Critères de test et génération de séquences de tests pour des systèmes réactifs synchrones modélisés par des équations flots de données et contrôlés par des automates étendus,

Christophe Junke

ÉCOLE CENTRALE PARIS

THÈSE DE DOCTORAT

Spécialité : Informatique

Présentée par

Christophe JUNKE

Soutenue le

9 janvier 2012

En vue de l'obtention du Titre de Docteur de l'École Centrale Paris

---

Critères de test et génération de séquences de tests pour des systèmes réactifs synchrones modélisés par des calculs flot de données et contrôlés par des automates étendus

---

Devant le jury composé de :

Pascale LE GALL	Directrice de thèse	Université d'Évry, ECP
Alain GIRAULT	Président du jury	INRIA
Ioannis PARISSIS	Rapporteur	INPG, LCIS
Jean-Pierre TALPIN	Rapporteur	INRIA, IRISA
Benjamin BLANC	Examinateur	CEA LIST, LSL

Résumé : Nous nous intéressons aux approches formelles pour le développement de systèmes réactifs critiques. Le langage synchrone Lustre pour la spécification de tels systèmes a subit des évolutions majeurs au cours des dernières années en intégrant dans sa sémantique à base flots de données synchrones des constructions de plus haut-niveau appelées automates de modes (dans le langage Scade 6). Ceux-ci mettent en œuvre l'activation de modes de calculs en fonction des états et des transitions de l'automate, et reposent pour cela sur la sémantique des horloges du langage Lustre. En particulier, nous étudiions la prise en compte des horloges et des automates de modes dans l'outil de génération de tests GATeL dédié à l'origine au langage Lustre mono-horloge (flots de données purs). GATeL génère automatiquement des séquences de tests pour un modèle à partir d'un objectif de test décrit en Lustre à travers une exploration en arrière des dépendances entre flots et selon des techniques de résolution de contraintes. Nous présentons ces différents domaines et la mise en œuvre des modifications apportées à l'outil pour prendre en compte les automates de modes. Enfin, nous définissons des critères de couverture structurelle pour les automates de modes et montrons alors comment, en les traduisant de manière automatique sous forme d'objectifs de tests, GATeL permet de générer des séquences couvrant ces critères.

Mots-clés systèmes réactifs synchrones, automates de modes, génération automatique de tests, programmation logique contrainte

Abstract : Lustre is a synchronous dataflow-oriented language for the specification of reactive systems. Since its definition, it has been extended to support mode automata, a formalism in which computation modes are activated according to an extended state-machine. The semantics of mode-automata is heavily based on an appropriate use of the clock sampling features of Lustre. We present the modifications made in GATeL, an automatic test sequences generator originally designed for a mono-rate subset of Lustre. GATeL performs a lazy goal-oriented test sequences generation, based on constraint logic programming. We modify it so that it can handle the temporal constraints of clocks internally and efficiently generate tests sequences from state-machines specifications. We also present some existing structural test criteria for state-machines and adapt them to the specific case of mode-automata.

Keywords synchronous reactive systems, mode automata, automatic test generation, constraint logic programming

Remerciements

Je remercie ma directrice de thèse pour son implication et sa disponibilité pendant le déroulement de ma thèse, notamment pour ses conseils nombreux lors de la rédaction des articles et du manuscrit. Elle m'a aidé tout au long de la thèse à avoir une vision claire et synthétique de mes travaux et à les resituer dans leur contexte scientifique.

Je remercie Alain Girault pour avoir accepté de présider mon jury ainsi que mes rapporteurs, Ioannis Parissis et Jean-Pierre Talpin, pour avoir pris le le temps de lire avec attention ce manuscrit et pour leur remarques le concernant. Je remercie aussi Ioannis pour m'avoir conforté dans ma décision de réaliser un doctorat lorsque j'étais en Master Professionel.

Je remercie mon encadrant Benjamin Blanc, pour sa participation active au bon déroulement de la thèse, que ce soit dans la relecture de mes écrits, lors de la répétition des présentations ou indirectement par son travail constant de recherche, de développement et de maintenance de l'outil GATeL.

Je suis reconnaissant envers Bruno Marre pour m'avoir permis de réaliser une thèse dans son équipe, ainsi que pour m'avoir encadré durant l'année que j'y ai passée en tant qu'ingénieur. J'ai bénéficié, de même que cette thèse, de son expertise scientifique et technique tout au long de mon séjour au CEA.

Je remercie à la fois Pascale, Benjamin et Bruno pour le soutien amical et leurs compétences d'encadrants.

J'ai été très bien accueilli par les membres du Laboratoire de Sûreté des Logiciels, qui font preuve à la fois de bonne humeur et d'une grande disponibilité malgré leurs occupations diverses et exigentes. Je pense par exemple à Sébastien, dont j'ai partagé le bureau et qui m'a régulièrement renseigné ou conseillé sur différents sujets, et ce bien souvent de sa propre initiative. Je salue donc le laboratoire dans son ensemble pour la bonne ambiance de travail qui y réside mais avant tout pour les compétences qu l'on y trouve.

Je remercie mes parents et mon grand-frère, pour m'avoir toujours montré le bon exemple et pour leur affection. Je pense aussi, et les remercie, aux parents d'Hélène qui m'ont aidé lorsque je terminais ma thèse. Je remercie enfin Hélène pour son amour, sa présence et son soutien inconditionnels, notamment durant la rédaction du manuscrit, alors qu'elle portait notre Elisabeth.

à mon grand-père
à ma fille

Table des matières

Introduction

Systèmes réactifs

Un système réactif est un logiciel qui fonctionne en temps-réel. On trouve des systèmes réactifs principalement dans les logiciels embarquées, où un composant matériel dédié à une tâche particulière est piloté par un programme informatique [SA93, LKYC02]. On s'intéresse à des systèmes de contrôle/commande comme des régulateurs de vitesse, où la prise d'information et les actions doivent être réalisées en continu dans le temps.

Les entrées d'un système correspondent à des données issues de son environnement (p. ex. : mesure de la vitesse instantanée d'un véhicule), alors que les sorties correspondent à des données envoyées à celui-ci pour le contrôler (p. ex. : commandes de frein, d'accélérateur). La mise en œuvre d'un tel système correspond en général à l'exécution en boucle d'un même traitement : (i) lire les entrées du système, (ii) prendre des décisions logiques, calculer les valeurs des sorties et (iii) émettre les sorties calculées au cycle courant. Cette mise en œuvre se détache des approches événementielles où le système réagit uniquement quand un événement est observé.

En général, chaque itération de la boucle réactive doit intervenir à une fréquence suffisamment grande pour réagir à temps à une évolution significative de l'environnement. Le temps de traitement entre l'instant où un événement est perçu par le système et le moment où celui-ci réagit ne doit pas dépasser une certaine limite, la borne temps-réel, qui dépend du système physique que l'on veut contrôler.

Approche synchrone

Aujourd'hui, les systèmes réactifs de grande taille constituent des programmes difficiles à modéliser et à tester. De plus, les systèmes critiques doivent satisfaire de fortes exigences de fiabilité et de sûreté, ce qui motive l'usage de méthodes formelles dans leur développement (p. ex. model-checking, interprétation abstraite, assistant de preuves, test de conformité). Différents langages formels de spécification ont été définis pour aider les automaticiens à spécifier des systèmes réactifs. Au cours des années 1980, la communauté des langages synchrones a défini des formalismes et des outils autour de l'approche synchrone mise en avant par Milner [Mil89] (SCCS [1]) et Berry (Esterel).

1. Synchronous Calculus of Communicating Systems

Parmi eux, on peut citer SIGNAL [BLG90], LUSTRE [BCH+85, CPHP87, Hal98, Hal05] et ESTEREL [Ber00, AB00, BPP99], qui reposent sur des styles différents de programmation [And05]. Nous nous intéressons au langage LUSTRE et ses variations.

Le langage LUSTRE et les flots de données

LUSTRE est l'abbréviation [Nav06] de « Lucid synchrone temps-réel », en référence au langage Lucid [WA85, AW75, AW76]. Le langage « [...] permet des manipulations formelles sur les programmes dans le but de faire des vérifications et des preuves de correction » [Ber86].

En LUSTRE, l'échelle de temps est consituée d'une succession d'étapes élémentaires appelées cycles : chaque cycle correspond à une itération de la boucle réactive. La structure de boucle est implicite car le langage manipule des séquences infinies de valeurs, des flots de données. Alors, les entrées d'un système comme ses sorties sont des flots ; les opérateurs du langage établissent des relations entre de telles suites de valeurs. L'approche « flot de données » permet de décrire explicitement les relations entre les données dans le temps.

Le langage LUSTRE fournit des opérateurs arithmétiques et logiques ainsi que des éléments de base du contrôle, comme les flots booléens, les opérateurs de comparaison, l'opérateur conditionnel « `if..then..else` ». Il fournit avant tout deux primitives importantes, à savoir l'opérateur d'initialisation, qui permet de distinguer les équations applicables au démarrage du système de celles présentes par la suite, et l'opérateur de décalage temporel : celui-ci observe la valeur d'un flot au cycle précédant le cycle courant ; en combinant plusieurs de ces opérateurs, on peut obtenir des programmes réactifs qui réalisent des traitements à partir de valeurs prises sur plusieurs cycles.

Enfin, une caractéristique importante est que les flots de données peuvent être cadencés à des ryhtmes différents, de sorte à activer ou non des calculs au cours du temps. Cela est fourni par les mécanismes d'horloges du langage : une horloge indique les instants où les flots sont présents ; les flots ainsi cadencés peuvent eux-même définir d'autres horloges, etc. Nous décrivons plus en détails le langage LUSTRE au chapitre 2.

Automates à états finis classiques

Le langage LUSTRE repose sur des équations de flots ce que le rend plutôt adapté pour représenter des traitements réguliers sur les entrées du système. En revanche, il est plus difficile de modéliser des états internes complexes et des transitions d'états conditionnelles. Il est reconnu que la dépendance entre les flots de contrôle et les flots de données est importante [MR00].

Pour décrire des systèmes à changement d'état, un formalisme qui vient naturellement à l'esprit est celui des automates à états finis classiques [BT01]. Un automate est un ensemble d'états états connectés entre eux par des transitions, activables selon des

conditions de franchissement. Le comportement logique d'un contrôleur au cours du temps est alors facile à décrire. Cependant, les automates finis classiques ne sont pas toujours adaptés aux systèmes réactifs réels et ne permettent par exemple pas de mélanger des automates à différents niveaux de granularité dans un même modèle. Il existe alors des formalismes étendus à base d'automates, adaptés aux besoins du domaine d'application des systèmes réactifs.

Automates étendus

Les outils Simulink et Matlab de l'éditeur *Mathworks*, apparus en 1984, sont largement utilisés pour modéliser des systèmes réactifs complexes. Ils facilitent le développement des logiciels critiques, d'un bout à l'autre de la chaîne de conception, notamment grâce au formalisme visuel des Statecharts [Har84, Har87] basé sur une description modulaire manipulant des automates hiérarchiques [Har88]. Dans un automate hiérarchique, il est possible d'imbriquer plusieurs automates les uns dans les autres et d'avoir une communication entre ces niveaux. On peut alors modéliser un système dans son ensemble, au niveau des composants de bas niveau comment au niveau global où ceux-ci intéragissent.

Des formalismes mixtes tels que SyncCharts [A$^+$96] ou Argos [Mar92] ont été définis et étendent de manière similaire la notion d'automates classiques. Par exemple, le langage SyncCharts définit des structures d'automates qui sont encodées dans le langage Esterel. Celui-ci opère sur des signaux et des blocs d'instructions imbriqués reposant sur de nombreux opérateurs de contrôle d'éxecution basés sur les signaux, comme « `await` », « `present` », « `emit` » ou encore l'opérateur de parallélisme « `||` ».

Il s'agit d'un formalisme adapté pour décrire des systèmes fortement orientés « contrôle », c'est-à-dire avec beaucoup d'états internes différents et des changements d'états fréquents. C'est un modèle modulaire qui permet de spécifier des composants à plusieurs niveaux. Cependant, les commandes effectivement calculées par un système décrit en Esterel sont reléguées à des fonctions externes au langage, qui doivent être liées par la suite lors de la compilation.

Automates de modes

Les automates de modes [MR98] héritent à la fois des automates hiérarchiques et des flots de données : les automates servent à définir des modes de calculs, ceux-ci étant définis par des équations de flots. Cela est adapté pour des contrôleurs où des séquences d'actions régulières doivent être coordonnées [MR00]. Le langage Stateflow étend le langage Statecharts pour intégrer ce principe de modes de calculs, mais ces deux langages partagent le même problème, à savoir une sémantique dans laquelle il est possible de modéliser des systèmes dont le comportement est indéterministe [HR04,

HGdR88]. Des travaux existent pour traduire un sous-ensemble sûr de ces langages vers Lustre [SSC+04], qui lui est formellement défini et déterministe.

Des approches hybrides multi-langages existent, comme dans le cas de l'intégration d'Argos [JLMR94, MH96] ou de AADL [JRLN07] avec Lustre, celle de SyncCharts avec SynDEx [PST03] ou encore celle de Statecharts avec Signal [BRG+01]. On peut aussi citer les formats « bas-niveau » commun aux langages synchrones, à savoir IC, OC, DC (GC), SC [PBM+93, GPSV+]. Ces approches suivent une démarche de traduction de modèles vers un langage commun. Cependant, le mélange de différents langages sources demande de s'assurer de la correction des outils de transformations de modèles. En revanche, avec un langage unique, il est possible de garantir que les modèles ont certaines propriétés intéressantes ; c'est ce qu'apporte le langage Scade 6 que l'on introduit dans la section suivante.

Langage Scade 6

Aujourd'hui, le langage et les outils autour de Lustre ont évolué et sont présents dans le milieu industriel. La plate-forme de développement de logiciels réactifs [2] Scade, d'Esterel Technologies [3], met l'accent sur les approches formelles dans la modélisation de systèmes réactifs. L'environnement de développement de Scade repose depuis ses débuts sur les langages Lustre et Esterel et fournit des outils d'analyse pour la validation et la vérification de systèmes, notamment son compilateur, qualifié pour le standard DO-178B/ED-12B (niveau A) pour l'avionique et les applications militaires.

Récemment encore, la plate-forme ne permettait de décrire des automates qu'à travers un langage appelé [And03, Dio04] SSM [4], basé sur SyncCharts, compilé alors vers des équations Lustre ne comportant qu'une seule horloge. Ces SSM peuvent entre autres être utilisés pour modéliser des automates de modes [LDB05]. Il s'agit là-aussi d'une approche hybride qui n'est pas satisfaisante : les flots mono-horloges sont tous calculés à chaque cycle, ce qui est pénalisant pour des gros systèmes ; en effet, la notion d'activation et de désactivation de tâches y est absente, bien qu'utile en pratique.

Des travaux récents successifs ont permis d'apporter à Lustre la notion d'horloges sur types énumérés [CP03], de flots réinitialisables et finalement d'automates de modes [CPP05a]. Ceux-ci exploitent la possibilité de combiner des flots cadencés selon plusieurs horloges. La dernière version du langage, Scade 6, est donc une évolution majeure qui propose le modèle d'automates de modes dans un langage unifié [Dor08]. L'ajout des opérateurs « `merge` » et « `reset` » dans le langage Lustre permet alors une compilation plus efficace et plus directe qu'avec des flots mono-horloges [BCHP08]. Il est ainsi possible de modéliser des sytèmes sur plusieurs niveaux d'abstraction, à l'aide de

2. Safety Critical Application Development Environment
3. `http://www.esterel-technologies.com/`
4. Safe State Machines

contrôleurs imbriqués, de synchroniser par rendez-vous des tâches parallèles, ou encore d'interrompre puis de reprendre/réinitialiser des calculs.

Approches formelles pour le test

Il est souvent nécessaire de s'assurer que les systèmes réactifs modélisés ne comportent pas d'erreurs, en particulier dans le cas des systèmes critiques où l'étape de validation et vérification constitue la part principale de l'effort de développement. Le test de logiciel [Whi00, Mat08, Mye08, Whi87] consiste à exécuter un programme dans le but de trouver ses défauts. L'activité de test complète d'autres approches de validation et vérification comme la preuve de programmes.

L'objectif d'un testeur est de confronter le comportement d'un système avec celui attendu. La génération de tests repose sur une formalisation du test logiciel [Ham94, Lay93, LGA96] et l'analyse statique ou dynamique de programmes. L'analyse directe d'un programme pour générer des tests est appelé *Code-Based Testing* et repose sur une analyse structurelle du programme sous test selon différentes techniques d'interpréation symbolique [Inc87, Cla76, Kin76].

Avec le test dirigé par les modèles, ou *model-based testing*, ce travail passe par l'utilisation d'un modèle, qui peut-être une représentation différente (complémentaire) du programme sous test [HP95, EFW01, Utt06, AD97, DJK+99]. Le modèle définit formellement le comportement attendu d'un système. Il s'agit d'une spécification qui a l'avantage d'être interpétable par des outils pour automatiser la création de jeux de test.

Le test à partir de modèles repose sur l'application d'approches formelles existantes [Kor90, Edv99, Sto93, CS94] et utilise ainsi des techniques diverses, comme le *model-checking*, les chaînes de Markov, et bien d'autres [BBLG97, PHR04, HMG06]. Un panorama intéressant des techniques de génération de séquences de tests peut-être trouvé dans [McM04]. Nous nous intéressons ici à l'utilisation des techniques de propagation et de résolution de contraintes dans le cadre du test logiciel, appelé *Constraint-Based Testing* [BBD+09, DO91, GBR00].

Test à base de contraintes

Le test à base de contraintes a été introduit en 1991 [DO91] mais suit des travaux antérieurs sur l'exécution symbolique de programmes [Cla76, Kin76]. Il vise à générer automatiquement des données de tests et utilise pour cela une traduction du programme/-modèle sous test en un problème de satisfaction de contraintes, ou CSP [5]. Nous présentons plus en détails la satisfaction de contraintes au chapitre **??**.

Parmi les outils qui mettent en œuvre du test à l'aide de contraintes, on peut citer INKA [CBG09, GBR98, GBW06, BGM+02, GLL08], pour la couverture structurelle de

5. Constraint Satisfaction Problem

programmes C, ainsi qu'EUCLIDE [Got09], qui hérite de la même approche. Dans ces deux cas, les programmes ainsi qu'un objectif de test sont convertis en un CSP. La modélisation d'un programme sous forme d'un CSP nous permet d'analyser son exécution et de forcer ses valeurs d'entrées à atteindre si possible le résultat requis par l'objectif de test.

Les outils comme PathCrawler [WMM04, WMMR05, BDK+09], CUTE [SMA05], DART [GKS05] ou EXE [CGP+08] raisonnent sur des prédicats de chemins pour tester des chemins d'exécution d'un programme (langage C). Dans PathCrawler, chaque prédicat de chemin est une conjonction de conditions de branchements, et une fois une branche couverte, la dernière condition est niée dans les contraintes puis résolue pour générer une entrée concrète qui visite un nouveau chemin du programme. Osmose [BH08, BH09] réalise simultanément la reconstruction et l'exploration du graphe de flots de contrôle d'un programme à partir d'un exécutable binaire, ainsi que l'évaluation concrète et symbolique de ce programme au niveau des représentations binaires des données. Les outils AGATHA [BFG+03], BZ-TT [BLP04] et LEIRIOS [JL06] sont d'autres outils de génération de tests qui reposent sur des techniques de résolution de contraintes. Nous nous intéressons dans cette thèse à GATeL.

GATeL

GATeL signifie Génération Automatique de séquences de Tests à partir de spécifications Lustre. Il produit des suites de valeurs pour les différents flots Lustre d'une spécification de sorte à atteindre un objectif de test fourni par le testeur. C'est un outil dérivé de travaux de formalisation et d'implémentation [BGM91b, BGM91a, ALGM96] sur la génération de tests à partir de spécifications algébriques [AAB+06, Mar91, AALGL09, Gau01] destiné à la génération automatique de séquences de tests pour les systèmes réactifs décrits dans le langage Lustre. Les premiers travaux [RTFA+24] autour de GATeL passaient par une interprétation du problème de la génération automatique de séquences de test selon des spécifications algébriques dans l'outil LOFT [Mar95] (LOgic for Functions and Testing). Ils ont donné lieu à une première version [MA00] de GATeL, qui a évolué pour prendre en compte les flots structurés (tuples, tableaux statiques) ainsi que le type `real` ou encore la simulation de modèles et l'analyse de couverture de séquences existantes [GRMB04, BM05, MB05]. L'outil peut être utilisé pour faire du test structurel ou fonctionnel [BDL+06]. Nous nous intéressons ici à la prise en charge des flots multi-horloges [JB09, BJM+10].

La programmation logique par contraintes est une approche souvent utilisée dans le cadre de la génération de données de test. Le programme sous test est généralement modélisé en un ensemble statique de contraintes, qui est ensuite résolu : on analyse un modèle, pour construire un système de contraintes, que l'on passe au solveur. Dans GATeL, les contraintes apparaissent et disparaissent au cours de la propagation de contraintes, c'est-à-dire aussi pendant la recherche de solutions. Cette manière

dynamique de filtrer les contraintes, avec une approche paresseuse, constitue l'originalité principale de GATeL par rapport aux autres outils de tests. En introduisant des nouvelles contraintes uniquement quand elles sont nécessaires, on peut explorer uniquement la partie du modèle qui correspond au cas de test souhaité et traiter des programmes Lustre de taille importante.

L'outil repose sur une introduction par chaînage arrière de contraintes, c'est-à-dire depuis le cycle final, où l'objectif de test est atteint, vers un cycle initial, et depuis les sorties d'un système vers ses entrées. GATeL tisse ainsi un réseau de contraintes entre des variables représentant les valeurs des flots dans le temps, de sorte à représenter les relations entre ces flots telles qu'elles sont décrites par les équations du système et par l'objectif de test. Le fait de contraindre des valeurs à des endroits de ce réseau permet de propager des réductions de domaines et de produire des séquences compatibles à la fois avec la spécification et l'objectif que l'on cherche à observer. L'introduction des contraintes est réalisée de manière dynamique, c'est-à-dire pendant le processus de filtrage des contraintes existantes. Nous détaillons cet outil au chapitre 3.

La génération de séquences de tests en arrière selon un objectif de test est complémentaire aux approches existantes pour le test de systèmes décrits en Lustre, basées sur des approches en avant comme c'est le cas pour les outils LURETTE [RWNH98], LUTESS [MPP10] et LESAR [HLR92], où les séquences sont construites depuis le cycle initial pour vérifier des invariants. Il existe par ailleurs un outil de vérification de systèmes Scade, *Design Verifier* [ADS+06, JH05], intégré à la plate-forme de développement du même nom et complémentaire à la méthode paresseuse de GATeL. Il s'agit d'un outil de preuve qui cible les propriétés de sûreté de fonctionnement par induction sur la taille des séquences. Il peut générer des contre-exemples lorsqu'une propriété n'est pas vérifiée.

Plan et contributions

Notre sujet de thèse consiste à concevoir un outil de test pour les modèles Scade 6 en partant de GATeL et en faisant en sorte que celui-ci prenne en compte les flots temporisés ainsi que les automates de modes. L'outil était à l'origine conçu pour générer des tests à partir de flots Lustre mono-horloges et nous faisons en sorte qu'il traite de manière interne, dans le noyau de génération de séquences, à la fois les horloges et les informations de haut-niveau concernant les automates.

Au chapitre 1, nous introduisons la propagation dynamique de contraintes et des éléments de syntaxe qui nous serviront tout au long de la thèse. GATeL repose sur la programmation logique par contraintes ; il est écrit en Prolog, pour le système `ECLiPSe` [6]. En pratique, les variables, domaines et contraintes sont ajoutées et/ou retirées dynamiquement du système à travers des mécanismes de suspension et de réveil de

6. ECLiPSe Constraint Logic Programming System

prédicats. Nous ne souhaitons pas décrire l'outil à ce niveau de détails, et en particulier, nous éviterons les problèmes d'ordonnancement des règles de filtrage. C'est pourquoi nous définissons le langage de contraintes $\mathcal{C}$ où le mécanisme de propagation est vu comme une réécriture d'un CSP vers un autre. Ces réécritures apparaissent comme des règles d'inférences, où d'autres prédicats interviennent et modélisent alors les raisonnements logiques et les fonctions auxiliaires. En passant par une réécriture de CSP, nous pouvons ajouter ou retirer à chaque étape de filtrage des contraintes et des variables et nous concentrer sur le niveau de détails qui nous intéresse.

