.2. Par exemple, la sortie du bloc `Add` a le type `double` (déduit par inférence arrière), donc le type de son unique entrée est un vecteur `double [ ]`*. Cela signifie que le bloc `Out` doit lire, lors de son exécution, au maximum (dans le système de types considéré) une valeur de type `double`. Ainsi, les types des ports définis par l'utilisateur doivent être au maximum ceux produits par inférence en arrière.

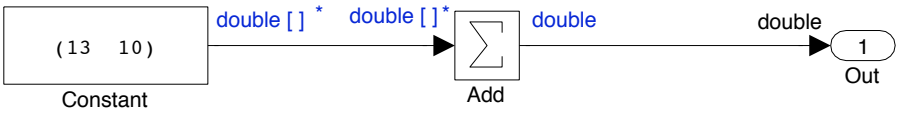


FIG. 7.2 – *Exemple d'inférence de type depuis les sorties vers les entrées*

Le fait d'effectuer seulement l'inférence en avant impose à l'utilisateur de tester son circuit à chaque fois qu'il précise un type de port d'un bloc interne (hors les blocs source et cible des librairies respectives `Source` et `Sink`). Il en est de même pour l'inférence en arrière, ce qui devient vite fastidieux, notamment quand le circuit est de taille importante. Or, en pratique, les utilisateurs ont le libre choix de préciser ou non les types des ports (d'entrée et/ou de sortie) des blocs. Que ce soit en SIMULINK ou encore en GENEAUTO, le circuit est respectivement simulé ou testé à chaque fois que l'utilisateur change les initialisations des types.

Dans le but d'offrir une solution générique, permettant de limiter certains des tests des circuits à chaque nouvelle initialisation des types, et de déterminer les contraintes de type des différents ports, nous proposons un algorithme pour typer les ports d'un circuit SIMULINK en effectuant deux inférences de type : une en avant et une en arrière. De plus, nous ne considérons les initialisations ou annotations de types, effectuées par les utilisateurs, que pour les blocs d'entrée et de sortie du circuit qui doivent être définis dans GENEAUTO. Cela permet de prendre en compte un grand nombre d'alternatives de typage correct des ports du circuit, indépendamment des types des ports explicités autres que les ports des blocs source et cible.

7.2 PRÉSENTATION DES ALGORITHMES DE TYPAGE

Dans le but de donner une vue globale de la démarche adoptée pour typer un circuit, nous présentons deux algorithmes d'inférence.

L'algorithme 4 (respectivement 5) résume les principales étapes pour effectuer une inférence de type en avant (respectivement en arrière) dans un circuit donné.

Initialement, que ce soit dans l'algorithme 4 ou dans l'algorithme 5, seuls les types des paramètres de blocs sont pris en compte, ainsi que les types des blocs source et cible. De plus, les deux algorithmes fonctionnent indépendamment l'un de l'autre. Le résultat des deux inférences sert ensuite à valider les types explicités par l'utilisateur.

Nous décrivons dans la suite les éléments de mise en œuvre de ces algorithmes.

Algorithme 4: Algorithme de typage pour inférence en avant

Entrées: Circuit c à s blocs atomiques.
$\mathcal{F}_A^-$: Fonction de typage des sorties de chaque bloc A.
$A.In$: Liste de blocs d'entrée de chaque bloc A.
Sorties: Un type pour chaque port des blocs du circuit.
tantque le point fixe n'est pas atteint **faire**
pour tout bloc A ayant un identifiant $j \in [0..s[$ **faire**
pour tout port de sortie o **faire**
Calculer le type de sortie de o obtenu par $\mathcal{F}_A^-$
fin pour
Mettre à jour les types de sortie
fin pour
pour tout bloc A ayant un identifiant $j \in [0..s[$ **faire**
pour tout port d'entrée i **faire**
Calculer la borne supérieure des types de i et du port de sortie de $A.In$ dans le treillis des types considéré.
fin pour
Mettre à jour les types d'entrée
fin pour
fin tantque

Algorithme 5: Algorithme de typage pour inférence en arrière

Entrées: Circuit c à s blocs atomiques.
$\mathcal{F}_A^+$: Fonction de typage des entrées de chaque bloc A.
$A.Out$: Liste de blocs de sortie pour chaque bloc A.
Sorties: Un type pour chaque port des blocs du circuit.
répéter
pour tout bloc A ayant un identifiant $j \in [0..s[$ **faire**
pour tout port d'entrée i **faire**
Calculer le type de l'entrée i obtenu par $\mathcal{F}_A^+$
fin pour
Mettre à jour les types d'entrée
fin pour
pour tout bloc A ayant un identifiant $j \in [0..s[$ **faire**
pour tout port de sortie o **faire**
Calculer, dans le treillis des types considéré, la borne inférieure du type de o et de tous les types des ports d'entrée de $A.Out$ qui lui sont reliés
fin pour
Mettre à jour les types de sortie
fin pour
jusqu'à ce que le point fixe soit atteint

7.3 CALCUL DES TYPES

Le cadre général présenté dans le chapitre 5 s'applique sur l'outil élémentaire de typage afin d'inférer les types des blocs d'un circuit donné. À la définition d'un bloc, un utilisateur peut déterminer le type des ports d'entrée et/ou de sortie, en fonction des types des paramètres du bloc si celui-ci en possède. Nous parlerons désormais de type de port et non de type de signal. En effet, le type d'un signal ne permet pas de distinguer les contraintes de type de ses deux extrémités (les ports source et cible respectifs). Néanmoins, les ports source et cible d'un signal ne peuvent être finalement de types différents en SIMULINK. Ceci est lié au processus de simulation où le transtypage, en anglais *type casting*, dans un signal n'est accepté que si un bloc `Data Type Converter` convertit le type du port source en le type du port cible. Dans notre analyse, nous acceptons qu'un signal ait les deux extrémités de types différents mais compatibles comme nous le verrons dans la section 7.7. C'est pourquoi nous parlons dans ce manuscrit de type de port au lieu de type de signal.

Ainsi, l'information calculée et propagée à travers la structure du circuit correspond aux types des ports des blocs de ce circuit. De plus, les types de base considérés dans cette étude correspondent aux types définis dans la section 3.2.4.

7.3.1 Domaine de calcul

Nous étendons les valeurs des types SIMULINK définies dans la section 3.2.4 par des types personnalisés. En effet, les utilisateurs peuvent tout à fait définir leurs propres types en rapport avec les applications développées et le processeur utilisé.

De plus, le système de types de GENEAUTO distingue, outre les types scalaires numériques et booléens ainsi que les types vectoriels, les événements, les fonctions et les caractères. Nous nous contentons dans notre étude des types scalaires numériques et booléens, vectoriels ainsi que les types personnalisés que peuvent définir les utilisateurs.

Notons ici que nous ne représentons pas le type des signaux d'activation et de contrôle. Ces derniers sont des événements qui servent à activer des sous-systèmes et n'interviennent pas dans leur typage.

Ainsi, le domaine abstrait Δ correspond à l'ensemble des valeurs que peut prendre le type d'un port d'un bloc en GENEAUTO lors de l'inférence de type. Les valeurs de type que nous représentons ici sont communes à tous les partenaires industriels du projet. Δ est alors décrit par le type inductif `simType` suivant.

```
Inductive simType : Set :=
 |custom : nat -> simType
 |basic : basicType -> simType
 |array : dim -> basicType -> simType
 |undef : simType
 |any : simType
 |Error : simType
 with basicType : Set :=
 |boolean : basicType
 |single : basicType
 |double : basicType
 |int8 : basicType
 |int16 : basicType
 |int32 : basicType
 |uint8 : basicType
 |uint16 : basicType
 |uint32 : basicType
 |fixed : nat -> basicType
 |compx8 : basicType
 |compx16 : basicType
 |compx32 :basicType
 |ucompx8 : basicType
 |ucompx16 : basicType
 |ucompx32 : basicType.
```

L'utilisateur peut associer explicitement à un port n'importe quel type de `simType` sauf `undef` et `Error`. Le type `Error` indique une erreur de type. Le type `any` est utilisé pour déterminer tout type différent de `Error`. Cela permet d'exhiber, lors de l'inférence de type, les vrais problèmes de typage quand ils ont lieu.

Un type de base (`basic`) peut être le type booléen `boolean`, ou un type numérique `single, double, ..., uint32, fixed n, ..., ucompx32` rencontré précédemment dans la section 3.2.4.

Un type de port peut être aussi : un type personnalisé défini par l'utilisateur (`custom`) qui est paramétré par un entier qui le distingue, ou un vecteur `array` de dimensions définies par `dim` et d'un certain type de base. Dans ce dernier cas, les dimensions (`dim`) d'un vecteur, lorsqu'elles sont connues, sont représentées par une liste d'entiers indiquant respectivement la taille de chaque dimension, ou par `lstar` lorsqu'il s'agit d'un vecteur de dimensions quelconques.
Le type `dim` est défini simplement par :

```
Inductive dim : Set :=
 |lv : list nat -> dim
 |lstar : dim.
```

Nous définissons la relation d'ordre `le_dim` qui compare les dimensions par :

```
Definition le_dim : dim -> dim -> Prop :=
fun x y  =>
      match x, y with
        |lv l1, lv l2 => l1=l2
        |_ , lstar => True
        |_ , _ => False
      end.
```

La relation `le_dim` est un ordre partiel défini en utilisant la signature décrite dans 5.3.2.1.

Un vecteur dont la dimension est une liste à un seul élément n correspond à un tableau de taille n. De même, si la dimension du vecteur est une liste à deux éléments n et m respectivement, il s'agit d'une matrice à n lignes et m colonnes.

Par exemple, la construction `array (lv (6::nil)) double` détermine un tableau de 6 éléments de type `double`.

Afin de simplifier la représentation des vecteurs, nous utilisons la notation $X\,[m_1]\ldots[m_n]$ pour désigner un vecteur à n dimensions de tailles respectives allant de m_1 à m_n. En pratique, seuls les tableaux et matrices sont manipulés. Cette notation se réduit à $X\,[m]$ pour désigner un tableau de m éléments de type X et à $X\,[m][n]$ pour désigner une matrice $m \times n$ de type X.

7.3.2 Famille de treillis des types

Le typage des circuits SIMULINK en GENEAUTO est restreint par rapport à ce qui est toléré par le système de types de SIMULINK dans la mesure où il n'existe pas autant de variantes pour le typage que dans SIMULINK. Il existe quasiment une variante de système de types pour chaque configuration de SIMULINK, comme il peut y avoir un système de types par partenaire de GENEAUTO. Il est donc judicieux de considérer la famille de treillis des types comme un paramètre de l'algorithme de typage que nous proposons. De ce fait, nous proposons de définir deux treillis pour établir l'inférence de type : une famille de treillis de typage à la SIMULINK (pour une configuration donnée) et une famille de treillis de typage dérivée des exigences de GENEAUTO.

Mais il aurait été également possible de définir autant de familles de treillis que de systèmes de types par partenaire industriel du projet et par configuration de SIMULINK.

Dans ce qui suit, la relation d'ordre partiel $\preceq$ sur les éléments de Δ désigne le *sous-typage*. Ainsi, les opérateurs $\sqcup$ et $\sqcap$ calculent respectivement le plus petit *super-type* et le plus grand *sous-type* de deux éléments de Δ.

7.3.2.1 Famille de treillis des types en Simulink

Une multitude de configurations sont possibles pour typer un circuit en SIMULINK. Nous avons vu dans le chapitre 3 qu'un bloc peut accepter des types d'entrée différents et éventuellement transtyper le type de la sortie en un type spécifié comme un paramètre dans la boîte de dialogue du bloc concerné. Il est également possible de préciser dans la boîte de dialogue d'un bloc que ses entrées doivent toutes être du même type.

En règle générale, pour un bloc donné, SIMULINK convertit les types de ses entrées en un type dit accumulateur. Il exécute ensuite l'opération correspondant au bloc considéré dont le résultat est converti au type de sortie spécifié en effectuant les arrondis nécessaires.

Toutefois, afin de simplifier la présentation, nous ne considérons que la configuration où, pour tous les blocs, *les types ne sont comparables que s'ils sont égaux*. Autrement dit, un bloc n'accepte en entrée que des signaux du même type. De plus, les booléens ne sont pas considérés comme des types numériques et vice-versa.

Les types de base sont comparés avec la relation `le_basic` de sorte que tous les types scalaires de base sont incomparables deux à deux.

Soit la fonction `eq_basic` qui détermine si deux types de base sont égaux.

```
Definition eq_basic (d : Data.type) (x y : basicType) : Prop :=
    x=y.
```

La relation d'ordre partiel $\preceq$ pour la configuration que nous avons choisie est décrite par la définition COQ `le` suivante :

```
Definition le : Data.type-> type -> type-> Prop :=
      fun d x y =>
        match x , y with
          |basic t1 , basic t2 => eq_basic d t1 t2
          |array l1 t1 , array l2 t2 =>  (le_dim tt l1 l2) /\ (eq_basic d t1 t2)
          |undef , _
          |_ , Error => True
          |custom n1 , custom n2 => n1=n2
          |_ , any =>
            match x with
                  |Error => False
              |_ => True
            end
          |_,_ => False
        end.
```

où tout type est un super-type de `undef` et sous-type de `Error`. Tous les types sauf `Error` sont des sous-types de `any`. Les vecteurs ne sont comparables par $\preceq$ que s'ils ont des dimensions comparables par `le_dim` et si leurs types de base respectifs sont égaux. Ainsi, un tableau n'est pas comparable avec une matrice par exemple et un tableau de booléens n'est pas comparable avec un tableau d'entiers.

L'ensemble partiellement ordonné $\langle \Delta, \preceq \rangle$, muni des éléments suivants, correspond à une famille de treillis, notée $\{\theta_d\}_{d \in unit}$, dont le paramètre d est de type `unit`, car il s'agit d'une famille de treillis à un seul élément (donc un treillis) :

- $\bot$ la plus petite borne inférieure du treillis $\{\theta_d\}_{d \in unit}$ qui vaut `undef` ;
- $\top$ la plus grande borne supérieure du treillis $\{\theta_d\}_{d \in unit}$ qui vaut `Error` ;
- $\sqcup$ l'opérateur binaire d'union décrit par la fonction `join`. Il calcule le plus petit super-type de deux types donnés, c'est-à-dire la borne supérieure de ces deux types.

  Dans un premier temps, nous définissons le plus petit super-type des types de base comme suit :

```
Definition superBasic : PreOrder.Data.type -> basicType -> basicType ->
    PreOrder.type :=
    fun d x y =>
      if (PreOrder.eq_basic_dec d x y)
        then basic x
        else
          any.
```

  Elle est ensuite utilisée pour définir la fonction `join`.

```
Definition join : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
   PreOrder.type :=
    fun d x y =>
      match (PreOrder.dec d x y) with
        |left _ => y
        |right _ =>
          match (PreOrder.dec d y x) with
            |left _ => x
            |right _ =>
              match x, y with
                |basic t1, basic t2 => superBasic d t1 t2
                |array n1 t1, array n2 t2 =>
                  if (dim_le_dec tt n1 n2) then
                    match (superBasic d t1 t2) with
                      |basic x => array n1 x
                      |_ => any
                    end
                    else any
                |_,_ => any
              end
          end
      end.
```

La propriété `dim_le_dec` définit la décidabilité de `le_dim` (voir le fichier TypeLattice.v).

– $\sqcap$ l'opérateur binaire d'intersection décrit par `meet`. Il calcule le plus grand sous-type de deux types donnés, c'est-à-dire la borne inférieure de ces deux sous-types.
Dans un premier temps, nous définissons le plus grand sous-type de deux types de base.

```
Definition subBasic : PreOrder.Data.type -> basicType -> basicType ->
    PreOrder.type :=
  fun d x y =>
    if (PreOrder.eq_basic_dec d x y)
      then basic x
      else
        undef.
```

Ainsi, l'opérateur $\sqcap$ dans $\{\theta_d\}_{d \in unit}$ est défini par :

```
Definition meet : PreOrder.Data.type -> PreOrder.type -> PreOrder.type
    -> PreOrder.type :=
  fun d x y =>
    match (PreOrder.dec d x y) with
      |left _ => x
      |right _ =>
        match (PreOrder.dec d y x) with
          |left _ => y
          |right _ =>
            match x, y with
              |basic t1, basic t2 => subBasic d t1 t2
              |array n1 t1, array n2 t2 =>
                if (le_dim_dec tt n1 n2) then
                  match (subBasic d t1 t2) with
                    |basic x => array n1 x
                    |_ => undef
                  end
                  else undef
              |_,_ => undef
            end
        end
  end.
```

Le treillis des types, en SIMULINK, est représenté par la figure 7.3 pour les types scalaires et par la figure 7.4 pour les vecteurs.

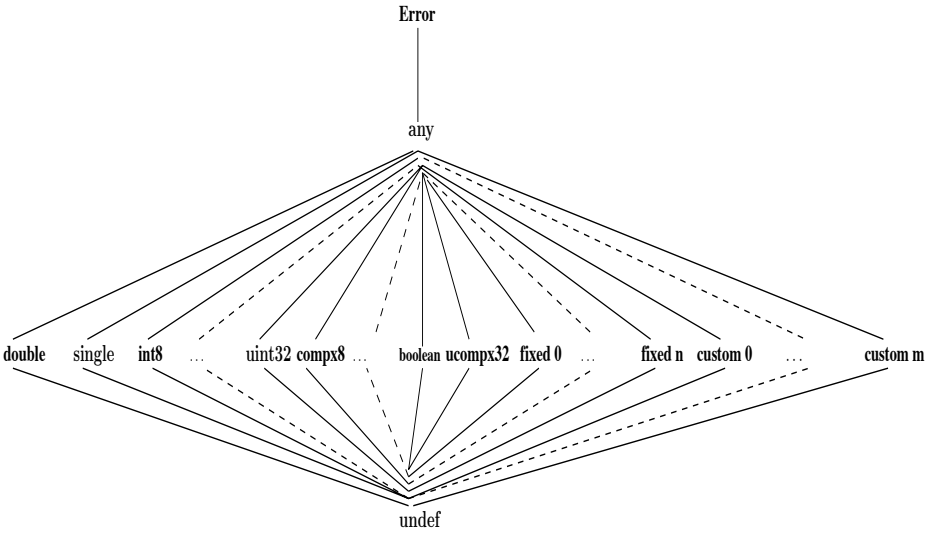


FIG. 7.3 – *Diagramme de Hasse du treillis des types scalaires d'une simple configuration en* SIMULINK

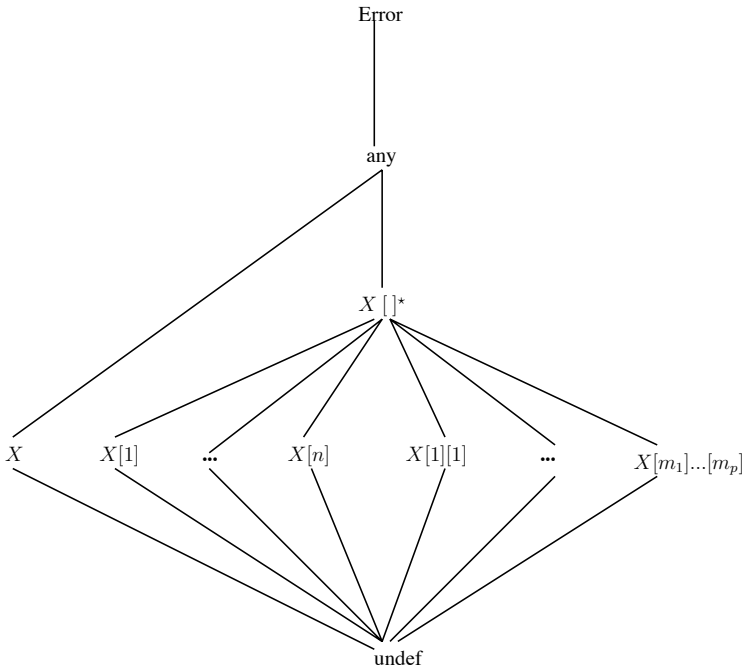


FIG. 7.4 – *Diagramme de Hasse du treillis des vecteurs selon la configuration* SIMULINK *choisie*

7.3.2.2 Famille de treillis des types en GeneAuto

Dans la configuration par défaut de SIMULINK, certains blocs peuvent effectuer des opérations sur des entrées de types différents, éventuellement incompatibles avec la sémantique d'exécution des blocs. Ces opérations peuvent être réalisées sans pour autant que le circuit ne soit rejeté.

C'est par exemple le cas des blocs logiques en SIMULINK qui acceptent des entrées numériques, ou encore le cas des blocs arithmétiques qui peuvent introduire une perte de précision ou effectuer des opérations sur des entrées booléennes.

Dans le but de restreindre certaines de ces possibilités, le système de types en GENEAUTO a mis en place quelques règles de sous-typage que nous citons dans ce qui suit et les traduisons dans la famille de treillis.

7.3.2.2.1 Contraintes de typage en GeneAuto

GENEAUTO a mis en place plusieurs contraintes pour limiter les ambiguïtés présentes dans le système de types de SIMULINK. Nous citons ici quelques contraintes principales concernant le sous-ensemble de blocs que nous étudions. En effet, pour chaque bloc, il y a une règle ou contrainte de typage.

T/R1 Les types entiers : `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` sont des sous-types de `single`;

T/R2 Le type `single` est sous-type de `double`;

T/R3 `int8` et `uint8` sont des sous-types de `int16`;

T/R4 `uint8` est sous-type de `uint16`;

T/R5 `int16` et `uint16` sont des sous-types de `int32`;

T/R6 `uint16` est sous-type de `uint32`;

T/R7 Les types booléen, complexe et virgule fixe ne sont compatibles avec aucun autre type ;

T/R8 Les différents partenaires peuvent définir leurs propres types qui ne sont comparables avec aucun autre type ;

T/R9 Un bloc combinatoire peut avoir une seule entrée. Il effectue alors l'opération élément par élément sur l'entrée s'il s'agit d'un vecteur, sinon produit une sortie égale à l'entrée ;

T/R10 Les blocs de la librairie `Source` et `Sink` possèdent toujours un paramètre dont le type doit être défini ;

Ces contraintes sont prises en compte directement lors de la construction de la famille de treillis des types.