Au chapitre 2, nous rappelons la sémantique de Lustre classique dit mono-horloge mais aussi les extensions concernant les horloges. Nous définissons alors un langage nommé $\mathcal{L}_+$ pour lequel Lustre mono-horloge est un sous-ensemble noté $\mathcal{L}_1$. Nous donnons alors une fonction d'évaluation en avant pour le langage multi-horloges. Celle-ci est paramétrée par une grille de valeurs G. Lorsqu'un CSP admet une solution, nous pouvons en extraire une grille de valeurs G et vérifier à l'aide de notre fonction d'évaluation si G est bien correcte et nous permet d'atteindre l'objectif de test désiré.

Au chapitre 3, nous avons une démarche symétrique à la fonction d'évaluation qui cherche à générer une grille de valeurs pour un objectif de test donné. Nous présentons l'approche choisie par GATeL en détaillant les travaux existants et en les unifiant dans notre langage de contraintes. Nous définissons ainsi le langage $\mathcal{L}_1^g$ qui étend $\mathcal{L}_1$ pour prendre en compte un objectif de test et des primitives propres à GATeL. Dans le langage $\mathcal{L}_1^g$, nous manipulons alors des Problèmes de Génération de Séquences de Test (PGST). et donnons la traduction de $\mathcal{L}_1^g$ vers une représentation sous forme de contraintes du langage $\mathcal{C}_1^g$, sous-ensemble de $\mathcal{C}$. Nous nous reposons sur le langage de contraintes vu au chapitre 1 pour décrire l'évolution dynamique du CSP et voir comment l'approche paresseuse de GATeL permet d'explorer des modèles Lustre mono-horloges. Nous montrons alors la sur-approximation d'expressions Lustre, les mécanismes de sélection de cas de tests et de résolution de contraintes.

Au chapitre 4, nous modifions les règles de propagation de GATeL mono-horloge, que l'on appelle alors GATeL_1, pour obtenir une version multi-horloges baptisée GATeL_+. Nous définissons de même que précédemment un langage $\mathcal{L}_+^g$, qui est le langage $\mathcal{L}_+$ muni de directives GATeL, et la traduction de ce langage en une base de faits du langage $\mathcal{C}_+^g$. Nous nous intéressons aux contraintes temporelles entre les horloges, les flots de données et le calcul des cycles initiaux d'un système. Nous exploitons ces contraintes pour faire circuler de l'information à travers la hiérarchie d'horloges d'un système et présentons alors les modifications importantes de GATeL liées à la propagation de contraintes. La figure 1 montre les différents langages manipulés dans la thèse.

	Mono-horloge	Multi-horloges
Lustre	$\mathcal{L}_1$	$\mathcal{L}_+$
PGST (Lustre + Directives GATeL)	$\mathcal{L}_1^g$	$\mathcal{L}_+^g$
Représentation sous forme de contraintes	$\mathcal{C}_1^g$	$\mathcal{C}_+^g$

Figure 1: Langages pour la modélisation respective des systèmes réactifs, des problèmes de génération de séquences de test et de leurs traductions en ensembles de contraintes.

Au chapitre 5, nous tirons parti des informations de haut-niveau que l'on peut extraire de modèles Scade 6 pour renforcer les contraintes de GATeL$_+$: les structures de haut-niveau des automates ayant disparues après traduction vers le langage intermédiaire $\mathcal{L}_+$, nous les réintroduisons en tant que contraintes dans le système après traduction. Nous nous intéressons par la suite à la couverture structurelle d'automates de modes de Scade 6. Il n'existe pas à notre connaissance de tels critères directement pour les automates, bien qu'il en existe pour le langage Lustre multi-horloges [PMDBP09]. Nous présentons une adaptation des critères propres à *Statecharts* aux automates de modes qui nous intéressent. De plus, nous montrons comment ces critères sont facilement exprimables sous forme de flots observateurs du langage $\mathcal{L}_+$ et ainsi exploitables par GATeL$_+$ pour générer automatiquement des séquences qui les couvrent. Pour cela, nous donnons une approche automatique de transformation des critères en flots observateurs par le biais d'une instrumentation du modèle sous test.

Chapitre 1

Programmation Logique par Contraintes

Le sujet de thèse fait appel à des notions de programmation logique par contraintes, une approche de modélisation et de résolution de problèmes combinatoires et/ou décisionnels [Fag96, JL87, JM94, Bar01]. La programmation logique par contraintes [VH91] combine l'aspect déclaratif et logique de la programmation logique [KoEDoCL73] avec la résolution de contraintes, qui elle consiste à représenter un problème de décision à travers des variables, des domaines de valeurs et des relations entre ces variables. Nous avons travaillé au cours de la thèse avec `ECLiPSe`, un système pour la résolution de contraintes basé sur le langage Prolog [Kow88, CR96].

Il existe de nombreuses applications à cette approche, notamment pour résoudre des problèmes d'ordonnancement et la planification de tâches [Sim01], mais aussi dans le cadre de l'analyse de programmes [GBR00, WMMR05, MA00, BBD$^+$09].

Nous rappelons dans ce chapitre les notions de programmations logiques par contraintes nécessaires aux chapitres suivants. Pour cela, nous montrons tout d'abord un exemple de problème de satisfaction de contraintes, sa modélisation ainsi que sa résolution. Ensuite, nous définissons un langage et une sémantique pour les mécanismes de filtrage et de résolution.

1.1 Send + More = Money

> **Définition 1**. *Problème de satisfaction de contraintes*
>
> Un problème de satisfaction de contrainte, ou CSP, est donné par un triplet (V, D, C), où V est un ensemble de variables, D l'ensemble des domaines de valeurs associés à chaque variable de V et C l'ensemble des contraintes portant sur ces variables. Chaque contrainte c de C est une relation portant sur un sous-ensemble V_c de V. La contrainte c définit alors l'ensemble des valeurs que peuvent prendre simultanément les variables de V_c.

Par exemple, soient X, Y et Z des variables d'un problème de satisfaction de

contraintes $P = (V, D, C)$. On pose :

$$V = \{X, Y, Z\}$$

On suppose que chaque variable peut prendre une valeur entre 0 et 10 ; pour tous entiers relatifs a et b avec $a \leq b$, la notation $a..b$ dénote un intervalle dans les entiers relatifs, à savoir l'ensemble des entiers compris entre a et b inclus. On peut donc noter $X \in 0..10$, $Y \in 0..10$ et $Z \in 0..10$. Alors, l'ensemble D est le suivant (implicitement, l'ensemble des domaines D est ordonné comme V) :

$$D = \{0..10, 0..10, 0..10\}$$

Enfin, on souhaite que les variables respectent la contrainte « $Z = X + Y$ ». On pose donc :

$$C = \{Z = X + Y\}$$

Une solution de ce CSP est un triplet de valeurs, pour X, Y et Z, qui appartiennent aux domaines souhaités et qui respectent l'égalité donnée par l'unique contrainte du problème. Par exemple, $(X, Y, Z) = (3, 4, 7)$ est une solution ; le triplet $(3, 3, 8)$ n'est pas une solution car la contrainte n'est pas respectée ; le triplet $(8, 3, 11)$ n'est pas une solution car la valeur de Z sort du domaine 0..10.

1.1.1 Exemple

Le problème suivant est un casse-tête logique dans lequel les lettres de l'addition doivent être remplacées par les chiffres appropriés afin que l'équation soit vraie.

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

Il a été publié pour la première fois en 1924 dans le volume 68 de *Strand Magazine*, par Henry Ernest Dudeney, et utilisé à plusieurs reprises pour illustrer des méthodes de résolutions automatiques, notamment à l'aide de la programmation par contraintes [Apt98, Fag96, Cod95]. Chacune des lettres S, E, N, D, M, O, R et Y correspond à un chiffre distinct. Par exemple, S et E ne peuvent tous les deux valoir 1. On considère de plus que ni M ni S, les chiffres de plus grand rang, ne peuvent valoir zéro. Nous modélisons ce problème à l'aide de contraintes.

1.1.2 Méthode de résolution

On souhaite résoudre le problème précédent ; pour cela, on donne une valeur à une seule des variables, par exemple X. On pose pour cela $X = 5$, une valeur qui est bien

dans le domaine de X. On vient d'*instancier* X à la valeur 5, ce qui réduit l'ensemble des solutions possibles au problème. La contrainte s'écrit maintenant « $Z = 5 + Y$ » ; en la projetant sur les variables Y et Z, on peut réduire leurs domaines respectifs.

Par exemple, on sait que Z ne peut pas valoir moins de 5, car Y est positive ou nulle. Donc, $Z \in 5..10$. De même, Y ne peut pas valoir plus de 5 : toute valeur supérieure ou égale à 6 donnerait un résultat supérieur à 10. Donc $Y \in 0..5$. On peut alors choisir une valeur pour Z, par exemple 9, auquel cas la projection de la contrainte nous donne $Y = 4$. On a ainsi retiré au fur et à mesure des instantiations de variables des ensembles de valeurs qui ne conduisaient pas à une solution, afin de restreindre les choix possibles par la suite.

Chaque contrainte admet des règles de filtrage, qui retirent les combinaisons de valeurs qui ne satisfont pas cette contrainte et ne font donc pas partie des solutions au problème général.

1.1.3 Modélisation du problème

Soit $S = (V, D, C)$ le problème de satisfaction de contraintes correspondant à l'exemple qui nous intéresse. L'ensemble V de variables est défini ainsi :

$$V = \{S, E, N, D, M, O, R, Y, C_1, C_2, C_3\}$$

Les variables C_1, C_2 et C_3 sont associées aux retenues de l'addition (*carry*), qui sont nécessaires pour le calcul. L'addition suivante détaille le problème en posant les retenues de manière explicite sur la première ligne :

$$\begin{array}{cccccc} & M & C_3 & C_2 & C_1 & 0 \\ & 0 & S & E & N & D \\ + & 0 & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

On note $dom(V)$ le domaine appartenant à D correspondant à toute variable V. La fonction *dom* est définie ainsi dans notre exemple :

- $dom(M) = dom(S) = 1..9$, car ni M ni S ne peuvent valoir zéro.
- $dom(X) = 0..9$, pour tout $X \in \{E, N, D, O, R, Y\}$.
- $dom(C_i) = \{0, 1\}$, pour tout $i \in \{1, 2, 3\}$, car la retenue pour une addition de deux nombres ne peut pas être supérieure à un.

L'ensemble des contraintes C est composé d'une contrainte *alldifferent* et de cinq contraintes *addcarry*.

alldifferent Dans notre exemple, il n'est pas possible que deux lettres représentent le même chiffre. La contrainte *alldifferent*(E) décrit le fait que les valeurs de chaque variable de l'ensemble E sont différentes. L'ensemble de contraintes C contient ainsi le terme *alldifferent*($\{S, E, N, D, M, O, R, Y\}$).

addcarry La contrainte $addcarry(A, B, IC, OC, R)$ est une contrainte représentant l'addition avec retenues au niveau d'une colonne, ou rang, d'une addition. Soient A, B, IC, OC et R des chiffres avec IC la retenue issue du calcul au rang précédent et OC la retenue produite au rang courant. Le filtre associé à cette contrainte s'assure que les propriétés suivantes sont toujours vraies :

$$A + B + IC = 10 * OC + R \qquad IC \in \{0, 1\} \qquad OC \in \{0, 1\}$$

$$\begin{array}{cccc} & (OC) & (IC) & \cdots \\ & \cdots & A & \cdots \\ + & \cdots & B & \cdots \\ \hline & \cdots & R & \cdots \end{array}$$

Cette contrainte est dupliquée autant de fois qu'il y a de colonnes dans l'addition. L'ensemble de contraintes C du problème qui nous intéresse contient les cinq contraintes *addcarry* suivantes :

$addcarry(0, 0, M, 0, M)$
$addcarry(S, M, C_3, M, O)$
$addcarry(E, O, C_2, C_3, N)$
$addcarry(N, R, C_1, C_2, E)$
$addcarry(D, E, 0, C_1, Y)$

$$\begin{array}{cccccccc} & 0 & M & C_3 & C_2 & C_1 & 0 \\ & & 0 & S & E & N & D \\ + & & 0 & M & O & R & E \\ \hline & & M & O & N & E & Y \end{array}$$

1.1.4 Propagation de contraintes

La propagation de contraintes consiste à appliquer des règles de filtrages pour chaque contrainte du problème.

> **Définition 2.** *Filtrage d'un problème de satisfaction de contraintes*
>
> Le filtrage consiste à réduire les domaines des variables d'un CSP en retirant les valeurs qui ne respectent pas les contraintes.

Pour illustrer le filtrage, imaginons un CSP avec deux variables A et B, telles que $A \in 1..5$ et $B \in 1..3$. De plus, ces variables sont contraintes par la relation $A + B = 7$. La situation avant le filtrage de cette contrainte correspond à la figure 1.1 : chaque axe correspond aux valeurs potentielles pour A et B et la région hachurée représente l'ensemble des combinaisons possibles pour A et B (seulement les points qui correspondent à des valeurs entières). La droite en diagonale représente l'ensemble des points tels que $A + B = 7$ (l'équation de la droite est donc $B = 7 - A$). Seuls les points qui appartiennent à cette droite peuvent être solution du problème. Le filtrage correspond alors à projeter la contrainte $A + B = 7$ sur les domaines de valeurs de A

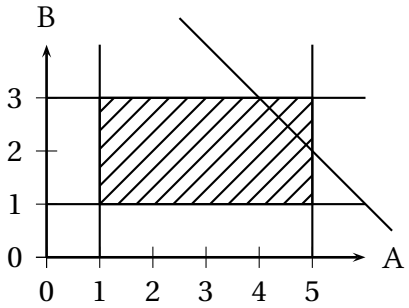


FIGURE 1.1: Avant filtrage

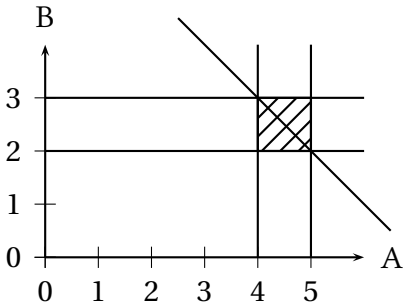


FIGURE 1.2: Après filtrage

et B, respectivement, comme on le voit à la figure 1.2, où les nouveaux domaines sont respectivement 4..5 et 2..3 pour A et B. En effet, la droite possède deux intersections avec la zone hachurée, aux points $(4,3)$ et $(5,2)$, qui sont les deux solutions possibles. Cette contrainte n'admet pas de propagation supplémentaire, mais si par la suite on choisissait une valeur pour A (resp. B) parmi son domaine, le filtrage pourrait alors s'appliquer pour trouver la valeur de B (resp. A) correspondante, de sorte que l'on ait bien $A+B=7$.

Comme les contraintes sont souvent liées à travers des variables partagées, elles échangent indirectement de l'information entre-elles, si bien qu'une réduction de domaine dans une contrainte peut permettre le filtrage d'une autre. La propagation de contraintes s'effectue sur un CSP, en y appliquant autant de filtrage que possible au niveau de chaque contrainte, jusqu'à produire un nouveau CSP dans lequel les domaines ont été réduits au maximum : les filtrages effectués localement sur les contraintes ne détectent plus de réduction de domaines possibles. Le CSP obtenu est le point fixe de la fonction de propagation, appliquée au problème initial.

1.1.5 Propagations pour SEND+MORE=MONEY

Nous reprenons l'exemple précédent pour montrer plusieurs étapes de propagation. Tout d'abord, la contrainte *addcarry*(S,M,C_3,M,O) impose que la retenue produite, ici M, ne puisse prendre comme valeur que 0 ou 1. Or, le domaine de M est initialement 1..9. On peut en déduire que la variable M est nécessairement égale à 1. De plus, mises à part les retenues, les autres variables ne pourront pas valoir 1, d'après la contrainte *alldifferent*. Le nouveau système S_1 correspond alors à cette équation :

$$\begin{array}{rccccc} & 1 & C_3 & C_2 & C_1 & 0 \\ & 0 & S & E & N & D \\ + & 0 & 1 & O & R & E \\ \hline & 1 & O & N & E & Y \end{array}$$

Par ailleurs, la contrainte $addcarry(S, 1, C_3, 1, O)$ exprime l'équation suivante :

$$\begin{aligned} S+1+C_3-10-O &= 0 \\ S+C_3-9-O &= 0 \\ S &= O+9-C_3 \end{aligned} \tag{1.1}$$

On sait que S est différent de 1 et de 0 : le domaine de S est $\{2,\ldots,9\}$. On a alors l'inégalité $2 \leq S \leq 9$. D'un autre côté, on a $dom(C_3) = \{0,1\}$ et $dom(O) = \{0,\ldots,9\}$. On peut donc en déduire un encadrement pour le membre droit de l'équation 1.1 précédente :

$$\begin{array}{lcccl} 0 & \leq & O & \leq & 9 \\ 9 & \leq & O+9 & \leq & 18 \\ 8 & \leq & O+9-C_3 & \leq & 18 \quad \text{(car on a } -1 \leq -C_3 \leq 0\text{)} \end{array}$$

Finalement, on en déduit un nouvel encadrement pour S :

$$\begin{array}{rccccc} \text{On a} & 8 & \leq & S & \leq & 18 \\ \text{Or} & 2 & \leq & S & \leq & 9 \\ \text{Donc} & 8 & \leq & S & \leq & 9 \end{array}$$

Le domaine de S est ainsi réduit à $\{8,9\}$. Toujours en considérant les intervalles de valeurs, l'équation 1.1 implique l'égalité $O = S+C_3-9$. Sachant que $S \in \{8,9\}$ et $C \in \{0,1\}$, la somme en membre droit mène à l'inégalité $-1 \leq O \leq 1$, qui permet de contraindre la variable O par l'inégalité $0 \leq O \leq 1$. Par ailleurs, on sait que O ne peut pas valoir 1, qui est déjà affecté à M. Nécessairement, $O = 0$. Il est encore possible de filtrer les domaines de valeurs, grâce à cette information, et l'on obtient ainsi $C_3 = 0$ et $S = 9$. Dans notre exemple, la première étape de propagation aboutit au système de contraintes S_2 représenté par l'addition suivante, avec $dom(X) = \{2,\ldots,8\}$ pour tout $X \in \{E,N,D,R,Y\}$:

$$\begin{array}{cccccc} & 1 & 0 & C_2 & C_1 & 0 \\ & 0 & 9 & E & N & D \\ + & 0 & 1 & 0 & R & E \\ \hline & 1 & 0 & N & E & Y \end{array}$$

Ce système de contraintes S_2 n'admet plus de filtrage : il s'agit par définition du point fixe de la fonction de propagation (l'image de l'ensemble par filtrage est le même ensemble). Cependant, il reste encore des variables non instanciées dans ce système.

1.1.6 Recherche de solutions

La résolution d'un système de contraintes passe par des techniques de recherches de solutions, dont l'objectif est de donner une valuation à chaque variable du CSP de sorte à ce que les contraintes soient satisfaites. La propagation de contraintes sert à réduire l'espace de recherche en retirant les valeurs qui provoquent des inconsistences locales au

niveau des contraintes, retirant quand elle est efficace un grand nombre de combinaisons inutiles.

Pour trouver une solution, une méthode simple de recherche consiste à instancier aléatoirement une variable, en lui donnant une valeur parmi celles couvertes par son domaine. Une fois la variable affectée, on obtient un sous-problème plus simple pour lesquels les règles de filtrage peuvent à nouveau s'exercer et potentiellement nous rapprocher d'une solution. Il est possible, cependant, que des contraintes puissent faire échouer la propagation. Si le problème est insatisfaisable, la procédure de recherche revient en arrière et choisit une autre valeur, ou une autre variable à instancier, par une procédé appelée *backtracking*. À chaque fois que le CSP n'admet plus de filtrage possible des domaines, une nouvelle hypothèse est faite sur la valeur d'une variable. En imbriquant ainsi les instantiations de variables et en parcourant les combinaisons par *backtrack*s successifs, on garantit que la procédure de résolution finit par tester toutes les combinaisons possibles de variables et de valeurs, pour des domaines et des ensembles de variables finis. Par exemple, la recherche de solution, aussi appelée communément *labeling*, peut suivre les étapes suivantes à partir du système S_2 précédent :

1.1.6.1 Instancier C_2 à 0

- la contrainte $addcarry(E, O, C_2, C_3, N)$ devient égale à $addcarry(E, 0, 0, 0, N)$, ce qui implique $E = N$; cela est impossible d'après la contrainte *alldifferent*, qui détecte l'insatisfaisabilité du sous-problème.
- La propagation échoue : on annule le dernier choix effectué (*backtrack*).

1.1.6.2 Instancier C_2 à 1

- La propagation réussit :

$$\begin{array}{cccccc} & 1 & 0 & 1 & C_1 & 0 \\ & 0 & 9 & E & N & D \\ + & 0 & 1 & 0 & R & E \\ \hline & 1 & 0 & N & E & Y \end{array}$$

$dom(E) = 2..7$, $dom(N) = 3..8$, $dom(D) = 2..8$, $dom(R) = 3..8$, $dom(Y) = 2..8$, $dom(C_1) = \{0, 1\}$

- Instancier C_1 à 0
  - L'hypothèse implique la fois $R = 9$ et $dom(R) = \{7, 8\}$ ce qui est contradictoire.
  - La propagation échoue car le sous-problème est insatisfaisable (*backtrack*).
- Instancier C_1 à 1
  - La propagation réussit
  - Instancier E à 5
    - La propagation réussit, toutes les variables sont instanciées.

– La solution trouvée correspond à l'équation suivante :

$$\begin{array}{cccccc} & 1 & 0 & 1 & 1 & 0 \\ & & 9 & 5 & 6 & 7 \\ + & & 1 & 0 & 8 & 5 \\ \hline & 1 & 0 & 6 & 5 & 2 \end{array}$$

1.2 Langage de contraintes

Nous avons introduit la programmation par contraintes en décrivant ce qu'est un CSP à l'aide d'un triplet (V, D, C), qui est la représentation usuelle. Cependant, pour pouvoir décrire les étapes de propagation de contraintes, nous préférons assimiler un CSP à un unique ensemble, contenant à la fois les contraines et les relations d'appartenance d'une variable à un domaine. Autrement dit, les domaines sont vus comme des contraintes, entre une variable et un terme représentant un ensemble de valeurs.

Nous utilisons cette représentation à l'aide d'un unique ensemble de termes pour décrire facilement des CSP dynamiques [JB09], où des contraintes apparaissent et disparaissent au cours des filtrages successifs. Pour cela, le filtrage est donné comme une transformation d'un ensemble de contraintes vers un autre ensemble. L'application répétée des règles correspond à la propagation de contraintes, qui, combinée à une stratégie de recherche, permet de passer d'un problème initial à un problème résolu.

Nous essayons de séparer la partie purement orientée contraintes des autres algorithmes combinant des règles de réécriture (contraintes) et des règles d'inférence (logique). Cette formalisation nous permet de décrire les règles de filtrage sans se préoccupper de la stratégie d'application des règles (du moment qu'elles sont applicables), et sans avoir à rentrer dans les détails ou les choix d'implémentation liés à la gestion des contraintes.

1.2.1 Ensemble de contraintes

Le système de contraines original de l'addition SEND+MORE=MONEY peut-être représenté par l'ensemble S_{smm} suivant :

$$\begin{aligned} S_{smm} = \{ \quad & M \in 1..9, S \in 1..9, \\ & E \in 0..9, N \in 0..9, D \in 0..9, \\ & O \in 0..9, R \in 0..9, Y \in 0..9, \\ & C_1 \in 0..1, C_2 \in 0..1, C_3 \in 0..1, \\ & \mathit{alldifferent}([S, E, N, D, M, O, R, Y]), \\ & \mathit{addcarry}(0, 0, M, 0, M), \mathit{addcarry}(S, M, C_3, M, O), \\ & \mathit{addcarry}(E, O, C_2, C_3, N), \mathit{addcarry}(N, R, C_1, C_2, E), \\ & \mathit{addcarry}(D, E, 0, C_1, Y) \qquad \} \end{aligned}$$

Avec ces notations, la propagation de contraintes est vue comme une réécriture d'un ensemble de contraintes à un autre. Chaque élément d'un CSP est un terme du langage $\mathcal{C}$ défini ci-après.

1.2.2 Langage de contraintes

La définition 3 introduit le langage de contraintes $\mathcal{C}$ et le langage de termes $\mathcal{T}$. Les mots acceptés par ce dernier sont appelés des termes : chaque terme est soit un terme composé de sous-termes, soit un terme atomique. Un terme atomique est alors un nombre réel, une variable ou un mot exprimant une valeur symbolique (un *atome*). Le langage de contraintes $\mathcal{C}$ est un sous-ensemble de $\mathcal{T}$: une contrainte peut-être un atome ou un terme composé (mais pas une valeur).

Définition 3. *Langages de contraintes et de termes*

On définit les langages $\mathcal{C}$ et $\mathcal{T}$ par la grammaire suivante, à travers les deux symboles non-terminaux respectifs *Contrainte* et *Terme* :

Contrainte	::=	**ATOME**	— Littéral symbolique
	|	**ATOME** (*Terme* { , *Terme* })	— Terme composé
Terme	::=	**VALEUR**	— Valeur numérique
	|	**VARIABLE**	— Variable logique
	|	*Contrainte*	— Atome/Terme composé
VALEUR	∈	$\mathbb{Z}$	
ATOME	∈	$\mathbb{A}$ = `([a-z][_a-z]*)` ∪ **INFIX**	
INFIX	∈	[≡≢∩∅ = + − * ∈ []]	
VARIABLE	∈	$\mathbb{V}$ = `[A-Z][A-Z_a-z0-9]*`	

Un *atome* est une valeur symbolique commençant par une minuscule, ou un des symboles mathématiques donnés ci-dessus. Les identifiants de *variables* commencent par une majuscule. Un terme atomique est soit un atome soit une valeur numérique. Pour un terme $f(T_1, \ldots, T_n)$ quelconque, avec $n > 0$ et $T_i \in \mathcal{C}$ pour tout $i \in \{1, \ldots, n\}$, l'atome $f \in \mathbb{A}$ est appelé foncteur et n l'arité du terme. Un atome est son propre foncteur et a pour arité zéro.

Par exemple, les termes suivants appartiennent au langage $\mathcal{C}$: *fail*, *true*, 10.5, $add(3, X, 7)$ ou encore « $\in (Y, fd([](1, [](8, []))))$ ». Toutes les contraintes sont sous une notation préfixée : elle est pratique car toutes les contraintes ont la même structure, mais elle peut être difficile à lire. Nous introduisons une syntaxe simplifiée, dans laquelle la contrainte précédente s'écrit de manière équivalente « $Y \in fd([1, 8])$ ».

1.2.2.1 Notation simplifiée

Intervalles Le domaine des intervalles d'entiers que l'on a vu précédemment, $x..y$, est une notation simplfiée qui cache un terme plus long, *domint(x,y)*, qui représente le domaine des valeurs entières entre x et y (inclus). L'intervalle vide est noté $\emptyset$.

Notation infixe Les termes dont le foncteur appartient à $\{\equiv, \not\equiv, \cap, \emptyset, =, +, -, *, \in\}$ et dont l'arité est 2 admettent une notation infixe, comme par exemple « $X \in D$ » ou « $A \equiv 10$ ».

Listes Tout terme $[](H, T)$ est interprété comme une liste de valeurs, où H est l'élément de tête (*head*) et T une sous-liste (*tail*). L'atome « [] » seul représente la liste vide (on l'assimile à l'ensemble vide « $\emptyset$ » dans les opérations sur les domaines). Le terme $[](H, T)$ est noté de préférence $[H|T]$. Il est possible d'écrire directement plusieurs éléments en tête d'une liste, en écrivant $[H_1, \ldots, H_n|T]$; ceci est équivalent au terme $[H_1|[\ldots|[H_n|T]\ldots]]$. Si la liste finale est vide, comme dans le terme $[H_1, \ldots, H_n|[]]$, la liste peut s'écrire plus simplement $[H_1, \ldots, H_n]$. On note $\alpha - list$ l'ensemble de termes représentant des listes dont les éléments appartiennent au même ensemble α.

Tuples Il arrive que l'on ait veuille représenter un tuple à n éléments x_1 à x_n. Or, tout terme composé doit avoir un foncteur. On note donc *tuple(x_1,…,x_n)* le terme composé représentant ce tuple, mais nous autorisons la notation simplifiée $(x_1, \ldots, x_n)$ pour représenter ce terme dans le langage. Le foncteur *tuple* est ansi omis pour plus de lisibilité.

1.2.3 Systèmes de contraintes

Avec le langage de termes ainsi défini, nous définissons un CSP comme étant un ensemble de contraintes de $\mathcal{C}$.

> **Définition 4.** *Problème de satisfaction de contraintes*
>
> On appelle problème de satisfaction de contraintes tout élément de $\mathscr{P}(\mathcal{C})$, l'ensemble des parties de $\mathcal{C}$.

1.2.4 Fonctions auxiliaires

Nous définissons trois fonctions auxiliaires portant à la fois sur les termes et sur les ensembles de termes. La première est l'égalité structurelle entre termes, étendue aux ensembles. La deuxième est la fonction *vars*, donnant l'ensemble des variables incluses dans un terme, étendue elle-aussi aux ensembles de contraintes. Enfin, nous définissons la substitution d'une variable logique par un terme, qui nous permettra de définir les règles d'unification de termes dans les sections suivantes.

1.2.4.1 Égalité structurelle

L'égalité entre termes se ramène à l'égalité structurelle classique ensembliste. Ainsi, on note « $A = B$ » le fait que deux termes A et B soient égaux (on dit aussi identiques).

1.2.4.2 Variables d'un terme

Nous définissons la fonction *vars* qui détermine l'ensemble des variables présentes dans un terme et ses sous-termes.

Définition 5. *Variables présentes dans terme*

La fonction $vars : \mathcal{T} \to \mathscr{P}(\mathbb{V})$ est définie inductivement, selon la nature du terme en paramètre, de la manière suivante :

$$\begin{array}{rcll} vars(x) & = & \emptyset & \forall\ x \in \mathbb{Z} \cup \mathbb{A} \\ vars(V) & = & \{V\} & \forall\ V \in \mathbb{V} \\ vars(f(T_1, \ldots, T_n)) & = & \bigcup\limits_{1 \leq i \leq n} vars(T_i) & \forall\ n \in \mathbb{N}^*, \forall\ f \in \mathbb{A}, \forall\ (T_1, \ldots, T_n) \in \mathcal{T}^n \end{array}$$

On dit qu'un terme $t \in \mathcal{T}$ est *clos* s'il ne dépend d'aucune variable, c'est-à-dire si $vars(t) = \emptyset$. Nous définissons aussi l'ensemble des variables d'un ensemble de contraintes. Pour cela nous surchargeons la fonction *vars* et l'étendons aux ensembles de termes :

Définition 6. *Variables d'un ensemble de termes*

Soit S un ensemble appartenant à $\mathscr{P}(\mathcal{T})$. L'ensemble des variables de S est noté $vars(S)$; il correspond à l'union des variables apparaissant dans les termes présents dans S :

$$vars(S) = \bigcup_{t \in S} vars(t)$$

1.2.4.3 Exemple

Soit $C_1 = \{geq(A, B), A \in 1..10, B \in 5..12\}$ un problème de satisfaction de contraintes. A et B sont des variables, chacune d'elle associée à un domaine. La contrainte $geq(A, B)$ signifie que A est supérieur ou égal à B (*greater or equal*). L'application de *vars* au CSP permet de retrouver les variables du problème, A et B :

$$\begin{array}{rcl} vars(C_1) & = & vars(geq(A, B)) \cup vars(A \in 1..10) \cup vars(B \in 5..12) \\ & = & (vars(A) \cup vars(B)) \cup \\ & & (vars(A) \cup vars(1..10)) \cup (vars(B) \cup vars(5..12)) \\ & = & \{A, B\} \cup \{A\} \cup \{B\} \\ & = & \{A, B\} \end{array}$$

1.2.4.4 Substitution

Les variables logiques d'un CSP peuvent être remplacées par des termes selon une fonction de susbtitution.

Définition 7. *Substitution d'une variable dans un terme*

La fonction $\cdot[\cdot/\cdot] : \mathcal{T} \times \mathbb{V} \times \mathcal{T} \longrightarrow \mathcal{T}$ représente la substitution d'une variable par un terme dans un autre terme. Elle n'est définie que si la variable remplacée n'apparaît pas dans le terme de substitution (le troisième argument) Soient $V \in \mathbb{V}$ et $T \in \mathcal{T}$ tels que $V \notin vars(T)$. La substitution de V par T est définie ainsi pour tout terme du langage $\mathcal{T}$:

$$\begin{array}{rcll} V[V/T] & = & T & \\ W[V/T] & = & W & \forall\ W \in \mathbb{V} \setminus \{V\} \\ x[V/T] & = & x & \forall\ x \in \mathbb{Z} \cup \mathbb{A} \\ f(T_1,\ldots,T_n)[V/T] & = & f(T_1[V/T],\ldots,T_n[V/T]) & \forall\ n \in \mathbb{N}^*, \forall\ f \in \mathbb{A}, \\ & & & \forall\ (T_1,\ldots,T_n) \in \mathcal{T}^n \end{array}$$

La substitution est étendue aux ensembles de termes selon la définition suivante :

Définition 8. *Substitution d'une variable dans un ensemble de termes*

Soient $S \in \mathscr{P}(\mathcal{T})$, $V \in \mathbb{V}$ et $T \in \mathcal{T}$ tels que $V \notin vars(T)$. La substitution de V par T dans S est définie ainsi :

$$S[V/T] = \{X[V/T] \mid X \in S\}$$

1.2.4.5 Exemple

Nous reprenons le CSP précédent, C_1, et y appliquons la substitution de la variable A par la valeur 7 :

$$\begin{array}{rcl} C_1[A/7] & = & \{geq(A,B), A \in 1..10, B \in 5..12\}[A/7] \\ & = & \{geq(A,B)[A/7], (A \in 1..10)[A/7], (B \in 5..12)[A/7]\} \\ & = & \{geq(A[A/7], B[A/7]), A[A/7] \in 1..10[A/7], B[A/7] \in 5..12[A/7]\} \\ & = & \{geq(7,B), 7 \in 1..10, B \in 5..12\} \end{array}$$

1.3 Règles de propagation

On peut remarquer, dans le système de contrainte C_1, que les domaines associés aux deux variables A et B peuvent être réduits : on a $A \in 1..10$ et $B \in 5..12$, mais ceux-ci pourraient tous deux être réduits à 5..10. En effet, si $A < 5$ ou si $B > 10$, il n'existe pas de

couple de valeurs, prises respectivement dans les domaines de A et de B, qui satisfait la contrainte $A \geq B$. Nous décrivons dans cette section comment réaliser un tel filtrage dans un CSP.

1.3.1 Filtrage des contraintes

Nous définissons la propagation de contraintes comme une réécriture d'un CSP en un autre, conditionnée par un ensemble de règles. Le but est de réduire les domaines de valeurs associés aux variables du problème en fonction de contraintes présentes.

Nous donnons un système de règles notées sous forme de règles d'inférences. Parmi ces règles, certaines sont appelées des règles de filtrage et servent à établir un jugement de la forme « $\vdash S \rightarrow S'$ ». Ce jugement signifie que l'ensemble de contraintes S peut se réécrire en l'ensemble S'. L'ensemble des règles permet d'établir une chaîne de réécritures successives :

$$\begin{aligned} \vdash \quad & S_0 \rightarrow S_1 \\ \vdash \quad & S_1 \rightarrow S_2 \\ & \vdots \\ \vdash \quad & S_{n-1} \rightarrow S_n \end{aligned}$$

Cette succession s'écrit aussi « $\vdash S_0 \xrightarrow{*} S_n$ », où la flèche étoilée correspond à zéro ou plusieurs applications successives de la flèche simple, définie par les règles suivantes :

$$\frac{}{\vdash A \xrightarrow{*} A} \qquad \frac{\vdash A \rightarrow B}{\vdash A \xrightarrow{*} B} \qquad \text{PROP-TRANS}\ \frac{\vdash A \xrightarrow{*} B \qquad \vdash B \xrightarrow{*} C}{\vdash A \xrightarrow{*} C}$$

1.3.2 Système de règles

Par la suite, nous donnerons des règles associées au filtrage des différentes contraintes qui nous intéressent. Par exemple, la règle qui suit est une des règles pour le traitement de la relation « $\equiv$ » que l'on reverra par la suite :

$$\equiv\text{-TRIVIAL}\ \frac{}{\vdash S \cup \{\alpha \equiv \alpha\} \rightarrow S}$$

Le CSP à gauche de la flèche symbolisant la réecriture est déstructuré en un ensemble S et un singleton contenant le terme $\alpha \equiv \alpha$. La règle retire ce terme, car après réecriture, il ne reste plus que l'ensemble S.

La règle est applicable quand le CSP a la forme décrite par la règle, c'est-à-dire quand un terme $\alpha \equiv \alpha$ y existe, avec α quelconque, mais aussi lorsque les conditions énoncée dans les prémisses de la règle sont vérifiées (ici il n'y en a pas).

Nous ne cherchons pas ici à le démontrer formellement, mais l'application répétée des règles termine bien sur un point fixe. Intuitivement, chacune d'elle réduit soit le nombre

de contraintes présentes soit le nombre (fini) de solutions possibles pour le système (par réduction de domaines, ou instanciation). Par exemple, nous évitons d'avoir des règles qui sont applicables à la fois dans le CSP précédent la réecriture et dans celui obtenu après.

Par ailleurs, nous faisons en sorte qu'il n'y ait qu'une seule règle applicable pour une contrainte, de sorte à ce que le filtrage d'une contrainte soit déterministe. Cependant, il est possible que plusieurs contraintes différentes soient filtrables simultanément. En toute généralité, cela pourrait donner lieu à des points fixes différents selon la stratégie d'application des règles. Nous ne donnons pas la fonction de choix qui guide la propagation de contraintes, mais en pratique, elle existe et rend le filtrage déterministe.

1.3.3 CSP dynamiques et nouvelles variables

Les CSPs considérés sont dynamiques, au sens où ils peuvent introduire de nouvelles contraintes au cours du filtrage. Les nouvelles contraintes doivent alors souvent introduire de nouvelles variables de l'ensemble $\mathbb{V}$. Il est essentiel que les variables introduites soient libres et ne rentrent pas en conflit avec des variables existantes.

La règle FRESH suivante garantit que les variables $V_1, \ldots, V_n$ ($n > 0$) choisies n'existent pas dans l'ensemble de contraintes et sont différentes les unes des autres :

$$\text{FRESH}\ \frac{\forall i \in 1..n : V_i \in \mathbb{V} \setminus \mathit{vars}(S) \qquad \forall i, j \in 1..n : (V_i = V_j) \Leftrightarrow (i = j) \qquad n \in \mathbb{N}^*}{S \vdash V_1, \ldots, V_n : \mathit{fresh}}$$

À l'aide du prédicat *fresh*, nous introduisons une ou plusieurs variables qui n'existent pas déjà dans le CSP courant et qui sont toutes différentes entre elles.

La règle est locale à un ensemble de contraintes, ce qui pourrait poser des problèmes de conflits de noms de variables si on venait à mélanger les éléments de deux ensembles différents. Cependant, on évite ce cas de figure dans les règles.

1.3.4 Exemple de l'unification

Parmi les symboles atomiques donnés dans la syntaxe, l'opérateur « ≡ » nous sert à représenter l'unification **??** de deux termes. Deux termes E et F sont unifiables s'il existe un ensemble de substitutions concernant les variables de E et F tel que l'application de ces substitutions respectivement à E et à F conduise à deux termes identiques. Ainsi, les termes $f(A, [a, b, c])$ et $f(z, B)$ sont unifiables, car si on substitue simultanément A par z et B par $[a, b, c]$ dans les deux termes, ceux-ci deviennent identiques. De même, les termes $g(X)$ et $g(Y)$ sont unifiables : dans ce cas, on peut substituer Y par X dans dans le second terme. L'unification de deux termes est réalisée selon l'approche récursive suivante :

- Deux termes atomiques s'unifient si et seulement s'ils sont identiques.

- Une variable non instanciée peut-être unifiée avec un atome, un terme composé ou encore une autre variable. Dans ce dernier cas, les deux variables sont en alias et l'une d'elle peut-être substituée par l'autre à chacune de ses occurrences sans modifier la sémantique du CSP.
- Deux termes composés sont unifiables si et seulement s'ils portent le même foncteur, ont la même arité et que leurs sous-termes respectifs sont deux à deux unifiables pour le même ensemble de susbsitutions.

Nous définissons une contrainte d'unification, notée « ≡ », qui est compatible avec cette définition et en particulier produit la contrainte *fail* si deux termes ne sont pas unifiables. De même, on note « $E \not\equiv F$ » la contrainte qui impose E et F ne soient pas unifiables : elle s'assure notamment que les domaines respectifs de variables non -unifiables sont bien disjoints.

1.3.4.1 Règles de filtrage

Nous définissons l'unification de deux termes dans un ensemble de contraintes à l'aide de règles d'inférences. En reprenant l'exemple précédént, si la contrainte « $f(A, [a, b, c]) \equiv f(z, B)$ » existe dans un CSP, alors on souhaite obtenir après propagation deux nouvelles contraintes, « $A \equiv z$ » et « $B \equiv [a, b, c]$ ». Ces contraintes seront elles-mêmes filtrées à leur tour, de manière à instancier les variables logiques dans le système de contraintes tout entier. En effet, lorsqu'une variable est unifiée avec un terme, on peut la substituer dans chacune de ses occurrences dans le CSP, et pas seulement dans les termes unifiés. Les règles d'unifications font appel à la substitution de variables, telle qu'elle a été définie précédemment. Ainsi, la règle suivante élimine le cas simple où les termes sont strictement identiques :

$$\equiv\text{-TRIVIAL}\ \frac{}{\vdash S \cup \{\alpha \equiv \alpha\} \rightarrow S}$$

Lorsqu'un des termes est une variable V, celle-ci peut-être remplacée par l'autre terme T dans le CSP. Pour cela, il est nécessaire que V ne soit pas une variable de T ; en effet, par définition de la substitution (p. 12), le remplacement de V par T donne le terme T, et non $T[V/T]$: on ne cherche pas à substituer les occurences de V dans T, car la substitution ne finirait pas. On évite d'avoir la variable V dans l'ensemble $vars(T)$ pour éviter qu'il n'y ait plus aucune occurence de V dans l'ensemble des contraintes après le filtrage de l'opérateur d'unification. Les règles suivantes prennent en compte les deux cas possibles, à savoir quand V est le terme à gauche ou celui à droite du symbole l'unification.

$$\equiv\text{-VAR-LEFT}\ \frac{V \in \mathbb{V} \qquad V \notin vars(T)}{\vdash S \cup \{V \equiv T\} \rightarrow S[V/T]} \qquad \equiv\text{-VAR-RIGHT}\ \frac{V \in \mathbb{V} \qquad V \notin vars(T)}{\vdash S \cup \{T \equiv V\} \rightarrow S[V/T]}$$

Autrement dit, le terme original « $V \equiv T$ » est retiré du CSP, et dans l'ensemble de contraintes restant, V est remplacée par le terme T. Pour les termes composés, l'unification se propage aux sous-termes.

$$\equiv\text{-TERM}\ \frac{a \in \mathbb{A} \qquad n \in \mathbb{N}^* \qquad (E_1, \ldots, E_n, F_1, \ldots, F_n) \in \mathcal{T}^{2n}}{\vdash S \cup \{a(E_1, \ldots, E_n) \equiv a(F_1, \ldots, F_n)\} \rightarrow S \cup \{E_1 \equiv F_1, \ldots, E_n \equiv F_n\}}$$

Les règles suivantes permettent de détecter l'insatisfaisabilité du CSP lorsque l'unification est impossible. Pour cela, le CSP est réécrit en un singleton $\{fail\}$ qui représente un problème trivialement insatisfaisable.

Nous définissons tout d'abord un prédicat auxiliaire, à travers la règle ATOMIC ; le jugement $\vdash \alpha_1, \ldots, \alpha_n : atomic$ est vérifié si et seulement si tous les termes α_i sont atomiques, c'est-à-dire soit des atomes soit des variables numériques.

$$\text{ATOMIC}\ \frac{n \in \mathbb{N}^* \qquad (\alpha_1, \ldots, \alpha_n) \in (\mathbb{A} \cup \mathbb{Z})^n}{\vdash \alpha_1, \ldots, \alpha_n : atomic}$$

Alors, nous définissons deux règles basées sur ce prédicat. La première fait échouer la propagation lors de l'unification de deux littéraux différents. La seconde prend en compte le cas de termes complexes n'ayant pas le même atome de tête ou ayant des arités différentes.

$$\equiv_{fail}\text{-ATOMIC}\ \frac{\vdash a, b : atomic \qquad a \neq b}{\vdash S \cup \{a \equiv b\} \rightarrow \{fail\}}$$

$$\equiv_{fail}\text{-COMPOUND}\ \frac{\begin{array}{c}(a, b) \in \mathbb{A}^2 \qquad (n, m) \in \mathbb{N}^* \times \mathbb{N}^* \\ (n \neq m) \vee (a \neq b) \qquad (E_1, \ldots, E_n, F_1, \ldots, F_m) \in \mathcal{T}^{n+m}\end{array}}{\vdash S \cup \{a(E_1, \ldots, E_n) \equiv b(F_1, \ldots, F_m)\} \rightarrow \{fail\}}$$

Nous avons alors ici un exemple de l'utilisation d'un prédicat auxiliaire dans l'application d'une règle.

1.3.4.2 Exemple

Soit $C_2 = \{f(A, [a, b, c]) \equiv f(z, B), A \equiv x\}$. Les différentes étapes de propagation sont données par les étapes successives suivantes.

$$\begin{array}{ll}
C_2 = \{f(A, [a, b, c]) \equiv f(z, B), A \equiv x\} & \\
\vdash C_2 \xrightarrow{*} \{A \equiv z, [a, b, c] \equiv B, A \equiv x\} & \equiv\text{-TERM} \\
\vdash C_2 \xrightarrow{*} \{[a, b, c] \equiv B, z \equiv x\} & \equiv\text{-VAR-LEFT} \\
\vdash C_2 \xrightarrow{*} \{fail\} & \equiv_{fail}\text{-ATOMIC}
\end{array}$$

Ainsi, C_2 n'admet pas de solution.

1.4 Familles de domaines

La contrainte d'appartenance « $V \in D$ » lie une variable V à un domaine D, qui est un terme atomique ou composé. L'ensemble des termes de même foncteur et de même arité que D représente une famille de termes que l'on appelle aussi domaine, par abus de langage. Dans notre exemple précédent, le domaine des intervalles d'entiers regroupe l'ensemble des termes de la forme « $a..b$ », avec a et b les bornes minimales et maximales entières. Chaque domaine est muni d'un ensemble de règles de propagation. Nous montrons quelques-unes de ces règles afin d'illustrer les particularités des contraintes de domaines.

1.4.1 Rôle des domaines

Plusieurs domaines différents peuvent donner des informations complémentaires sur un même ensemble de valeurs. Par exemple des variables représentant des valeurs entières peuvent être liées à la fois à des intervalles et à des relations de congruences : si une variable X est dans l'intervalle 0..10 et qu'on s'aperçoit qu'elle est impaire, alors cet intervalle de valeurs peut-être restreint à 1..9. Ainsi, les différentes représentations d'un ensemble de valeurs se recoupent pour avoir une vision plus précise des valeurs acceptables par une variable.