7.3.2.2.2 Famille de treillis des types en GeneAuto

Nous allons dans un premier temps élaborer la relation $\preceq$ définie dans Δ avant de présenter le treillis abstrait.
L'équivalence des types de base `eq_basic` est la même que celle définie pour SIMULINK (voir Section 7.3.2.1). Les types de base sont triés par la relation `le_basic` définie comme suit.

```
Definition le_basic : Data.type -> basicType -> basicType -> Prop :=
    fun d x y =>
      match x, y with
        |int8, int16
        |int8, int32
        |int16, int32
        |uint8, uint16
        |uint8, uint32
        |uint8, int16
        |uint8, int32
        |uint16, uint32
        |uint16, int32 => True
        |_, double =>
          match x with
            |single |int8 |uint8 |int16 |uint16 |int32 |uint32 |double => True
            |_ => False
          end
        |_, single =>
          match x with
            |int8 |uint8 |int16 |uint16 |int32 |uint32 |single  => True
            |_ => False
          end
        |_ , _  => eq_basic d x y
      end.
```

De même que pour SIMULINK, le type `undef` est la plus petite valeur du domaine Δ et tout autre type est super-type de `undef`. De plus, le type `Error` est la plus grande valeur de Δ, et le type `any` est le super-type de tous les types définis.

Toutefois, quelques différences existent pour GENEAUTO relativement à ce qui était présenté dans la section 7.3.2.1. D'une part, tous les types numériques de base sont des sous-types du type `single` qui lui-même est un sous-type de `double`. D'autre part, les vecteurs sont comparables par la relation `le` si leurs dimensions respectives sont comparables par `eq_dim` et si leurs types de base respectifs sont comparables par `le_basic`. Une autre différence concerne les scalaires : ils sont considérés comme des vecteurs de n'importe quelles dimensions de n'importe quelles tailles. Cela est requis par plusieurs blocs, notamment par les blocs arithmétiques comme `Add` par exemple. Un type scalaire de base est donc un sous-type de tout vecteur dont le type de base est super-type du type scalaire.

La relation d'ordre partiel $\preceq$ est donc spécifiée par le script Coq suivant conformément aux contraintes de typage (de T/R1 à T/R8) décrites dans la section 7.3.2.2.1.

```
Definition le : Data.type-> type -> type-> Prop :=
    fun d x y =>
      match x, y with
        |basic t1, basic t2 => le_basic d t1 t2
        |array l1 t1, array l2 t2 => (le_dim tt l1 l2) /\ (le_basic d t1 t2)
        |undef , _
        |_, Error => True
        |_, any =>
          match x with
            |custom _ |basic _ |array _ _ |any |undef => True
            |_ => False
          end
        |custom n1, custom n2 => n1=n2
        |basic t1, array l t2 => le_basic d t1 t2
        |_,_ => False
      end.
```

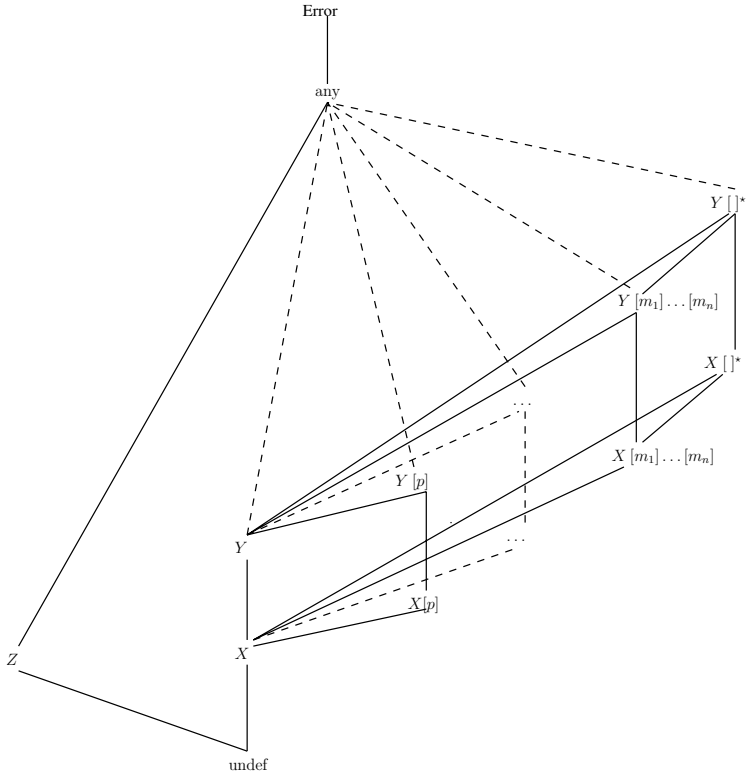


FIG. 7.5 – *Diagramme de Hasse de la famille de treillis des types en* GENEAUTO

La figure 7.5 contient le diagramme de Hasse représentant le poset $\langle \Delta, \preceq \rangle$. Les notations X et Y représentent des types scalaires de base (de `basicType`) et Z dénote le reste des types scalaires (`custom`). Afin de simplifier la représentation, nous avons distingué dans la figure 7.6 le treillis des types scalaires. Ainsi, le sous-typage en GENEAUTO peut être directement déduit des diagrammes de Hasse présentés dans les figures 7.5 et 7.6.

Le poset $\langle \Delta, \preceq \rangle$, muni des éléments suivants, est une famille de treillis notée $\{\theta_d\}_{d \in unit}$.

– $\perp$ la plus petite borne inférieure de la famille de treillis $\{\theta_d\}_{d \in unit}$ qui vaut `undef` ;

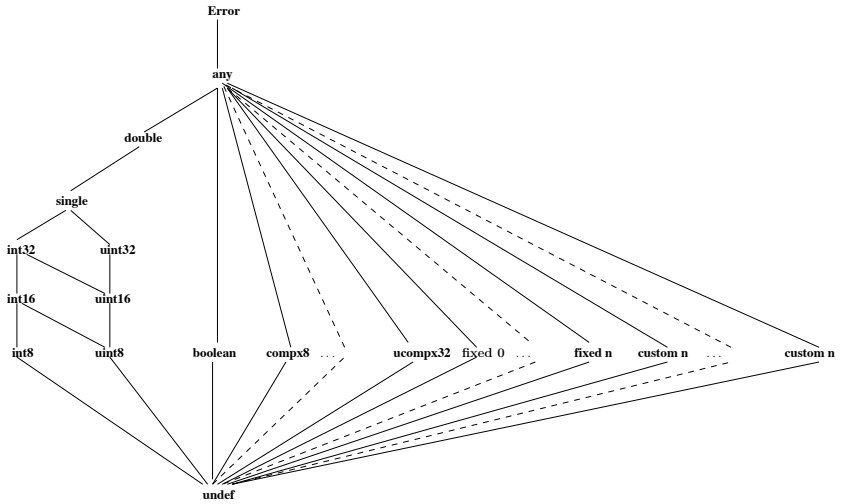


FIG. 7.6 – *Diagramme de Hasse de la famille de treillis des types scalaires en* GENEAUTO

```
Definition bot := undef.
```

– $\top$ la plus grande borne supérieure de la famille de treillis $\{\theta_d\}_{d \in unit}$ qui vaut `Error`;

```
Definition top := Error.
```

– $\sqcup$ l'opérateur d'union qui correspond au plus petit super-type de deux types. Définissons dans un premier temps `superBasic` qui désigne le plus petit super-type de deux types de base incomparables par la relation $\preceq$. Le type `int16` est le plus petit super-type de `int8` et de `uint8` (`int8` $\sqcup$ `uint8`), `int32` = `int16` $\sqcup$ `uint16`, et également `int32` = `uint16` $\sqcup$ `int8`.

```
Definition superBasic : PreOrder.Data.type -> basicType -> basicType ->
    PreOrder.type :=
    fun d x y =>
      match x, y with
        |int8, uint8
        |uint8, int8 => basic int16
        |int16, uint16
        |uint16, int16
        |uint16, int8
        |int8, uint16 => basic int32
        |int8, uint32
        |uint32, int8
        |int32, uint32
        |uint32 , int32
        |int16, uint32
        |uint32, int16 => basic single
        |_, _ => any
      end.
```

Le plus petit super-type de deux vecteurs de dimensions comparables est un vecteur ayant les mêmes dimensions et dont le type de base est égal au plus petit super-type de leurs types de base respectifs. Ainsi, l'opérateur $\sqcup$ est décrit par la définition `join` qui suit.

```
Definition join : PreOrder.Data.type -> PreOrder.type -> PreOrder.type ->
    PreOrder.type :=
    fun d x y =>
      match (PreOrder.dec d x y) with
        |left _ => y
        |right _ =>
          match (PreOrder.dec d y x) with
            |left _ => x
            |right _ =>
              match x, y with
                |basic t1, basic t2 => any
                |array n1 t1, array n2 t2 =>
                  if (le_dim_dec n1 n2) then
                    match (superBasic d t1 t2) with
                      |basic x => array n1 x
                      |_ => any
                    end
                    else any
                |basic t1, array n t2 => match (superBasic d t1 t2) with
                                           | basic x => array n x
                                           |_ => any
                                         end
                |array n t1 , basic t2 => match (superBasic d t1 t2) with
                                            | basic x => array n x
                                            |_ => any
                                          end
                |_,_ => any
              end
          end
      end.
```

– $\sqcap$ l'opérateur binaire d'intersection qui calcule le plus grand sous-type de deux types. Soit, d'abord, `subBasic` la fonction qui calcule le plus grand sous-type de deux types incomparables avec la relation $\preceq$.

```
Definition subBasic : PreOrder.Data.type -> basicType -> basicType ->
    PreOrder.type :=
    fun d x y =>
      match x, y with
        |int32, uint32
        |uint32, int32 => basic uint16
        |int16, uint16
        |uint16, int16
        |uint32, int16
        |int16, uint32 => basic uint8
        |_, _ => undef
      end.
```

Le plus grand sous-type de deux types comparables correspond au plus petit d'entre eux selon la relation $\preceq$. Tandis que le plus grand sous-type de deux types incomparables par $\preceq$ correspond à leur plus grand sous-type commun. C'est ce qui est défini par `subBasic`.
Par conséquent, nous disposons de `uint16` $=$ `int32` $\sqcap$ `uint32`, de `uint8` $=$ `int16` $\sqcap$ `uint16`, et également de `uint8` $=$ `int16` $\sqcap$ `uint32`. Enfin, l'opérateur $\sqcap$ est décrit par la fonction `meet`.

```
Definition meet : PreOrder.Data.type -> PreOrder.type -> PreOrder.type
    -> PreOrder.type :=
  fun d x y =>
    match (PreOrder.dec d x y) with
      |left _ => x
      |right _ =>
        match (PreOrder.dec d y x) with
          |left _ => y
          |right _ =>
            match x, y with
              |basic t1, basic t2 => undef
              |array n1 t1, array n2 t2 =>
                if (le_dim_dec n1 n2) then
                  match (subBasic d t1 t2) with
                    |basic x => array n1 x
                    |_ => undef
                  end
                  else undef
              |basic t1, array n t2 => match (subBasic d t1 t2) with
                                         | basic x => array n x
                                         |_ => undef
                                       end
              |array n t1 , basic t2 => match (subBasic d t1 t2) with
                                          | basic x => array n x
                                          |_ => undef
                                        end
              |_,_ => undef
            end
        end
    end.
```

L'implantation de la famille de treillis des types en GENEAUTO peut être consultée dans le fichier TypeLattice.v.

Nous décrirons, dans ce qui suit, les éléments nécessaires pour effectuer l'inférence de type.

7.3.3 Environnement de l'inférence des types

L'inférence de type d'un circuit donné dépend des types des ports des différents blocs constituant ce circuit, mais aussi de la sémantique des blocs ainsi que des connexions qui les relient.

Le type d'un port donné dépend du bloc auquel il appartient. Dans le cas d'étude de l'inférence de type, la manipulation du paramètre $d \in \mathfrak{D}$ s'avère nécessaire. Celui-ci permet de définir un environnement qui associe pour chaque bloc un type pour chacun de ses ports.
Considérons donc un environnement γ qui associe un type pour chaque port d'un bloc et un environnement Y qui associe pour chaque bloc l'environnement γ.

Les types sont associés aux ports d'entrée et de sortie de tout bloc. Comme cité précédemment, un utilisateur peut préciser les types des ports d'entrée et de sortie. Toutefois, ils ne sont pas pris en compte par l'algorithme que nous proposons. Ils sont uniquement utilisés pour valider le résultat de l'inférence de type. Des détails supplémentaires sont fournis dans la section 7.8.

Néanmoins, l'algorithme proposé doit être le plus général possible permettant à la fin de l'analyse de valider les initialisations des utilisateurs. Ainsi, l'inférence de type que nous proposons est établie dans deux sens : depuis les entrées vers les sorties du circuit (qualifiée d'inférence en avant) et inversement des sorties vers les entrées du circuit (qualifiée d'inférence en arrière). Il en résulte la définition d'un environnement par sens d'inférence. Mais cette approche nécessite la manipulation d'un treillis dans lequel l'inférence de type

reste monotone quel que soit le sens de la propagation.

Une solution serait de ne manipuler que le treillis des types $\{\theta_d\}_{d\in unit}$. Rappelons que l'inférence en avant calcule le plus petit type que peut prendre chaque port, alors que l'inférence en arrière calcule le plus grand type que peut prendre chaque port. Par conséquent, le résultat de l'analyse serait obtenu par le calcul du plus petit point fixe pour l'inférence de type en avant et par le calcul du plus grand point fixe pour l'inférence en arrière. Ce qui demanderait une extension du cadre général, notamment en ce qui concerne les preuves de terminaison de l'algorithme et de croissance de la fonction qui décrit l'inférence en arrière.

Afin de garder la généricité du cadre général proposé et de calculer seulement le plus petit point fixe quelle que soit l'analyse effectuée, nous proposons de manipuler la famille de treillis des types $\{\theta_d\}_{d\in unit}$ pour l'inférence en avant et la famille de treillis inverse (voir section 5.3.3.2), notée $\{\theta_d^{-1}\}_{d\in unit}$, pour l'inférence en arrière.

7.3.3.1 Environnement des ports

L'environnement des ports pour l'inférence en avant, noté γ^-, consiste à associer pour chaque port d'un bloc donné un type du domaine Δ. Dans notre analyse, nous distinguons l'environnement des ports d'entrée, noté γ_{in}^-, et l'environnement des ports de sortie, noté γ_{out}^-.

Les deux environnements ont la même définition. L'environnement γ^-, pour une taille p donnée (nombre de ports d'entrée ou de sortie), est une famille de treillis définie comme suit.

Définition 7.3.1. *Environnement des ports pour l'inférence en avant*

$$\gamma^- \triangleq \{\{n \in \mathbb{N} | \, n < p\} \rightarrow \{\theta_{f(n)}\}\}_{p\in\mathbb{N},\, f\in\mathbb{N}\rightarrow unit}$$

où p est le nombre de ports dans un bloc donné. $(p\,,\,f)$ est le paramètre de la famille de treillis γ^-. Ce paramètre est de type $\mathbb{N}\times(\mathbb{N}\rightarrow unit)$. Notons ce type par $\mathfrak{D}'$.

Le script Coq qui implante γ^- est :

```
Module TypingPortEnvironmentFwd := MakeDepEnvironment(SimTypingLattice).
```

avec `SimTypingLattice` la définition en Coq de $\{\theta_d\}_{d\in unit}$.

Réciproquement, l'environnement des ports pour une inférence en arrière, noté γ^+, se définit en utilisant la famille du treillis inverse de $\{\theta_d\}_{d\in unit}$, notée $\{\theta_d^{-1}\}_{d\in unit}$. γ^+ associe un type de Δ pour chaque port d'un bloc donné.

Définition 7.3.2. *Environnement des ports pour l'inférence en arrière*

$$\gamma^+ \triangleq \{\{n \in \mathbb{N} | \, n < p\} \rightarrow \{\theta_{f(n)}^{-1}\}\}_{p\in\mathbb{N},\, f\in\mathbb{N}\rightarrow unit}$$

$\{\theta_d^{-1}\}_{d\in unit}$ est spécifié par `SimTypingLatticeInv` en Coq. Ainsi, γ^+ est implanté par le script suivant.

```
Module TypingPortEnvironmentBwd := MakeDepEnvironment(SimTypingLatticeInv).
```

Ici, aussi, nous distinguons l'environnement des ports d'entrée, noté γ^+_{in}, et l'environnement des ports de sortie, noté γ^+_{out}.

Pour des raisons de simplicité, les ports d'entrée et de sortie sont distingués par leurs identifiants entiers comme pour les blocs. Pour l'inférence en avant, le résultat des calculs concerne les ports de sortie, et inversement, pour l'inférence en arrière le résultat concerne les ports d'entrée.

7.3.3.2 Environnement des blocs

L'environnement des blocs, noté Y, consiste à associer un type du domaine Δ à chaque port de chaque bloc d'un circuit de taille s. Autrement dit, il s'agit d'associer à chaque bloc, en fonction de l'inférence, un environnement des ports. Ainsi, comme pour l'environnement des ports, nous manipulons deux environnements de blocs pour chaque inférence : un environnement des blocs qui associe pour chaque bloc un environnement des ports d'entrée et un autre qui associe pour chaque bloc un environnement des ports de sortie. Les deux environnements des blocs sont définis de la même manière.

Le paramètre de l'environnement Y est de type $\mathbb{N} \times (\mathbb{N} \to \mathfrak{D}')$.

Ainsi, l'environnement des blocs pour l'inférence en avant, noté Y^-, est défini comme suit.

Définition 7.3.3. *Environnement des blocs pour l'inférence en avant*

$$\mathrm{Y}^- \triangleq \{\{n \in \mathbb{N} | \, n < s\} \to \{\gamma^-{}_{f(n)}\}\}_{s \in \mathbb{N}, f \in \mathbb{N} \to \mathfrak{D}'}$$

L'environnement des blocs Y^- est défini en Coq par :

```
Module TypingBlockEnvironmentFwd :=
  MakeDepEnvironment(TypingPortEnvironmentFwd).
```

De manière symétrique, l'environnement des blocs pour l'inférence en arrière, noté Y^+, est défini en utilisant l'environnement des ports γ^+.

Définition 7.3.4. *Environnement des blocs pour l'inférence en arrière*

$$\mathrm{Y}^+ \triangleq \{\{n \in \mathbb{N} | \, n < s\} \to \{\gamma^+{}_{f(n)}\}\}_{s \in \mathbb{N}, f \in \mathbb{N} \to \mathfrak{D}'}$$

En Coq, il est défini par :

```
Module TypingBlockEnvironmentBwd :=
  MakeDepEnvironment(TypingPortEnvironmentBwd).
```

7.3.3.3 Initialisation de l'environnement

En vertu de l'absence de ports d'entrée (comme le bloc `In`) ou de sortie (comme le bloc `Out`) dans certains blocs SIMULINK, nous considérons que tous les blocs possèdent au moins un port d'entrée et un port de sortie. Cela permet d'homogénéiser l'inférence de type sur l'ensemble des blocs traités et de vérifier la propriété d'adjonction qui sera présentée dans la section 7.4.1.

Nous introduisons donc des ports virtuels dont l'unique but est de définir les fonctions de typage du bloc en question et de propager les types calculés. En outre, il est nécessaire de disposer de ces ports pour récupérer le résultat du calcul qui sera utilisé pour vérifier le bon typage des blocs en fin d'analyse. L'ajout de ports virtuels n'a aucun impact sur les autres outils élémentaires, vu qu'ils ne sont reliés à aucun signal.

Définition 7.3.5. *Ports virtuels*
Un port virtuel est un port qui n'existe pas réellement. Il est ajouté explicitement à l'entrée (respectivement à la sortie) de chaque bloc ne possédant pas de ports d'entrée (respectivement de ports de sortie). Ce port n'est relié à aucun signal et n'a pas d'existence réelle dans le circuit.

L'initialisation des environnements de blocs dépend du sens de l'inférence. Pour une inférence en avant, dans l'environnement initial des blocs, noté Y_0^-, les ports d'entrée virtuels sont initialisés à `any`, et les autres ports d'entrée sont initialisés à la plus petite borne inférieure $\bot$ de la famille de treillis $\{\theta_d\}_{d\in unit}$, ce qui correspond à `undef`. De même, pour une inférence en arrière, dans l'environnement initial, noté Y_0^+, les ports de sortie virtuels sont initialisés à `any`, et les autres ports de sortie sont initialisés à la plus petite borne inférieure $\bot$ de la famille de treillis inverse $\{\theta_d^{-1}\}_{d\in unit}$, ce qui correspond à $\top$ (ou `Error`) dans $\{\theta_d\}_{d\in unit}$.