1.4.2 Opérations sur les domaines

Un domaine de valeurs, comme un intervalle de la forme $a..b$ ou une liste finie de valeurs telle que [*false, true*], est une représentation dans le langage de contraintes d'un ensemble de valeurs admissibles pour une variable. L'intervalle $a..b$, qui est un moyen plus court d'écrire *domint(a,b)*, représente l'ensemble des entiers relatifs compris entre a et b inclus, alors que la liste [*false, true*] représente directement l'ensemble {*false, true*}. Pour tout couple d'entiers (a, b) on définit $\mathscr{D}_{a..b}$ comme suit :

$$\begin{aligned} \mathscr{D}_{\emptyset} &= \emptyset \\ \mathscr{D}_{a..b} &= \{x \in \mathbb{Z} \mid a \leq x \leq b\} \end{aligned}$$

De même, on peut définir récursivement l'ensemble associé à une liste d'atomes ; soient L une liste d'atomes et $H \in \mathbb{A}$ un atome ; on a :

$$\begin{aligned} \mathscr{D}_{[]} &= \emptyset \\ \mathscr{D}_{[H|L]} &= \{H\} \cup \mathscr{D}_L \end{aligned}$$

On peut ainsi retrouver les ensembles symbolisés par ces termes spéciaux du langages $\mathcal{T}$ que sont les domaines. Par ailleurs, nous surchargeons les opérations ensemblistes usuelles pour les appliquer sur les domaines directement. Ainsi, nous notons $2..6 \cap 0..4$

l'intersection des domaines 2..6 et 0..4, qui est 2..4. Autrement dit, quelques soient a, b, c et d quatre entiers relatifs, on définit cette intersection comme suit :

$$a..b \cap c..d = \begin{cases} \emptyset & \text{si } \mathscr{D}_{a..b} \cap \mathscr{D}_{b..c} = \emptyset \\ min(D)..max(D) & \text{sinon, avec } D = \mathscr{D}_{a..b} \cap \mathscr{D}_{b..c} \end{cases}$$

Nous définissons les autres opérations de manière similaire, c'est-à-dire en les appliquant sur les ensembles que représentent les domaines, puit en convertissant le résultat vers un terme approprié dans le langage de contraintes. Par ailleurs, la relation $x \in 5..10$ est vraie si et seulement si la relation $x \in \mathscr{D}_{5..10}$ l'est aussi. Les opérations qui nous intéressent sont en particulier les unions et les intersections de domaines, mais aussi l'inclusion et le test d'égalité. En particulier, le test d'égalité entre domaines prend le dessus sur le test structurel entre termes du langage $\mathcal{T}$: les deux listes [*false, true*] et [*true, false*] sont structurellement différentes (l'ordre est inversé), mais la relation d'égalité [*false, true*] = [*true, false*] est vérifiée, car elle est définie relativement aux ensembles de valeurs représentés.

En résumé, jusqu'à présent, nous avons introduit différentes notations sur les ensembles, comme « $V \in D$ » qui est la représentation infixe de la contrainte « $\in (V, D)$ » dans un CSP, où « $\in$ » fait partie des quelques atomes du langage qui sont des symboles ; il s'agit ici d'une contrainte du CSP. Nous utilisons par ailleurs le symbole « $\in$ » tel qu'il est défini de manière usuelle pour les ensembles. Finalement, dans cette section, nous surchargeons cet opérateur, ainsi que les opérateurs $\cup$, $\cap$, $\notin$,…pour réaliser ces mêmes opérations sur les domaines portés par les variables, et obtenir en retour des nouveaux domaines, sous la forme de termes du langage de contraintes.

Ces différentes notations se superposent, mais le contexte de leur utilisation doit normalement permettre de les distinguer facilement. Par exemple, la règle suivante définit une règle de filtrage qui permet d'éliminer la contrainte $n \in D$ d'un ensemble lorsque le membre gauche, n, est bien un élément du domaine D :

$$\frac{n \in D}{\vdash S \cup \{n \in D\} \rightarrow S}$$

Dans l'expression « $S \cup \{n \in D\}$ », l'union est celle des ensembles de termes, et porte sur le CSP, alors que le terme « $n \in D$ » est une contrainte. En revanche, la relation $n \in D$ dans les prémisses de la règle permet de tester si la contrainte est vérifiée ; elle fait appel pour cela à la relation « $\in$ » sur les domaines que l'on a définie ici.

1.4.3 Règles d'introduction et de propagation

Les domaines de valeurs sont introduits dans un CSP comme les autres contraintes. L'ensemble de contraintes évolue par propagations successives, et il est possible d'atteindre une situation où deux variables sont unifiées, ce qui peut amener à avoir

plusieurs domaines pour une même variable. Par exemple, si on ajoute la contrainte $V \equiv W$ à l'ensemble de contraintes $\{V \in 0..10, W \in 5..20\}$, on peut arriver à l'ensemble suivant après propagation de l'opérateur d'unification :

$$\{V \in 0..10, V \in 5..20\}$$

C'est pour cela qu'il existe des règles mettant en commun les informations provenant de plusieurs variables. Ici, on définit la règle suivante :

$$\frac{D = D_1 \cap D_2}{\vdash S \cup \{V \in D_1, V \in D_2\} \rightarrow S \cup \{V \in D\}}$$

Dans cette règle, le domaine est réduit selon l'opérateur « $\cap$ » qui représente l'intersection des domaines respectifs D_1 et D_2. Le domaine résultant pour notre exemple précédent est ainsi « 5..10 ». Par la suite, si on instancie V à 8 en ajoutant la contrainte $V \equiv 8$ à l'ensemble de contraintes précédent, on obtient après substitution « $8 \in 5..10$ ». La règle suivante vérifie que l'instantiation est correcte et retire la contrainte d'appartenance ; il s'agit de la règle que l'on a vue dans l'exemple de la section précédente :

$$\frac{n \in D}{\vdash S \cup \{n \in D\} \rightarrow S}$$

L'opération « $\in$ » signifie que le terme atomique n appartient au domaine représenté par le domaine D. Nous pouvons aussi détecter l'insatisfaisabilité de la contrainte liant une variable à un domaine. Dans ce cas, la propagation échoue, ce qui se traduit par la réécriture vers le singleton $\{fail\}$:

$$\frac{n \notin D}{\vdash S \cup \{n \in D\} \rightarrow \{fail\}}$$

Ainsi, les règles que l'on vient de voir permettent de déclencher de nouvelles propagations de contraintes dans le système lors de l'unification ou de l'instantiation d'une variable. Nous avons présenté quelques-unes des règles essentielles liées à la réduction de domaines. En pratique, n'importe quelle contrainte peut décider de réduire un domaine selon sa signification, et les domaines eux-mêmes peuvent intéragir entre eux pour exploiter les différentes informations qu'ils représentent.

1.5 Recherche de solution

La résolution d'un CSP utilise la propagation de contraintes pour rechercher des solutions. Elle réalise des hypothèses sur les valeurs des variables d'un problème et lance la propagation pour trouver une solution ou détecter une insatisfaisabilité. Les différents essais/erreurs guident la recherche de solutions jusqu'à finalement trouver une valuation pour chaque variable du problème, ou conclure sur l'insatisfaisabilité du problème initial. Nous montrons l'algorithme principal de résolution.

1.5.1 Contraintes et introduction d'hypothèses

Dans un CSP, chaque contrainte représente une relation ou un fait qui est admis comme vrai dans cet ensemble de contraintes :

$$\text{CSTR-TRUE}\ \frac{}{S \cup \{C\} \vdash C}$$

Il arrive que l'on souhaite réaliser une propagation de contraintes dans le contexte d'une ou plusieurs hypothèses. Dans ce cas, on note à gauche du signe « $\vdash$ » un ou plusieurs termes $H_1, \ldots, H_n$ qui représentent autant de contraintes supplémentaires à prendre en compte dans la propagation :

$$\text{HYP-INTRO}\ \frac{\vdash A \cup \{H_1, \ldots, H_n\} \xrightarrow{*} B}{H_1, \ldots, H_n \vdash A \xrightarrow{*} B}$$

1.5.2 CSP sous forme résolue

Lorsque l'on veut résoudre un CSP, il nous faut un moyen d'arrêter la recherche si on sait que le problème n'est pas satisfaisable, ou si on a atteint une solution. On a vu précédemment que certains filtrages peuvent introduire la contrainte atomique *fail*, qui rend alors le problème insatisfaisable. Nous cherchons ici à savoir si un ensemble est une solution du problème initial.

Généralement, un ensemble de contraintes S est sous une forme résolue si toutes les variables sont instanciées ($vars(S) = \emptyset$). On peut aussi s'arrêter dès que les variables ne sont plus liées entre-elles par des contraintes : elles sont alors indépendantes et on peut choisir n'importe quelle valeur dans les domaines associés. Nous verrons cependant par la suite que l'on a besoin d'ajouter d'autres conditions pour les problèmes particuliers qu'on souhaite résoudre.

C'est pourquoi nous utilisons un prédicat intermédiare noté $\vdash S : solution$ pour représenter qu'un CSP S est sous forme résolue. Nous le définissons pour le moment comme suit, en sachant qu'il sera plus détaillé pour la classe particulière de CSP qui nous intéresse :

$$\text{SOLUTION}\ \frac{fail \notin S \qquad \forall C \in S, |vars(C)| \leq 1}{\vdash S : solution}$$

On arrete ainsi la recherche de solution quand les variables du problèmes sont indépendantes les unes des autres et ne contiennent pas le terme *fail*. C'est le cas dans la propagation ci-dessous, où l'ensemble résultant peut être considéré comme une solution sans pour autant instancier explicitement les variables A et B :

$$\begin{array}{ll} \vdash & \{gt(A,5), lt(B,4), A \in 0..10, B \in 0..10\} \\ \xrightarrow{*} & \{A \in 6..10, B \in 0..3\} \end{array}$$

On remarque que si S n'est pas sous une forme résolue, cela signifie seulement qu'aucune solution ne peut être extraite des contraintes actuelles, mais pas forcément qu'il n'existe pas de solution atteignable depuis S (il peut être nécessaire de filtrer davantage les contraintes). Avec cette notation, on peut définir la méthode générale de résolution d'un CSP.

1.5.3 Résolution d'un CSP

Intuitivement, la résolution consiste à choisir une variable V dans l'ensemble de contraintes S, ainsi qu'une valeur v dans le domaine de V, à instancier V par v et à résoudre le sous-problème ainsi créé [1]. Si le sous-problème échoue systématiquement, on peut retirer à coup sûr la valeur v du domaine de V. Si, au contraire, il existe une solution atteignable par la résolution du sous-problème, alors on finit la résolution en gardant cette solution. Nous utilisons à nouveau des ensembles de règles pour décrire l'étape de résolution et introduisons pour cela une autre notation.

> **Définition 9**. *Résolution d'un CSP*
>
> Soient S et Sol des ensembles de contraintes. On note « $\vdash S \rightsquigarrow Sol$ » le fait que :
>
> 1. Sol est sous forme résolue ; et
> 2. Sol est atteignable par résolution de contraintes depuis S.

La notation « $\vdash S \rightsquigarrow Sol$ » fait le lien entre deux CSP qui ne sont pas nécessairement atteignables par propagation de contraintes, mais à travers des hypothèses supplémentaires ajoutées au cours de la résolution. La figure 1.3 illustre la démarche utilisée à travers un exemple fictif : l'ensemble S_0 peut se réecrire en un CSP appelé S_1 ; avec une hypothèse supplémentaire H_1, S_1 se réecrit en $S_1 1$ (les hypothèses sont ajoutées par la procédure de recherche) ; en ajoutant une autre hypothèse H_2 à ce CSP, on obtient Sol qui est sous forme résolue. Alors, comme Sol est une solution obtenue par l'ajout d'une hypothèse H_2 à S_{11}, il s'agit d'une solution atteignable par S_{11} (étape 1 à 2). Or, S_{11} est lui-même construit en ajoutant l'hypothèse H_1 à S_1, et donc Sol est une solution pour S_1 (étape 2 à 3). Finalement, comme S_1 et obtenu par propagation de contraintes depuis S_0, Sol est aussi une solution pour S_0.

Les hypothèses introduites correspondent toujours à des sous-problèmes du problème initial, ce qui nous permet de faire le lien entre les CSP et dire qu'une solution au sous-problème est bien une solution au problème original. Le choix des hypothèses repose en pratique sur des heuristiques guidant une démarche par essais/erreurs.

1. En toute généralité, il n'est pas nécessaire de choisir une valeur v en particulier ; il suffit de choisir un sous-domaine du domaine courant de V.

1.	$\vdash S_0 \xrightarrow{*} S_1$	$\vdash S_1 \cup H_1 \xrightarrow{*} S_{11}$	$\vdash S_{11} \cup H_2 \xrightarrow{*} Sol$	$\vdash Sol : solution$
2.	$\vdash S_0 \xrightarrow{*} S_1$	$\vdash S_1 \cup H_1 \xrightarrow{*} S_{11}$	$\vdash S_{11} \rightsquigarrow Sol$	
3.	$\vdash S_0 \xrightarrow{*} S_1$	$\vdash S_1 \rightsquigarrow Sol$		
4.	$\vdash S_0 \rightsquigarrow Sol$			

FIGURE 1.3: Exemple de résolution de contraintes. Chaque ligne représente les jugements connus et notamment la progression de la relation $\rightsquigarrow$.

1.5.3.1 Règles de résolution

La première règle fait le lien avec la propagation, dans le cas où une solution est directement atteignable par filtrage. La deuxième indique l'atteignabilité d'une solution *Sol* depuis un ensemble A, sachant que A donne B par propagation et que *Sol* est atteignable depuis B.

$$\text{RESOL-PROP}\ \frac{A \xrightarrow{*} B \qquad \vdash B : solution}{A \rightsquigarrow B} \qquad\qquad \text{RESOL-TRANS}\ \frac{A \xrightarrow{*} B \qquad B \rightsquigarrow C}{A \rightsquigarrow C}$$

Notre objectif lors d'une résolution est de montrer qu'il existe une solution *Sol* atteignable par l'ensemble de contraintes initial. Le cas traité ci-dessous par RESOL-TRUE lie un CSP à une solution trouvée *Sol* à l'aide d'une hypothèse sur la valuation d'une variable. L'unification qui est faite en hypothèse correspond bien à un cas représenté par S. La solution *Sol* est donc bien à une des solutions possibles de S.

$$\text{RESOL-TRUE}\ \frac{S \vdash heuristics(V, v) \qquad \vdash S \cup \{V \equiv v\} \rightsquigarrow Sol}{\vdash S \rightsquigarrow Sol}$$

Le prédicat $S \vdash heuristics(V, v)$ n'est pas défini ici car il est lui aussi dépendant du type de problème à résoudre : il met en œuvre les heuristiques de sélection d'une variable et de sa valuation, signifiant ainsi que V a été choisie parmi les variables de S pour être instanciée à v. D'après la règle, si S auquel on ajoute la contrainte $V \equiv v$ conduit à un CSP solution *Sol*, alors S permet d'atteindre la solution *Sol*. L'hypothèse peut ne pas aboutir à une solution, et on peut souhaiter étudier le cas où V est différent de v.

$$\text{RESOL-FALSE}\ \frac{S \vdash heuristics(V, v) \qquad \vdash S \cup \{V \not\equiv v\} \rightsquigarrow Sol}{\vdash S \rightsquigarrow Sol}$$

Ainsi, les deux règles principales de la résolution que l'on vient de voir font intervenir des résolutions de CSP intermédiaires, qui elles-même peuvent demander de résoudre d'autres sous-problèmes. On passe potentiellement en revue toutes les valuations de toutes les variables d'un problème avant de trouver une solution, ce qui peut ne pas

terminer si l'espace de recherche est infini. L'efficacité de la procédure de résolution dépend fortement de la capacité des règles de filtrage à trouver au plus tôt des insatisfaisabilités dans le CSP, d'une part, et des heuristiques d'instantiation d'autre part.

La vision de la résolution exposée ici ne prend pas en compte les techniques implémentées en pratique, comme par exemple la gestion de la profondeur de la recherche (le nombre d'hypothèses déjà faites) ou le décompte du nombre d'échecs à cette profondeur.

1.5.3.2 Remarque

Selon la nature du problème, on peut ou non vouloir extraire certaines informations de l'ensemble *Sol*, comme par exemple les valuations des variables. Par exemple, dans le cas du puzzle arithmétique SEND+MORE=MONEY, toutes les contraintes mènent à une instantiation des variables ; comme généralement ces contraintes disparaissent du système une fois qu'elles portent sur des arguments atomiques valides, l'ensemble résolu *Sol* est finalement l'ensemble vide, qui indique seulement qu'il existe une solution, sans donner les valeurs des variables. Cependant, si nous ajoutons le terme *sendmory(S,E,N,D,M,O,R,Y)* à l'ensemble initial, la résolution instancie les variables pour finalement donner le système $Sol = \{sendmory(9,5,6,7,1,0,8,2)\}$, qui représente l'information souhaitée. Le terme auxiliaire est une contrainte sans règle de filtrage qui reste toujours dans l'ensemble et sert à collecter les valeurs récupérées.

1.5.4 Exemple

Nous reprenons les étapes de résolution de l'exemple « SEND+MORE=MONEY ». Pour cela, nous suivons un autre chemin de résolution que celui présenté dans la section 1.1.6 afin d'avoir un aperçu de l'application de chacune des règles propres à la résolution. Dans notre exemple, un CSP est une solution si et seulement s'il ne contient plus que le terme *sendmory* que l'on a décrit précédemment.

Nous montrons dans la figure 1.4 les différentes étapes permettant d'atteindre l'ensemble solution *Sol* à partir du système initial (l'ensemble S_{smm} est donné page 8) :

$$S = S_{smm} \cup \{sendmory(S, E, N, D, M, O, R, Y)\}$$

Ces étapes ont été réalisées dans le but de montrer l'application de chacune des règles que l'on a vu pour la résolution.

Nous détaillons les étapes dans les paragraphes qui suivent. Jusqu'à la ligne 10, les règles mettent en œuvre la recherche proprement dite. Au-delà, elles mettent en évidence l'atteignabilité de cette solution à partir de l'ensemble initial. Tout d'abord, la propagation depuis S vers S_1 à la ligne 1 correspond au premier filtrage à partir de l'ensemble de contraintes initial. Le système de contraintes résultant correspond ainsi à l'équation

1	$\vdash S \xrightarrow{*} S_1$	(propagation initiale)
2	$S_1 \vdash \mathit{heuristics}(D, 7)$	(heuristique de résolution)
3	$S_2 = S_1 \cup \{D \equiv 7\}$	
4	$\vdash S_2 \xrightarrow{*} S_3$	(point fixe par propagation)
5	$S_3 \vdash \mathit{heuristics}(C_2, 0)$	(deuxième hypothèse)
6	$S_4 = S_3 \cup \{C_2 \equiv 0\}$	
7	$\vdash S_4 \xrightarrow{*} \{\mathit{fail}\}$	(insatisfaisabilité)
8	$S_5 = S_3 \cup \{C_2 \not\equiv 0\}$	(tentative dans le cas opposé)
9	$\vdash S_5 \xrightarrow{*} \mathit{Sol}$	(point fixe depuis S_5)
10	$\vdash \mathit{Sol} : \mathit{solution}$	(Sol est une solution)
11	$\vdash S_5 \rightsquigarrow \mathit{Sol}$	(RESOL-PROP avec 9 et 10)
12	$\vdash S_3 \rightsquigarrow \mathit{Sol}$	(RESOL-FALSE avec 11, 8 et 5)
13	$\vdash S_2 \rightsquigarrow \mathit{Sol}$	(RESOL-TRANS avec 4 et 12)
14	$\vdash S_1 \rightsquigarrow \mathit{Sol}$	(RESOL-TRUE avec 13, 3 et 2)
15	$\vdash S \rightsquigarrow \mathit{Sol}$	(RESOL-TRANS avec 1 et 14)

FIGURE 1.4: Étapes de résolution pour le problème « SEND+MORE=MONEY »

suivante, où les domaines attachés aux variables sont précisés :

	1	0	$C_2 \in \{0,1\}$	$C_1 \in \{0,1\}$	0
	0	9	$E \in \{2,\ldots,8\}$	$N \in \{2,\ldots,8\}$	$D \in \{2,\ldots,8\}$
+	0	1	0	$R \in \{2,\ldots,8\}$	E
	1	0	N	E	$Y \in \{2,\ldots,8\}$

Ce système contient des variables non instanciées et aucune règle de filtrage n'est applicable. On construit alors S_2 (lignes 2 et 3) un système identique à S_1 mis à part l'hypothèse supplémentaire que la variable D vaut 7. L'ensemble S_2 admet des propagations de contraintes, jusqu'à atteindre le point fixe S_3 (étape 4). Celui-ci représente l'équation suivante :

	1	0	$C_2 \in \{0,1\}$	1	0
	0	9	$E \in \{5,6,8\}$	$N \in \{5,6,8\}$	7
+	0	1	0	$R \in \{2,\ldots,6,8\}$	E
	1	0	N	E	$Y \in \{2,\ldots,5\}$

L'ensemble n'est pas une solution car il reste des variables. Cependant, plus aucun filtrage n'est applicable. On réalise alors une autre hypothèse en instantiant C_2 à zéro (lignes 5 et 6), définissant ainsi un système de contraintes S_4 dont la propagation de contraintes échoue (ligne 7). En effet, la contrainte *addcarry* initiale conduit à l'équation

$E + C_2 - N = 0$ dans S_3. L'ajout de la contrainte $C_2 \equiv 0$ à S_3 mène à une insatisfaisabilité (ligne 5) puisqu'elle entraîne « $E \equiv N$ » d'une part (*addcarry*), alors que l'on sait d'autre part que « $E \not\equiv N$ » (*alldifferent*). L'hypothèse étant responsable de l'échec de la résolution, on essaye le cas opposé en ajoutant cette fois la contrainte « $C_2 \not\equiv 0$ » ; cela définit l'ensemble S_5 (ligne 8). Or, l'ensemble S_5 admet des filtrages et ceux-ci mènent à un ensemble solution *Sol* (lignes 9 et 10).

Les étapes suivantes servent à un faire le lien entre l'ensemble initial et l'ensemble solution. Tout d'abord, comme S_5 peut se réécrire en *Sol*, on sait que l'on peut atteindre la solution depuis S_5 (ligne 11). Or, S_5 est obtenu en niant l'hypothèse « $C_2 \equiv 0$ » ; on peut donc atteindre la solution à partir de S_3 (ligne 12). Comme S_3 est atteint par propagation depuis S_2, celui-ci peut atteindre la solution (ligne 13). Enfin, on rappelle que S_2 est construit à partir de l'hypothèse « $D \equiv 7$ ». Comme il existe une solution atteignable depuis S_2, on en déduit à la ligne 14 qu'il existe une solution atteignable depuis S_1. Finalement, cette solution est aussi atteignable depuis S, étant donné que l'ensemble S_1 est issu de la propagation de contraintes depuis l'ensemble initial (ligne 15). On a donc bien trouvé un ensemble *Sol* tel que « $\vdash S \rightsquigarrow Sol$ ».

1.6 Conclusion

Nous avons présenté les différents mécanismes liés à la programmation logique par contraintes qui nous seront utiles par la suite. Pour cela, nous avons défini un langage de contraintes dans lequel celles-ci expriment à la fois les relations entre les variables, mais aussi entre les variables et leurs domaines. Nous pouvons ainsi manipuler un CSP sous la forme d'un ensemble de contraintes et appliquer de manière uniforme des règles de transformations locales à cet ensemble. Ces règles permettent de définir l'évolution dynamique des contraintes, notamment l'ajout de nouvelles variables et contraintes lors de la propagation, à un niveau plus ou moins détaillé. Enfin, nous avons présenté le schéma général de résolution de contraintes à l'aide d'heuristiques.

Chapitre 2

Le langage Lustre multi-horloges étendu

Nous nous intéressons aux systèmes réactifs modélisés selon l'approche synchrone Lustre et aux évolutions récentes du langage concerntant les horloges multiples. L'objectif du chapitre est de définir le langage $\mathcal{L}_+$, basé sur Lustre, qui étend celui-ci pour prendre en compte des extensions issues de travaux récents. Dans un premier temps, nous faisons un rappel sur les systèmes réactifs et leur modélisation à l'aide de Lustre. Nous décrivons ensuite le langage de manière informelle, en passant par l'approche flots de données et le cadencement des flots à l'aide d'horloges multiples. Nous introduisons les extensions que nous souhaitons traiter par la suite, et définissons le langage $\mathcal{L}_+$ ainsi que sa sémantique d'évaluation. Enfin, nous définissons à la fin du chapitre le sous-ensemble $\mathcal{L}_1$ qui symbolise le langage Lustre avec horloge unique qui nous servira de point de comparaison par la suite.

2.1 Présentation informelle

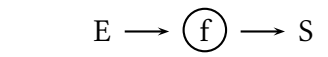
2.1.1 Approche synchrone Lustre

2.1.1.1 Systèmes réactifs

De nombreux programmes informatiques ont pour but d'entretenir en continu une interaction avec leur environnement. Il peut s'agir d'interfaces homme-machine [MP08], ou encore de systèmes embarqués [Hea03], qui réalisent des tâches de contrôle et de commande d'appareils électroniques (p. ex. téléphone portable, lave-linge, robotique industrielle). Lorsque de tels systèmes doivent réagir à une vitesse donnée, déterminée par l'environnement, ces systèmes sont modélisés avec des contraintes temps-réel. C'est le cas par exemple dans certains systèmes critiques, où le temps de réaction d'un programme est aussi important que sa correction. Les systèmes réactifs, critiques ou non, sont donc des systèmes réagissant continuellement à des stimulis extérieurs [Ber89, Ber00], qui garantissent un temps de réaction faces aux variations de l'environnement borné.

Les systèmes réactifs sont souvent comparés aux systèmes transformationnels [HP85], qui eux produisent un calcul (une transformation) entre un ensemble de valeurs en entrées et de valeurs en sorties. Un système transformationnel est schématisé de la manière

suivante, où E représente l'ensemble des valeurs d'entrées et S l'ensemble des valeurs en sortie, c'est-à-dire produites par la fonction f avec les valeurs de E :



Un système réactif consiste à exécuter de tels calculs en boucle, c'est-à-dire de manière continue dans le temps. La figure 2.1 suivante représente une boucle réactive entre un contrôleur et son environnement. Les flèches illustrent le passage d'un ou plusieurs signaux de l'un à l'autre, à travers des capteurs ou des actionneurs. Ici, il s'agit d'un contrôle en boucle fermée : le contrôleur peut recevoir des retours d'informations de la part de l'environnement suivant les actions qu'il effectue.

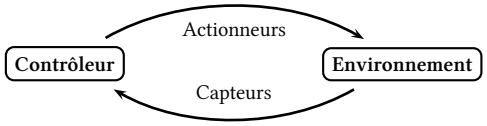


FIGURE 2.1: Boucle fermée réactive entre un contrôleur et son environnement

Par exemple, un moteur asservi (servomoteur) fonctionne en boucle fermée : il reçoit une consigne de position, et ajuste en permanance sa propre position. Pour cela, le système mesure à chaque instant l'écart entre la position désirée (la consigne) et la position courante. La différence mesurée donne lieu à une réaction. Le capteur ainsi que la consigne constituent les entrées du contrôleur, alors que la commande de déplacement en est une sortie.

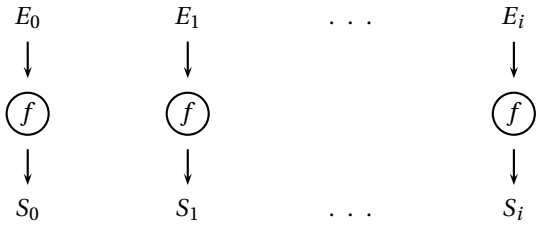
LUSTRE [Hal05, CPHP87] est un langage de spécification pour les systèmes réactifs issu de l'approche synchrone [BCE+03, PBdST05]. Il fournit un ensemble de primitives pour manipuler les flots de données entrant, sortant et circulant dans un système réactif au cours du temps. Le langage peut-être compilé (transformé) vers des instructions séquentielles de plus bas-niveau [HRR91, Ray91], tout en garantissant un temps d'exécution borné de chaque itération de la boucle réactive, ainsi qu'une utilisation bornée de la mémoire (ces deux points sont importants pour les systèmes embarqués). Le formalisme LUSTRE assure ainsi de bonnes propriétés aux programmes modélisés et s'inscrit dans une méthode de conception familière aux programmeurs de systèmes réactifs [Hal98, Ber00].

LUSTRE se place dans une famille de modèles qui segmente le temps en successions de cycles, comme c'est le cas généralement avec des contrôleurs numériques ; contrairement aux contrôleurs analogiques capables de traiter les signaux en continu, les contrôleurs numériques reposent sur une représentation discrète des grandeurs physiques [JS05]. Dans notre cas, un cycle de calcul correspond à un aller-retour entre l'environnement et le contrôleur. On numérote chaque cycle avec un entier, en commançant par zéro au cycle initial. Les temps de calculs réalisés sont supposés négligeables, car on suppose que le temps d'exécution d'un cycle est suffisamment rapide pour respecter les contraintes temporelles du système et ainsi observer toute variation significative d'une des entrées.

2.1.1.2 Cycles de calculs

Pour expliquer la démarche mise en œuvre autour de Lustre, nous reprenons la boucle réactive de la figure 2.1, du point de vue du contrôleur, et la déplions sur les différents cycles d'exécution. Les ensembles de valeurs E_i et S_i sont celles des entrées lues et des sorties produites par un système donné au cycle i. Un système réactif est alors vu comme l'application répétée dans le temps d'une fonction f associée à celui-ci. À tout cycle i, on a la relation suivante :

$$S_i = f(E_i)$$



Un tel modèle peut convenir pour un système réactif simple sans état interne. Cependant, il est rare que les entrées seules suffisent à définir complètement un contrôleur. Ceux-ci sont souvent complexes et peuvent changer d'état au cours du temps. Par exemple, on peut avoir besoin de calculer la dérivée d'une grandeur physique mesurée par un capteur, ce qui nécessite la mémorisation des observations passées de ces mesures. Il est aussi possible que le contrôleur doivent réaliser une succession de tâches (p. ex. : déplacer un bras robotisé, prendre une pièce, la déplacer, etc.), et dans ce cas, l'état du système mémorise le traitement actuel à réaliser et décide s'il doit passer ou non à l'état suivant au cours du temps.

2.1.1.3 État interne

La mémorisation d'un cycle à un autre est comprise dans la notion d'état, noté σ, variable dans le temps. En plus de réaliser un calcul des entrées vers les sorties, la fonction f peut maintenant aussi modifier l'état du contrôleur à chaque cycle. Soit i un entier représentant un cycle de calcul. On détermine l'état courant et les sorties à l'aide de l'état précédent et des entrées.

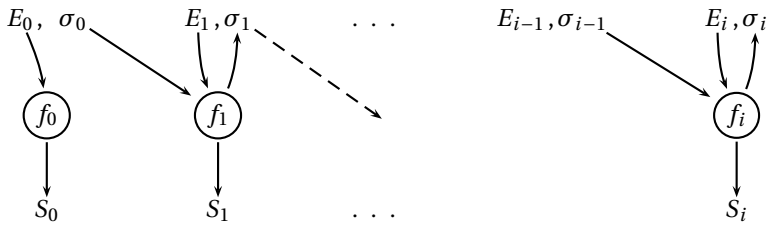
L'état initial σ_0 du système est défini à part, et l'égalité suivante est valable pour tout cycle $i > 0$:

$$S_i, \sigma_i = f(E_i, \sigma_{i-1})$$

La relation entre l'état courant, les sorties et les entrées est donnée à travers la fonction f qui prend cette fois deux arguments, les entrées et l'état précédent, et retourne deux résultats, à savoir les sorties courantes et le prochain état. Nous ajoutons à la trace d'exécution abstraite précédente l'état courant σ.

```
1. v = 0          - Initialiser v avec la valeur 0
2. LIRE e         - Lire la valeur courante de e
3. s = e + v      - Calculer la somme s
4. v = e          - La nouvelle valeur de v est e
5. ECRIRE s       - Envoyer la sortie vers l'environnement
6. ALLER A 2      - Repartir au point 2 pour le prochain cycle
```

FIGURE 2.2: Pseudo-code séquentiel équivalent au calcul du nœud somme.



2.1.1.4 Exemple

On considère un contrôleur simple qui adment une valeur entière « e » en entrée et une sortie « s ». À chaque cycle, la valeur de « s » est la somme des deux dernières valeurs en entrée, c'est-à-dire celles au cycle courant et celle au cycle précédent. On introduit une variable « v » qui contient la valeur précédente de « e ». Les relations entre les variables au cours du temps sont les suivantes :

- Au cycle zéro, v contient une valeur par défaut, zéro.
- Quelque soit le cycle i suivant, v_i prend la valeur e_{i-1}.
- Quelque soit le cycle i, $s_i = e_i + v_i$.

cycle	0	...	i-1	i	...
e	e_0		e_{i-1} →	e_i	
v	$v_0 = 0$			$v_i = e_{i-1}$	
s	$s_0 = e_0$			$s_i = e_i + v_i$	

Le langage LUSTRE permet de décrire un tel système en termes simples, puis de le compiler dans un langage séquentiel similaire au pseudo-code de la figure 2.2. Le système donné en exemple s'écrit de la manière suivante en LUSTRE [1] :

1. Ici, les équations sont écrites pour ressembler à la formulation originale, avec une variable intermédiaire. Il n'est cependant pas nécessaire d'introduire la variable v puisque le langage permet d'écrire l'équation plus simplement sous la forme `s = e → (pre(e) + e)`.

```
node somme(e: int) returns (s: int)
var v: int;
let
    v = 0 → pre(e);
    s = e + v ;
tel
```

La première ligne déclare un nouveau *nœud* Lustre appelé `somme` qui tient lieu de système réactif. Il admet une entrée `e` et une sortie `s`, toutes deux de type `int` (des valeurs entières). Tout variable locale, comme `v` ici, est déclarée dans le partie délimitée par les mot-clés `var` et `let`. Les mots-clés `let` et `tel` délimitent un bloc d'équations, où chaque variable calculée doit être associée à une expression de définition, à travers une affectation de la forme *Variable = Expression*.

La variable `v` est définie par l'expression « `0 → pre(e)` » : l'opérateur flèche est placé entre deux sous-expressions. À gauche de la flèche se trouve la définition de `v` au cycle initial. À droite de la flèche, on a l'expression associée à `v` à tous les autres cycles de calculs, à savoir la valeur précédente de l'entrée `e`. La sortie `s`, quant-à-elle, est définie à tous les cycles comme étant la valeur courante de `v` plus celle de `e`.

Le langage Lustre est ainsi muni d'opérateurs temporels, à savoir la flèche « → » et l'opérateur « `pre` ». Le cycle de calcul n'est pas précisé dans la spécification, car le temps est implicite.

Nous décrivons dans la section suivante de manière informelle le langage Lustre ainsi que les extensions du langage que nous voulons traiter.

2.1.2 Noyau Lustre

Un modèle Lustre est composé d'un nœud, c'est-à-dire un système d'équations où les flots de sortie sont exprimés en fonction de flots d'entrée par l'intermédiaire d'expressions arithmétiques, logiques ou temporelles Un système réactif est ainsi représenté par un unique nœud principal appelé « nœud racine ». Dans le langage Lustre original, il est possible de définir d'autres nœuds, auxiliaires, qui jouent le rôle de sous-systèmes. Nous ignorons cet aspect du langage dans ce document pour simplifier la présentation.

2.1.2.1 Flots de données

Le langage Lustre définit des opérateurs qui s'appliquent non pas sur des valeurs ponctuelles dans le temps, mais sur des flots de données. Un flot de données est séquence infinie de valeurs. Par exemple, la constante 3 de Lustre symbolise la séquence infinie de valeurs $(3, 3, 3, \ldots)$.

Les variables du langage servent à nommer des flots du systèmes : les identifiants de variables sont notés, dans ce document, en minuscules (p. ex. `x`, `y`, `position`, `vitesse`, …).

Pour toute variable LUSTRE dont l'identifiant est x, on note x_i la valeur de x au cycle $i \geq 0$:

$$\mathtt{x} = (x_0, x_1, x_2, \ldots)$$

Nous considérons uniquement deux types de base, à savoir `bool` (*true* ou *false*) et `int` (entiers relatifs). Nous ignorons le type `real` usuel car il introduit des notions complexes qui ne nous intéressent pas directement ici. Nous verrons par la suite qu'il est possible de déclarer des types supplémentaires à l'aide du langage (§2.1.3.1).

Le langage définit un ensemble classique d'opérateurs combinatoires sur les flots, comme l'addition, la multiplication. L'opérateur « * » correspond à la multiplication, cycle-à-cycle, des paires de valeurs portées par les deux opérandes :

$$\mathtt{x*y} = \big((x_0 * y_0), (x_1 * y_1), \ldots\big)$$

En plus des opérations arithmétiques usuelles (+, -, `div`, `mod`, *), LUSTRE définit les opérations booléennes (`and`, `or`, `not`), les opérateurs de comparaisons ($<, \leq, \geq, >$), d'égalité et d'inégalité ($=, \neq$) et l'opérateur conditionnel ternaire « `if..then..else` ». Le tableau de la figure 2.3 illustre le résultat de l'application d'opérateurs combinatoires sur des flots de valeurs quelconques [2].

cycle	0	1	2	3	4	5	6	7	8	9	...
x	1	1	2	3	-4	-7	13	-5	8	10	
y	0	1	1	2	3	5	5	1	0	16	
a = x + y	1	2	3	5	-1	-2	18	-4	8	26	
b = y > x	T	f	T	T	f	f	T	f	T	f	
c = not(b)	f	T	f	f	T	T	f	T	f	T	
d = if c then x else y	0	1	1	2	-4	-7	5	-5	0	10	

FIGURE 2.3: Exemples de calculs simples : (a) somme des valeurs courantes de x et y (b) la comparaison est effectuée à chaque cycle et donne un résultat booléen (c) négation du flot booléen b calculé précédemment (d) si le flot c vaut vrai, d prend la valeur courante de x ; sinon, celle de y.

2.1.2.2 Primitives temporelles

Les deux opérations temporelles principales du langage sont les suivantes :

– **Initialisation**. La valeur retournée par l'expression x → y dépend du cycle d'évaluation : elle coïncide avec la valeur de x au cycle initial, puis avec celle de y à tous les autres cycles. Le flot de valeurs est alors :

$$(x_0, y_1, y_2, \ldots)$$

2. Les valeurs booléenes *true* et *false* sont respectivement abrégées « T » et « f » dans les grilles de valeurs, pour plus de lisibilité.

– **Décalage unitaire**. Les valeurs sont décalées de sorte à être disponibles au cycle suivant. Le décalage n'est pas défini au cycle initial (il n'existe pas de cycle précédent), ce qui est symbolisé ci-dessous par le symbole « NIL ». Il s'agit d'une erreur de spécification si le système nécessite d'évaluer un opérateur `pre` au cycle initial. La séquence de valeurs pour l'expression `pre(y)` est alors :

$$(\text{NIL}, y_0, y_1, y_2, \ldots)$$

Les deux opérateurs sont complémentaires et généralement utilisés conjointement, de sorte que l'opérateur d'initialisation complète le cas initial laissé indéterminé par un décalage temporel. On obtient alors une combinaison de la forme :

```
x → pre(y)
```

La séquence de valeurs est alors complètement définie :

$$(x_0, y_0, y_1, y_2, \ldots)$$

L'imbrication de plusieurs retards unitaires couplés avec des flèches permet de faire référence à des valeurs antérieures au cycle précédent. L'exemple de la figure 2.4 illustre ceci : le flot `F` contient à chaque cycle n la valeur de la suite [3] de Fibonacci au rang n. Les flots `pF` et `ppF` correspondent respectivement à un décalage de 1 et de 2 cycles du flot `F`, complétés par des valeurs initiales bien choisies.

```
F   = pF + ppF      ;
pF  = 0 → pre(F)    ;
ppF = 1 → pre(pF)   ;
```

cycle	0	1	2	3	4	5	6	7	8	9
F	1	1	2	3	5	8	13	21	34	55
pF	0	1	1	2	3	5	8	13	21	34
ppF	1	0	1	1	2	3	5	8	13	21

FIGURE 2.4: Suite de Fibonacci en LUSTRE à l'aide d'opérateurs temporels imbriqués.

2.1.2.3 Propriétés invariantes

Nous terminons la présentation du langage en rappelant que LUSTRE permet de définir, au sein d'un nœud, un ensemble de propriétés invariantes exprimées à l'aide d'expressions booléenes. Parmi le bloc d'équations associé à un nœud, il est en effet possible d'introduire des expressions de la forme suivante :

```
assert Exp ;
```

3. Pour tout $n > 1$, $F_n = F_{n-1} + F_{n-2}$; $F_0 = F_1 = 1$

Ici, `Exp` est n'importe quelle expression booléene LUSTRE formée autour des variables visibles dans le nœud qui la contient. L'asssertion signifie qu'à chaque cycle d'exécution, l'expression `Exp` s'évalue à la valeur *true*. L'intérêt d'exprimer les invariants de cette manière permet de préciser aux outils d'analyse, comme un compilateur ou un *model-checker*, les hypothèses sur lesquelles repose un système.

Par exemple, on suppose que l'expression `Exp` est « `i > 0` ». Si `i` est une entrée du système, l'assertion est une précondition du système réactif, qui nécessite que `i` soit toujours strictement positive. À l'inverse, si `i` est le résultat d'un calcul, cette expression affirme que celui-ci est toujours positif au sein du système. Dans un cas ou l'autre, la propriété doit être prise en compte dans la conception du système.

2.1.3 LUSTRE multi-horloges et ses extensions

Jusqu'à présent, les flots de données des exemples précédents étaient tous cadencés sur le rythme global du système, que l'on appelle l'horloge de base : ils admettent une valeur exactement à tous les cycles d'exécution. Pour cela, on s'est restreint aux opérateurs usuels qui constituent le sous-ensemble de LUSTRE qualifié de « mono-horloge ». Le langage LUSTRE classique définit des constructions qui permettent de cadencer des flots de données sur des horloges calculées. Les horloges permettent d'activer ou de désactiver des calculs pendant le fonctionnement du système. Le langage que l'on présente maintenant est donc qualifié de « multi-horloges ».

Initialement, les horloges ont été assimilées à des flots booléens [CPHP87] : à chaque cycle où une horloge vaut *true*, les flots cadencés sur cette horloge peuvent être calculés ; sinon, ces flots n'admettent pas de valeur au cycle considéré. Comme une horloge est définie à l'aide d'un flot, il est possible de construire différents niveaux de calculs, chacun évoluant à son rythme propre tout en synchronisant les traitements.

Depuis la définition originale du langage, la notion d'horloges a été légèrement étendue à travers différents travaux portant sur les langages LUSTRE et LUCID SYNCHRONE [CP96, CGHP04, CPP05a]. Une horloge est alors vue comme une information de type associée à un flot et n'est plus matérialisée par un flot booléen. Nous décrivons dans la suite de cette section les mécanismes d'horloges, les opérateurs `when` et `merge` de filtrage et de projection de flots, ainsi que la réinitialisation asynchrone de flots de données. Mais tout d'abord, nous nous intéressons aux types énumérés, car toutes les autres notions en dépendent.

2.1.3.1 Types énumérés

Un type énuméré est une collection finie de symboles, identifiée par un nom. Chaque symbole est une valeur de ce type. Une déclaration de type énuméré a lieu en dehors d'un nœud, et se présente comme dans l'exemple ci-dessous :

```
type mouvement = enum{recule, stop, avance};
```

Avec cette déclaration, le programmeur définit un type appelé `mouvement`, dont les trois valeurs possibles sont les constantes `recule`, `stop` et `avance`. Il est alors possible de définir un flot de données de type `mouvement`, et chacune des valeurs qu'il prendra dans le temps sera une des valeurs symboliques du type. Les types énumérés permettent de définir des flots à valeurs logiques au-delà du type booléen qui ne peut représenter que deux valeurs diférentes. Ils remplacent les constantes numériques entières qui jouent habituellement le même rôle. Par exemple, avec la déclaration suivante, on peut représenter aussi un ensemble de trois valeurs qui ont signification particulière dans le modèle :

```
const RECULE = 0;
const STOP   = 1;
const AVANCE = 2;
```

Cependant, les types énumérés donnent plus d'informations que des constantes entières et permettent entre autres d'éviter des erreurs potentielles de manipulation de ces variables. Par exemple, l'expression `RECULE+2` est reconnue par un compilateur comme étant valide, alors que l'équivalent avec le type énuméré précédent n'est pas autorisé par le langage. Les seules opérations réalisables entre deux valeurs d'un même type énuméré sont les tests d'égalité et d'inégalité [4].

Les types énumérés sont pratiques pour modéliser un système sans recourir à des constantes, mais ils se révèlent surtout importants par le fait qu'il peuvent servir à cadencer des calculs sur des horloges complémentaires, ce qui, on le verra ensuite, permet de simuler un ensemble d'états d'un automate.

2.1.3.2 Horloges

Chaque flot de données F est lié à une information de type appelée *horloge*, notée $\acute{h}(F)$. À chaque cycle d'exécution, une horloge peut être soit *absente* soit *présente*. L'horloge de base est notée *base* et est présente à tous les cycles. Toute horloge h qui n'est pas l'horloge de base du système est de la forme (s, v), telle que :

1. le type associé à s est un type énuméré T
2. v est une valeur possible pour le type T

Alors, pour tout cycle c d'exécution du système, h est présente au cycle c si et seulement si le flot s admet une valeur au cycle c et que cette valeur est v. La variable de flot s est appelée *flot support* de l'horloge h.

Si une horloge h est présente à un cycle, tous les flots F d'horloge h admettent une valeur à ce cycle ; sinon, ces flots n'admettent pas de valeur à l'instant considéré. On peut

4. On verra avec l'opérateur « `merge` » que la position d'une valeur symbolique dans la déclaration de son type est importante, et qu'ainsi, les types énumérés sont ordonnés. Cependant, les opérateurs de comparaison ne s'appliquent pas sur les valeurs symboliques.

donc avoir des flots qui évoluent à des rythmes plus lents que le système, ces rythmes étant donnés par les instants de présence de leurs horloges respectives. Le traitement des flots temporisés est possible grâce à deux mécanismes principaux :

1. La construction d'un ou plusieurs niveaux d'horloges par filtrages de flots existants.
2. La reconstruction d'un flot par entrelacement de flots d'horloges différentes.

Nous présentons le filtrage à travers un exemple, décrivons les relations de parenté entre les horloges ainsi construites, puis montrons comment les flots ralentis peuvent être combinés entre eux et servir finalement à reconstruire, par entrelacement de leurs valeurs respectives, un flot plus rapide [5]. Nous décrivons en particulier les changements d'interprétation des opérateurs temporels et des assertions en présence d'horloges multiples.

2.1.3.3 Filtrage de flots

Le filtrage d'un flot par une horloge consiste à ne retenir d'un flot que les valeurs qui coïncident avec la présence de cette horloge. Il se présente sous la forme suivante :

```
y = x when (c match v) ;
```

Le flot `y` est le résultat du filtrage du flot `x` aux instants où le flot `c` vaut la valeur `v`. Ainsi, l'opérateur `when` prend deux flots, `x` et `c`, de même horloges, et construit le flot `y` d'horloge (c, v), présent uniquement aux cycles où `c` vaut `v`. Il existe deux constructions alternatives, où le membre droit de l'opérateur `when` est une variable booléenne ou sa négation :

```
y = x when b      ;
y = x when not(b) ;
```

Le type `bool` est assimilé à un type énuméré : les notations sont équivalentes aux filtrages selon les horloges respectives $(b, true)$ et $(b, false)$.

Exemple La figure 2.5 illustre l'utilisation de l'opérateur `when`. Le type énuméré `t` possède trois valeurs `a`, `b` et `c`. La flot `s` de type `t` sert à filtrer le flot entier `x`, pour chacune de ces valeurs, donnant ainsi les flots `xa`, `xb` et `xc` d'horloges respectives (s, a), (s, b) et (s, c). La valeur de `xc` sert à construire un flot booléen nommé `pos`, vrai à tous les cycles où `xc` est supérieur à zéro ; il faut pour cela filtrer la constante zéro pour que la comparaison puisse s'effectuer sur deux expressions de même horloge. Finalement, `xc` permet de construire deux flots, les valeurs de `xc` quand celles-ci sont positives (`xcp`) ou négatives (`xcn`).