Exemple 7.3.1. *Considérons le circuit de la figure 7.7. Il s'agit d'un circuit simple qui calcule la somme de deux entrées : un vecteur et un scalaire.*

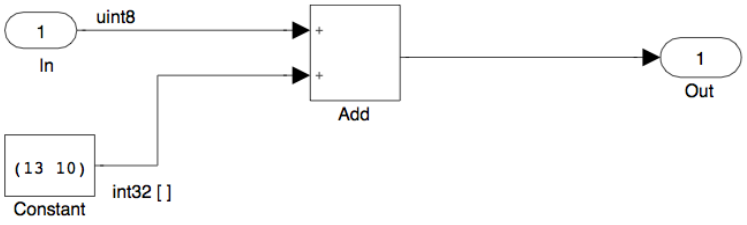
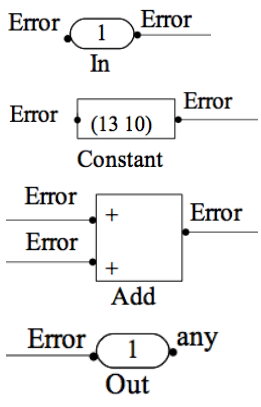


FIG. 7.7 – *Exemple de circuit à typer*

Pour simplifier la représentation des environnements de ports, nous référençons les ports d'entrée (ou de sortie) par les blocs auxquels ils appartiennent. L'environnement initial pour l'inférence en avant correspond à :

$$Y_0^- = \left\{ \begin{array}{l} \text{any} \rightarrow \boxed{1}\ \text{In} \rightarrow \text{undef} \\ \text{any} \rightarrow \boxed{(13\ 10)}\ \text{Constant} \rightarrow \text{undef} \\ \text{undef},\ \text{undef} \rightarrow \boxed{+\ +}\ \text{Add} \rightarrow \text{undef} \\ \text{undef} \rightarrow \boxed{1}\ \text{Out} \rightarrow \text{undef} \end{array} \right.$$

De même, l'environnement initial pour l'inférence en arrière correspond à :

$Y_0^+ =$ 

Ainsi, à chaque port p est associé un couple de types $\langle \tau^-, \tau^+ \rangle$ *où* $\tau^- = \gamma^-(p)$ *et* $\tau^+ = \gamma^+(p)$*. Initialement, ce couple est donné par* $\langle \gamma_0^-, \gamma_0^+ \rangle$ *avec les valeurs données plus haut.*

7.4 FONCTIONS DE TYPAGE

Chaque bloc possède ses propres caractéristiques de typage. Il faut donc définir une fonction de typage par bloc. Le but de cette fonction est de calculer pour chaque bloc du circuit le type de ses ports d'entrée par inférence en arrière et le type de ses ports de sortie par inférence en avant. Nous définissons alors deux fonctions de typage pour tout bloc du circuit :

- la fonction de typage des ports de sortie en fonction des ports d'entrée d'un bloc, notée $\mathcal{F}^-$;
- et la fonction de typage duale qui calcule les types des ports d'entrée en fonction des ports de sortie , $\mathcal{F}^+$.

La fonction $\mathcal{F}^-$ calcule, pour un bloc donné, la borne inférieure des types de ses ports de sortie, c'est-à-dire, le plus petit type dans la famille de treillis $\{\theta_d\}_{d \in unit}$, que peuvent prendre les sorties. De manière symétrique, la fonction $\mathcal{F}^+$ calcule, pour un bloc donné, la borne supérieure des types de ses ports d'entrée, c'est-à-dire, le plus grand type dans la famille de treillis $\{\theta_d\}_{d \in unit}$, que peuvent prendre les entrées.

De plus, les fonctions de typage $\mathcal{F}^-$ et $\mathcal{F}^+$ doivent être décrites de façon à ce qu'elles soient cohérentes. Soit b un bloc donné, notons dans ce qui suit par X la séquence de ses types d'entrée, par Y la séquence de ses types de sortie et par α le type du paramètre du bloc s'il en accepte un.

7.4.1 Propriété d'adjonction

Afin de propager correctement les informations de type dans le circuit, les fonctions de typage $\mathcal{F}^-$ et $\mathcal{F}^+$ doivent être cohérentes. Cette cohérence est garantie par la propriété d'adjonction définie comme suit.

Propriéte 7.4.1. *Adjonction*
$\mathcal{F}^-(X) \preceq_1 Y \Leftrightarrow \mathcal{F}^+(Y) \preceq_2 X$
tel que,
$\mathcal{F}^- : \gamma_{in}^- \rightarrow \gamma_{out}^-$
$\mathcal{F}^+ : \gamma_{out}^+ \rightarrow \gamma_{in}^+$

avec γ_{in}^{-} l'environnement des ports d'entrée pour l'inférence en avant d'un bloc à p entrées et γ_{out}^{-} l'environnement des ports de sortie pour l'inférence en avant du même bloc à n sorties. De même, γ_{in}^{+} l'environnement des ports d'entrée pour l'inférence en arrière d'un bloc à p entrées et γ_{out}^{+} l'environnement des ports de sortie pour l'inférence en arrière du même bloc à n sorties. X représente les entrées du bloc considéré et Y représente ses sorties.

La relation d'ordre $\preceq_1$ correspond à la relation $\preceq_{\gamma_{out}^{-}}$ définie dans la famille de treillis γ_{out}^{-} et la relation $\preceq_2$ correspond à la relation $\preceq_{\gamma_{in}^{+}}$ définie dans γ_{in}^{+}. Ces relations, étant définies l'une sur une famille de treillis et l'autre sur la famille de treillis inverse, sont des relations duales l'une de l'autre.

Pour clarifier les fonctions de typage et simplifier l'écriture des preuves dans cette section, nous utilisons une seule relation d'ordre définie sur des environnements de ports de tailles différentes.

Soient γ_{in}^{-} et γ_{out}^{-} les environnements de ports d'entrée et de sortie respectivement dont les tailles respectives sont p et n. $\preceq_1$ est la relation d'ordre définie dans γ_{in}^{-} et $\preceq_2$ correspond à la relation d'ordre définie dans γ_{out}^{-}. Ainsi, la propriété d'adjonction est redéfinie comme suit.

Propriéte 7.4.2. *Adjonction*
$\mathcal{F}^{-}(X) \preceq_1 Y \Leftrightarrow X \preceq_2 \mathcal{F}^{+}(Y)$
Avec,
$\mathcal{F}^{-} : \gamma_{in}^{-} \rightarrow \gamma_{out}^{-}$
$\mathcal{F}^{+} : \gamma_{out}^{-} \rightarrow \gamma_{in}^{-}$

L'adjonction assure que, pour un bloc donné, les types des ports de sortie calculés par la fonction $\mathcal{F}^{-}$ sont des sous-types respectifs de leurs types explicités par l'utilisateur, si et seulement si les types des ports d'entrée explicités de ce bloc sont des sous-types respectifs de leurs types calculés par la fonction de typage $\mathcal{F}^{+}$.

Elle permet, d'une part, de garantir la propagation des erreurs de type si elles ont lieu et, d'autre part, de limiter le coût de vérification de la correction du typage. Au lieu de vérifier que le type de chaque port de chaque bloc du circuit est correct, grâce à l'adjonction, il suffit de vérifier la correction de typage des blocs qui constituent les entrées ou les sorties du circuit.

Pour simplifier la présentation des fonctions de typage ainsi que les preuves associées, nous travaillons dans ce qui suit directement sur l'image de l'environnement. Cette image correspond à la séquence des types respectifs des ports. Autrement dit, lors de la représentation des fonctions de typage, il suffit de considérer les types car les ports ne changent pas et n'interviennent pas dans les calculs. Ainsi, les relations d'ordre $\preceq_1$ et $\preceq_2$ correspondent, en fonction du nombre d'éléments à comparer, soit à la relation $\preceq$ du treillis $\{\theta_d\}_{d\in\texttt{unit}}$, soit à la relation $\preceq_{\theta^n}$ du produit cartésien de $\{\theta_d\}_{d\in\texttt{unit}}$ facteur n.

Nous citons quelques propriétés de base sur l'adjonction. Quel que soit le bloc, la fonction de typage en avant d'une entrée indéfinie reste indéfinie.

Propriéte 7.4.3. *Élément identité pour $\mathcal{F}^{-}$*

$$\mathcal{F}^{-}(\bot) = \bot$$

Pour tout bloc, la fonction de typage en arrière d'une sortie erronée donne une entrée de type erroné.

Propriéte 7.4.4. *Élément identité pour* $\mathcal{F}^+$

$$\mathcal{F}^+(\top) = \top$$

Nous donnons dans ce qui suit quelques définitions accompagnées de leurs propriétés d'adjonction.

7.4.1.1 Composition de blocs

L'adjonction de composition est applicable dans le cas de séquences de blocs pour construire un seul bloc composite.

Définition 7.4.1. *Composition*
Soient les deux couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ *et* $(\mathcal{F}_2^-, \mathcal{F}_2^+)$*, les fonctions de typage respectives de deux blocs A et B. Le couple de fonctions de typage du bloc composé de A connecté à l'entrée du bloc B correspond à* $(\mathcal{F}_2^- \circ \mathcal{F}_1^-, \mathcal{F}_1^+ \circ \mathcal{F}_2^+)$.

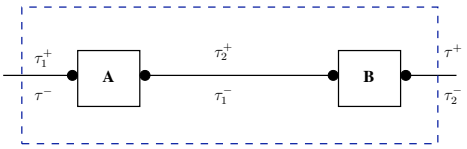


FIG. 7.8 – *Composition de deux blocs*

Par exemple, dans le circuit de la figure 7.8, τ_1^- est le type de la sortie du bloc A obtenu par $\mathcal{F}_1^-(\tau^-)$ et τ_1^+ le type de l'entrée du bloc A obtenu par la fonction $\mathcal{F}_1^+(\tau_2^+)$. Il en est de même pour le bloc B. Lorsque la sortie du bloc A est connectée à l'entrée du bloc B, le type de la sortie de cette combinaison (qui est celle de B) est le type obtenu par $\mathcal{F}_2^- \circ \mathcal{F}_1^-$, selon l'inférence en avant. Le type de l'entrée de cette combinaison (qui est celle du bloc A) est obtenu par la fonction $\mathcal{F}_1^+ \circ \mathcal{F}_2^+$, selon l'inférence en arrière.

Propriéte 7.4.5. *Adjonction de composition*
Si les deux couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ *et* $(\mathcal{F}_2^-, \mathcal{F}_2^+)$ *respectent la propriété d'adjonction, alors le couple de fonctions issu de leur composition respecte également l'adjonction.*

Preuve. Soient les couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ et $(\mathcal{F}_2^-, \mathcal{F}_2^+)$ liés respectivement par l'adjonction. Nous avons donc :
$\mathcal{F}_1^-(X) \preceq Y \Leftrightarrow X \preceq \mathcal{F}_1^+(Y)$
$\mathcal{F}_2^-(X) \preceq Y \Leftrightarrow X \preceq \mathcal{F}_2^+(Y)$

Remplaçons X dans la seconde équivalence par $\mathcal{F}_1^-(X)$. Il en résulte :
$\mathcal{F}_2^-(\mathcal{F}_1^-(X)) \preceq Y \Leftrightarrow \mathcal{F}_1^-(X) \preceq \mathcal{F}_2^+(Y)$.
Remplaçons Y dans la première équivalence par $\mathcal{F}_2^+(Y)$. Il en résulte :
$\mathcal{F}_1^-(X) \preceq \mathcal{F}_2^+(Y) \Leftrightarrow X \preceq \mathcal{F}_1^+(\mathcal{F}_2^+(Y))$.
Par transitivité de l'équivalence, nous obtenons :
$(\mathcal{F}_2^- \circ \mathcal{F}_1^-)(X) \preceq X \Leftrightarrow X \preceq (\mathcal{F}_1^+ \circ \mathcal{F}_2^+)(Y)$.

7.4.1.2 Superposition de blocs

Définition 7.4.2. *Superposition*
Soient les deux couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ *et* $(\mathcal{F}_2^-, \mathcal{F}_2^+)$*, les fonctions de typage respectives de deux blocs A et B. Le couple de fonctions de typage de la superposition de ces deux blocs correspond à* $(\mathcal{F}_2^- \sqcup \mathcal{F}_1^-, \mathcal{F}_2^+ \sqcap \mathcal{F}_1^+)$.

La superposition est utile dans le cas des blocs constitués, comme cela est illustré dans la figure 7.9.

Par exemple, τ_1^- est le type de la sortie du bloc A obtenu par $\mathcal{F}_1^-(\tau^-)$ et τ_1^+ le type de l'entrée du bloc A obtenu par la fonction $\mathcal{F}_1^+(\tau^+)$. Le même raisonnement est valable pour le bloc B. Lorsque A et B sont superposés, le type de la sortie de cette superposition est celui obtenu par $\mathcal{F}_2^- \sqcup \mathcal{F}_1^-$, selon l'inférence en avant. Le type de l'entrée de cette superposition est obtenu par la fonction $\mathcal{F}_1^+ \sqcap \mathcal{F}_2^+$, selon l'inférence en arrière.

Un cas d'utilisation de cette propriété sera donné plus en détails dans la section 7.4.6.

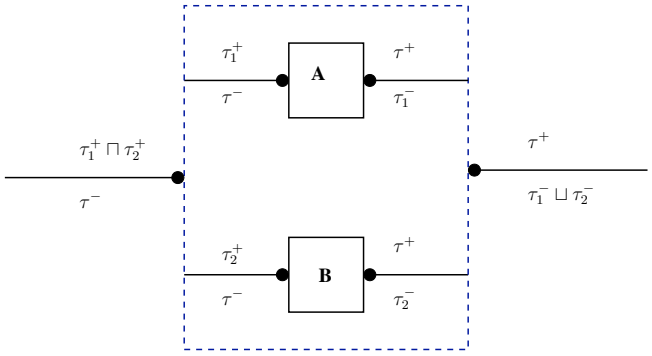


FIG. 7.9 – *Superposition de deux blocs*

Propriété 7.4.6. *Adjonction de superposition*
Si les deux couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ *et* $(\mathcal{F}_2^-, \mathcal{F}_2^+)$ *respectent la propriété d'adjonction, alors le couple de fonctions issu de leur superposition respecte également l'adjonction.*

Preuve. Soient les couples $(\mathcal{F}_1^-, \mathcal{F}_1^+)$ et $(\mathcal{F}_2^-, \mathcal{F}_2^+)$ liés respectivement par l'adjonction. Nous avons donc :
$\mathcal{F}_1^-(X) \preceq Y \Leftrightarrow X \preceq \mathcal{F}_1^+(Y)$
$\mathcal{F}_2^-(X) \preceq Y \Leftrightarrow X \preceq \mathcal{F}_2^+(Y)$
En prenant respectivement la conjonction des membres gauches et droits et en exploitant la définition de $\sqcup$ et $\sqcap$, il en résulte $\mathcal{F}_1^-(X) \sqcup \mathcal{F}_2^-(X) \preceq Y \Leftrightarrow X \preceq \mathcal{F}_1^+(Y) \sqcap \mathcal{F}_2^+(Y)$.

Afin de factoriser les fonctions pour les blocs dont le typage est identique, celles-ci sont décrites par classes de blocs SIMULINK. Nous avons considéré, dans cette étude expérimentale du typage, quelques blocs SIMULINK illustrés par la figure 7.10.

7.4.2 Blocs identité

Les blocs identité sont tous les blocs dont les sorties sont de mêmes types que les entrées respectives. Ce bloc n'a aucun rôle fonctionnel, il est constitué de fils reliant respectivement chaque entrée à une sortie. En réalité, ce bloc n'existe pas dans les librairies SIMULINK. Nous l'avons défini pour exploiter ses fonctions de typage lors de la superposition avec d'autres blocs. Cela permet de simplifier la définition des fonctions de typage de blocs complexes. Ainsi, les preuves d'adjonction de blocs complexes seront construites en exploitant les preuves des blocs qui les composent.

Les fonctions de typage de ce dernier sont données comme suit.

Bloc	règle de typage
 	$\alpha \times \ldots \times \alpha \rightarrow \alpha$, $\alpha \notin \{\texttt{boolean}, \texttt{boolean}\,[m_1]\ldots[m_n]\}$
	$\alpha \times \ldots \times \alpha \rightarrow \alpha, \quad \alpha \in \{\texttt{boolean}, \texttt{boolean}\,[m_1]\ldots[m_n]\}$
 	α, pas de type d'entrée, mais le type α du paramètre doit être spécifié
	Pas de type de sortie, mais le type α du paramètre doit être spécifié
	$\alpha \times \beta \rightarrow \sigma$, avec α le type du paramètre, β le type de l'entrée

FIG. 7.10 – *Liste des blocs* SIMULINK *étudiés pour le typage*

Définition 7.4.3. *Fonction de typage en avant d'un bloc identité*

$$\mathcal{F}^{-}(X) = X$$

La définition en COQ de cette fonction, dans l'environnement γ^{-}, est décrite par le code suivant.

```
Definition identityFunctionFwd
    (b : BlockBaseType)
    (currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    fun blockPort =>
      if (leb (b.(outputPortsSize)) blockPort)
        then
          SimTypingLattice.top
        else
          if (eq_nat_dec b.(outputPortsSize) b.(outputPortsSize))
            then
              currentInputTypes blockPort
              else
                SimTypingLattice.top.
```

Définition 7.4.4. *Fonction de typage en arrière d'un bloc identité*

$$\mathcal{F}^{+}(Y) = Y$$

La définition en COQ, dans l'environnement γ^{+}, de la fonction $\mathcal{F}^{+}$ est décrite par le script suivant.

```
Definition identityFunctionBwd
    (b : BlockBaseType)
    (currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
    fun blockPort =>
      if (leb (b.(inputPortsSize)) blockPort)
        then SimTypingLattice.bot
        else
          if (eq_nat_dec b.(outputPortsSize) b.(outputPortsSize))
            then
              currentOutputTypes blockPort
              else
                SimTypingLattice.bot.
```

La preuve que ces fonctions sont liées par l'adjonction est triviale. Elle consiste simplement à exploiter les définitions.

7.4.3 Blocs source

La classe des blocs source regroupe les blocs, ne possédant pas de ports d'entrée et disposant d'une ou plusieurs sorties, de la bibliothèque `Source`. Dans la version actuelle du typage, nous ne considérons que les blocs source à une seule sortie. Le type du port de sortie d'un bloc source dépend du type de son paramètre d'initialisation α. Par exemple, la fonction de typage du bloc `In` est similaire à celle du bloc `Constant`. Les deux blocs ne disposent pas de ports d'entrée mais ont un paramètre dont le type, noté α, influe sur le type de sortie.

7.4.3.1 Fonction de typage en avant

La fonction de typage $\mathcal{F}^-$ d'un bloc source renvoie un type indéfini (`undef`), si le type du paramètre n'est pas précisé, et le type α, si le type du paramètre est précisé et ne vaut pas `Error`. Dans le cas contraire, la fonction renvoie `Error`.

Définition 7.4.5. *Fonction de typage en avant d'un bloc source*

$$\mathcal{F}^-(X) = \begin{cases} \bot & si\ (X \preceq \bot)\ sinon \\ \alpha & si\ (X \preceq any)\ sinon \\ \top & \end{cases}$$

La définition en Coq de cette fonction, dans l'environnement γ^-, correspond au code suivant.

```
  Definition sourceFunctionFwd
    (b : BlockBaseType)
    (currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    fun blockPort =>
      if (eq_nat_dec blockPort 0)
        then
          if (SimTypingLattice.PreOrder.dec tt (currentInputTypes 0)
               SimTypingLattice.bot)
            then SimTypingLattice.bot
            else
              let alpha:=
                match b.(parameters) with
                  |(initial t) :: nil => t
                  | _ => SimTypingLattice.top
                end
                in
                if SimTypingLattice.PreOrder.dec tt (currentInputTypes blockPort)
                     any
                  then alpha
                  else SimTypingLattice.top
        else SimTypingLattice.top.
```

7.4.3.2 Fonction de typage en arrière

La fonction de typage $\mathcal{F}^+$ des blocs source consiste à renvoyer une erreur de type si la sortie est de type `Error`, le type `any` si le type du paramètre est un sous-type de de la sortie, un type indéfini sinon.

Définition 7.4.6. *Fonction de typage en arrière d'un bloc source*

$$\mathcal{F}^+(Y) = \begin{cases} \top & si\ (\top \preceq Y)\ sinon \\ any & si\ (\alpha \preceq Y)\ sinon \\ \bot & \end{cases}$$

La fonction de typage $\mathcal{F}^+$ des blocs source est spécifiée, dans l'environnement γ^+, par le script Coq suivant.

```
Definition sourceFunctionBwd
    (b : BlockBaseType)
    (currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
    fun blockPort =>
      if (eq_nat_dec blockPort 0)
        then
          if SimTypingLatticeInv.PreOrder.dec tt (currentOutputTypes blockPort)
               SimTypingLatticeInv.bot
            then SimTypingLatticeInv.bot
            else
              let alpha:=
                match b.(parameters) with
                  |(initial t) :: nil => t
                  | _ => SimTypingLatticeInv.top
                end
                in
                if SimTypingLatticeInv.PreOrder.dec tt (currentOutputTypes
                     blockPort) alpha
                  then any
                  else SimTypingLatticeInv.top
        else SimTypingLattice.top.
```

7.4.3.3 Correction des fonctions de typage des blocs source

Nous utilisons la propriété d'adjonction pour vérifier la cohérence des fonctions de typage l'une par rapport à l'autre.

Preuve. ($\Rightarrow$).
Cas 1 : $X \preceq \bot$. Cela signifie que $X = \bot$, alors $\bot \preceq \mathcal{F}^+(Y)$ est vrai.
Cas 2 : $X \preceq any$. Cela se traduit par $\mathcal{F}^-(X) = \alpha$ et $\alpha \preceq Y$. Il en résulte $\mathcal{F}^+(Y) = any$. Ainsi, $X \preceq any$ est vérifié par hypothèse.
Cas 3 : $X = \top$. Nous avons donc $\mathcal{F}^-(X) = \top$. Donc $\top \preceq Y$, ce qui entraîne $\mathcal{F}^+(Y) = \top$. Par conséquent, $X \preceq \top$ est vérifiée.
($\Leftarrow$)
Cas 1 : $\top \preceq Y$. Cela signifie $Y = \top$. Ainsi, $\mathcal{F}^-(X) \preceq Y$.
Cas 2 : $\alpha \preceq Y$. Il en résulte $X \preceq any$. Cela implique que $\mathcal{F}^-(X) = \alpha$ et donc $\alpha \preceq Y$ est vérifiée par hypothèse.
Cas 3 : $\neg(\top \preceq Y \wedge \alpha \preceq Y)$. D'une part, $\mathcal{F}^+(Y) = \bot$, et donc nous avons par hypothèse $X \preceq \bot$. D'autre part, $\mathcal{F}^-(X) = \bot$ en exploitant cette hypothèse. Ce qui implique $\mathcal{F}^-(X) \preceq Y$.