5. Autrement dit, présent plus fréquemment que chacun des flots qui le composent.

```
type t = enum{a, b, c};

...

xa = (x when s match a);
xb = (x when s match b);
xc = (x when s match c);

pos = (xc >= (0 when s match c));

xcp = xc when pos;
xcn = xc when not(pos);
```

Cycle	0	1	2	3	4	5	6	7	8
x	25	10	31	-41	-10	23	-5	-15	10
s	a	c	b	c	a	b	a	c	a
xa	25				-10		-5		10
xb			31			23			
xc		10		-41				-15	
pos		T		f				f	
xcp		10							
xcn				-41				-15	

FIGURE 2.5: Exemple d'utilisation de l'opérateur when

2.1.3.4 Relations de parenté

Dans l'exemple 2.5, les horloges manipulées sont les suivantes : l'horloge de base, les horloges (s, a), (s, b), (s, c), ainsi que $(pos, true)$ et $(pos, false)$. Elles forment une hiérarchie, à travers la relation de parenté donnée par les horloges de leurs flots supports :

- *base* est l'horloge mère de (s, a), (s, b) et (s, c) car l'horloge de s est *base* ;
- (s, c) est l'horloge mère de $(pos, true)$ et $(pos, false)$, car $\acute{h}(pos) = (s, c)$.

2.1.3.5 Synchronisation des calculs

L'évaluation des autres opérateurs du langage, comme les opérations cycle-à-cycle, sont compatibles avec les horloges. Ainsi, les opérations arithmétiques et logiques sont applicables uniquement à des flots de même horloge : une expression telle que « `y = x + (x when c)` » n'a pas de sens, puisque les membres gauche et droit de l'addition sont nécessairement d'horloges différentes. Par exemple, si $\acute{h}(x) = base$, alors l'horloge du membre droit est (x, c) ; même si en pratique `c` peut valoir systématiquement *true* lors de l'exécution du système, de sorte que les horloges soient toujours présentes simultanément, l'expression est mal construite est n'est pas acceptée par le langage.

2.1.3.6 Initialisation et décalage

La sémantique des opérateurs temporels définis pour LUSTRE mono-horloge, à savoir le décalage unitaire et l'initialisation de flots, est généralisée en présence d'horloges multiples.

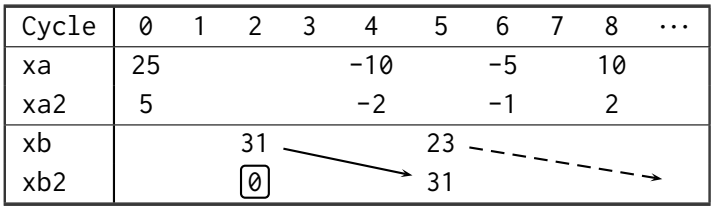


FIGURE 2.6: Opérations sur des flots ralentis

- L'expression E définie par « `A` → `B` », où `A` et `B` ont nécessairement la même horloge, s'évalue comme précédemment selon le membre `A` à l'instant initial. Cependant, cet instant initial est relatif à l'horloge de E : en effet, le premier instant où E est défini est le premier cycle auquel son horloge $\hbar(E)$ est présente.
- L'instant précédent d'un flot est lui-aussi relatif aux instants de présence de son horloge. Par exemple, l'expression F définie par « `A → pre(B)` », où `A` et `B` ont la même horloge h, fait référence aux valeurs passées de `B` dans le membre droit de son équation. Or, la valeur passée de `B` est celle au dernier cycle où son horloge était présente, qui n'est pas forcément le cycle précédent.

Exemple La trace d'exécution de la figure 2.6 réutilise les flots `xa` et `xb` obtenus précédemment à travers les équations suivantes :

```
-- Divison de xa par 5
xa2 = xa / (5 when s match a) ;

-- Decalage des valeurs de xb
xb2 = (0 when s match b) → pre(xb) ;
```

Le flot `xa2` est un exemple de calcul simple réalisé avec des flots de données ralentis : ici, les valeurs de `xa` sont divisés par cinq. La constante a été ralentie elle-aussi pour satisfaire aux règles du calcul d'horloges.

Le flot `xb2` réalise le décalage d'un cycle du flot `xb2`, relativement à son horloge. La valeur initiale du nouveau flot est zéro à sa première évaluation, puis la valeur précédente de `xb` aux autres cycles. Le zéro mis en évidence dans la figure est celui issu de l'initialisation, et marque le cycle initial du flot `xb`. On peut voir que ce cycle correspond à la première occurence de la valeur `b` pour `s`. Le décalage temporel aussi dépend de l'horloge, puisque la valeur précédente de `xb` correspond à la dernière valeur que le flot avait, la dernière fois que son horloge était présente.

Cycle	0	1	2	3	4	5	6	7	8
pos		T		f				f	
xcp		10							
xcn		↓		-41				-15	
xc2		10		41				15	
xa2	5	↓		↓	-2		-1	↓	2
xb2	↓	↓	0	↓	↓	31	↓	↓	↓
m	5	10	0	41	-10	31	-5	15	10
s	a	c	b	c	a	b	a	c	a

FIGURE 2.7: Exemple d'utilisation de l'opérateur merge

2.1.3.7 Projection de flots

L'opérateur d'entrelacement de flots « merge » réalise une projection de plusieurs flots d'horloges complémentaires vers un unique flot. Soient $n > 1$ et T un type énuméré de n éléments notés t_1 à t_n. Soient s un flot de données de type T et n expressions de flots notées $E_1, \ldots, E_n$, tels que :

- Tous les flots E_1 à E_n ont le même type T_E (indépendant de T)
- Quelque soit i entre 1 et n, $\acute{h}(E_i) = (s, t_i)$

Alors, les horloges (s, t_1) à (s, t_n) sont dites « horloges complémentaires » car elles reposent sur le même flot support s pour des valeurs différentes du type T. Par ailleurs, comme ces valeurs sont exactement les n valeurs de T, à chaque cycle où s admet une valeur, il existe exactement un k entre 1 et n tel que $s = t_k$ à ce cycle, et tel que E_k, d'horloge (s, t_k), admette elle-aussi une valeur à ce cycle. Ainsi, l'expression « merge(s, $E_1, \ldots, E_n$) » est un flot de type T_E et de même horloge que s, valant à chaque cycle la même valeur que l'unique expression E_k définie au moment considéré. Étant donnée la syntaxe de l'opérateur « merge », les types énumérés sont nécessairement *ordonnés*. La figure 2.7 est un exemple de recomposition de flots d'horloges différentes. Elle reprend les équations des exemples précédentes et les recompose pour obtenir un nouveau flot de données appelé m.

```
-- Valeur absolue par entrelacement de xcp et de (- xcn)
xc2 = merge(pos, xcp, (- xcn)) ;

-- Fusion des differents flots
m = merge(s, xa2, xb2, xc2);
```

2.1.3.8 Assertions temporisées

Nous avons vu précdemment qu'un système réactif peut rendre explicite certains de ses invariants, par le biais d'une expression de la forme « `assert Exp` ». Or, il est possible, en présence d'horloge, que l'expression `Exp` possède une horloge différente de l'horloge de base. Par exemple, si un système possède une entrée entière `i` et une entrée booléenne `b`, l'expression suivante est valide :

```
assert ((i > 0) when b);
```

L'invariant exprimé ici est qu'à chaque instant où `b` vaut *true*, la relation « `i > 0` » est vérifiée. En particulier, l'assertion ne dit rien à propos des instants où l'horloge du filtrage est absente.

2.1.3.9 Réinitialisations de flots

La réinitialisation de flots est une extension du langage Lustre qui propose une primitive de reconfiguration pour les flots de données [HP00]. L'opérateur « $\rightarrow$ » de Lustre est modifié pour pouvoir retourner dans un état initial suivant la valeur de flots booléens.

Soient A et B deux expressions Lustre de même type et de même horloge h. Soit $R = R_1, \ldots, R_n$ un ensemble d'expressions booléennes Lustre d'horloges respectives *quelconques*. Soit c un cycle auquel l'horloge h est présente. On note E l'expression « $A \rightarrow * \ (R_1 ; \ldots ; R_n) \ B$ ».

Alors, l'opérateur « $\rightarrow *$ » est appelé *opérateur de réinitialisation* (ou de *redémarrage*) ; les flots R_1 à R_n sont appelés *flots de réinitialisation*. La sémantique de l'opérateur est la suivante : si c est le premier cycle auquel h est présente (le cycle intial de l'expression), alors E s'évalue comme la valeur de A au cycle c ; sinon, l'expression s'évalue comme B, à moins que l'expression ait été réinitialisée par un des flots de réinitialisation depuis la dernière fois que E admettait une valeur.

Plus précisément, pour un cycle c qui n'est pas le cycle initial de l'expression, on appelle *fenêtre (temporelle) de réinitialisation* de l'expression E au cycle c l'ensemble des cycles compris entre c (inclus) et le dernier cycle (exclu) de présence de l'horloge h.

- S'il existe au moins une occurence de la valeur *true* pour au moins un des flots R_i ($1 \leq i \leq n$) dans la fenêtre de réinitialisation, alors E s'évalue aussi comme la valeur au cycle c du membre gauche, ici A.
- Sinon, l'expression E s'évalue comme la valeur de B à ce même cycle.

L'opérateur « $\rightarrow *$ » permet donc de réinitialiser des calculs en revenant dans le membre gauche de la flèche, lorsqu'au moins un des flots de réinitialisation devient vrai. Dans les cas où l'ensemble de flots R est vide ou ces flots sont systématiquement faux ou absents, la sémantique est équivalente à celle de l'opérateur « $\rightarrow$ » usuel. On peut remarquer que les flots de réinitialisation peuvent avoir des horloges indépendantes les

Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
h	f	f	T	f	f	T	f	f	f	f	T	f	f	T	f
f			0			1					0			0	
g			0			1					2			3	
r		f		f		f		f		f		f		T	
s	f			f			T			f			f		

FIGURE 2.8: Réinitialisation de flots

unes des autres, et indépendantes de l'horloge commune aux deux branches de la flèche. Pour cette raison, l'opérateur est qualifié d'opérateur de redémarrage *asynchrone*.

Exemple La figure 2.8 illustre les différents termes que l'on vient de définir et la sémantique de l'opérateur. Les flots `f` et `g` sont définis par les équations suivantes :

```
f = (0 when h) →* (r; s) (pre(f) + (1 when h)) ;
g = (0 when h) → (pre(g) + (1 when h)) ;
```

Le flot booléen `h` est une entrée du système, de même que les flots `r` et `s`. On a $\acute{h}(f) = (h, true)$ et $\acute{h}(h) = base$; l'horloge de `r` est présente aux cycles impairs, celle de `s` est présente aux cycles multiples de 3. L'expression portée par `f` est réinitialisable par les flots `s` et `r`. Le flot `g` représente la même équation que `f` mais avec une flèche simple, sans redémarrage. Dans la figure 2.8, les numéros de cycles encadrés représentent les fenêtres temporelles successives pendant lesquelles les flots `r` et `s` peuvent réinitialiser le flot `f` : par exemple, aux cycles 3, 4 et 5, toute occurence de *true* pour `r` ou `s` réinitialise la valeur de `F` au cycle 5. Au cycle 6, le flot `s` prend la valeur *true* ; bien que l'horloge de `f` ne soit pas présente à ce cycle, la réinitialisation est prise en compte pour la prochaine évaluation de ce flot, au cycle 10. De même, au cycle 16, le flot `r` force `f` à valoir zéro. Le flot `g`, quant à lui, incrémente progressivement au cours du temps.

2.1.3.10 Grammaire du langage

Nous résumons la syntaxe du langage LUSTRE multi-horloges étendu en rappelant sa syntaxe concrète. Il s'agit de celle que nous adoptons dans les extraits de code à travers nos différents exemples. Chaque symbole non-terminal est constitué de règles de production impliquant des symboles terminaux et non-terminaux. Les symboles terminaux sont donnés de manière littérale, à l'aide d'ensembles de mots ou d'expressions rationnelles. Le mot vide est représenté par ϵ. Les doubles crochets $[\![E]\!]$ autour d'une expression E indiquent zéro ou une occurence de cette expression. Les doubles accolades

⦃*E*⦄ représentent zéro ou plusieurs occurences successives de l'expression *E*. Nous commentons les règles à la suite de la grammaire.

Règles de production

Model	::=	⦃ *TypeDecl* ⦄ *Node*	— Modèle
TypeDecl	::=	`type` **ID** `=` `enum` `{` **SYMB** ⦃ `,` **SYMB** ⦄ `}` `;`	— Déclaration de type
Node	::=	`node` **ID**	— Déclaration du nœud
		`(` *Var* ⦃ `;` *Var* ⦄ `)`	— Entrées
		`returns` `(` *Var* ⦃ `;` *Var* ⦄ `)` `;`	— Sorties
		⟦ `var` *Var* ⦃ `;` *Var* ⦄ ⟧	— Variables locales (si nécessaire)
		`let` ⦃ *BodyElt* ⦄ `tel` `;`	— Corps du nœud
Var	::=	**ID** `:` *Type* ⟦ `when` *ClockExp* ⟧	— Déclaration d'une variable
Type	::=	`int` | `bool` | **ID**	— Types associés aux flots
ClockExp	::=	**ID** `match` **SYMB**	— Cas général avec types énumérés
	|	**ID** | `not` **ID**	— Notation simplifiée pour les booléens
BodyElt	::=	**ID** `=` *Exp* `;`	— Équation
	|	`assert` *Exp* `;`	— Assertion
Exp	::=	`(` *Exp* **BINOP** *Exp* `)`	— Opérateur binaire
	|	`(` **UNOP** *Exp* `)`	— Opérateur unaire
	|	`(` `if` *Exp* `then` *Exp* `else` *Exp* `)`	— Expression conditionnelle
	|	`(` *Exp* `when` *ClockExp* `)`	— Filtrage
	|	`merge` `(` **ID** `,` *Exp* `,` *Exp* ⦃ `,` *Exp* ⦄ `)`	— Fusion de flots
	|	`(` *Exp* $\rightarrow *$ `(` *Exp* ⦃ `;` *Exp* ⦄ `)` *Exp* `)`	— Réinitialisation
	|	**ID**	— Référence à une variable
	|	**BOOL** | **INT** | **SYMB**	— Valeurs littérales

INT	$\in$	$\mathbb{Z}$
BOOL	$\in$	$\{true, false\}$
UNOP	$\in$	$\{-, pre, not\}$
BINOP	$\in$	$\{+, -, *, div, mod, \rightarrow, and, or, =, \neq, >, <, \leq, \geq\}$
ID	$\in$	[a-z][a-zA-Z0-9]*
SYMB	$\in$	[a-z][a-zA-Z0-9]*

Commentaires

Model – Un modèle LUSTRE est donné par un ensemble facultatif de déclarations de type,

suivi d'un unique nœud.

TypeDecl – Dans une déclaration des types énumérés, chaque type possède un identifiant, distinct des autres idenfitiants de type. Un type définit alors plusieurs valeurs symboliques, propres à ce type.

Node – Un nœud met en relation un ensemble d'entrée avec un ensemble de sorties et admet des variables intermédiaires. La déclaration d'un nœud comporte un en-tête, un bloc optionnel pour la déclaration de variables, ainsi que le *corps* du nœud qui contient les équations et les assertions.

Var – Une déclaration de variable dans l'en-tête d'un nœud consiste à associer un type à un identifiant, avec éventuellement une horloge. Chaque identifiant est unique au sein du nœud. Les variables locales et en sorties sont nécessairement associées à une équation dans le corps du nœud.

Type – Les deux types de bases que l'on considère sont les entiers et les booléens. Par ailleurs, une déclaration de variable peut faire référence à un type énuméré.

ClockExp – Le membre droit d'un opérateur « `when` » est est une expression de filtrage, donnée par la correspondance entre une variable et une valeur symbolique, par une variable booléenne ou encore par la négation d'une variable booléenne.

BodyElt – Le corps d'un nœud contient des équations associées à des variables, ainsi que des assertions. Toute variable calculée doit posséder une équation de définition. À l'inverse, une variable d'entrée ne peut pas être liée à une équation.

Exp – Les expressions de flots sont définies inductivement, en combinant sous-expressions à l'aide d'opérateurs arithmétiques, logiques et temporels. Les expressions atomiques du langage sont soit des littéraux numériques, booléens ou symboliques, soit des références à d'autres variables du nœud. Deux variables ne peuvent pas dépendre mutuellement l'une de l'autre à travers leurs définition.

Espaces de noms Les identifiants de types, de variables, du nœud, mais aussi les symboles présents dans les déclarations de types sont tous des mots composés de symboles alphanumériques, dont la première lettre est en minuscule. Nous ne donnons pas explicitement les règles qui permettent d'éviter les conflits de nommage de ces différents objets. Certaines de ces règles seront détaillées dans la présentation formelle du langage. Dans les exemples que nous prenons, les différents identifiants sont distincts pour éviter les ambigüités.

2.2 Définition du langage $\mathcal{L}_+$

Le langage Lustre mutli-horloges étendu que l'on vient de présenter nous servira tout au long du document. Nous définissons ici le langage de manière formelle, que l'on notera désormais $\mathcal{L}_+$. Nous définissons formellement l'ensemble des modèles bien formés, en introduisant notamment les types énumérés, les déclarations conjointes de types, les ensembles de nœuds bien formés, et finalement les expressions bien formées dans ces nœuds. Nous donnons ensuite une sémantique d'évaluation basée sur des grilles de valeurs, compatible avec l'approche prise dans la génération de tests.

2.2.1 Déclarations de types bien formées

Nous commençons par nous intéresser aux types énumérés, qui définissent les types et les valeurs disponibles dans les nœuds Lustre.

2.2.1.1 Valeurs symboliques

Soit $\mathcal{S}ymb$ un ensemble, disjoint de $\mathbb{Z}$, de symboles distincts. Ces symboles sont les valeurs littérales que l'on rencontre dans les types énumérés. Pour des raisons pratiques, le type `bool` de Lustre est assimilé à un type énuméré. On pose :

$$\{true, false\} \subseteq \mathcal{S}ymb$$

2.2.1.2 Identifiants

On note $Idents$ l'ensemble, disjoint de $\mathbb{Z}$, des identifiants du langage $\mathcal{L}_+$. Il regroupe les identifiants de types, de variables et de nœuds.

2.2.1.3 Type énuméré

On note $\mathcal{E}nums$ l'ensemble des types énumérés que l'on peut construire à partir des symboles de $\mathcal{S}ymb$. Un type énuméré est un ensemble fini de symboles, pris dans $\mathcal{S}ymb$, dont les éléments sont indexés par un entier.

Ordre des valeurs Pour tout $E \in \mathcal{E}nums$ de cardinal $n > 0$, il existe une application injective [6] de $1..n$ vers E reliant un indice entier à un unique élément de E. On note E_i l'élément d'indice i d'un tel type énuméré E. Nous adoptons la convention que les types énumérés sont toujours représentés avec les éléments triés selon leur indices croissants.

Identifiant de type L'ensemble $\mathcal{E}nums$ est muni d'une relation $ident$ associant à un type énuméré un identifiant. L'application va des types énumérés vers l'ensemble $Idents \backslash \{int\}$, afin d'éviter le conflit de nom avec le type prédéfini int.

6. Dans le contexte d'un nœud, celle-ci sera bijective.

Type booléen Le type « bool » de LUSTRE est représenté par le type énuméré $\mathcal{B}ool$:

- $\mathcal{B}ool = \{false, true\}$
- $\mathcal{B}ool_1 = false$
- $\mathcal{B}ool_2 = true$
- $ident(\mathcal{B}ool) = bool$

2.2.1.4 Déclarations de types

On note $\mathcal{D}ecls$ l'ensemble de toutes les déclarations de types bien formées. Une déclaration de types est une *partition* finie de symboles appartenant à $\mathcal{S}ymb$, où chaque partie est un type énuméré ayant un identifiant propre. Chaque symbole est donc unique dans l'union des valeurs symboliques définies à travers la déclaration. Toute déclaration de types comprend au moins le type booléen $\mathcal{B}ool$:

$$\forall D \in \mathcal{D}ecls, \mathcal{B}ool \subseteq D$$

Les identifiants des types énumérés sont distincts ; soit $D \in \mathcal{D}ecls$:

$$\forall (E, F) \in D^2, (E \neq F) \Rightarrow (ident(E) \neq ident(F))$$

Exemple L'ensemble suivant est une déclaration de types [7] :

$$D_1 = \{\{true, false\}, \{on, off, standby\}, \{nominal, failsoft\}\}$$

Identifiants des types On définit l'application injective *idents* qui associe à chaque déclaration de types D l'ensemble des identifants de types énumérés que la déclaration contient.

$$\forall D \in \mathcal{D}ecls : idents(D) = \bigcup_{E \in D} ident(E)$$

Ensemble de valeurs On définit l'application *symbols* qui associe à chaque déclaration de types D l'ensemble des valeurs symboliques déclarées dans les types énumérés de D.

$$\forall D \in \mathcal{D}ecls : symbols(D) = \bigcup_{E \in D} E$$

Type associé à un symbole On définit enfin, pour tout $D \in \mathcal{D}ecls$, l'application $type_D$ qui associe à chaque valeur symbolique définie dans D le type énuméré auquel appartient cette valeur dans D :

$$\forall D \in \mathcal{D}ecls, \forall E \in D, \forall s \in E : type_D(s) = E$$

7. Les identifiants des types énumérés ne sont pas détaillés dans cet exemple. On suppose qu'ils sont tous différents les uns des autres.

Exemple Avec la déclaration D_1 précédente on a :

$$type_{D_1}(failsoft) = \{nominal, failsoft\}$$

2.2.1.5 Types de données

Soit $D \in \mathcal{D}ecls$. L'ensemble des types de données que l'on peut affecter à un flot LUSTRE est noté $\mathcal{T}ypes_D$. Il contient les identifiants des types énumérés (notamment *bool*), ainsi que le type prédéfini *int*.

$$\mathcal{T}ypes_D = idents(D) \cup \{int\}$$

2.2.1.6 Valeurs

Soit $D \in \mathcal{D}ecls$. L'ensemble des valeurs que l'on peut affecter à un flot LUSTRE de type quelconque est noté $\mathcal{V}alues_D$. Cet ensemble regroupe l'ensemble des valeurs symboliques de la déclaration D et l'ensemble des entiers relatifs associés au type *int*.

$$\mathcal{V}alues_D = symbols(D) \cup \mathbb{Z}$$

2.2.1.7 Horloges

Une horloge peut-être le symbole *base*, appelée horloge de base, ou bien une horloge relative, donnée par un couple associant un identifiant de flot, de type énuméré, à une valeur de ce flot (un symbole). Soit $D \in \mathcal{D}ecls$. L'ensemble des horloges $\mathcal{C}locks_D$ est ainsi défini :

$$\mathcal{C}locks_D = \{base\} \cup \big(Idents \times symbols(D)\big)$$

Pour toute horloge relative $C = (F, V) \in \mathcal{C}locks_D$, le flot identifié par F est appelé flot support de l'horloge C, et V sa valeur de présence. La définition des horloges est volontairement incomplète ici. Pour garantir que les horloges des flots sont bien formées, nous devons nous placer dans le contexte d'un nœud, où l'on peut, par exemple, s'assurer que F fait référence à un flot existant, ce que nous faisons à la section 2.2.2.1.

2.2.1.8 Variable de flot

Soit $D \in \mathcal{D}ecls$. On note $\mathcal{V}ars_D$ l'ensemble des variables de flots LUSTRE construites à partir de D. Une variable $v \in \mathcal{V}ars_D$ est donnée par un triplet (Id, T, C) où Id est un identifiant, T un type et C une horloge, avec les propriétés suivantes :

- $Id \in Idents \setminus \big(\mathcal{V}alues_D \cup \mathcal{T}ypes_D\big)$
  Id est un identifiant, distinct à la fois d'une valeur symbolique et d'un identifiant de type. Cette définition exclut la possibilité d'avoir un type et une variable qui

partage le même nom. Cette restriction nous permet de simplifier la manipulation des identifants par la suite.
- $T \in \mathcal{T}ypes_D$
- $C \in \mathcal{C}locks_D$

Accesseurs On note respectivement $id(v)$, $type(v)$ et $clock(v)$ l'identifiant Id, le type T et l'horloge C qui composent la variable v.