7.4.4 Blocs cible

Il s'agit principalement des blocs de la bibliothèque `Sink` de SIMULINK. La classe des blocs cible regroupe tous le blocs qui disposent d'une entrée et n'ont pas de sortie.

Un bloc cible peut contenir un paramètre qui détermine le type des entrées admises par le bloc, c'est par exemple le cas du bloc `Out` qui peut avoir une valeur initiale pour son entrée.

7.4.4.1 Fonction de typage en avant

La fonction de typage $\mathcal{F}^-$ des blocs cible consiste à associer à la sortie le type `undef` si le type de l'entrée n'est pas défini, le type `any` si le type de l'entrée (différent de `undef`) est un sous-type de α, une erreur de type (`Error`) sinon.

Définition 7.4.7. *Fonction de typage en avant d'un bloc cible*

$$\mathcal{F}^-(X) = \begin{cases} \bot & si\ (X \preceq \bot)\ sinon \\ any & si\ (X \preceq \alpha)\ sinon \\ \top & \end{cases}$$

Cette fonction est spécifiée, dans l'environnement γ^-, en Coq par le script :

```
Definition sinkFunctionFwd
    (b: BlockBaseType)
    ( currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    fun blockPort =>
      if (leb (b.(outputPortsSize)) blockPort)
        then SimTypingLattice.top
        else
          if (SimTypingLattice.PreOrder.dec tt (currentInputTypes blockPort)
               SimTypingLattice.bot)
            then SimTypingLattice.bot
            else
              let alpha:=
                match b.(parameters) with
                  |(initial t) :: nil => t
                  | _ => SimTypingLattice.top
                end
                in
                if (SimTypingLattice.PreOrder.dec tt (currentInputTypes blockPort)
                      alpha)
                  then any
                  else SimTypingLattice.top
  else SimTypingLattice.top.
```

7.4.4.2 Fonction de typage en arrière

Le type de l'entrée d'un bloc cible par $\mathcal{F}^+$ est `Error` si le type de la sortie est `Error`, α si la sortie est de type `any`, `undef` sinon.

Définition 7.4.8. *Fonction de typage en arrière d'un bloc cible*

$$\mathcal{F}^+(Y) = \begin{cases} \top & si\ (\top \preceq Y)\ sinon \\ \alpha & si\ (any \preceq Y)\ sinon \\ \bot & \end{cases}$$

L'implantation Coq correspondant à la fonction de typage $\mathcal{F}^+$, dans l'environnement γ^+, est la suivante.

```
Definition sinkFunctionBwd
   (b: BlockBaseType)
   ( currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
   fun blockPort =>
     if (leb (b.(inputPortsSize)) blockPort)
       then SimTypingLatticeInv.bot
       else
         if (SimTypingLatticeInv.PreOrder.dec tt (currentOutputTypes blockPort)
              SimTypingLatticeInv.bot)
           then SimTypingLatticeInv.bot
           else
             let alpha:=
               match b.(parameters) with
                 |(initial t) :: nil => t
                 | _ => SimTypingLatticeInv.top
               end
               in
               if (SimTypingLatticeInv.PreOrder.dec tt (currentOutputTypes
                    blockPort) any)
                 then alpha
                 else SimTypingLatticeInv.top
   else SimTypingLatticeInv.top.
```

7.4.4.3 Correction des fonctions de typage des blocs cible

En utilisant la propriété d'adjonction, nous vérifions la cohérence des fonctions de typage des blocs cible définies précédemment.

Preuve. ($\Rightarrow$)
Cas 1 : $X \preceq \bot$. Nous avons donc $X = \bot$. Il en résulte alors $X \preceq \mathcal{F}^+(Y)$ qui est toujours vérifiée.
Cas 2 : $X \preceq \alpha$. D'une part, $\mathcal{F}^-(X) = any$ et $any \preceq Y$. D'autre part, $\mathcal{F}^+(Y) = \alpha$. Ainsi, $X \preceq \alpha$ est correcte par hypothèse.
Cas 3 : $\neg(X \preceq \bot \wedge X \preceq \alpha)$. Nous avons $\mathcal{F}^-(X) = \top$. Il en résulte alors $\top \preceq Y$, ce qui signifie $Y = \top$. Ainsi, $\mathcal{F}^+(Y) = \top$. Par conséquent, $X \preceq \top$ est correcte dans tous les cas.
($\Leftarrow$)
Cas 1 : $\top \preceq Y$. Une seule solution à cette inégalité : $Y = \top$. Ce qui signifie $\mathcal{F}^-(X) \preceq \top$.
Cas 2 : $any \preceq Y$. Nous avons alors $\mathcal{F}^+(Y) = \alpha$. Supposons donc $X \preceq \alpha$. Nous avons alors $\mathcal{F}^-(X) = any$. Donc, $\mathcal{F}^-(X) \preceq Y$ est correct par hypothèse.
Cas 3 : $\neg(\top \preceq Y \wedge any \preceq Y)$. Soit $X \preceq \bot$ car $\mathcal{F}^+(Y) = \bot$; soit $\mathcal{F}^-(X) = \bot$. Par conséquent, $\mathcal{F}^-(X) \preceq Y$ est correct.

7.4.5 Blocs combinatoires

Cette classe de blocs regroupe tous les blocs SIMULINK arithmétiques (`Add`, `Product`, `Max`, etc.) et logiques (`And`, `Or`, etc.). Nous avons choisi de typer les blocs combinatoires arithmétiques (par exemple `Add` et `Product`) ainsi que les blocs logiques (tels que `And` et `Or`) par une fonction de typage générique. Ces blocs, selon qu'ils possèdent une ou plusieurs entrées, seront typés différemment.

Notons ici, que pour un bloc `Product` qui se comporte comme une division (dans le cas où le paramètre de division est spécifié), la valeur du diviseur dans le cas où elle est nulle n'est pas vérifiée. Une erreur sera alors détectée à l'exécution et non à la phase de typage.

7.4.5.1 Fonction de typage en avant

Considérons l'exemple du bloc `Add`. Il additionne plusieurs entrées de types numériques $(\tau_1, \ldots, \tau_n)$ et fournit une sortie de type numérique compatible avec les types des entrées.

Dans le cas d'un seul port d'entrée, si l'entrée est un vecteur $X\,[m_1]\ldots[m_n]$ de n dimensions de tailles respectives allant de m_1 à m_n, le bloc effectue alors l'opération indiquée sur les éléments du vecteur. Ainsi, le type du port de sortie est de type X. Il en est de même pour les blocs logiques, à l'exception qu'ils acceptent des entrées booléennes et fournissent une sortie booléenne. Dans le cas d'un bloc combinatoire à une seule entrée scalaire, le bloc se comporte comme un bloc identité.

La fonction de typage en avant des blocs combinatoires est définie en deux parties.

– $\mathcal{F}^-$ des blocs combinatoires à une seule entrée.
Définition 7.4.9. *Fonction de typage en avant d'un bloc combinatoire à une entrée*

$$\mathcal{F}^-(X) = \begin{cases} X' & si\ X = X'\,[m_1]\ldots[m_n]\ sinon \\ X & \end{cases}$$

La spécification en Coq de la fonction $\mathcal{F}^-$ d'un bloc combinatoire à une seule entrée, dans l'environnement γ^-, correspond au code suivant.

```
Definition combinBlock_OneInputFwd (b : BlockBaseType)
    (currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    fun blockPort =>
      if (eq_nat_dec blockPort 0) then
        let indatatype := currentInputTypes blockPort
          in match indatatype with
               |array n t => basic t
               |_ => indatatype
             end
        else SimTypinLattice.top.
```

– $\mathcal{F}^-$ des blocs combinatoires à plusieurs entrées.
Soit $\overrightarrow{X} = \langle X_1, \ldots, X_n \rangle$ la séquence des types d'entrée d'un bloc combinatoire à n entrées. Notons par $\sqcup_{i=1}^{n} X_i$ le plus petit super-type de la séquence des types $\overrightarrow{X}$.
Définition 7.4.10. *Fonction de typage en avant d'un bloc combinatoire à n entrées*

$$\mathcal{F}^-(\overrightarrow{X}) = \sqcup_{i=1}^{n} X_i$$

La fonction de typage $\mathcal{F}^-$ pour les blocs combinatoires possédant des entrées multiples est implantée, dans l'environnement γ^-, comme suit.

```
Definition combinBlock_MultipleTypesFwd
    (b : BlockBaseType)
    (currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    fun blockPort =>
      if (eq_nat_dec blockPort 0)
        then
          let indatatype := currentInputTypes blockPort
            in
            let linPorts := envPortFwd2list b.(inputPortsSize)
                 currentInputTypes
              in fold_right (fun t queue => SimTypingLattice.join tt t queue
                   ) SimTypingLattice.bot linPorts
        else SimTypingLattice.top.
```

Ainsi, la fonction de typage en avant d'un bloc combinatoire est un simple appel à l'une des deux fonctions :

```
Definition combinFunctionFwd (b : BlockBaseType)
    (currentInputTypes : TypingPortEnvironmentFwd.PreOrder.type) :=
    if (leb b.(inputPortsSize) 0)
      then SimTypingLattice.top
      else
        if (leb b.(inputPortsSize) 1)
          then combinBlock_OneInputFwd b currentInputTypes
          else combinBlock_MultipleTypesFwd b currentInputTypes.
```

7.4.5.2 Fonction de typage en arrière

La fonction de typage des entrées des blocs combinatoires consiste à renvoyer le même type de base que le type de sortie. Par exemple, si le type de sortie d'un bloc à une seule entrée est τ, le port d'entrée est de type vecteur de τ de dimensions et de tailles quelconques, noté $\tau\,[\,]^\star$. D'autre part, si le type de sortie d'un bloc à plusieurs entrées est τ, le type de chacune des entrées est également τ.

De même que le typage en avant, la fonction de typage des entrées de blocs combinatoires $\mathcal{F}^+$ est définie en deux parties :

- $\mathcal{F}^+$ des blocs combinatoires à une seule entrée.
  Définition 7.4.11. *Fonction de typage en arrière d'un bloc combinatoire à une entrée*

$$\mathcal{F}^+(Y) = \begin{cases} \top & si\ \top \preceq Y\ sinon \\ any & si\ any \preceq Y\ sinon \\ \bot & si\ Y = \bot\ sinon \\ Y & si\ Y = custom\ k\ sinon \\ Y\,[\,]^\star & \end{cases}$$

  Ceci correspond au code donné par la fonction suivante.

```
Definition combinBlock_OneInputBwd
    (b : BlockBaseType)
    (currentOutputTypes: TypingPortEnvironmentBwd.PreOrder.type) :=
    fun blockPort =>
      if (eq_nat_dec blockPort 0)
        then
          if (SimTypingLaticeInv.dec tt (currentOutputTypes blockPort)
               SimTypingLattice.bot)
            then SimTypingLatticeInv.bot
              else
                if (SimTypingLaticeInv.dec tt (currentOutputTypes blockPort)
                      any)
                  then any
                    else
                      if (SimTypingLaticeInv.eq_dec tt (currentOutputTypes
                            blockPort) SimTypingLatticeInv.top)
                        then SimTypingLatticeInv.top
                          else
                            match (currentOutputTypes blockPort) with
                              |array _ t
                              |basic t => array lstar t
                              |custom _ => (currentOutputTypes blockPort)
                              |_ => SimTypingLatticeInv.top
                            end
        else SimTypingLaticeInv.top.
```

- $\mathcal{F}^+$ des blocs combinatoires à plusieurs entrées.
  Pour un bloc combinatoire à n entrées, $\mathcal{F}^+$ est donnée par la définition suivante.

Définition 7.4.12. *Fonction de typage en arrière d'un bloc combinatoire à n entrées*

$$\mathcal{F}^{+}(Y) = \overrightarrow{Y}$$

En Coq, elle est décrite, dans l'environnement γ^{+}, par :

```
Definition combinBlock_MultipleTypesBwd
    (b : BlockBaseType)
    (currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
    fun blockPort =>
      if (leb b.(inputPortsSize) blockPort)
        then SimTypingLatticeInv.Top
        else currentOutputTypes 0.
```

Ainsi, la fonction de typage en arrière d'un bloc combinatoire est un simple appel à l'une des deux fonctions :

```
Definition combinFunctionBwd (b : BlockBaseType)
    (currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
    if (leb b.(inputPortsSize) 0)
      then SimTypingLatticeInv.top
      else
        if  (leb b.(inputPortsSize) 1)
          then combinBlock_OneInputBwd b currentOutputTypes
          else combinBlock_MultipleTypesBwd b currentOutputTypes.
```

7.4.5.3 Correction des fonctions de typage des blocs combinatoires

Nous vérifions l'adjonction sur les fonctions de typage des blocs combinatoires.

Preuve. Soit un bloc combinatoire à une entrée :
($\Rightarrow$)
Cas 1 : $X = X'[m_1]\dots[m_n]$. Dans ce cas, $X \in \texttt{SimType} \setminus \{\bot, \top, any, custom\,k\}$. Nous avons, $\mathcal{F}^{-}(X) = X'$ et disposons donc de l'hypothèse $X' \preceq Y$.

$$\mathcal{F}^{+}(Y) = \begin{cases} \top & \text{si} \quad \top \preceq Y \text{ sinon} \\ any & \text{si} \quad any \preceq Y \text{ sinon} \\ Y\,[\,]^{\star} & \text{si} \quad Y \neq \bot \end{cases}$$

Ainsi, nous disposons de l'un des cas respectifs suivants :

$$X \preceq \begin{cases} \top & \text{si} \quad \top \preceq Y \text{ sinon} \\ any & \text{si} \quad any \preceq Y \text{ sinon} \\ Y\,[\,]^{\star} & \text{si } Y \neq \bot \end{cases}$$

Ce dernier cas résulte de la transitivité de $X'[m_1]\dots[m_n] \preceq Y[m_1]\dots[m_n]$ et de $Y[m_1]\dots[m_n] \preceq Y\,[\,]^{\star}$.
Cas 2 : $X \neq X'\,[\,]$. Nous avons donc $\mathcal{F}^{-}(X) = X$. Soit $\mathcal{F}^{-}(X) \preceq Y$. Ce qui signifie que :

$$X \preceq \begin{cases} \top & \text{si} \quad \top \preceq Y \text{ sinon} \\ any & \text{si} \quad any \preceq Y \text{ sinon} \\ \bot & \text{si} \quad Y = \bot \text{ sinon} \\ Y & \text{si} \quad Y = custom\,k \text{ sinon} \\ Y\,[\,]^{\star} & \end{cases}$$

Ce dernier cas résulte de la transitivité de $X \preceq Y$ et $Y \preceq Y\,[\,]^{\star}$. Par conséquent, $X \preceq \mathcal{F}^{+}(Y)$.
($\Leftarrow$)
Cas 1 : $\top \preceq Y$. Donc, $\mathcal{F}^{-}(X) \preceq Y$ car $Y = \top$.

Cas 2 : $any \preceq Y \wedge Y \neq \top$. Nous avons donc $\mathcal{F}^+(Y) = any$. Soit $X \preceq \mathcal{F}^+(Y)$. Cela signifie que $\mathcal{F}^-(X) \preceq Y$.
Cas 3 : $Y = \bot$. Nous disposons de $\mathcal{F}^+(Y) = \bot$. Soit $X \preceq \mathcal{F}^+(Y)$. Cela signifie que $X = \bot$ et que $\mathcal{F}^-(X) = \bot$. Ainsi, $\mathcal{F}^-(X) \preceq Y$.
Cas 4 : $Y = custom\ n$. Nous avons $\mathcal{F}^+(Y) = Y$. Supposons $X \preceq Y$. D'un autre coté, $\mathcal{F}^-(X) = X$. Ce qui signifie $\mathcal{F}^-(X) \preceq Y$ est satisfaite.
Cas 5 : $Y \notin \{\top, \bot, any, custom\ k\}$. Ainsi, d'une part $\mathcal{F}^+(Y) = Y\,[\,]^\star$ et l'hypothèse $X \preceq \mathcal{F}^+(Y)$. D'autre part,

$$\mathcal{F}^-(X) = \begin{cases} X' & \text{si} \quad X = X'\,[m_1]\ldots[m_n] \\ X & \text{sinon} \end{cases}$$

Soit $\mathcal{F}^-(X) = X'$. Étant donné $X'\,[\,] \preceq Y\,[\,]^\star$ alors $X' \preceq Y$. Il en résulte $\mathcal{F}^-(X) \preceq Y$.
Dans le cas où $\mathcal{F}^-(X) = X$, le type X est un scalaire. Ainsi, de l'hypothèse $X \preceq Y\,[\,]^\star$ résulte $X \preceq Y$. Par conséquent, $\mathcal{F}^-(X) \preceq Y$.

Soit un bloc combinatoire à n entrées :
($\Rightarrow$)
Par définition, $\mathcal{F}^-(\overrightarrow{X}) = \sqcup_{i=1}^n X_i$. Supposons alors $\mathcal{F}^-(\overrightarrow{X}) \preceq Y$. Par définition $\mathcal{F}^+(Y) = \overrightarrow{Y}$. Ce qui signifie $\langle \sqcup_{i=1}^n X_i, \ldots, \sqcup_{i=1}^n X_i \rangle \preceq_{\theta^n} \langle Y, \ldots, Y \rangle$, avec n la taille des séquences $\overrightarrow{X}$ et $\overrightarrow{Y}$. Étant donné $\overrightarrow{X} \preceq_{\theta^n} (\overrightarrow{\sqcup_{i=1}^n X_i})$. Nous déduisons par transitivité $\overrightarrow{X} \preceq_{\theta^n} \mathcal{F}^+(Y)$.
($\Leftarrow$)
Nous avons d'une part $\mathcal{F}^+(Y) = \overrightarrow{Y}$. Supposons $\overrightarrow{X} \preceq_{\theta^n} \mathcal{F}^+(Y)$. Selon la définition de $\sqcup$, cela signifie $\sqcup_{i=1}^n X_i \preceq \sqcup_{i=1}^n Y_i$ avec $Y_i = Y$. Ainsi, par définition de $\sqcup$ et de la relation $\preceq$, il en découle que $\sqcup_{i=1}^n X_i \preceq Y$. Et d'autre part, $\mathcal{F}^-(\overrightarrow{X}) = \sqcup_{i=1}^n X_i$. Par conséquent, $\mathcal{F}^-(\overrightarrow{X}) \preceq Y$.

7.4.5.4 Limite des fonctions de typage des blocs combinatoires

Considérons un bloc `Add` dont les entrées, par inférence en avant, sont de type booléen (voir la figure 7.11a) et un bloc `Add` dont les entrées sont incompatibles (voir la figure 7.11b).

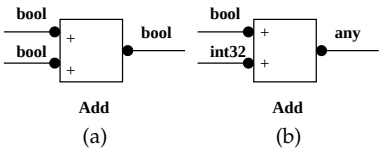


FIG. 7.11 – *Anomalies de typage des blocs arithmétiques*

En appliquant la fonction de typage $\mathcal{F}^-$ des blocs combinatoires, la sortie du bloc `Add` de la figure 7.11a est de type booléen, et celle du bloc `Add` de la figure 7.11b est de type `any`.

À l'issue du typage des blocs combinatoires, il est alors nécessaire d'effectuer une analyse pour :

- signaler une erreur lorsqu'un bloc arithmétique manipule des entrées/-sorties de type booléen ou réciproquement lorsqu'un bloc logique manipule des entrées/sorties de type numérique ;

– laisser à l'utilisateur le choix de configurer le transtypage souhaité : transtyper les booléens en numériques (dans le cas des blocs arithmétiques) ou inversement dans le cas des blocs logiques, comme cela est possible en SIMULINK.

Notre point de départ était de factoriser le plus de fonctions de typage afin de garder leur généricité. Cela nous évite d'écrire des fonctions de typage pour chaque bloc et de vérifier si elles satisfont l'adjonction. Toutefois, nous n'avions pas le temps de traiter les cas d'erreurs des blocs combinatoires cités plus haut. Néanmoins, une solution est proposée et discutée dans la section 7.8.

7.4.6 Blocs séquentiels

Rappelons que les blocs séquentiels contiennent un paramètre d'initialisation dont dépend le type de sortie, et que cette catégorie de blocs calcule la sortie en fonction des valeurs calculées dans des cycles antérieurs. C'est le cas du bloc `Unit Delay` rencontré précédemment. Il est composé d'une valeur initiale (paramètre), d'une entrée et d'une sortie.

Le type de la sortie est indéfini (`undef`) si le port d'entrée est indéfini ou bien est égal au plus petit super-type de X et de α quand le type de l'entrée X est un type correct. Dans le cas contraire, si l'entrée est erronée alors le type de sortie est `Error`.

De même, la fonction de typage $\mathcal{F}^+$ d'un bloc séquentiel consiste à affecter `Error` au type d'entrée dans le cas d'une sortie erronée, $(Y \sqcap \texttt{any})$ dans le cas où le type du paramètre (α) est un sous-type de Y, le type indéfini `undef` dans le cas contraire.

Ces fonctions de typage sont données par superposition des blocs identité et source.

Définition 7.4.13. *Fonction de typage en avant d'un bloc séquentiel*

$$\mathcal{F}^-(X) = \begin{cases} \bot & si\ X \preceq \bot\ sinon \\ (\alpha \sqcup X) & si\ X \preceq any\ sinon \\ \top & \end{cases}$$

Définition 7.4.14. *Fonction de typage en arrière d'un bloc séquentiel*

$$\mathcal{F}^+(Y) = \begin{cases} \top & si\ \top \preceq Y\ sinon \\ Y & si\ \alpha \preceq Y\ sinon \\ \bot & \end{cases}$$

Nous obtenons les mêmes fonctions de typage en superposant les blocs identité et source tel que cela est illustré par la figure 7.12. Le typage d'un bloc séquentiel est le même que la superposition des blocs identité et source.

Il ne s'agit pas de transformer le bloc séquentiel mais seulement d'exploiter le résultat du typage.

Comme, d'une part, les fonctions de typage du bloc identité vérifient l'adjonction et, d'autre part, celles du bloc `source` vérifient également l'adjonction, alors les fonctions de typage de leur superposition vérifient l'adjonction (voir Section 7.4.1.2).

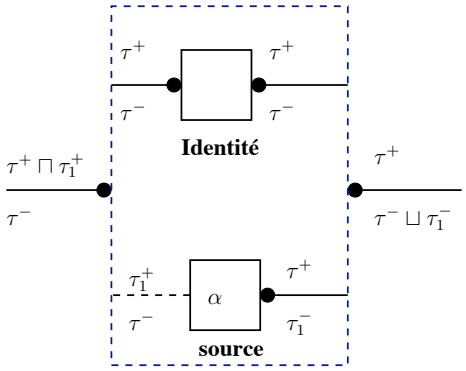


FIG. 7.12 – *Typage équivalent à celui d'un bloc* `Unit Delay`

En pratique, nous disposons d'une fonction de typage par nature de bloc pour les inférences de type en avant et en arrière.

`blockTypingFunctionFwd` regroupe les fonctions de typage en avant des blocs rencontrés.

```
Definition blockTypingFunctionFwd
  (b: BlockBaseType)
  (currentInputTypes  : TypingPortEnvironmentFwd.PreOrder.type) :=
  match b.(operator) with
    |Combin (sum _ _)
    |Combin (product _ _)
    |Combin (div _ _)        => arithFunctionFwd b currentInputTypes
    |Combin (logic _ _)      => logicFunctionFwd b currentInputTypes
    |Combin (inport _ _)     => sourceFunctionFwd b currentInputTypes
    |Combin (outport _ _) => sinkFunctionFwd b currentInputTypes
    |Seq (unitDelay _ _)     => delayFunctionFwd b currentInputTypes
    |_ => TypingPortEnvironmentFwd.top
  end.
```

Il en est de même pour le typage en arrière. `blockTypingFunctionBwd` regroupe les fonctions de typage en arrière des blocs étudiés.

```
Definition blockTypingFunctionBwd
  (b: BlockBaseType)
  (currentOutputTypes : TypingPortEnvironmentBwd.PreOrder.type) :=
  match b.(operator) with
    |Combin (sum _ _)
    |Combin (product _ _)
    |Combin (div _ _)        => arithFunctionBwd b currentOutputTypes
    |Combin (logic _ _)      => logicFunctionBwd b currentOutputTypes
    |Combin (inport _ _)     => sourceFunctionBwd b currentOutputTypes
    |Combin (outport _ _) => sinkFunctionBwd b currentOutputTypes
    |Seq (unitDelay _ _)     => delayFunctionBwd b currentOutputTypes
    |_ => TypingPortEnvironmentBwd.top
  end.
```

L'implantation des fonctions de typage ainsi que les propriétés et preuves d'adjonction sont consultables dans le fichier TypingFunction.v.

7.5 ÉQUATIONS DE TYPAGE

Les fonctions de typage que nous avons présentées plus haut servent à calculer le type de chaque port de chaque bloc indépendamment de ses connexions avec le reste des blocs. Tandis que les équations de typage servent à prendre en compte les liens entre les blocs et à inférer les types calculés par les fonctions de typage à chaque étape.

Dans le reste du manuscrit, nous reprenons le domaine de définition des fonctions de typage de la propriété 7.4.1, c'est-à-dire en manipulant les environnement à la place des types. Nous posons dans ce qui suit quelques règles de notation et de présentation des équations.

Soient $\mathcal{T}^-$ un élément de γ^- et $\mathcal{T}^+$ un élément de γ^+. Les équations de typage $\mathcal{T}^-$ et $\mathcal{T}^+$ sont donc représentées en manipulant respectivement les opérateurs de la famille de treillis γ^- et ceux de la famille de treillis inverse γ^+.

Notons par $\mathcal{T}^-(p_A)$ l'équation de typage par inférence en avant qui associe un type au port p du bloc A, selon le flot de propagation en avant (depuis les entrées vers les sorties du circuit). De manière symétrique, $\mathcal{T}^+(p_A)$ désigne l'équation de typage par inférence en arrière qui associe un type au port p_A, selon le flot de propagation en arrière (depuis les sorties vers les entrées du circuit en question). Notons pour un bloc A, un port d'entrée par i_A et un port de sortie par o_A. Rappelons qu'un port cible d'un signal a un seul port source alors qu'un port source d'un signal peut avoir plusieurs ports cibles. Notons le port source du port i par $i_{A.In}$ et les ports cibles du port o par $o_{A.Out}$. Pour un bloc A, la fonction de typage en avant est notée par $\mathcal{F}_A^-$, et la fonction de typage en arrière est notée par $\mathcal{F}_A^+$.

7.5.1 Équations de typage des ports pour l'inférence en avant

Soit $\mathcal{T}_{in}^- \in \gamma_{in}^-$, le typage en avant des ports d'entrée d'un bloc donné. Soit également $\mathcal{T}_{out}^- \in \gamma_{out}^-$, le typage en avant des ports de sortie d'un bloc donné.

7.5.1.1 Typage en avant des ports d'entrée d'un bloc

Soit l'exemple présenté dans la figure 7.13. Le type du port d'entrée du bloc A (i_A) calculé par inférence en avant, noté $\mathcal{T}_{in}^-(i_A)$, dépend :

- du dernier type calculé pour ce port d'entrée par inférence en avant ($\mathcal{T}_{in}^-(i_A)$) ;
- du type du port source relié au port i_A calculé par inférence en avant ($\mathcal{T}_{out}^-(i_{A.In})$).

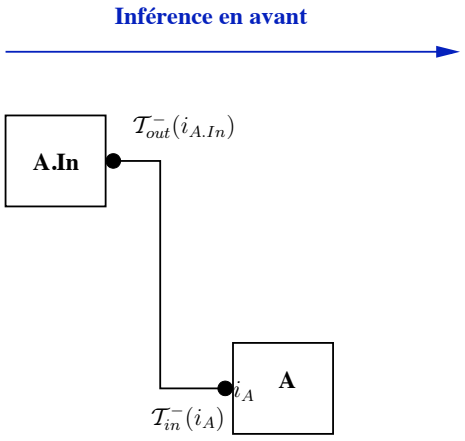


FIG. 7.13 – *Typage du port d'entrée du bloc A par inférence en avant*

Définition 7.5.1. *Typage en avant d'un port d'entrée i d'un bloc A*

$$\mathcal{T}_{in}^{-\,k+1}(i_A) \triangleq \mathcal{T}_{in}^{-\,k}(i_A) \sqcup \mathcal{T}_{out}^{-\,k}(i_{A.In}) \tag{7.1}$$

Il convient de préciser que le type du port d'entrée d'un bloc doit être compatible avec chaque port source qui lui est relié via un signal de données. Cela est assuré par l'opérateur $\sqcup$ du treillis de types choisi.

7.5.1.2 Typage en avant des ports de sortie

Considérons l'exemple de la figure 7.14. Pour un bloc A, le type de son port de sortie o_A calculé par inférence en avant, noté $\mathcal{T}_{out}^{-}(o_A)$, dépend :
- du dernier type calculé pour ce port de sortie par inférence en avant ;
- du type du port o_A calculé par la fonction de typage en avant $\mathcal{F}_A^-$.

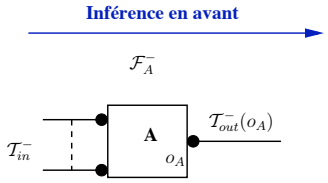


FIG. 7.14 – *Typage du port de sortie du bloc A par inférence en avant*

Définition 7.5.2. *Typage en avant d'un port de sortie o d'un bloc A*

$$\mathcal{T}_{out}^{-\,k+1}(o_A) \triangleq \mathcal{T}_{out}^{-\,k}(o_A) \sqcup (\mathcal{F}_A^-(\mathcal{T}_{in}^{-k}))(o_A) \tag{7.2}$$

7.5.2 Équations de typage des ports pour l'inférence en arrière

Soit $\mathcal{T}_{in}^{+} \in \gamma_{in}^{+}$, le typage en arrière des ports d'entrée d'un bloc donné. Soit également $\mathcal{T}_{out}^{+} \in \gamma_{out}^{+}$, le typage en arrière des ports de sortie d'un bloc donné. Notons par $\sqcup^{-1}$ l'union de l'environnement γ^+ (il correspond à l'intersection $\sqcap$ de γ^-).

7.5.2.1 Typage en arrière des ports d'entrée d'un bloc

Considérons l'exemple de la figure 7.15. Le type d'un port d'entrée i du bloc A, par inférence en arrière, dépend :
- du dernier type $\mathcal{T}_{in}^{+}(i_A)$ calculé pour le port i_A, par inférence en arrière ;
- du type calculé pour le port i_A par la fonction de typage en arrière $\mathcal{F}_A^+$.

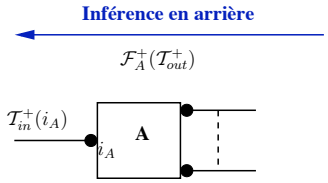


FIG. 7.15 – *Typage du port d'entrée du bloc A par inférence en arrière*

Définition 7.5.3. *Typage en arrière d'un port d'entrée i d'un bloc A*

$$\mathcal{T}_{in}^{+\,k+1}(i_A) \triangleq \mathcal{T}_{in}^{+\,k}(i_A) \sqcup^{-1} (\mathcal{F}_A^+(\mathcal{T}_{out}^{+\,k}))(i_A) \tag{7.3}$$

7.5.2.2 Typage en arrière des ports de sortie

Soit l'exemple de la figure 7.16, le type d'un port de sortie o du bloc A, par inférence en arrière, dépend :
- du dernier type $\mathcal{T}^{+}_{out}(o_A)$ calculé pour le port o_A par inférence en arrière ;
- des types des ports cibles reliés à o_A calculés par inférence en arrière.

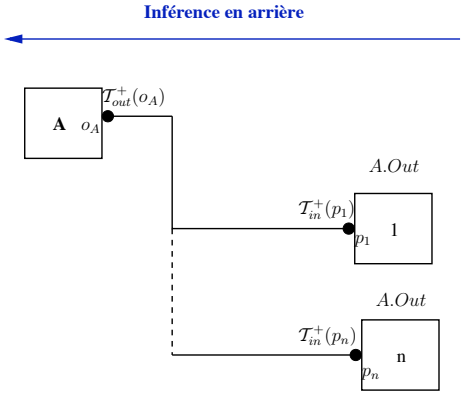


FIG. 7.16 – *Typage du port de sortie du bloc A par inférence en arrière*

Définition 7.5.4. *Typage en arrière d'un port de sortie o d'un bloc A*

$$\mathcal{T}^{+\,k+1}_{out}(o_A) \triangleq \mathcal{T}^{+\,k}_{out}(o_A) \sqcup^{-1} (\sqcup^{-1}{}_{p \in o_{A.out}} \mathcal{T}^{+k}_{in}(p)) \tag{7.4}$$

7.5.2.2.1 Fonctions de propagation des types

Il y a une fonction de propagation des types pour chaque inférence.

A Fonction de propagation des types selon l'inférence en avant

La fonction de propagation des types (d'entrée et de sortie) par inférence en avant, notée ρ^{-}, est définie par $\rho^{-} \triangleq \rho^{-}_{in} \times \rho^{-}_{out}$. La fonction de propagation ρ^{-}_{in} propage les types des ports d'entrée des blocs du circuit considéré. Elle consiste à associer à chaque bloc du circuit l'environnement des ports d'entrée γ^{-}_{in} et à les propager selon l'inférence en avant. ρ^{-}_{in} est définie en utilisant l'équation 7.1. Cette fonction est implantée en Coq par la définition suivante :

```
Definition ForwardInputTypes
  (diagram : ModelType)
  (currentInputTypes currentOutputTypes : TypingBlockEnvironmentFwd.PreOrder.
      type)
  : TypingBlockEnvironmentFwd.PreOrder.type :=
  fun blockIndex =>
    if (leb (size diagram) blockIndex)
      then errBlock_type blockIndex
      else
        match (lookForBlockStructure blockIndex diagram) with
          |Some block =>
            let dataSourcesPorts := dataSourceForPorts (dataSignals
                ModelElementType diagram) blockIndex
              in
              mergePorts blockIndex dataSourcesPorts currentInputTypes
                  currentOutputTypes
          |None => errBlock_type blockIndex
        end.
```

La fonction de propagation ρ^{-}_{out} propage les types des ports de sortie des blocs du circuit considéré. Elle consiste à associer à chaque bloc l'environnement des ports de sortie γ^{-}_{out} et de les propager selon l'inférence en avant. ρ^{-}_{out} est définie en utilisant l'équation 7.2.

```
Definition ForwardOutputTypes
    (diagram : ModelType)
    (currentInputTypes currentOutputTypes : TypingBlockEnvironmentFwd.PreOrder.
        type)
    : TypingBlockEnvironmentFwd.PreOrder.type :=
    fun blockIndex =>
      if (leb (size diagram) blockIndex)
        then errBlock_type blockIndex
        else
          match (lookForBlockStructure blockIndex diagram) with
            |Some block =>
              (fun blockPort => SimTypingLattice.join tt
                ((blockTypingFunctionFwd block (currentInputTypes blockIndex) )
                    blockPort)
                ((currentOutputTypes blockIndex ) blockPort)))
            |None => errBlock_type blockIndex
          end.
```

B Fonction de propagation des types selon l'inférence en arrière

La fonction de propagation des types (d'entrée et de sortie) par inférence en arrière, notée ρ^{+}, est définie par $\rho^{+} \triangleq \rho^{+}_{in} \times \rho^{+}_{out}$. La fonction de propagation ρ^{+}_{in} propage les types des ports d'entrée des blocs du circuit considéré. Elle consiste à associer à chaque bloc du circuit l'environnement des ports d'entrée γ^{+}_{in} et à les propager selon l'inférence en arrière. ρ^{+}_{in} est définie en utilisant l'équation 7.3. Cette fonction est implantée en Coq par la définition suivante :

```
Definition BackwardInputTypes
    (diagram : ModelType)
    (currentInputTypes currentOutputTypes : TypingBlockEnvironmentBwd.PreOrder.
        type)
    : TypingBlockEnvironmentBwd.PreOrder.type :=
    fun blockIndex =>
      if (leb (size diagram) blockIndex)
        then errBlock_type blockIndex
        else
          match (lookForBlockStructure blockIndex diagram) with
            |Some block =>
              (fun blockPort => SimTypingLatticeInv.join tt
                ((blockTypingFunctionBwd block (currentOutputTypes blockIndex) )
                    blockPort)
                ((currentInputTypes blockIndex ) blockPort)))
            |None => errBlock_type blockIndex
          end.
```

La fonction de propagation ρ^{+}_{out} propage les types des ports de sortie des blocs du circuit considéré. Elle consiste à associer à chaque bloc l'environnement des ports de sortie γ^{+}_{out} et à les propager selon l'inférence en arrière. ρ^{+}_{out} est définie en utilisant l'équation 7.4.

```
Definition BackwardOutputTypes
    (diagram : ModelType)
    (currentInputTypes currentOutputTypes : TypingBlockEnvironmentBwd.PreOrder.
        type)
    : TypingBlockEnvironmentBwd.PreOrder.type :=
    fun blockIndex =>
      if (leb (size diagram) blockIndex)
        then errBlock_type blockIndex
        else
          match (lookForBlockStructure blockIndex diagram) with
            |Some block =>
              let dataTargetsPorts := dataTargetForPorts (dataSignals
                  ModelElementType diagram) blockIndex
                in mergePortsBwd blockIndex dataTargetsPorts currentInputTypes
                    currentOutputTypes
            |None => errBlock_type blockIndex
          end.
```

Les fonctions de propagation ρ^- et ρ^+ sont itérées jusqu'à ce que le point fixe soit atteint (voir les algorithmes 4 et 5). Les résultats des inférences de type sont $lfp(\rho^-)$ (dans la famille de treillis $\{\theta_d\}_{d\in\texttt{unit}}$) pour l'inférence en avant, et $lfp(\rho^+)$ (dans la famille de treillis inverse $\{\theta_d^{-1}\}_{d\in\texttt{unit}}$) pour l'inférence en arrière.

L'implantation de l'algorithme de typage peut être consultée dans le fichier Typer.v.

7.6 Vérification de l'outil élémentaire de typage

La satisfaction de la propriété d'adjonction par les fonctions de typage proposées est une première étape de la vérification de l'outil élémentaire de typage (voir section 7.4.1).

Dans un premier temps, afin d'exploiter les propriétés du calcul de point fixe $lfp(\rho)$, présentées dans la section 5.3.6, l'environnement initial Y_0 doit être un pré-point fixe.

Propriété 7.6.1. *L'environnement initial est un pré-point fixe*

$$Y_0 \preceq_Y \rho(Y_0)$$

Preuve. Les ports virtuels sont initialisés à `any` et les autres ports à $\bot$ (de la famille de treillis concernée). Le type des ports virtuels reste inchangé par les fonctions de typage, tandis que le cas des autres ports est traité en exploitant la propriété `bot_least`.

Nous nous focalisons, dans la suite, sur la vérification des algorithmes d'inférence de type. Ainsi, les propriétés à vérifier se classent en : propriétés de croissance des fonctions de propagation (avant et arrière) et propriété de correction du typage du circuit.

7.6.1 Croissance des fonctions de propagation

Il faut s'assurer que chaque appel récursif des fonctions d'itération ρ^- et ρ^+ pour le calcul des types fait progresser le calcul qui doit alors converger si le treillis est de profondeur finie.

Théorème 7.6.2. *Croissance de la fonction de propagation en avant* ρ^-
$\forall\, e_1\, e_2 : Y^-_{in} \times Y^-_{out},\ e_1 \preceq_{Y^-_{in}\times Y^-_{out}} e_2 \Rightarrow \rho^-(e_1) \preceq_{Y^-_{in}\times Y^-_{out}} \rho^-(e_2)$

Preuve. La preuve consiste à remplacer ρ^- par sa définition. Par définition, $\rho^- \triangleq \rho^-_{in} \times \rho^-_{out}$.

Premièrement, ρ^-_{in} est croissante car cette fonction est définie seulement par l'opérateur d'union ($\sqcup$) qui est lui-même croissant dans Y^-_{in}.

Deuxièmement, ρ^-_{out} comporte la fonction $\mathcal{F}^-$ qui dépend de la nature des blocs du circuit. Étant donné que la fonction $\mathcal{F}^-$ est monotone, et que l'équation 7.2 n'est composée que d'unions de types ($\sqcup$) qui est un opérateur croissant, alors la fonction de propagation des types de sorties en avant ρ^-_{out} est croissant.

Enfin, comme les fonctions ρ^-_{in} et ρ^-_{out} sont croissantes, alors la fonction de propagation des types en avant ρ^- est croissante.

Théorème 7.6.3. *Croissance de la fonction de propagation en arrière* ρ^+
$\forall\, e_1\, e_2 : Y^+_{in} \times Y^+_{out},\ e_1 \preceq_{Y^+_{in} \times Y^+_{out}} e_2 \Rightarrow \rho^+(e_1) \preceq_{Y^+_{in} \times Y^+_{out}} \rho^+(e_2)$

Le même raisonnement est suivi pour prouver la monotonie de la fonction de propagation ρ^+, où la seule différence est l'application de la définition des équations 7.3 et 7.4.

7.6.2 Correction de typage

Un bloc est dit bien typé si, pour chacun de ses ports, le type calculé par ρ^- est un sous-type du type calculé par ρ^+.

Propriété 7.6.4. *Correction de typage des blocs d'un circuit*
$\forall\, A :\ D, \forall\, p,\ \rho^-(p_A) \preceq \rho^+(p_A)$

Le calcul du point fixe permet de propager, s'il y a lieu, les erreurs de type en avant et en arrière grâce à l'adjonction. En revanche, il n'est pas nécessaire de vérifier la propriété de correction de typage pour chaque bloc du circuit qui est d'autant plus longue quand le circuit prend une taille importante. Ainsi, seule une vérification au niveau des extrémités du circuit est suffisante. Par extrémités, nous entendons les entrées qui alimentent le circuit ou les sorties qui accumulent les résultats. Ce choix réduira considérablement le coût de cette vérification.

7.7 EXEMPLE

Nous présentons dans cette section un exemple de typage d'un circuit simple. Considérons le circuit de la figure 7.17. Dans le circuit, le paramètre du bloc `In` est initialisé à `uint16`, celui du bloc `Constant` à `int8 [2]`, celui du bloc `Out` à `double [ ]`*, celui du bloc `Unit Delay` à `int16` et celui du bloc `Out1` à `single`.

Dans l'exemple, nous montrons seulement les types des paramètres des blocs sources et cibles.

Pour simplifier la représentation des environnements de ports, nous avons choisi de référencer les ports d'entrée (ou de sortie) par les blocs auxquels ils appartiennent.