Identifiants L'application auxiliaire ids collecte, pour tout ensemble de variables, l'ensemble des identifiants de ces variables ; soit $D \in \mathcal{D}ecls$:

$$\forall V \subseteq \mathcal{V}ars_D : ids(V) = \bigcup_{v \in V} id(v)$$

2.2.2 Nœuds et modèles bien formés

Nous définissons un nœud par le tuple (D, I, L, O, A) associant une déclaration de types D associée à trois ensembles de flots I, L, et O bien formés vis-à-vis de D ainsi qu'un ensemble A d'expressions booléennes représentant les invariants du modèle. Ainsi, un modèle Lustre est donné exactement par un nœud, où la déclaration de types est une composante du nœud. L'ensemble $\mathcal{L}_+$ représentant le langage Lustre multi-horloges étendu est donc à la fois l'ensemble des modèles et des nœuds bien formés dans ce langage.

Nous définissons l'ensemble $\mathcal{L}_+$ des nœuds bien formés, ainsi que l'ensemble $\mathcal{E}xprs_N$ des expressions que l'on peut écrire dans un nœud $N \in \mathcal{L}_+$. Un tel nœud N est donné par un quintuplet (D, I, L, O, A), avec :

1. $D \in \mathcal{D}ecls$.
2. $I \subseteq \mathcal{V}ars_D$ est un ensemble non vide de flots sans équations de définition. De tels variables correspondent à des flots de valeurs prises en entrée du système.
3. L et O deux sous-ensembles de $\mathcal{V}ars_D$ respectivement associés aux variables locales et aux sorties du nœud. L peut être vide, O contient au moins une variable. Les deux ensembles regroupent les flots calculés du systèmes, définis par une expression de flot. Soit $V = L \cup O$. L'ensemble V est muni de l'application def :

   $$def \ : \ V \rightarrow \mathcal{E}xprs_N$$

   Les applications T et C représentent respectivement le type et l'horloge d'une expression bien formée appartenant à $\mathcal{E}xprs_N$. Elles sont définies un peu plus loin (§2.2.2.3 et §2.2.2.4), mais cependant, on peut déjà préciser que les expressions de définition des variables calculées d'un nœud doivent partager le même type et la même horloge. Ainsi, pour tout $v \in V$, on a :

   $$\vdash_N \ type(def(v)) = type(v)$$
   $$\vdash_N \ clock(def(v)) = clock(v)$$

4. A un sous-ensemble de $\mathcal{E}xprs_N$ d'expressions booléennes :

$$\forall a \in A, \vdash_N type(a) = \mathcal{B}ool$$

Identification des variables Dans le contexte du nœud N, il est maintenant possible de définir les liens entre les identifiants et les variables qu'ils référencent. On définit pour tout nœud N l'ensemble suivant :

$$\mathcal{V}_N = I \cup L \cup O$$

Alors, on note $var_N(i)$ la variable référencée dans N par tout identificateur « i » appartenant à $ids(\mathcal{V}_N)$:

$$\forall v \in \mathcal{V}_N : var_N(id(v)) = v$$

Pour éviter les conflits, les ensembles de variables ne partagent pas les mêmes identifiants :

$$\begin{array}{lcll} ids(I) & \cap & ids(L) & = \emptyset \\ ids(I) & \cap & ids(O) & = \emptyset \\ ids(L) & \cap & ids(O) & = \emptyset \end{array}$$

2.2.2.1 Flots bien formés vis-à-vis des horloges

Les variables d'un nœud, $\mathcal{V}_N$, respectent des règles de bonne construction vis-à-vis de leurs horloges.

Horloge valide Tout d'abord, leurs horloges respectent les propriétés générales suivantes :

1. Les identifiants des flots supports correspondent à des variables déclarées
2. Les valeurs de présence appartiennent au type du flot support

Nous avons défini l'ensemble des horloges construits à partir d'une déclaration de types D, à savoir $\mathcal{C}locks_D$, à la section 2.2.1.7. Pour tout nœud $N = (D, I, L, O, A) \in \mathcal{L}_+$, nous définissons maintenant l'ensemble $\mathcal{C}locks_N$ des horloges bien formées dans le nœud N. L'horloge c appartient à $\mathcal{C}locks_N$ si et seulement si :

1. $c \in \mathcal{C}locks_D$,
2. Si $c = (id, v)$ est une horloge relative :
   - il existe $f = var_N(id)$ une variable de flot déclarée dans N dont l'identifiant est id.
   - La valeur de présence v appartient au type de f, qui est un nécessairement un type énuméré :
     $v \in \mathcal{S}ymb \cap type(f)$

On dit alors que c est une horloge valide dans le nœud N.

Hiérarchie d'horloges En plus de ces règles isolées, les flots de données sont dépendants les uns des autres, par l'intermédiaire des horloges. Ainsi, il ne peut pas exister de dépendance cyclique entre les variables : les horloges constituent une hiérarchie construite inductivement à travers les couples d'identifiants et de valeurs de présence. Pour cette relation de dépendance induite par les horloges, il existe des règles supplémentaires pour les ensembles I, L et O :

1. Les horloges des flots de I sont construites à partir de variables d'entrées de I.
2. Les horloges des flots de O sont construites à l'aide de variables de I ou d'autres variables de O.
3. Les horloges des variables locales présentes dans L peuvent être construite par l'intermédiare de variables locales aussi bien que de variables en entrée ou en sortie.

Ces propriétés sont vérifiées si les ensembles « I », « $I \cup O$ » et « $I \cup L \cup O$ » sont respectivement bien formés, au sens de la définition 10 suivante.

Définition 10. *Ensembles de flots bien formés dans un nœud*

Soit $W \subseteq \mathcal{V}_N$ un sous-ensemble des variables d'un nœud $N = (I, L, O)$. On dit que W est bien formé, si :
- $W = \emptyset$, ou
- Il existe $w = (i, t, c) \in W$ tel que :
  - $W = \{w\} \cup W'$, avec $W' \subset W$ un sous-ensemble bien formé,
  - L'horloge c est valide dans N,
  - De plus, si c est une horloge relative (id, v), on a aussi $var_N(id) \in W'$: le flot support identifié par id appartient à W', ce qui empêche les dépendances cycliques entre les flots à travers les horloges.
  - L'équation associée à w a le type t : $\vdash_N type(def(w)) = t$
  - L'équation associée à w a pour horloge c : $\vdash_N clock(def(w)) = c$

La définition repose sur un découpage de l'ensemble W en sous-ensembles bien formés, de manière inductive sur les *étages* de dépendances entre flots :
- Les flots sur l'horloge de base sont bien formés
- Les flots dont les horloges dépendent de flots bien formés et qui satisfont aux contraintes de types et d'horloges sont eux-aussi bien formés.

Exemple La figure 2.9 est un exemple de modèle bien formé, exprimé dans la syntaxe concrète du langage. La représentation formelle est donnée par le nœud $N = (D, I, L, O, A)$ appartenant à $\mathcal{L}_+$, où $D \in \mathcal{D}ecls$ correspond à la déclaration de types :

$$D = \{\mathcal{B}ool, \{a, b, c\}\}$$

```
type abc = enum { a, b, c } ;
node exempleABC (h: bool; x: int; s: abc when h)
     returns (y: int when s match a);
let
    y = (x when h) when s match a ;
tel;
```

FIGURE 2.9: Modèle bien formé avec filtrages successifs.

On a :

$$\begin{aligned} I &= \{(h, \mathcal{B}ool, base), (x, int, base), (s, \{a, b, c\}, (h, true))\} \\ L &= \{\} \\ O &= \{(y, int, (s, a))\} \end{aligned}$$

L'ensemble I est bien formé car :

1. Le flot h est un flot booléen, le flot x est un flot d'entiers ; tous deux sont définis sur l'horloge de base, et donc $I_0 = \{(h, \mathcal{B}ool, base), (x, int, base)\}$ est bien formé.
2. Le flot « s » a pour horloge $(h, true)$, avec h un flot bien formé dans l'ensemble I_0 et $true$ un symbole appartenant à $type(h)$.

De même, l'ensemble $I \cup O$ est bien formé, car l'horloge du flot « y » est composée du flot support « s », bien formé dans I, et de la valeur de présence « a » appartenant à $type(s)$. L'ensemble L étant vide, on sait aussi que $\mathcal{V}_N = I \cup L \cup O$ est bien formé.

Finalement, dans l'exemple, l'unique variable calculée y est bien associée à une équation dont le type et l'horloge correspondent bien à sa déclaration. Cela est donné par les règles de bonnes constructions des expressions dans un nœud, que nous allons étudier dans la section suivante.

2.2.2.2 Expressions bien formées

Étant donné un nœud $N \in \mathcal{L}_+$, nous définissons l'ensemble des expressions bien formées $\mathcal{E}xprs_N$. Une expression $E \in \mathcal{E}xprs_N$ est bien formée si :

1. E respecte la syntaxe concrète donnée à la section 2.1.3.10.
2. E respecte les règles de typage et de calcul d'horloges décrites aux sections 2.2.2.3 et 2.2.2.4.
3. Les identifiants auquels E fait référence renvoient à des variables et des valeurs symboliques qui sont définies dans le contexte du nœud. Cette propriété est vérifiée lorsque l'expression satisfait aux règles précédentes.

4. E ne possède pas de dépendance cyclique par l'intermédiaire des variables dont elle dépend. En d'autres termes, un équation associée à une variable v n'a pas le droit de faire référence, directement ou par le biais d'une autre variable, à la valeur de v (au même cycle). Si cela était possible, la définition de v à un cycle serait récursive, comme dans l'exemple suivant :

   ```
   v = -2 * v + 3;
   ```

   Bien que dans ce cas particulier, il existe une solution à l'équation, la définition n'est pas permise par le langage. Cette restriction s'étend aux ensembles de variables, qui n'ont pas le droit de faire référence directes ou indirectes à des variables dont la défintion est cyclique :

   ```
   x = 2 * y ;
   y = 5 + x ;
   ```

   Cependant, on rappelle qu'il est correct de faire référence à des valeurs passées d'un flot. Nous ne détaillons pas les règles dans ce document.

5. En plus des règles précédentes, les expressions calculées en sortie d'un nœud doivent être bien intialisées. Cela signifie qu'il n'existe pas d'expression portée par un « `pre` » qui ne soit pas incluse comme sous-expression dans le membre droit d'un opérateur d'initialisation « → », et ce, directement, où à travers des références à des variables intermédiaires.

   L'exemple de la figure 2.10 montre un nœud dans lequel un calcul intermédiaire, porté par la variable `v`, est mal initialisé. Cependant, toute contribution de `v` au calcul d'une variable en sortie complète la valeur intiale, ce qui explique pourquoi le nœud est considéré comme étant bien formé. Nous ne détaillons pas non plus les règles de vérification de cette propriété.

Les sections suivantes décrivent les règles de bonne formation des expressions, dans le contexte d'un nœud $N \in \mathcal{L}_+$. Dans ces règles, nous adoptons ces notations :

- La lettre τ représente le type que l'on cherche à associer à une expression.
- La lettre $\mathscr{E}$ représente un type énuméré, définissant n valeurs symboliques.
- La lettre σ représente un des symboles de $\mathscr{E}$.
- La notation $\mathscr{E}_i$ fait référence au i^{me} élément de $\mathscr{E}$.
- Les expressions sont notées en majuscules, avec si nécessaire des indices, comme A, B, E, F, E_i *(merge)* ou R_j $(\rightarrow *)$.

2.2.2.3 Règles de typage

La figure 2.11 détaille les règles de typage des expressions. Une expression $E \in \mathcal{E}xprs_N$ est bien formée s'il existe un type $T \in \mathcal{T}ypes_D$ tel que les règles permettent de définir un

```
node initcheck(x: int)
     returns (y: int; b: bool);
var
    v: int;
let
    v = pre(y);
    y = (x → (v + x)) ;
    b = (false → (y > v)) ;
tel;
```

FIGURE 2.10: Nœud illustrant la bonne intialisation des variables en sortie.

jugement de la forme suivante :

$$\vdash_{\mathrm{N}} type(\mathsf{E}) = T$$

Ce jugement se lit : le type de l'expression E est T dans le nœud N. La relation de typage est définie de manière inductive sur les termes du langage.

Combinaison de flots Les règles (T1) à (T11) ainsi que la règle (T13) pour l'opérateur conditionnel sont des règles simples : les valeurs de bases, ainsi que les valeurs symboliques, sont affectées au type correspondant ; les expressions se combinent de manière usuelle. Par exemple, la règle (T7) indique que si les deux expressions A et B sont des flots d'entiers, alors l'expression combinée « $A \mathscr{R} B$ » est de type booléen, quelque soit la relation de comparaison $\mathscr{R}$.

Projection de flots Dans la règle (T12) de typage de l'opérateur « `merge` », La lettre $\mathscr{E}$ représente un type énuméré définissant n valeurs symboliques. Le nombre d'argument de l'opérateur est alors $n+1$, à savoir la variable de contrôle F dont le type est le type énuméré $\mathscr{E}$, et n expressions de même type τ. Le résultat de la projection a alors pour type τ.

Filtrage de flots La règle (T14) concerne le filtrage de flots selon une horloge. Dans la règle, l'horloge en question est (id, σ), où id est l'identifiant d'un flot f et σ un symbole appartenant au type énuméré $\mathscr{E}$ associé à f. Le type résultat est le même que le type de l'expression filtré, à savoir τ, le type de E.

2.2.2.4 Règles de calculs d'horloges

Les règles de la figure 2.12 donnent l'horloge d'une expression dans le contexte d'un nœud (et d'une déclaration de types). Elles sont elles aussi définies inductivement, de

$$\frac{z \in \mathbb{Z}}{\vdash_N type(z) = int}\ (T1) \qquad \frac{\sigma \in symbols(D)}{\vdash_N type(\sigma) = type_D(\sigma)}\ (T2) \qquad \frac{id \in Idents}{\vdash_N type(id) = type(var_N(id))}\ (T3)$$

$$\frac{\vdash_N type(\texttt{E}) = int}{\vdash_N type(\texttt{-E}) = int}\ (T4) \qquad \frac{\vdash_N type(\texttt{E}) = \mathcal{B}ool}{\vdash_N type(\texttt{not E}) = \mathcal{B}ool}\ (T5) \qquad \frac{\vdash_N type(\texttt{E}) = \tau}{\vdash_N type(\texttt{pre E}) = \tau}\ (T6)$$

$$\frac{\mathcal{R} \in \{<, >, \leq, \geq\} \qquad \vdash_N type(\texttt{A}) = int \qquad \vdash_N type(\texttt{B}) = int}{\vdash_N type(\texttt{A } \mathcal{R} \texttt{ B}) = \mathcal{B}ool}\ (T7)$$

$$\frac{\mathcal{R} \in \{=, \neq\} \qquad \vdash_N type(\texttt{A}) = \tau \qquad \vdash_N type(\texttt{B}) = \tau}{\vdash_N type(\texttt{A } \mathcal{R} \texttt{ B}) = \mathcal{B}ool}\ (T8)$$

$$\frac{\mathcal{R} \in \{+, *, div, mod\} \qquad \vdash_N type(\texttt{A}) = int \qquad \vdash_N type(\texttt{B}) = int}{\vdash_N type(\texttt{A } \mathcal{R} \texttt{ B}) = int}\ (T9)$$

$$\frac{\vdash_N type(\texttt{A}) = \tau \qquad \vdash_N type(\texttt{B}) = \tau}{\vdash_N type(\texttt{A} \rightarrow \texttt{B}) = \tau}\ (T10)$$

$$\frac{\begin{array}{c}\vdash_N type(R_1) = \mathcal{B}ool \\ \vdots \\ \vdash_N type(R_n) = \mathcal{B}ool\end{array} \qquad \vdash_N type(\texttt{A}) = \tau \qquad \vdash_N type(\texttt{B}) = \tau}{\vdash_N type(\texttt{A} \rightarrow * \texttt{ } (R_1, \ldots, R_n) \texttt{ B}) = \tau}\ (T11)$$

$$\frac{\begin{array}{c}\vdash_N type(E_1) = \tau \\ \vdots \\ \vdash_N type(E_n) = \tau\end{array} \qquad \vdash_N type(\texttt{F}) = \mathcal{E} \qquad |\mathcal{E}| = n}{\vdash_N type(\texttt{merge(F}, E_1, \ldots, E_n\texttt{)}) = \tau}\ (T12)$$

$$\frac{\vdash_N type(\texttt{I}) = \mathcal{B}ool \qquad \vdash_N type(\texttt{T}) = \tau \qquad \vdash_N type(\texttt{E}) = \tau}{\vdash_N type(\texttt{if I then T else E}) = \tau}\ (T13)$$

$$\frac{\vdash_N type(\texttt{E}) = \tau \qquad \vdash_N type(id) = \mathcal{E} \qquad \mathcal{E} \in D \qquad \sigma \in \mathcal{E}}{\vdash_N type(\texttt{E when } id \texttt{ match } \sigma) = \tau}\ (T14)$$

FIGURE 2.11: Règle de typage des expressions

$$\frac{v \in \mathcal{V}alues_D}{\vdash_N clock(v) = base}\ (C1) \qquad \frac{id \in Idents}{\vdash_N clock(id) = clock_{\zeta}(var_N(id))}\ (C2)$$

$$\frac{\vdash_N clock(\mathtt{E}) = \gamma \quad \mathscr{U} \in \{-, not, pre\}}{\vdash_N clock(\mathscr{U}\ \mathtt{E}) = \gamma}\ (C3) \qquad \frac{\vdash_N clock(\mathtt{A}) = \gamma \quad \vdash_N clock(\mathtt{B}) = \gamma \quad \mathscr{R} \in \{<, >, \leq, \geq, =, \neq, +, *, div, mod, \rightarrow\}}{\vdash_N clock(\mathtt{A}\ \mathscr{R}\ \mathtt{B}) = \gamma}\ (C4)$$

$$\frac{\vdash_N clock(\mathtt{I}) = \gamma \quad \vdash_N clock(\mathtt{T}) = \gamma \quad \vdash_N clock(\mathtt{E}) = \gamma}{\vdash_N clock(\mathtt{if\ I\ then\ T\ else\ E}) = \gamma}\ (C5) \qquad \frac{\vdash_N clock(id) = \gamma \quad \vdash_N clock(\mathtt{E}) = \gamma \quad \mathscr{E} \in D \quad \sigma \in \mathscr{E}}{\vdash_N clock(\mathtt{E\ when}\ id\ \mathtt{match}\ \sigma) = (id, \sigma)}\ (C6)$$

$$\frac{\begin{array}{c}\vdash_N clock(R_1) = \gamma_1 \\ \vdots \\ \vdash_N clock(R_n) = \gamma_n\end{array} \quad \vdash_N clock(\mathtt{A}) = \Gamma \quad \vdash_N clock(\mathtt{B}) = \Gamma}{\vdash_N clock(\mathtt{A} \rightarrow * \ (R_1, \ldots, R_n)\ \mathtt{B}) = \Gamma}\ (C7)$$

$$\frac{\begin{array}{c}\vdash_N clock(E_1) = (F, \mathscr{E}_1) \\ \vdots \\ \vdash_N clock(E_n) = (F, \mathscr{E}_n)\end{array} \quad \mathscr{E} \in D \quad |\mathscr{E}| = n \quad \vdash_N type(\mathtt{F}) = \mathscr{E} \quad \vdash_N clock(\mathtt{F}) = \gamma}{\vdash_N clock(\mathtt{merge}(\mathtt{F}, E_1, \ldots, E_n)) = \gamma}\ (C8)$$

FIGURE 2.12: Règles du calcul d'horloges

sorte à former des jugements de la forme :

$$\vdash_N clock(\mathtt{E}) = H$$

Un tel jugement signifie : dans le nœud N, l'horloge de l'expression E est H. Nous détaillons ci-dessous les significations des différentes règles données dans la figure 2.12.

Valeurs littérales – (C1) Les valeurs littérales comme « 5 », « *true* » ou une autre valeur symbolique prédéfinie ont pour horloge celle de base.

Identifiants – (C2) Une expression de la forme « *id* » fait référence à une variable définie dans le nœud. L'horloge de l'expression est celle donnée dans la déclaration du nœud.

Compositions – (C3), (C4) et (C5) Les opérateurs cycle-à-cycle s'appliquent sur des flots ayant la même horloge, qui devient celle du flot résultant de l'opération.

Filtrage – (C6) Pour pouvoir filtrer une expression E selon un couple (id, σ), il est nécessaire que l'horloge associée au flot référencé par id soit la même que l'horloge de l'expression E. La règle présentée ici, comme les autres, est complétée par la règle de typage vue précédemment.

Réinitialisation – (C7) Les horloges des flots de réinitialisation sont quelconques, alors que les membres gauche et droit de la flèche doivent être synchronisés.

Fusion de flots – (C8) Une expression « `merge(F,E1,...,En)` » porte n sous-expressions, dont exactement une d'elle est évaluée, à chaque cycle où le flot de contrôle F admet une valeur. Ces expressions sont d'horloges complémentaires. En effet, chaque sous-expression E_i (avec $1 \leq i \leq n$) a pour horloge le couple $(F, \mathscr{E}_i)$. Elle partage le même flot support F que les horloges des autres expressions et a comme valeur de présence la valeur symbolique de rang i dans le type énuméré $\mathscr{E}$, le type de F. La syntaxe de l'opérateur `merge` est d'ailleurs la raison pour laquelle les types énumérés admettent un ordre. L'horloge de l'expression résultant d'un tel entrelacement de flots est la même que celle du flot de contrôle F.

2.2.3 Règles d'évaluation

Nous présentons la fonction d'évaluation à un cycle du langage $\mathcal{L}_+$. Celle-ci nous donne la valeur d'une expression de flot Lustre à un instant donné, sachant les valeurs successives prises par les flots depuis l'instant initial. L'évaluation au sein d'un nœud dépend ainsi d'un ensemble de valuations pour les entrées de ce nœud dans le temps. Cet ensemble de valeurs est appelé « grille de valeurs », et constitue le contexte d'une fonction d'évaluation.

Grille de valeurs Nous présentons tout d'abord la structure de la grille de valeurs. Soit $N = (D, I, L, O, A) \in \mathcal{L}_+$. Une valuation de flot à un cycle est un terme de la forme :

$$id\ [c] \leftarrow v$$

1. $c \geq 0$ est un numéro de cycle absolu, c'est-à-dire exprimé selon la cadence de l'horloge de base. Le cycle initial est le cycle 0.
2. L'identifiant id renvoie à une variable du nœud N :
   $\exists F = (id, t, c) \in I$ tel que $var_N(id) = F$
3. La valeur v appartient au type du flot F :
   $\vdash_N type(v) = t$

Une grille de valeurs d'entrée est alors un ensemble de telles valuations. On notera généralement Γ cet ensemble, et Γ_N l'ensemble des grilles que l'on peut construire étant donné un nœud $N \in \mathcal{L}_+$.

La grille peut-être vue comme une fonction (partielle) liant les couples formés d'identifiants de flots et de cycles à une valeur. En particulier, chaque couple de l'ensemble de départ n'admet pas plus d'une image, ce qui interdit d'avoir plusieurs valuations pour une même variable à un même cycle.