Rappelons que les ports d'entrée virtuels sont initialisés à `any`, tandis que les autres ports sont initialisés à `undef` pour l'inférence en avant. Initialement,

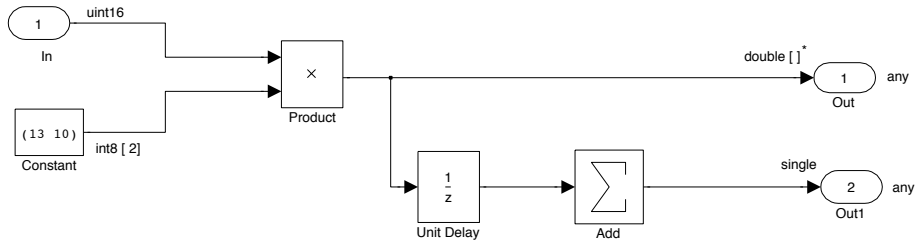


FIG. 7.17 – *Exemple de circuit à typer*

l'environnement des ports d'entrée correspond à :

$$(\gamma_{in}^{-})_0 = \begin{cases} \texttt{In} & \mapsto \texttt{any} \\ \texttt{Constant} & \mapsto \texttt{any} \\ \texttt{Product} & \mapsto \texttt{undef} \times \texttt{undef} \\ \texttt{Out} & \mapsto \texttt{undef} \\ \texttt{Unit Delay} & \mapsto \texttt{undef} \\ \texttt{Add} & \mapsto \texttt{undef} \\ \texttt{Out1} & \mapsto \texttt{undef} \end{cases}$$

Par inférence en avant, il en résulte les types indiqués dans le circuit de la figure 7.18. Par exemple, l'entrée supérieure du bloc `Product` qui était de

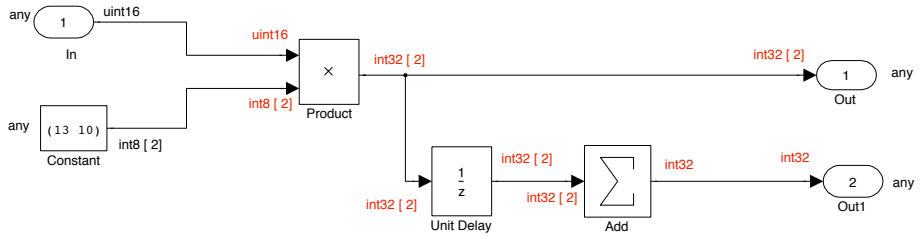


FIG. 7.18 – *Inférence en avant du circuit donné*

type `undef`, au départ, devient de type `uint16` ⊔ `undef`=`uint16` après inférence en avant. De même, l'inférence en avant produit le type `int8 [2]` pour l'entrée inférieure du bloc `Product`. Ainsi, en appliquant la définition de la fonction de typage du bloc `Product`, la sortie est de type `int32 [2]` car `int32` est le plus petit super-type des types de base `uint16` et `int8` et l'entrée inférieure du bloc est un vecteur à une dimension de taille 2. La sortie est donc également un vecteur avec la même dimension et la même taille (voir figure 7.5).

L'inférence en arrière produit les types notés dans la figure 7.19. Rappelons que initialement, les ports de sortie virtuels sont initialisés à `any`, tandis que tous les autres ports sont initialisés à `Error` (ce qui correspond à $\bot_{\gamma^+}$).

L'environnement initial des ports de sortie des blocs est donné par :

$$(\gamma^{+}_{out})_0 = \begin{cases} \texttt{In} & \mapsto \texttt{Error} \\ \texttt{Constant} & \mapsto \texttt{Error} \\ \texttt{Product} & \mapsto \texttt{Error} \\ \texttt{Out} & \mapsto \texttt{any} \\ \texttt{Unit Delay} & \mapsto \texttt{Error} \\ \texttt{Add} & \mapsto \texttt{Error} \\ \texttt{Out1} & \mapsto \texttt{any} \end{cases}$$

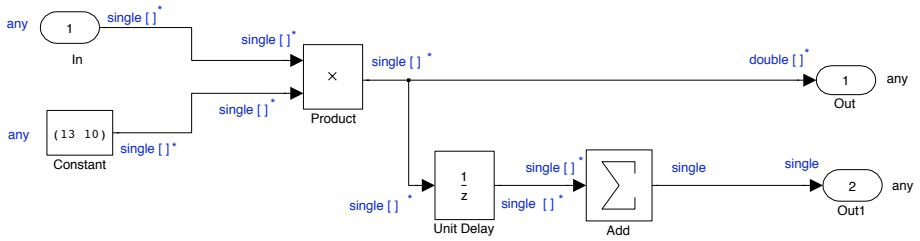


FIG. 7.19 – *Inférence en arrière du circuit donné*

Par exemple, après inférence en arrière, le port d'entrée du bloc `Unit Delay` est de type `single [ ]`* et le port d'entrée du bloc `Out` est de type `double [ ]`*. Le type du port de sortie du bloc `Product` correspond donc à (`double [ ]`$^\star \sqcap$ `single [ ]`*) qui est `single [ ]`*.

À l'issue des deux inférences, chaque port possède la plus petite borne et la plus grande borne de type, calculées respectivement par inférence en avant et par inférence en arrière. Pour chaque port, l'intervalle obtenu par ces deux bornes contient la (ou les) solution(s) qui permet au circuit d'être bien typé. L'initialisation de l'utilisateur, pour un port donné, doit être comprise dans l'intervalle calculé, autrement il s'agit d'une erreur de type.

Les initialisations des utilisateurs pour les blocs `In`, `Constant`, `Out` et `Out1` sont donc correctes car elles sont comprises dans les intervalles calculés, de plus la propriété de correction 7.6.4 est vérifiée. Le type du paramètre du bloc `Unit Delay` est correct car les fonctions de typage de ce bloc ne produisent pas d'erreurs.

Par exemple, le circuit reste bien typé si le type du paramètre du bloc `In` est `int32`. En revanche, le circuit est mal typé si ce paramètre est de type `boolean`. Cela résulte de la fonction de typage du bloc `Product` qui produit `Error` pour sa sortie et qui sera propagé (en avant) tout au long du reste du circuit. Il en est de même si le paramètre du bloc `Out` est précisé à `double`. L'inférence en arrière produit une erreur de typage pour la sortie du bloc `Product` car les types `double` et `single[]`* ne sont pas comparables (`double` $\sqcap$ `single[]`$^\star = \bot$).

Remarque Remarquons que le circuit est bien typé même si le signal reliant les blocs `Product` et `Out` n'est pas du même type de part et d'autre (voir figure 7.19). Cela s'explique par le fait que le bloc `Product` écrit le résultat en `single [ ]`* et le bloc `Out` lit une valeur de type `double [ ]`*. La possibilité qu'un signal ait des extrémités de types différents dépend de la manière

dont est implantée la génération de code. Dans le cadre de GeneAuto, une structure est initialisée avec toutes les variables à utiliser dans le circuit. Ces variables représentent les valeurs des signaux du circuit et possèdent un seul type déclaré, alors que nos inférences en produisent deux.

Toutefois, une solution peut être apportée en vérifiant pour chaque signal que les bornes inférieure et supérieure sont égales.

7.8 Synthèse et discussion

Notre but était de définir une analyse générique qui permettra à chaque partenaire de définir son propre treillis de types et de lancer le calcul du point fixe pour vérifier si le circuit est bien typé. Le calcul et l'inférence des types peuvent être effectués avec tout langage de programmation sans utiliser un assistant de preuve. La vérification de type peut également être établie avec un autre langage de vérification qu'un assistant de preuve. Toutefois, notre point de départ était de définir un outil élémentaire de typage qui soit également un cas d'application du cadre général que nous avons défini dans le chapitre 5. L'outil a été expérimenté avec quelques blocs de base d'un sous-ensemble de circuits Simulink.

L'information calculée, ici le type, se propage par inférence de type à travers les signaux de données qui connectent les ports. Toutefois, une vérification de type est quelquefois nécessaire pour s'assurer qu'un signal relie deux ports de même type, notamment dans le cas où l'utilisateur a explicité le type.

Le typage par inférence que nous avons présenté établit une propagation des types en avant et en arrière du circuit. La propriété d'adjonction garantit alors la correction des fonctions de typage des blocs et assure que toute erreur est propagée tout au long du circuit. Cela est un point fort de l'algorithme car il permet de définir la plage des types corrects $[\tau^-, \tau^+]$ pour chaque port de tout bloc du circuit. En dehors de cet intervalle, le circuit est mal typé, c'est également le cas lorsque la propriété de correction n'est pas vérifiée. Toutefois, la plage des types corrects pour les ports ne permet pas de déduire que toutes les valeurs de cet intervalle conduisent au bon typage du circuit. Les intervalles calculés contiennent la ou les solution(s) pour que le circuit soit bien typé.

En pratique, plusieurs blocs Simulink peuvent être configurés pour hériter leurs informations, par exemple les types, via la propagation en avant ou la propagation en arrière. Cette possibilité réduit la lecture des informations du circuit considéré à l'entrée du générateur de code et simplifie l'échange des types des signaux et/ou ports tant que le modèle évolue. Tel est souvent le cas des applications réelles qui sont souvent de grande taille en nombre de blocs. Cet *héritage*, dans le cadre de notre analyse, n'est rien d'autre qu'une inférence de type.

SimCheck [RS10] est un travail récent sur la vérification de types des circuits Simulink. L'approche permet de générer des obligations de preuve qui seront résolues par le solveur Yices [Yic]. Ce dernier génère par la suite des cas de tests ou des contre-exemples. La vérification se focalise sur les dimensions des types manipulés, leurs unités (centimètres, litres, etc.) ou les contraintes sur les valeurs, par exemple ne pas tolérer une division par une valeur nulle. Cependant, les circuits vérifiés doivent être entièrement annotés pour la géné-

ration d'obligations de preuve. La vérification de types du circuit nécessite des contraintes éditées par le concepteur du circuit. L'outil vérifie alors avec le solveur Yices que cette contrainte est bien respectée par le circuit. De plus, le lien entre les types de SIMULINK n'est pas clairement défini. Enfin, la génération d'obligations de preuve est nécessaire à chaque fois que l'utilisateur change les types.

Solution pour pallier les limites des fonctions combinatoires

Les fonctions de typage proposées ne permettent pas d'effectuer des opérations sur des combinaisons d'entrées booléennes avec d'autres types. Cependant, ces fonctions ne génèrent pas d'erreurs dans le cas de sortie booléenne pour un bloc arithmétique, ou inversement, elles ne signalent pas d'erreurs dans le cas de sortie numérique pour un bloc logique.

La réécriture des équations de typage des blocs combinatoires, pour prendre en compte les erreurs de typage lorsqu'un bloc arithmétique manipule des entrées booléennes, ou lorsqu'un bloc logique manipule des entrées numériques, n'est pas possible pour la fonction $\mathcal{F}^+$ (voir section 7.4.5.2). Cela s'explique par le fait que le système de types ne dispose pas de type qui exprime $\neg boolean$ pour les blocs arithmétiques par exemple.

Une solution serait de composer, selon le principe de la composition présenté en section 7.4.1.1, chaque bloc combinatoire avec un bloc de détection d'erreurs. Ce bloc n'a pas de rôle fonctionnel, il est spécifié uniquement pour capturer les erreurs.

Les fonctions de typage pour le bloc de détection d'erreurs sont décrites comme suit.

Définition 7.8.1. *Fonction de typage en avant du bloc de détection d'erreurs*

$$\mathcal{F}^-_{\neg\alpha}(X) = \begin{cases} X & si\ (X \sqcap \alpha = \bot)\ sinon \\ \top & \end{cases}$$

Définition 7.8.2. *Fonction de typage en arrière du bloc de détection d'erreurs*

$$\mathcal{F}^+_{\neg\alpha}(Y) = \begin{cases} \top & si\ (\top \preceq Y)\ sinon \\ Y \sqcap \neg\alpha & \end{cases}$$

La définition de $\mathcal{F}^+_{\neg\alpha}(Y)$ nécessite, donc, de définir $\neg\alpha$ qui n'est pas représentée dans le treillis des types manipulés. Ainsi, cette solution consiste à rajouter de nouveaux types dans le treillis des types considéré. Par exemple, le bloc de détection d'erreurs pour les blocs arithmétiques nécessite la définition du type $\neg\, boolean$. Ce dernier serait le type *numerical* en l'absence des types personnalisés. Par conséquent, lors de la composition d'un bloc arithmétique et d'un bloc de détection de booléens, une erreur sera signalée quand le bloc arithmétique produit une sortie booléenne.

Prise en compte des initialisations de l'utilisateur

L'algorithme d'inférence de type proposé ne prend pas en compte les initialisations des types ou annotations effectuées par les utilisateurs. Ceci peut être vu comme une limite par les partenaires industriels. Toutefois, une réflexion est menée à ce sujet qui permet de prendre en compte les annotations de types des utilisateurs pendant la phase d'inférence sans avoir à changer l'algorithme.

FIG. 7.20 – *Circuit annoté par un type*

Soit l'exemple de la figure 7.20. Afin de prendre en compte le type α, le typage de cette branche de circuit est équivalent au typage de la branche illustrée dans la figure 7.21. Il ne s'agit pas, ici, de transformer le circuit de départ, mais seulement d'exploiter le typage du circuit de la figure 7.21.

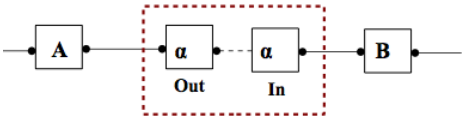


FIG. 7.21 – *Transformation en circuit équivalent en typage*

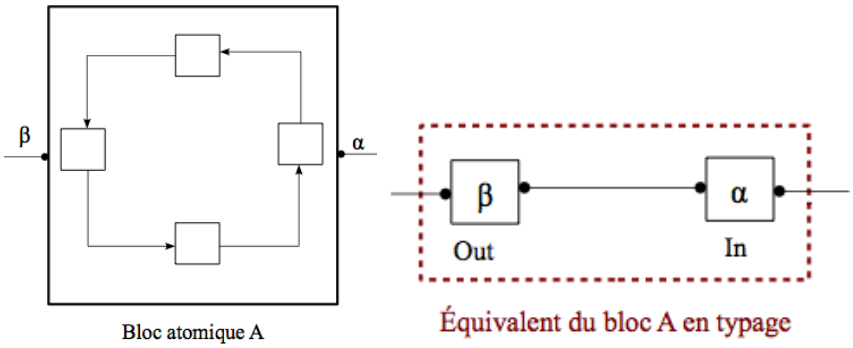
Il s'agit d'introduire un bloc *fantôme* entre les deux blocs `A` et `B`. Il est constitué en entrée du bloc cible `Out` et en sortie du bloc source `In`. Ces deux blocs sont eux-mêmes reliés par un signal virtuel. De plus, le type du port initialisé par l'utilisateur est celui du paramètre des deux blocs. Par conséquent, par propagation en avant, le port de sortie du bloc `In` est α.

Ainsi, pour toute annotation que ce soit pour un port de sortie comme l'exemple présenté ici ou pour un port d'entrée, le typage est équivalent à la même transformation de circuit que la figure 7.21.

Une solution simple peut être apportée pour considérer les types que définissent les utilisateurs. Il s'agit d'initialiser l'environnement des ports directement avec ces valeurs. Cependant, cela nécessite de vérifier pour chaque port que le résultat de la fonction de typage correspond bien à l'initialisation des types sinon il faut propager une erreur. Autrement dit, il sera nécessaire de changer l'algorithme et de refaire les preuves de correction et de croissance des fonctions de propagation.

Modularisation des blocs Simulink

Les circuits que nous traitons dans cette thèse sont plats (préalablement dépliés). D'un point de vue modularisation, il serait préférable de ne pas déplier un bloc atomique pour le typage. Pour cela, nous pouvons considérer que le circuit de la figure 7.22a soit transformé comme indiqué dans la figure 7.22b. Il s'agit, d'une part, d'affecter l'annotation de l'entrée du bloc `A` (β) au paramètre du bloc cible `Out` et, d'autre part, de considérer le paramètre du bloc source `In` de type α. Le bloc `Out` est relié au bloc `In` par un signal *virtuel* (ou fantôme) et l'ensemble forme un bloc fantôme qui servira au typage du bloc `A`. Le seul but est d'exploiter les fonctions de typage pour exprimer le type d'un sous-système. Cela peut être vu comme l'équivalent de l'utilisation d'une signature en programmation.



(a) *Bloc atomique annoté à typer* (b) *Transformation du bloc A*

FIG. 7.22 – *Transformation en sous-système ayant un typage équivalent*

Transtypage en Simulink

En SIMULINK, les utilisateurs exploitent souvent la possibilité de considérer les réels comme des booléens ou des entiers et réciproquement. Cependant, il peut être dangereux de transtyper implicitement avec des précisions différentes notamment dans les applications numériques critiques. Dans la solution que nous avons proposé, cela peut être résolu en considérant une famille de treillis de types sans aucun sous-typage.

En GENEAUTO, pour régler les problèmes de typage les utilisateurs transtypent le résultat de simulation du modèle au type interne qu'ils souhaitent obtenir. En revanche, cette façon de faire peut entraîner un cumul d'imprécision propagé tout au long du modèle.

Une solution fondée sur les méthodes formelles existe pour les applications numériques critiques. Il s'agit d'exploiter des outils comme FLUCTUAT [DGP+09]. Ce dernier est un analyseur statique par interprétation abstraite de programme C qui se focalise sur les propriétés numériques. Il permet de détecter et de mesurer les pertes de précisions éventuelles introduites par le transtypage, ainsi que le passage des réels mathématiques aux nombres à virgule flottante ou fixe informatique. Plusieurs utilisateurs exploitent cet outil.

Quatrième partie

Synthèse

Conclusion & Perspectives

8

8.1 Bilan sur les travaux présentés

Notre principale contribution dans le projet GeneAuto a été l'expérimentation de l'usage des méthodes formelles pour le développement d'outils qualifiés selon une approche acceptable par les partenaires industriels et les autorités de certification. L'objectif principal est la réduction des activités de vérification par tests tout en augmentant la qualité des outils de génération de code développés, en s'appuyant sur des techniques de vérification formelle.

Les travaux présentés dans ce manuscrit ont satisfait les exigences des différents partenaires et ont conduit à une implantation correcte par construction apte à traiter des systèmes réels. Le prototype GeneAuto, actuellement distribué, exploite l'outil formel d'ordonnancement présenté dans ce document. Celui-ci a été testé sur plusieurs applications industrielles de taille réelle issues du domaine du transport.

L'approche choisie consiste à utiliser l'assistant de preuve Coq pour la spécification, la conception, l'implantation et la vérification des outils élémentaires. Ceci permet ensuite de disposer de modules dont la correction est vérifiée une fois pour toutes et non pas à chaque utilisation du générateur. Deux formes de preuves sont nécessaires : d'une part, prouver que la spécification est bien formée (correction de l'ordre construit pour les blocs, correction des types des blocs, correction des algorithmes proposés) ; et, d'autre part, prouver que l'analyse spécifiée de chaque outil élémentaire respecte les exigences des partenaires industriels du projet.

La taille du code actuel est donnée dans la figure 8.1. Ces chiffres incluent les différentes versions des algorithmes présentées dans ce document.

Spécification et conception (Cadre)	4100
Ordonnancement des blocs	4000
Typage des blocs	4500

Fig. 8.1 – *Tailles du code* Coq *développé dans le cadre cette thèse*

Les erreurs signalées par les utilisateurs de GeneAuto pour l'ordonnanceur ont été très peu nombreuses et ne concernaient que des erreurs au niveau de la spécification en langage naturel ou son interprétation en Coq. Cette expérience montre donc que les applications industrielles doivent s'adosser de plus en plus sur des méthodes formelles pour une assurance plus élevée et une sûreté des systèmes critiques.

8.1.1 Cadre Formel

Le développement des outils élémentaires en Coq est factorisé grâce au cadre formel que nous avons proposé. Il représente la conception commune des outils élémentaires développés en Coq. Ce cadre générique s'appuie sur le principe de l'analyse statique par interprétation abstraite et, en particulier, sur les ensembles partiellement ordonnés, les treillis de profondeur finis et le calcul de point fixe par application de l'algorithme de Kleene. Le cadre formel regroupe toutes les preuves relatives à la spécification et à la conception (indépendantes des outils élémentaires). Chaque outil élémentaire est ensuite une instanciation de ce cadre pour laquelle des preuves supplémentaires sont nécessaires pour vérifier les propriétés spécifiques aux algorithmes implantés.

Nous avons choisi de travailler directement sur la sémantique abstraite correspondant à l'analyse statique réalisée pour deux raisons : la première est liée à l'absence de sémantique concrète des langages manipulés par les outils élémentaires, et la seconde relève de la nécessité de nous adosser sur les mêmes exigences pour les deux formes de développement (classique et formel), afin de rester conforme à la norme de certification DO-178B.

Il serait évidemment intéressant de compléter nos travaux en spécifiant la sémantique concrète (lorsque celle-ci est disponible) et de vérifier la correspondance entre les deux sémantiques abstraite et concrète au sein des différents outils élémentaires.

Nous nous sommes focalisés sur l'applicabilité des méthodes formelles en termes de spécification, de développement et de vérification de certains outils élémentaires du générateur de code ainsi que sur l'intégration de ceux-ci dans une application industrielle réelle qualifiable.

Il existe de nombreuses façons pour représenter les circuits. Nous avons choisi des structures de données simples qui reflètent les données manipulables dans les analyses et qui facilitent la construction des preuves que nous devions réaliser. Nous avons également choisi de construire les preuves les plus simples possibles sans nous appuyer sur des tactiques plus sophistiquées existantes ou que nous aurions développées spécialement pour le cadre. Cela conduit à de nombreuses preuves semblables

8.1.2 Spécifications précises

Le développement formel des outils élémentaires nous a permis de confirmer que l'utilisation des méthodes formelles au plus tôt dans le processus de développement, notamment dans la spécification, est indispensable. Elle permet d'améliorer significativement la qualité des outils et de réduire les coûts de vérification de l'implantation en détectant les problèmes plus tôt. En effet, l'écriture des exigences en Coq a révélé des incomplètudes, des ambiguïtés et des incohérences au niveau des spécifications écrites en langage naturel qui avaient été validées et vérifiées par les techniques usuelles de relecture croisée indépendante par les différents partenaires industriels du projet GeneAuto. Ces défauts ont été détectés lors des preuves réalisées en exploitant ces spécifications. Il n'aurait pas été possible de détecter toutes les anomalies de spécification dans un développement en langage Java ou un autre langage classique, car, même en vérifiant par un grand nombre de tests, ces derniers ne peuvent être exhaustifs. Une première illustration a été exhibée au niveau de l'ordonnanceur lors du calcul des dépendances. L'ordre suggéré par les exigences qui est obtenu par composition lexicographique de l'ordre partiel d'inclusion des

dépendances et de l'ordre total des priorités ne satisfait certaines autres exigences car les exigences sur la causalité des données sont contradictoires avec celles des priorités. Nous avons donc dû modifier les exigences en conséquence et introduire l'ordre basé sur la comparaison des dépendances des blocs triées selon les priorités. Un second exemple a été observé au niveau de l'outil élémentaire de typage qui existe dans sa version Java et Coq. Les règles de typage du bloc `Product` sont incomplètes dans l'implantation en Java. Citons un troisième exemple, le type booléen est spécifié dans certaines exigences comme un sous-type du type double, alors que, comme dans l'implantation du typage des blocs logiques, il est sous-entendu que le type booléen est un super-type du type double.

L'utilisation du même langage formel pour la spécification, le développement et la vérification des outils élémentaires renforce la confiance en ces derniers sans se soucier de la cohérence entre ces trois phases.

8.1.3 Processus de développement spécifique

Le développement exploitant l'assistant de preuve Coq nous a amené à distinguer le processus de développement formel du processus de développement qui exploite des technologies classiques (comme la conception et la programmation par objets en Uml et Java). Ce dernier cas exige beaucoup plus d'activités de vérification et en particulier de tests pour vérifier la cohérence de l'implantation par rapport à la spécification et à la conception, et pour rester conforme à la norme de certification DO-178B qui exige une couverture MC/DC à base de tests unitaires.

Un autre point important est l'adaptation du plan de vérification de la norme de qualification DO-178B pour le développement formel. La vérification de la conformité de la spécification formelle par rapport à celle écrite en langage naturel se fait par relecture. Les tests unitaires sont cependant supprimés pour le développement formel, ce qui réduit considérablement le coût de la vérification.

8.1.4 Un pas vers la certification selon les normes de certification

Le code OCaml extrait préserve les propriétés prouvées en Coq, car l'extraction se contente de supprimer les parties purement logiques de la spécification. Il s'agit néanmoins d'une forme de génération de code qui nécessite d'être qualifiée ou dont le résultat doit être vérifié. En suivant la norme DO-178B, nous proposons d'effectuer une relecture du code extrait par rapport aux fonctions écrites en Coq. Néanmoins, les échanges avec les autorités de certification, pour une pré-qualification, ont tenu compte des travaux sur la vérification du noyau de Coq [Bar99], ainsi que des travaux en cours sur la certification du mécanisme d'extraction de Coq [Glo09]. Ce qui aurait pour effet la suppression de la phase de vérification par relecture du code extrait, et apporte ainsi un niveau de confiance qui semble suffisant pour les autorités de certification quant à l'utilisation de l'assistant de preuve Coq.

Un point important qui mérite d'être évoqué concerne la qualification de l'intégration des outils élémentaires formels. Étant donné que la chaîne d'outils élémentaires à laquelle nous participons comporte des outils élémentaires en Java et d'autres en Coq, il était indispensable de développer des interfaces pour connecter un outil composé du code extrait en OCaml depuis Coq aux autres

outils écrits en JAVA. Ces interfaces suivent le même processus de développement que les outils classiques, en particulier la vérification par tests unitaires et couverture MC/DC. Le choix de format très simple à base de grammaire régulière contenant le minimum nécessaire d'informations permet de simplifier grandement ces activités et éventuellement de s'appuyer sur une simple relecture des fonctions d'analyse et de synthèse des données.

8.1.5 Ordonnancement

Plusieurs versions de l'algorithme d'ordonnancement ont été proposées à partir des exigences issues de SIMULINK. Nous avons montré que le calcul des dépendances reflète naturellement l'ordre d'exécution des blocs souhaité par les exigences. L'ordonnanceur traite tout type de circuits aplatis combinant des flots de données et de contrôle.

Plusieurs cas d'étude industriels réalistes ont été élaborés dans le cadre du projet GENEAUTO pour valider le générateur complet et chaque outil élémentaire dont l'ordonnancement. Ils ont tous montré que les dépendances événementielles rassemblent toutes les informations du circuit nécessaires pour fournir le bon ordre d'exécution des blocs, mais aussi, pour détecter la présence de boucles algébriques au niveau du circuit. L'outil développé en COQ fait actuellement partie du prototype GENEAUTO [1].

Le développement de l'ordonnanceur était loin d'être une tâche simple. Beaucoup de temps a été consacré à la recherche d'informations qui auraient dû être présentes dans une spécification complète et cohérente, et particulièrement les exigences liées aux flots de contrôle. En SIMULINK, la documentation n'est pas très claire au sujet de l'ordonnancement et le meilleur moyen d'obtenir des informations complémentaires est l'expérimentation en SIMULINK, avec des exemples et des contre-exemples afin d'en déduire les exigences.

Cela confirme que pour limiter le coût de développement avec un outil formel comme COQ, notamment dans le cadre de systèmes critiques contraint par des standards de certification et qualification, la spécification présentée dans les documents doit être concise, claire, précise et complète.

8.1.6 Limitations liées à l'usage des entiers de Peano

Le code extrait reflète les types de COQ utilisés pour les spécifications. En COQ, les structures de données représentant les blocs et les circuits ainsi que les rangs sont définies en utilisant le type inductif `nat` qui spécifie les entiers naturels à la façon de Peano. Un entier de Peano est une suite de `S` (successeurs) appliqués à `O`. La manipulation de ces entiers est coûteuse en mémoire et en temps d'exécution. Une simple addition $n + m$ est exécutée instantanément dans le cas des entiers primitifs, tandis qu'elle coûte $O(n)$ en temps dans le cas des entiers de Peano. Ce dernier cas s'explique par la récursion à partir de n pour atteindre la valeur zéro. De plus, un entier de Peano occupe $O(n)$ en taille mémoire.

La solution adoptée actuellement consiste à remplacer les entiers de Peano par les entiers primitifs dans le code extrait et à utiliser la programmation défensive pour éviter les dépassements de capacité. Toutefois, cette solution n'est pas vérifiée formellement, mais est complètement qualifiable par relecture du

[1] http ://www.geneauto.org

code extrait. Cette relecture est, de toute manière, nécessaire dans le processus de qualification du code extrait actuellement proposé.

Il n'est cependant pas souhaitable d'intervenir dans le code extrait. Une autre possibilité serait alors d'utiliser directement des entiers binaires de Coq, dont les opérations sont en $O(log\ n)$, qui garde toujours l'avantage de l'efficacité en temps calcul et en occupation mémoire.

8.1.7 Manipulation efficace de l'application de fonctions

Le mécanisme d'extraction de la version de Coq utilisée (v8.2) remplace certains identificateurs par leurs définitions, et recalcule ainsi la valeur de la définition à chaque fois que cette dernière est exploitée. Des η-expansions sont également effectuées par le mécanisme d'extraction pour mettre en place une sémantique d'appel par nom. Celles-ci consistent à remplacer des expressions directement évaluables par des fonctions dont l'expression initiale est le corps qui sera exécuté à chaque utilisation de la fonction. Dans le cadre formel, nous avons choisi de mettre en œuvre les environnements Y comme des fonctions de l'indice vers la valeur mémorisée dans l'environnement. Tandis que des structures de données spécifiques telles que les listes ou les tables de hachage auraient mérité une attention particulière et des preuves supplémentaires. Ces fonctions sont soumises à ces η-expansions, si bien que, lors du calcul du point fixe des fonctions de propagation dans le code extrait, les informations de chaque bloc nécessitent de recalculer celles de tous les blocs reliés, et ceci pour chaque itération. Ceci a été résolu en mettant en œuvre un mécanisme de cache similaire à de la "mémorisation" au cours de l'extraction. Le cache met une fois pour toutes les valeurs calculées au cours d'une itération dans un vecteur local. La fonction représentant l'environnement exploite ensuite ce vecteur. Le *garbage collector* permet de réutiliser ces vecteurs.

La gestion de ces caches a été soigneusement conçue afin d'éviter tout effet de bord ou effet secondaire qui ruinerait les efforts de développement en Coq. La confiance dans la correction des caches est actuellement obtenue par la relecture du code extrait.

8.1.8 Typage des circuits Simulink

Nous avons proposé un algorithme de typage générique basé sur l'inférence des types en avant et en arrière. Cette approche s'appuie sur le calcul d'un intervalle de types corrects (bornes inférieure et supérieure) pour chaque port d'un bloc du circuit considéré. Ces intervalles sont calculés en considérant seulement les initialisations des paramètres des blocs d'entrée et de sortie du circuit, ainsi que les paramètres des blocs séquentiels. Les annotations des utilisateurs ne sont donc pas prises en compte lors de l'inférence de type mais seulement par la vérification en fin d'analyse.

Le treillis des types est fourni comme paramètre de l'analyse. Il est possible de créer autant de treillis de types que de configurations souhaitées par l'utilisateur.

L'intervalle des types, calculé par l'algorithme pour chaque port, permet contrairement à Simulink ou GeneAuto de ne pas effectuer de nouvelles inférences de types, lorsque les types des blocs d'entrée et de sortie restent inchangés. Les intervalles de types calculés contiennent le ou les types pour que le circuit soit bien typé. En dehors de cet intervalle, le circuit est considéré mal typé.

Deux fonctions de typage sont nécessaires par catégorie de blocs (une fonction de typage pour l'inférence en avant et une pour l'inférence en arrière). Ces fonctions doivent être cohérentes selon la propriété d'adjonction. Ainsi, les preuves de cohérence des fonctions de typage s'ajoutent à celles relatives à l'algorithme. La propriété d'adjonction assure une propagation correcte des types ou des erreurs de type. Grâce à elle et à l'inférence de type, il n'est pas nécessaire de vérifier, une fois que le point fixe est atteint, la correction de typage de chaque port des blocs du circuit. En effet, il suffit de vérifier, quand le point fixe est atteint, la correction de typage des blocs qui constituent soit les entrées soit les sorties du circuit.

Le typage a été expérimenté sur quelques blocs Simulink de base. Pour prendre en compte davantage de blocs, il est impératif de définir deux fonctions de typage par bloc supplémentaire et de vérifier leur cohérence. Ce qui devient rapidement coûteux. Néanmoins, le gain en termes de coût de tests et de confiance en l'outil est bien plus avantageux de notre point de vue.

8.2 Perspectives

Nous présentons maintenant quelques perspectives de nos travaux.

8.2.1 Vers une sémantique concrète ?

Une sémantique pour un circuit Simulink consiste à associer des informations sémantiques à chaque composante du circuit (bloc, port, signal). Si nous considérons l'exécution ou simulation du circuit, la sémantique concrète fournie par le simulateur de Simulink correspond à la valeur associée à chaque signal et à la mémoire associée à chaque bloc séquentiel, et ce à chaque cycle d'une exécution non bornée du circuit. La sémantique de Simulink change régulièrement selon la version de l'outil et de la configuration adoptée par l'utilisateur. Il est donc difficile de maintenir une sémantique concrète.

Cependant, le développement formel d'un générateur de code nécessite une spécification concise de la génération de code ainsi que de la sémantique du langage cible. Une chaîne de compilation complète et vérifiée pourrait alors être obtenue du modèle Simulink jusqu'au code machine, en exploitant le compilateur C prouvé Compcert [Ler06] et l'extracteur de code en voie de vérification [Glo09].

8.2.2 Prendre en compte des exigences d'optimisation

Il serait intéressant d'expérimenter les travaux présentés sur l'ordonnancement en considérant des exigences spécifiques à l'optimisation du code et de la mémoire.

Soit le circuit illustré dans la figure 8.2. Si les trois blocs possèdent les mêmes priorités explicites, l'ordre obtenu, selon les exigences considérées dans l'ordonnancement, est tel que le bloc A est exécuté avant B qui est exécuté avant C. Cela signifie que le résultat produit par le bloc A doit être sauvegardé (en utilisant une variable locale) en attendant que le bloc B soit exécuté. Les exigences concernant l'optimisation du code généré doivent être prises en compte en premier lieu avant celles des priorités. Autrement dit, même si le bloc C est moins prioritaire que le bloc B, C s'exécute après le bloc A afin d'optimiser l'utilisation de la mémoire.

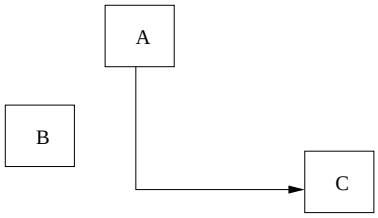


FIG. 8.2 – *Réduire les variables locales*

Afin d'optimiser le code généré, il serait également important de considérer par exemple la propagation de constante. Celle-ci peut être implantée comme une instanciation du cadre formel générique mais demande de disposer d'une sémantique formelle de chaque bloc atomique qui permette de calculer la valeur de la sortie d'un bloc dès que l'on dispose des valeurs de ces entrées. Cette optimisation restera toutefois limitée à un seul cycle de calcul. Soit l'exemple présenté dans la figure 8.3. Pour réduire la durée de vie des signaux ainsi que les variables locales utilisées pour ranger les valeurs des deux constantes de la figure 8.3a, il faudrait considérer le circuit équivalent de la figure 8.3b.

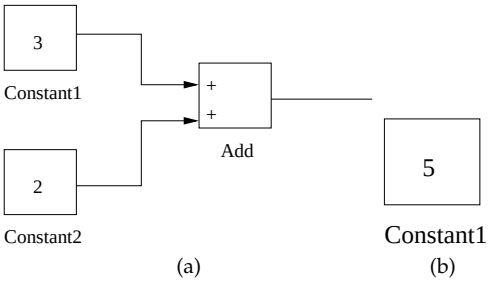


FIG. 8.3 – *Propagation de constante*

8.2.3 Explorer le typage au sens large

Nous avons considéré, dans l'étude du typage des circuits SIMULINK, les types de base de SIMULINK et les types personnalisés par les partenaires de GENEAUTO. Il est également envisageable d'étendre le système de type pour prendre en compte des objets et des classes qui n'étaient pas considérés par GENEAUTO mais qui sont présents dans SIMULINK.

Beaucoup de blocs SIMULINK méritent d'être expérimentés, notamment ceux qui manipulent des structures de types particulières. Nous citons par exemple, les multiplexeurs et démultiplexeurs qui manipulent des bus (n-uplets de types ou enregistrements). Ces derniers peuvent être représentés par un produit cartésien des types considérés. Il faut pour cela associer deux fonctions de typage par bloc supplémentaire ainsi que la preuve qu'elles sont cohérentes.

Une piste pertinente consiste à considérer d'autres mécanismes élémentaires de type que la composition et la superposition et leur preuve d'adjonction, qui permettraient de définir efficacement les fonctions de typage des blocs par simple assemblage de fonctions élémentaires déjà prouvées et de simplifier, ainsi, les preuves de cohérence. Il serait également intéressant d'automatiser la génération des fonctions de typage à partir de spécification de cas possible, ainsi que la génération des preuves d'adjonction de ces fonctions.

8.2.4 Application du cadre à une sémantique synchrone

Dans une sémantique synchrone, le cadre pourrait être appliqué pour le calcul d'horloges de la même manière que l'inférence de type. Plusieurs travaux ont été élaborés à ce sujet, nous citons par exemple [CP03, BCHP08] pour le langage Lustre (ou son extension Lucid Synchrone). Il s'agit de calculer l'horloge des signaux pour les langages synchrones aux flots de données. S. Boulmé a présenté dans [BH01a, BH01b] une sémantique synchrone (de Lustre) et sa correction en Coq. Ces travaux cités peuvent être exploités lors de la transformation de Simulink vers Lustre [PAA+03, CCM+03].

Des travaux récents ont été réalisés sur la génération de spécifications exécutables à partir de contraintes d'horloges [YTB+11]. Ces contraintes sont exprimées en Ccsl (Clocks Constraint Specification Language) de l'outil Marte. Ils sont notablement exploitables dans le cas de systèmes à horloges multiples.

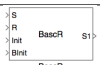
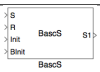
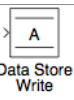
L'ordonnancement donne la structure du code qui doit être généré et l'ordre dans lequel les blocs doivent être exécutés lorsqu'ils sont tous activés. À ce stade, il n'est pas possible de déterminer les blocs qui ne sont pas activés. C'est pourquoi l'ordre d'exécution des blocs est un sous-ordre de l'ordre que nous avons fourni. En effet, les flots de contrôle ne nous donnent pas les détails sur l'activation des sous-systèmes, car la valeur des signaux n'est pas accessible dans la structure. De ce point de vue, Simulink/Stateflow est plus complexe que Scade.

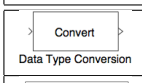
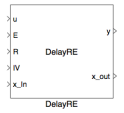
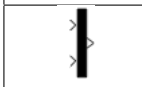
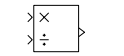
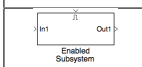
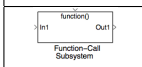
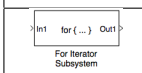
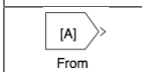
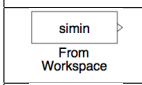
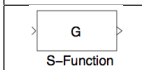
Le calcul d'horloge permet alors de déterminer quand les blocs sont activés et exécutés. Mais pour prendre en compte le calcul d'horloge en Simulink, il est indispensable de disposer de plus de détails : les utilisateurs doivent fournir des informations sur les horloges et leurs relations. Le cadre peut alors être exploité pour définir le calcul d'horloge. Cela consiste à représenter les horloges, par exemple, par des valeurs symboliques et vérifier si les horloges initialisées respectent bien l'ordre calculé en fonction des relations qui les relient.

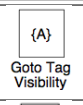
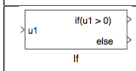
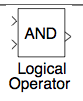
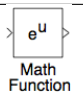
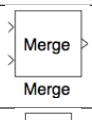
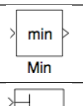
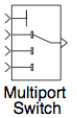
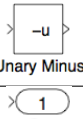
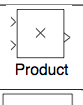
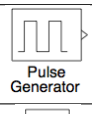
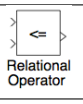
Liste des blocs

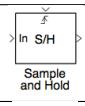
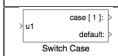
A

La liste des blocs Simulink considérés dans le projet GeneAuto est décrite dans le tableau A.1. Les blocs appartenant à la bibliothèque GeneAuto sont des blocs spécifiques aux besoins des partenaires industriels du projet.

Bloc	Description	Bibliothèque
	Valeur absolue	Math Operations
	Affectation	Math Operations
	Set/Reset avec priorité à Reset	GeneAuto
	Set/Reset avec priorité à Set	GeneAuto
	Opérateur logique bit à bit	Logic and Bit Operations
	Création de bus	Signal Routing
	Sélection de signaux dans un bus	Signal Routing
	Diagramme d'états	Stateflow
	Constante	Sources
	Définition de mémoire	Signal Routing
	Lecture mémoire	Signal Routing
	Écriture mémoire	Signal Routing

Bloc	Description	Bibliothèque
	Conversion de type	Signal Attributes
	Délai avec réinitialisation	GeneAuto
	Démultiplexeur	Signal Routing
	Multiplexeur	Signal Routing
	Affichage	Sinks
	Division	Math Operations
	Port d'activation	Ports & Subsystems
	sous-système activable	Ports & Subsystems
	Sous-système appelé comme une fonction	Ports & Subsystems
	Interpolation linéaire	User Defined Functions
	Itérateur	Ports & Subsystems
	Données pour branchement	Signal Routing
	Lecture depuis le workspace	Sources
	Interpolation linéaire dimension 2	User Defined Functions
	Multiplication par une constante	Math Operations
	Données pour branchement	Signal Routing

Bloc	Description	Bibliothèque
	Accessibilité du Goto	Signal Routing
	Connection des entrées non reliées	Sources
	Contrôle conditionnel	Ports & Subsystems
	Entrée	Sources
	Limitation de l'amplitude	Discontinuities
	Opérateurs logiques	Logic and Bit Operations
	Fonctions mathématiques	Math Operations
	Maximum	Math Operations
	Fusion de signaux	Signal Routing
	Minimum	Math Operations
	Switch multiple	Signal Routing
	Négation unaire	Math Operations
	Sortie	Sinks
	Multiplication	Math Operations
	Génération d'impulsions régulières	Sources
	Opérateurs relationnels	Logic and Bit Operations

Bloc	Description	Bibliothèque
	Échantillonnage	GeneAuto
	Affichage des signaux générés durant la simulation	Sinks
	Séquenceur	GeneAuto
	Commutateur	Signal Routing
	Commutateur de flot de contrôle	Ports & Subsystems
	Connection des ports de sorties non connectés	Sinks
	Écriture dans le workspace	Sinks
	Ajout de port de déclenchement	Ports & Subsystems
	sous-système à déclenchement	Ports & Subsystems
	Fonction trigonométrique	Math Operations
	Délai sur une période	Discrete
	Conversion de signal temps-discret en signal temps-continu	Discrete

Tab. A.1 – *Liste des blocs* Simulink *considérés dans* GeneAuto

Fichiers Coq

B

Tous ces fichiers sont regroupés dans la page web http://izerrouken.perso.enseeiht.fr/Code/.

Structure de données du langage d'entrée

- GeneAutoLanguage.v : définition de la structure de données des circuits lus en entrée de GeneAuto ;
- GeneAutoLibrary.v : définition de la bibliothèque des blocs et de leurs paramètres.

Cadre formel de développement

- Domain.v : définition des ensembles ordonnés et familles de treillis nécessaires pour le cadre formel ;
- EnvironmentLattice.v : définition de l'environnement de calcul ;
- ElementaryTool.v : Définition d'un outil élémentaire.

Ordonnancement des circuits Simulink

- RankLattice.v : construction de la famille de treillis des rangs ;
- RankSequencer.v : outil élémentaire d'ordonnancement des blocs Simulink par calcul des rangs ;
- DependencyLattice.v : construction de la famille de treillis des ensembles de dépendances ;
- SequencerDependency.v : outil élémentaire d'ordonnancement des blocs Simulink par calcul des ensembles de dépendances ;
- DependencyEventsLattice.v : construction de la famille de treillis des ensembles de dépendances événementielles ;
- SequencerDependencyEvents.v : outil élémentaire d'ordonnancement des blocs Simulink par calcul des ensembles de dépendances événementielles.

Typage des circuit Simulink

- SimulinkTypeLattice.v : construction de la famille de treillis des types Simulink ;
- TypeLattice.v : construction de la famille de treillis des types GeneAuto ;

– Typer.v : outil élémentaire de typage des blocs Simulink.

Liste des communications

Articles dans des Revues Nationales

Nassima Izerrouken, Marc Pantel, Xavier Thirioux, Olivier Ssi-Yan Kai, *Exprimentations en* Coq *pour un Gnrateur de Code Qualifiable. Revue des Sciences et Technologies de l'Information (TSI), Numro spcial : Application des Mthodes Formelles l'Analyse Statique et la Compilation. Herms Science Publications, Vol. 30, N. 4/2011, 2011.*

Confrences et Workshops Internationaux avec actes dits et comit de lecture

Nassima Izerrouken, Olivier Ssi-Yan Kai, Marc Pantel, Xavier Thirioux, *Use of Formal Methods for Building Qualified Code Generator for Safer Automotive Systems. In Critical Applications Robustness & Safety (CARS@EDCC). ACM, Valencia, Spain, (Position Paper), pages 53–56, 2010.*

Andres Toom, Nassima Izerrouken, Marc Pantel, Olivier Ssi-Yan Kai, Tonu Naks, *Towards Reliable Code Generation with an Open Tool : Evolutions of the GeneAuto Toolset. In International Conference on Embedded Real Time Software and Systems (ERTS2'10). Toulouse, France, SIA/3AF/SEE, (electronic support), 2010.*

Nassima Izerrouken, Marc Pantel and Xavier Thirioux, *Machine-Checked Sequencer for Critical Embedded Code Generator. In Proceeding of 11th International Conference in Formal Engineering Methods (ICFEM'09), Rio De Janeiro Brazil, LNCS Springer, pages 521–540, 2009.*

Nassima Izerrouken, Marc Pantel, Xavier Thirioux, Olivier Ssi-Yan Kai, *Integrated Formal Approach for Qualified Critical Embedded Code Generator. In 14th International Workshop on Formal Methods for Industrial Critical Systsems (FMICS'09). LNCS, Eindhoven, The Netherlands, (Short paper), pages 199–201, 2009*

Nassima Izerrouken, Marc Pantel, Xavier Thirioux, Martin Strecker, *Certifying an Automated Code Generator Using Formal Tools : Preliminary Experiments in the GeneAuto Project. In International Conference on Embedded Real Time Software and Systems (ERTS'08). Toulouse, France, SIA/3AF/SEE, (electronic support), 2008.*

Autres publications

Nassima Izerrouken, Marc Pantel, Olivier SSi-Yan Kai, GeneAuto Coq *Development Guidelines*. Rapport de Recherche, prsent au comit de certification CEAT, Novembre 2008

Nassima Izerrouken, Marc Pantel, Olivier SSi-Yan Kai, GeneAuto *Block Sequencer tool Requirements*. Rapport technique D2.36, Aout 2008

Nassima Izerrouken, Marc Pantel, GeneAuto *Block Sequencer tool Design*. Rapport technique D2.51, Septembre 2008

Bibliographie

[Ack82] William B. Ackerman. Data flow languages. *IEEE Computer*, 15(2) :15–25, 1982. (Cité page 32.)

[Acz77] Peter Aczel. An introduction to inductive definitions. In K. Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, Amsterdam, 1977. (Cité page 81.)

[A.L01] Edward A.Lee. Overview of the ptolemy project. Technical report, University of California, Berkeley, March 2001. (Cité page 49.)

[And09] Charles André. Syntax and semantics of the clock constraint specification language (ccsl). Technical report, INRIA, 2009. (Cité page 50.)

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, November 1999. (Cité pages 53, 77 et 199.)

[BBF+00] Gérard Berry, Amar Bouali, Xavier Fornari, Emmanuel Ledinot, Eric Nassor, and Robert de Simone. Esterel : a formal method applied to avionic software development. *Science of Computer Programming*, 36(1) :5–25, 2000. (Cité page 9.)

[BC04] Yves Bertot and Pierre Casteran. Interactive Theorem Proving and Program Development : Coq-Art : The Calculus of Inductive Constructions. *EATCS Texts in Theoretical Computer Science. Springer-Verlag*, 2004. (Cité pages 5, 6 et 53.)

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-criticalreal-time embedded software. In *The Essence of Computation*, pages 85–108, 2002. (Cité page 18.)

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003. (Cité pages 4 et 18.)

[BCG+07] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas P. Jensen, and David Pichardie. The mobius proof carrying code infrastructure. In *FMCO*, pages 1–24, 2007. (Cité page 19.)

[BCHP08] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation of synchronous data-flow languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tuscon, Arizona, June 2008. (Cité pages 8 et 204.)

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *In Proceedings of Formal Methods, 2006 (FM 2006)*, volume 4085/2006, pages 460–475. Springer-Verlag, 2006. (Cité pages 5, 6, 9 et 20.)

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, 1990. (Cité page 15.)

[BG92] Gerard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992. (Cité pages 47 et 48.)

[bGJ91] Albert benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations : The signal language and its semantics. *Science of Computer Programming*, 16(2) :103–149, 1991. (Cité pages 9, 47 et 48.)

[BH01a] Sylvain Boulmé and Grégoire Hamon. Certifying Synchrony for Free. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 2250, La Havana, Cuba, December 2001. Lecture Notes in Artificial Intelligence, Springer-Verlag. Short version of *A clocked denotational semantics for Lucid-Synchrone in Coq*, available as a Technical Report (LIP6), at `www.lri.fr/~pouzet`. (Cité page 204.)

[BH01b] Sylvain Boulmé and Grégoire Hamon. *A Clocked Denotational Semantics for Lucid Synchrone in Coq*, November 2001. (Cité page 204.)

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3) :263–288, 2009. (Cité pages 20 et 63.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977. (Cité pages 4, 10, 18 et 102.)

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*, pages 269–282, 1979. (Cité pages 10 et 18.)

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), 1992. (Cité pages 4, 10, 18 et 102.)

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier. The astrée analyser. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*. Springer-Verlag, 2005. (Cité pages 4 et 18.)

[CCM+03] Paul Caspi, Adrian Curic, Aude Maignan, Christos Sofronis, and Stavros Tripakis. Translating discrete-time simulink to lustre. In *ACM International Conference on Embedded Software (EMSOFT'03)*. EMSOFT, 2003. (Cité pages 9, 47 et 204.)

[CCN09] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, 2009. (Cité page 4.)

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. Model checking. *MIT Press*, 1999. (Cité pages 5 et 17.)

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76, 1988. (Cité page 53.)

[Cha08] Alexandre Chapoutot. *Simulation Abstraite : une Analyse Statique de Modèles Simulink*. PhD thesis, École Polytechnique de Paris, December 2008. (Cité page 52.)

[CHP06] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *ACM International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, October 2006. (Cité page 8.)

[CHP07] Paul Caspi, Grégoire Hamon, and Marc Pouzet. *Real-Time Systems : Models and verification —Theory and tools*, chapter Synchronous Functional Programming with Lucid Synchrone. ISTE, 2007. (Cité page 8.)

[CM04] Séverine Colin and Leonardo Mariani. Run-time verification. In *Model-Based Testing of Reactive Systems*, pages 525–555, 2004. (Cité page 17.)

[Cod] Real Time Workshop Embedded Coder. http ://www.mathworks.com/products/rtwembedded/. (Cité page 46.)

[Col52] Kleene Stephen Cole. *Introduction to metamathematics*. Amsterdam, North Holland, 1952. (Cité pages 10 et 81.)

[Cou81] Patrick Cousot. Semantic foundations of program. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis : Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs , New Jersey, 1981. (Cité pages 18 et 102.)

[Cou05] Patrick Cousot. Integrating physical system in the static analysis of embedded control software. In *APLAS*, pages 135–138, 2005. (Cité page 19.)

[CP00] Paul Caspi and Marc Pouzet. Lucid synchrone, a functional extension of lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000. (Cité page 47.)

[CP03] Jean Louis Colaco and Marc Pouzet. Clocks as first class abstract types. In *Third International Conference on Embedded Software (EMSOFT'03)*, 2003. (Cité page 204.)

[CP04] Jean-Louis Colaço and Marc Pouzet. Type-based initialization analysis of a synchronous data-flow language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004. (Cité page 8.)

[Dav03] Maulik A. Dave. Compiler verification : a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6) :2–2, November 2003. (Cité pages 8, 16 et 20.)

[dB70] Nicolaas Govert de Bruijn. The mathematical language automath, its usage and some of its extentions. In *Symposium on Automatic Demonstration*, volume 125, pages 29–61, 1970. (Cité page 5.)

[DGP+09] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *FMICS*, pages 53–69, 2009. (Cité page 194.)

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982. (Cité page 151.)

[Dor08] François-Xavier Dormoy. Scade 6 a model based solution for safety critical software development. In *European Congress Embedded Real-Time Software (ERTS)*, Toulouse, France, 2008. (Cité pages 4, 6 et 47.)

[DSp] DSpace. http ://www.dspace.fr/. (Cité page 46.)

[ECS] ECSS. http ://www.ecss.nl/. (Cité page 14.)

[Eur] Eurocae. http ://www.eurocae.net/. (Cité page 14.)

[FGH06] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & desing language (aadl) : An introduction. Technical report, Carnegy Mellon University, 2006. (Cité page 49.)

[FJJV96] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. Using on-the-fly verification techniques for the génération of test suites. In *Computer Aided Verification*, pages 348–359. Lecture Notes in Computer Science, 1996. (Cité page 17.)

[Flo67] Robert W Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. AMS, 1967. (Cité page 5.)

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/-caduceus plateform for deductive program verification. In *CAV*, pages 173–177, 2007. (Cité page 18.)

[Fra] FramaC. http ://frama-c.cea.fr/. (Cité page 18.)

[Glo09] Stéphane Glondu. Extraction certifiée dans coq-en-coq. In *Journées Francophones des Langages Applicatifs (JFLA)*, Saint-Quentin sur Isère, France, January 2009. (Cité pages 7, 60, 77, 199 et 202.)

[Gro01] IEC Working Group. Iec 61508 - functional safety of electrical/electroni/programmable electronic safety-related systems (7 parts). *International Electrotechnical Commission*, 2001. (Cité pages 5 et 14.)

[GTL02] Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12 :261–304, 2002. (Cité page 48.)

[Har87] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987. (Cité page 32.)

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991. (Cité pages 6 et 47.)

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of The ACM*, 12(10) :576–580, 1969. (Cité pages 5 et 18.)

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4) :366–381, 2000. (Cité page 17.)

[HRR91] Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. (Cité page 9.)

[IEE05] IEEE. *Ieee Standard 1012-2004 for Software Verification and Validation*, 2005. (Cité page 4.)

[IPT09] Nassima Izerrouken, Marc Pantel, and Xavier Thirioux. Machine checked sequencer for critical embedded code generator. In *International Conference on Formal Engineering Methods (ICFEM'09)*, pages 521–540. Springer-Verlag, December 2009. (Cité pages 127 et 130.)

[IPTK11] Nassima Izerrouken, Marc Pantel, Xavier Thirioux, and Olivier Ssi Yan Kai. Expérimentations en coq pour un générateur de code qualifiable. *Sciences et Technologies de l'Information, Méthodes Formelles à l'Analyse et la Compilation*, 30(4), 2011. (Cité page 127.)

[ISO] ISO. http ://www.iso.org/. (Cité page 14.)

[ITPS08] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, page (electronic medium), http ://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile. (Cité pages 11 et 109.)

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974. (Cité page 47.)

[kN03] Gerwin klein and Tobias Nipkow. Verified bytecode verifiers. *TCS*, 298 :583–626, 2003. (Cité page 20.)

[kN06] Gerwin klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4) :619–695, 2006. (Cité page 20.)

[KNT00] Mickaël Kerboeuf, David Nowak, and Jean-Pierre Talpin. Specification and verification of a stream-boiler with signal-coq. In *13th International Conference of Theorem Proving in Higher Order Logics (TPHOL'00)*. Springer-Verlag, 2000. (Cité page 48.)

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006. (Cité pages 5, 6, 9, 20 et 202.)

[Ler09] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasonning*, 43(4), December 2009. (Cité pages 5 et 6.)

[Let04] Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, 2004. (Cité pages 53, 60 et 77.)

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *IEEE*, volume 75, pages 1235–1245, 1987. (Cité page 49.)

[LPP05] Dirk Leinenbach, Wolfgag Paul, and Elena Petrova. Towards the formal verification of a c0 compiler : Code generation and correctness. In *International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 2–12. IEEE Computer Society Press, 2005. (Cité page 20.)

[M-J88] M-J-C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin. (Cité page 53.)

[mar07] *UML Profile for MARTE*. Object Management Group, August 2007. (Cité page 50.)

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3) :348–375, 1978. (Cité page 151.)

[Mil79] Robin Milner. LCF : A Way of Doing Proofs with a Machine. In *MFCS*, pages 146–159, 1979. (Cité pages 5 et 20.)

[Mon98] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Master's thesis, Université Paris 7, 1998. (Cité page 82.)

[Moo98] Gordon E. Moore. Cramming more components onto integrated circuits. In *IEEE*, volume 86, 1998. (Cité page 3.)

[MW72] Robin Milner and R. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, (7), 1972. (Cité page 20.)

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, New York, NY, USA, 1979. (Cité page 6.)

[Mye04] Glenford J. Myers. *The Art of Software Testing, Second Edition*. John Wiley & Sons, 2004. (Cité page 15.)

[Nec00] George C. Necula. Translation validator for optimizing compilers. *SIGPLAN Not.*, 35(5) :83–94, 2000. (Cité page 19.)

[NL97] George C. Necula and Peter Lee. Research on proof-carrying code for untrusted-code security. In *IEEE Symposium on Security and Privacy*, page 204. IEEE Computer Society, 1997. (Cité page 19.)

[Nor88] Bengt Nordström. Terminating general recursion. Technical Report 46, Programming Methodology Group, University of Goteborg and Chalmers University of Technology, June 1988. (Cité page 81.)

[OSR95] Sam Owre, Natarajan Shankar, and John M. Rushby. *User Guide for the PVS Specification and Verification System*. CSL, 1995. (Cité pages 5 et 53.)

[PAA+03] Caspi Paul, Curic Adrian, Maignan Aude, Sofronis Christos, Tripakis Stavros, and Niebert Peter. From simulink to scade/lustre to tta : a layered approach for distributed embedded applications. In *ACM-SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003. (Cité pages 9, 47 et 204.)

[PAC+08] Bruno Pagano, Olivier Andrieu, Benjamin Canou, Emmanuel Chailloux, Jean Louis Colaco, Thomas Moniot, and Philippe Wang. Certified development tools implementation in objective caml. In *PADL*, pages 2–17, Toulouse, France, 2008. (Cité page 6.)

[Pai67] John Mc Carthyand James Painter. Correctness of a compiler for arithmetic expressions. In *Symposium in Applied Mathematics*, pages 33–41, 1967. (Cité page 16.)

[Pau89] Lawrence C Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5, 1989. (Cité pages 5 et 20.)

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, 1993. (Cité page 53.)

[Pic05] David Pichardie. *Interprétation Abstraite en Logique Intuitionniste : Extraction d'Analyseurs Java Certifiés*. PhD thesis, Université de Rennes 1, December 2005. (Cité pages 82 et 102.)

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. *In Tools and Algorithms for Construction and Analysis of Systems, TACAS 98*, 1384 :151–166, 1998. (Cité pages 19 et 63.)

[RBN+03] M. Raulet, M. Babel, J.-F. Nezan, O. Déforges, and Y. Sorel. Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures. In *Proceedings of IEEE Workshop on Signal Processing Systems, SiPS'03*, Seoul, Korea, August 2003. (Cité page 48.)

[RD] Anna-Elelena Rugina and Jean-Charle Dalbin. Experiences with the geneauto code generator in the aerospace industry. (Cité page 150.)

[Riv04] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. *SIGPLAN Not.*, 39(1) :1–13, 2004. (Cité pages 19 et 63.)

[RS10] Pritam Roy and Natarajan Shankar. Simcheck : An expressive type-system for simulink. In NASA, editor, *Second NASA Formal Methods Symposium*, volume 216215, pages 149–160, 2010. (Cité page 191.)

[RTC99] RTCA. *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*, 1999. (Cité pages 5, 10 et 14.)

[Sel06] Bran Selic. UML 2 : A model-driven development tool. *IBM Systems Journal*, 45(3) :607–620, 2006. (Cité page 32.)

[Shr02] Bernd S. W. Shroder. *Ordered Sets : An Introduction*. Birkhäuser, 2002. (Cité page 79.)

[Sim] Simulink. http ://www.mathworks.com/products/simulink/. (Cité pages 4 et 26.)

[Som06] Ian Sommerville. *Software Engineering*. Addison-Wesley, 8th edition edition, 2006. (Cité page 65.)

[Sta] Stataflow. http ://www.mathworks.com/products/stateflow/. (Cité page 32.)

[Str02] Martin Strecker. Formal verification of a java compiler in isabelle. In LNCS, editor, *In Proc. Conference in Automated Deduction (CADE)*, volume 2392, pages 63–77. Springer-Verlag, 2002. (Cité page 20.)

[Str05] Martin Strecker. Compiler verification for c0. Technical report, Université Paul Sabatier, Toulouse, April 2005. (Cité page 20.)

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific Journal of Mathematics*, volume 5, pages 285–309, 1955. (Cité page 18.)

[TIN+] Andres Tooms, Nassima Izerrouken, Tonu Naks, Marc Pantel, and Olivier Ssi Yan Kai. Towards reliable code generation with an open tool : Evolutions of the gene-auto toolset. (Cité page 150.)

[TL08] Jean Baptiste Tristan and Xavier Leroy. Formal Verification of translation Validators- A Case Study on Instruction Scheduling Optimizations. *In POPL'08, 35th Symposium on Principles Of Programming Languages*, January 2008. (Cité pages 19 et 20.)

[uml07] *Object Management Group*. Unified Modeling Language, Superstructure, November 2007. (Cité page 50.)

[Val92] Franck Vallée. Statemate : une méthode et un outil pour la spécification des systèmes réactifs. *Revue générale de l'électricité*, 1992. (Cité page 32.)

[Win93] Glynn Winskell. The formal semantics of programming languages. *MIT Press*, 1993. (Cité page 79.)

[Wor] Real Time Workshop. http ://www.mathworks.com/products/rtw/. (Cité page 46.)

[Yic] Yices. http ://yices.csl.sri.com/. (Cité page 191.)

[YTB+11] Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Hervé Marchand, and Paul Le Guernic. Polychronous controller synthesis from marte ccsl timing specifications. In *ACM-IEEE Conference on Methods and Models for Codesign (MEMOCODE'11)*, July 2011. (Cité pages 50 et 204.)

[ZPFG02] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC : A translation validator for optimizing compilers. In ENTCS, Elsevier Science, 2002. (Cité page 19.)

Titre Développement prouvé de composants formels pour un générateur de code embarqué critique pré-qualifié

Résumé Nous nous intéressons au développement prouvé de composants formels pour un générateur de code pré-qualifié. Ce dernier produit un code séquentiel (C et Ada) pour des modèles d'entrée qui combinent les flots de données et de contrôle et qui présentent des possibilités d'exécution concurrente (Simulink/Stateflow et Scicos). Le développement prouvé permet de réduire le coût des tests et d'augmenter l'assurance des outils développés avec cette approche vis-à-vis de la qualification.
Les phases de spécification, de développement et de vérification des outils développés sont effectuées avec l'assistant de preuve Coq. Ce dernier permet d'extraire le contenu calculatoire des composants en préservant les propriétés prouvées en Coq. Ce code extrait est ensuite intégré dans une chaîne complète de développement (chaîne de GeneAuto).
Nous présentons un cadre formel, inspiré de l'analyse statique, qui s'appuie sur la sémantique abstraite et qui est instanciable sur plusieurs composants du générateur de code. Nous nous basons sur les ensembles partiellement ordonnés et sur le calcul de point fixe pour définir le cadre et effectuer les différentes analyses des composants du générateur de code. Ce cadre formel comporte toutes les preuves communes aux composants et indépendantes des analyses effectuées. Deux composants sont étudiés : l'ordonnanceur et le typeur des modèles d'entrée.

Mots-clés développement prouvé, Coq, Simulink, assistants de preuve, générateur de code

Title Proved Development of Formal Components for a Pre-Qualified Critical Embedded Code Generator

Abstract We are interested in the proved development of formal components for a pre-qualified code generator. This produces a sequential code (C and Ada) for input models that combine data and control flows, with potential concurrent execution (Simulink/Stateflow and Scicos). The proved development reduces test cost and increases insurance of components developed with this approach regarding the qualification.
Phases of specification, development and verification of the developed components are done with the Coq proof assistant. This allows to extract the computational content of the components preserving the properties proved in Coq. The extracted code is then integrated into the complete development tool-chain (GeneAuto tool-chain).
We present a formal framework, inspired from static analysis, based on the abstract semantics which is instantiable to several components of the code generator. We rely on partially ordered sets and fixed-point to define de formal framework and to perform the various analysis of components of the code generator. This formal framework includes all proofs common to the components and independent from the performed analyses. Two components are studied : the scheduler and the type checker of input models.

Keywords proved development, Coq, Simulink, proof assistants, code generator