Dimension d'une grille de valeurs On appelle *longueur* d'une grille de valeurs Γ le plus grand cycle c tel qu'il existe une valuation au cycle c dans Γ ; cette taille est notée $\mathit{length}(\Gamma)$:

$$\mathit{length}(\Gamma) = max\{c \mid id\ [c] \leftarrow v \in \Gamma\}$$

L'ensemble des *variables* (de flots) d'une grille Γ est noté $\mathit{vars}(\Gamma)$ et contient l'ensemble des identifiants de flots qui admettent une valuation dans cette grille :

$$\mathit{vars}(\Gamma) = \{id \mid id\ [c] \leftarrow v \in \Gamma\}$$

Fonction d'évaluation Soient $N = (D, I, L, O, A) \in \mathcal{L}_+$ un modèle Lustre, $\Gamma \in \Gamma_N$ une grille de valeurs, $E \in \mathcal{E}xprs_N$ une expression de flot, $c \geq 0$ un cycle et $v \in \mathcal{V}alues_D$ une valeur littérale. Le jugement suivant exprime le fait que dans le nœud N, l'expression E s'évalue comme la valeur v au cycle c, étant donné les valuations présentes dans Γ :

$$\Gamma \vdash_N E =_c v$$

Nous donnons dans la suite les règles d'évaluations du langage, en supposant que les expressions à évaluer sont effectivement bien formées. Il n'est pas garanti que toute expression bien formée soit évaluable, puisque celle-ci dépend du contexte d'évaluation, c'est-à-dire de la grille de valeurs fournie pour les entrées du système. Cependant, il n'est pas nécessaire que toute variable d'entrée soit valuée à tous les cycles pour qu'une expression soit évaluable : en effet, il suffit que seulent les variables qui influencent le résultat de l'expression soient valuées.

Résultats indéterminés Le langage Lustre définit une valeur indéterminée « nil » pour les cas où une opération n'est pas totalement définie (division par zéro, décalage au cycle initial, fonctions externes partielles, …). La valeur nil, une fois produite, se propage aux autres valeurs à travers les opérateurs du langage.

Or, nous utilisons la fonction d'évaluation pour vérifier qu'une grille de valeurs donnée (générée) est bien correcte vis-à-vis de la sémantique Lustre. Pour cela, nous faisons en sorte que la fonction d'évaluation ne soit pas définie dans les cas où un calcul repose sur des résultats intermédiaires indéfinis. Si, par exemple, la grille de valeurs est telle qu'on puisse observer une division par zéro à un cycle, alors l'évaluation dans le contexte de cette grille n'est pas définie pour toutes les expressions et tous les cycles possibles.

Cela nous permet de savoir que si on arrive à évaluer l'objectif de test d'un PGST au cycle 0, alors la valeur obtenue ne dépend que d'opérations intermédiaires totalement définies : la grille ne repose donc pas sur une interprétation particulière de la valeur NIL.

Plan Nous introduisons les notations en présentant l'évaluation des expressions littérales (§2.2.3.1). Nous voyons ensuite comment les identifiants de flots sont évalués, en faisant la distinction entre les variables calculées et les varibles d'entrées (§2.2.3.3), et en introduisant notamment la notion d'horloge présente à un cycle. Dans la section 2.2.3.2, nous nous intéressons aux opérateurs cycle-à-cycle dont l'évaluation est indépendante de l'horloge des opérandes. Enfin, à partir de la section 2.2.3.7, nous traitons les opérateurs temporels plus ou moins complexes.

2.2.3.1 Expressions littérales

$$\frac{c \in \mathbb{N} \qquad v \in \mathcal{V}alues_D}{\Gamma \vdash_N v =_c v} \text{(EVAL1)}$$

le règle (EVAL1) se lit ainsi : la valeur de l'expression de flot v prend la valeur v à tout cycle c positif ou nul. Les valeurs littérales du langage s'évaluent donc de manière immédiate.

2.2.3.2 Horloges

Nous définissons un prédicat auxiliaire « *present* » à l'aide des règles (EVAL2) et (EVAL3) suivantes. Ce prédicat indique si une horloge est présente à un cycle, et nous servira par la suite pour simplifier les notations.

$$\frac{c \in \mathbb{N}}{\Gamma \vdash_N base : present_c} \text{(EVAL2)} \qquad \frac{\Gamma \vdash_N id =_c v}{\Gamma \vdash_N (id, v) : present_c} \text{(EVAL3)}$$

Il s'agit d'un raccourci signifiant, d'une part, que l'horloge de base est toujours présente, et d'autre part qu'une horloge relative est présente si et seulement si son flot support s'évalue à sa valeur de présence au cycle considéré.

Par l'intermédiaire des règles suivantes d'évaluation des variables LUSTRE, nous verrons que la définition du prédicat *present* est récursive : pour que le flot support s'évalue à la valeur de présence, ce flot support doit lui aussi être évaluable, ce qui n'est le cas que si sa propre horloge est présente.

2.2.3.3 Identifiants de variables

Lorsqu'une expression fait référence à une variable, deux cas peuvent se présenter :

- La variable admet une expression de définition :
  son évaluation correspond à celle de cette expression.
- La variable est une variable d'entrée :
  l'expression n'est évaluable que si cette variable admet une valuation dans la grille de valeurs au cycle considéré.

Dans tous les cas, la règle d'évaluation des variables vérifie que l'horloge de la variable est bien présente au cycle de l'évaluation. Cette vérification fait le lien avec la définition précédente du prédicat *present*, qui repose sur le fait que l'évaluation d'une variable à un cycle n'est possible que si son horloge est présente.

Flots calculés

$$\frac{\begin{array}{c} V = var_N(id) \qquad \Gamma \vdash_{\mathrm{N}} h(V) : present_c \\ def(V) = E \qquad \Gamma \vdash_{\mathrm{N}} \mathtt{E} =_c e \end{array}}{\Gamma \vdash_{\mathrm{N}} id =_c e} \text{(EVAL4)}$$

Flots en entrée

$$\frac{V = var_N(id) \qquad \Gamma \vdash_{\mathrm{N}} h(V) : present_c \qquad V \in I}{\Gamma \cup \{id\ \ [c] \leftarrow v\} \vdash_{\mathrm{N}} id =_c v} \text{(EVAL5)}$$

Dans la règle (EVAL5) précédente, on rappelle que l'ensemble I est l'ensemble des variables d'entrées du nœud N.

2.2.3.4 Opérations combinatoires

Nous donnons les règles de manière générale pour les opérateurs unaires et binaires, et détaillons celles de l'opérateur de branchement conditionnel `if..then..else`.

Opérations arithmétiques et logiques

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathrm{N}} \mathtt{E} =_c e \\ \mathscr{U} \in \{-, not\} \end{array}}{\Gamma \vdash_{\mathrm{N}} \mathscr{U}\ \mathtt{E} =_c (\mathscr{U}\ e)} \text{(EVAL6)} \qquad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{N}} \mathtt{A} =_c \alpha \qquad \Gamma \vdash_{\mathrm{N}} \mathtt{B} =_c \beta \\ \mathscr{R} \in \{<, >, \leq, \geq, =, \neq, +, *, div, mod\} \end{array}}{\Gamma \vdash_{\mathrm{N}} \mathtt{A}\ \mathscr{R}\ \mathtt{B} =_c (\alpha\ \mathscr{R}\ \beta)} \text{(EVAL7)}$$

Dans les règles (EVAL6) et (EVAL7), les lettres $\mathscr{U}$ et $\mathscr{R}$ représentent les relations appliquées à la fois aux expressions, mais aussi aux valeurs retournées par les sous-évaluations. Ainsi, les opérations sur les flots se concrétisent en calculs sur les valeurs ponctuelles de ces flots, selon les règles de calcul usuelles. Par exemple, l'expression Lustre « `A + B` » s'évalue à un cycle c comme la somme des évaluations respectives de `A` et `B` au cycle c. De même, l'expression « `A > B` » retourne une valeur booléenne propre aux valeurs entières prises respectivement par les sous-termes `A` et `B` au cycle de l'évaluation.

Les opérateurs `div` et `mod` donnent respectivement le quotient et le reste de la division euclidienne du premier argument par le deuxième. Quand celui-ci vaut zéro, cette division n'est pas définie.

Opérateur conditionnel

$$\frac{\Gamma \vdash_N \texttt{I} =_c \mathit{true} \quad \Gamma \vdash_N \texttt{T} =_c t \quad \Gamma \vdash_N \texttt{E} =_c e}{\Gamma \vdash_N \texttt{if I then T else E} =_c t}\ (\text{EVAL8}) \qquad \frac{\Gamma \vdash_N \texttt{I} =_c \mathit{false} \quad \Gamma \vdash_N \texttt{T} =_c t \quad \Gamma \vdash_N \texttt{E} =_c e}{\Gamma \vdash_N \texttt{if I then T else E} =_c t}\ (\text{EVAL9})$$

Les règles d'évaluation (EVAL8) et (EVAL9) sont celles exprimant la sémantique de l'opérateur conditionnel `if..then..else`. L'opérateur ne court-circuite pas les évaluations des sous-termes : quelle que soit la valeur de la condition, les sous-termes `T` et `E` sont évalués, ce qui correspond à la sémantique originale de Lustre, une sémantique stricte, qui impose que l'opérateur évalue systématiquement les deux branches possibles. C'est seulement ensuite, en fonction de la valeur courante de la condition, qu'il redirige le calcul approprié vers sa propre sortie. On ne peut pas empêcher l'éxécution d'un bloc d'équations en fonction d'une condition, comme par exemple une division par zéro :

```
(* Incorrect : division lorsque x = 0 *)
y = if (x = 0) then -1 else (1/x) ;

(* Correct: le membre droit n'est jamais nul *)
y = 1 / (if (x = 0) then -1 else x) ;
```

Dans l'exemple précédent, on suppose que l'on veuille retourner -1 lorsque `x` vaut zéro. Or, seule la deuxième formulation est évaluable, car elle empêche d'avoir un valeur nulle dans le rôle de diviseur.

Un des avantages des horloges est de pouvoir exprimer des branchements conditionnels tels qu'on les conçoit naturellement dans les autres langages de programmation, c'est-à-dire selon une sémantique paresseuse : la condition est d'abord évaluée, puis seul la branche correspondant à la valeur de la condition est calculée (l'autre branche est inactive). Nous verrons comment l'exemple précédent peut-être réalisé à l'aide d'un filtrage et d'un « `merge` » dans les sections suivantes.

2.2.3.5 Filtrage

L'opération de filtrage donnée par l'opérateur « `when` » admet plusieurs syntaxes possibles, une qui est le cas général avec un filtrage selon un type énuméré, et deux autres qui acceptent un filtrage selon une variable booléene ou sa négation. Ces deux cas particuliers sont traités en faisant appel à la règle générale :

$$\frac{\Gamma \vdash_N \texttt{A when B match } \mathit{true} =_c \alpha}{\Gamma \vdash_N \texttt{A when B} =_c \alpha}\ (\text{EVAL10}) \qquad \frac{\Gamma \vdash_N \texttt{A when B match } \mathit{false} =_c \alpha}{\Gamma \vdash_N \texttt{A when not(B)} =_c \alpha}\ (\text{EVAL11})$$

Dans le cas général, l'expression filtrée admet une valeur seulement aux cycles où l'horloge du filtrage est présente.

$$\frac{\Gamma \vdash_N (f, v) : \mathit{present}_c \qquad \Gamma \vdash_N \texttt{A} =_c \alpha}{\Gamma \vdash_N \texttt{A when f match v} =_c \alpha}\ (\text{EVAL12})$$

Cet opérateur est illustré dans le prochain exemple.

2.2.3.6 Entrelacement de flots

Nous définissons l'évaluation de l'opérateur `merge` de la manière suivante :

$$\frac{1 \leq j \leq n \qquad \vdash_N \mathit{type}(\texttt{F}) = \mathscr{E} \qquad \Gamma \vdash_N \texttt{F} =_c \mathscr{E}_j \qquad \Gamma \vdash_N \texttt{E}_j =_c e}{\Gamma \vdash_N \texttt{merge(F,E}_1\texttt{,...,E}_n\texttt{)} =_c e}\ (\text{EVAL13})$$

À chaque cycle c, il existe un unique indice j tel que `F` prenne la valeur symbolique $\mathscr{E}_j$ définie dans le type énuméré $\mathscr{E}$ de `F`. Alors, l'expression E_j dans l'opérateur `merge` est celle qu'il faut évaluer à ce cycle pour connaître la valeur e produite par l'opérateur.

Exemple Nous reprenons l'exemple de la division en exploitant cette fois les horloges : la division `y/x` n'est pas réalisée aux cycles où `x` est nul. Pour cela, le flot booléen `zero` nous sert d'horloge. Il vaut `true` uniquement aux cycles où `x` est nul. Les différentes équations sont les suivantes :

```
zero = (x = 0);
invx = (1 when not zero) / (x when not zero);
y    = merge(zero; (-1 when zero); invx);
```

Dans cet exemple, seule une sous-expression est évaluée suivant la valeur courante du flot booléen `zero` :

- Si `zero = true`, la valeur courante de `y` est donnée, d'après l'opérateur `merge`, par le terme `-1 when zero`, actif au cycle courant.
- Si `zero = false`, la valeur de `y` est cette fois donnée par la variable intermédiaire `invx` (l'inverse de `x`), qui admet bien une valeur selon les règles de calcul des horloges.

L'opérateur `merge` recompose ainsi les flots d'horloges complémentaires pour calculer finalement les valeurs de `y` à tous les cycles. Les différents opérateurs `when` construisent les expressions temporisées.

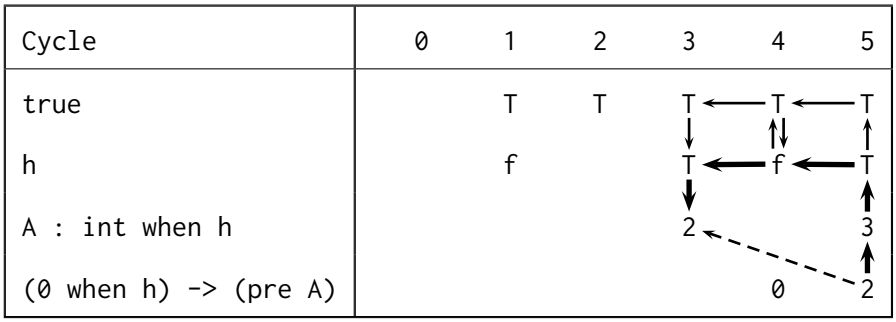


FIGURE 2.13: Exemple de l'évaluation de l'opérateur pre.

2.2.3.7 Décalage temporel

L'évaluation de l'opérateur de décalage temporel « pre » est définie en parcourant à la fois les cycles de calculs, mais aussi la relation de parenté entre horloges. Nous illustrons ceci dans la figure 2.13.

Exemple Nous définissons un flot entier A, dont l'horloge est $(h, true)$, avec h un flot booléen sur l'horloge de base. La dernière ligne de la trace d'éxécution représente les valeurs de l'expression suivante :

```
(0 when h) → pre(A)
```

Cette expression réalise un simple décalage temporel du flot A, avec une valeur intiale de zéro. Le ralentissemnt de la constante par l'horloge h , *true* est nécessaire pour satisfaire les règles de bonne formation vis-à-vis des horloges. La première ligne du même tableau représente la constante *true* : elle nous permet de représenter les instants de présence de l'horloge de base du système.

Les flèches représentent les itérations nécessaires pour évaluer « pre(A) » au cycle 5. Pour cela, on recherche le dernier instant où A admettait une valeur avant le cycle 5. Ce cycle correspond au dernier instant où l'horloge de A était elle-même présente, c'est-à-dire au dernier cycle où le flot h valait *true*. Cette recherche du cycle précédent consiste à itérer sur les valeurs précédentes de l'horloge h, jusqu'à trouver la correspondance h = *true*. Or, l'itération sur les valeurs précédentes nécessite de calculer aussi pre(h) à différents cycles. C'est pourquoi l'évaluation de l'opérateur pre est récursive. On voit ainsi que pour itérer sur les valeurs de h, on doit rechercher le dernier cycle où *l'horloge de* h était présente, ce qui dans notre cas est facile, puisque l'horloge de base est toujours présente.

Règle d'évaluation La règle d'évaluation de l'opérateur pre fait appel à un nouveau prédicat auxiliaire intitulé *precycle* : ce prédicat accepte une horloge H et un cycle c, et nous donne le plus grand numéro de cycle $\gamma < c$ tel que l'horloge H soit présente. Alors,

la règle (EVAL14) est ainsi définie :

$$\frac{\vdash_N \mathit{clock}(\mathtt{E}) = H \qquad \Gamma \vdash_N \mathit{precycle}_c(H) = \gamma \qquad \Gamma \vdash_N \mathtt{E} =_\gamma e}{\Gamma \vdash_N \mathtt{pre(E)} =_c e} \text{(EVAL14)}$$

Elle signifie que :

- si l'horloge de E est H, et
- si le cycle précédent de présence de H est γ (depuis c),
- alors l'évaluation de pre(E) au cycle c équivaut celle de E au cycle γ.

Nous donnons maintenant les règles de calcul pour le prédicat *precycle*.

Horloge de base Le cycle précédent se calcule simplement dans le cas de l'horloge de base :

$$\frac{c \in \mathbb{N} \qquad c > 0}{\Gamma \vdash_N \mathit{precycle}_c(\mathit{base}) = c - 1} \text{(EVAL15)}$$

Le cycle qui est directement précédent, à savoir $c - 1$, est la valeur recherchée. Il est nécessaire que le cycle c ne soit pas nul pour que la règle soit applicable.

Horloge relative Dans le cas général, le calcul du cycle précédent de (f, v) consiste à itérer sur les instants de présence précédents de l'horloge de f, jusqu'à ce que l'on atteigne un cycle où f s'évalue à v.

$$\frac{\vdash_N \mathit{clock}(f) = H \qquad \Gamma \vdash_N \mathit{precycle}_c(H) = \gamma \qquad \Gamma \vdash_N f =_\gamma v}{\Gamma \vdash_N \mathit{precycle}_c((f, v)) = \gamma} \text{(EVAL16)}$$

$$\frac{\vdash_N \mathit{clock}(f) = H \qquad \Gamma \vdash_N \mathit{precycle}_c(H) = \gamma \qquad \Gamma \vdash_N f =_\gamma w \qquad v \neq w \qquad \Gamma \vdash_N \mathit{precycle}_\gamma((f, v)) = \delta}{\Gamma \vdash_N \mathit{precycle}_c((f, v)) = \delta} \text{(EVAL17)}$$

1. Dans les deux règles, on fait appel à *precycle* pour trouver le cycle précédent de H, l'horloge mère de (f, v). Ce cycle est noté γ.
2. La différence entre les deux règles réside dans le fait que f s'évalue ou non à la valeur de présence v au cycle γ.
   - (EVAL16) Lorsque c'est le cas, l'itération s'arrête avec γ le cycle recherché.
   - (EVAL17) Sinon, on continue la recherche, en repartant cette fois de γ :

     $$\Gamma \vdash_N \mathit{precycle}_\gamma((f, v)) = \delta$$

     Cet appel récursif donne finalement la valeur δ, qui est le cycle recherché.

2.2.3.8 Opérateur de réinitialisation

On rappelle que la flèche classique de Lustre, à savoir l'opérateur « → », est un cas particulier de l'opérateur de réinitialisation où il n'existe pas de flot de redémarrage. On définit donc simplement l'évaluation de la flèche classique à partir de la flèche étendue :

$$\frac{\Gamma \vdash_{\mathrm{N}} \mathtt{A} \rightarrow * \ () \ \mathtt{B} =_c v}{\Gamma \vdash_{\mathrm{N}} \mathtt{A} \ \rightarrow \ \mathtt{B} =_c v} \text{(EVAL18)}$$

Finalement, nous définissons le cas général, c'est-à-dire l'opérateur de réinitialisation avec un ensemble fini quelconque de flots de redémarrage. Son évaluation a besoin de déterminer si :

1. le cycle de l'évaluation est le cycle initial du flot
2. un des flots de réinitialisation s'évalue à *true* entre le cycle courant et le cycle précédent de l'expression

Cycle initial Nous rappelons que les flots qui partagent la même horloge partagent aussi le même cycle initial : celui-ci est l'instant où l'horloge est présente pour la première fois. Nous définissons le prédicat $H : initial_c$ dans le système de règles. Il est vrai si et seulement si l'horloge H est présente pour la première fois au cycle c :

- Pour l'horloge de base, il s'agit du cycle zéro.
- Pour une horloge relative $H = (f, v)$, le prédicat est vrai au cycle c si :
  1. le flot support f vaut la valeur de présence v, et
  2. à tout cycle $d < c$, le flot support f ne vaut pas v (son horloge est absente, ou sa valeur est différente de v).

Ces conditions sont vérifiées par les règles suivantes :

$$\frac{}{\Gamma \vdash_{\mathrm{N}} base : initial_0} \text{(EVAL19)}$$

$$\frac{\begin{array}{c}\vdash clock(\mathtt{f}) = H \qquad \vdash \Gamma \vdash_{\mathrm{N}} f =_c v \\ (\forall d < c)(\Gamma \vdash_{\mathrm{N}} H : present_d \Rightarrow \Gamma \vdash_{\mathrm{N}} \mathtt{f} =_d w; w \neq v)\end{array}}{\Gamma \vdash_{\mathrm{N}} (f, v) : initial_c} \text{(EVAL20)}$$

Règles d'évaluation Nous définissons maintenant l'évaluation de l'opérateur de réinitialisation pour toute expression E portant un ensemble R quelconque de flots booléens :

$$E = \mathtt{A} \rightarrow * \ (R) \ \mathtt{B}$$

Si l'horloge du flot E, qui est la même que ses deux opérandes principales, est présente pour la première fois au cycle de l'évaluation, il s'agit du cycle initial, et dans ce cas on évalue selon le membre gauche.

$$\frac{\vdash_N clock(\mathtt{A}) = H \qquad \vdash_N clock(\mathtt{B}) = H \qquad \Gamma \vdash_N H : initial_c \qquad \Gamma \vdash_N \mathtt{A} =_c \alpha}{\Gamma \vdash_N \mathtt{A} \rightarrow * \ (R)\ \mathtt{B} =_c \alpha} \text{(EVAL21)}$$

Sinon, il existe un cycle précédent pour cette horloge. Entre le cycle précédent et le cycle courant, il est possible qu'un des flot de réinitialisation vaille *true*. Dans ce cas, l'évaluation de l'expression au cycle courant est celle du membre gauche de la flèche. La règle suivante recherche un cycle δ dans la fenêtre temporelle $\gamma < \delta \leq c$ tel qu'au moins un des flots de redémarrage s'y évalue comme *true*.

$$\frac{\vdash_N clock(\mathtt{A}) = H \qquad R = R_1, \ldots, R_n \qquad n \geq 0 \qquad 0 < i \leq n \qquad \Gamma \vdash_N precycle_c(H) = \gamma \qquad \gamma < \delta \leq c \qquad \Gamma \vdash_N R_i =_\delta true \qquad \Gamma \vdash_N \mathtt{A} =_c \alpha}{\Gamma \vdash_N \mathtt{A} \rightarrow * \ (R)\ \mathtt{B} =_c \alpha} \text{(EVAL22)}$$

Sinon, tous les flots de réinitialisation, lorsqu'ils sont présents dans la fenêtre de temps considérés, valent *false*. Dans ce cas, l'évaluation de la flèche se fait selon son membre droit.

$$\frac{\begin{array}{c}\vdash_N clock(\mathtt{A}) = H \qquad n \geq 0 \qquad R = R_1, \ldots, R_n \\ \Gamma \vdash_N precycle_c(H) = \gamma \qquad \Gamma \vdash_N \mathtt{B} =_c \beta \\ (\forall \delta \mid \gamma < \delta \leq c)(\forall i \mid 0 < i \leq n) : \\ \big((\vdash_N clock(R_i) = H_i \wedge \Gamma \vdash_N H_i : present_\delta) \Rightarrow (\Gamma \vdash_N R_i =_\delta false)\big)\end{array}}{\Gamma \vdash_N \mathtt{A} \rightarrow * \ (R)\ \mathtt{B} =_c \beta} \text{(EVAL23)}$$

2.2.3.9 Grilles de valeurs correctes par évaluation

Soit $N = (D, I, L, O, A) \in \mathcal{L}_+$. On appelle Γ_N^* l'ensemble des grilles de valeurs qui sont évaluables, selon les règles précédentes, pour le nœud N. Soit $\Gamma \in \Gamma_N$ une grille de taille $c \in \mathbb{N}$. Alors, par définition, $\Gamma \in \Gamma_N^*$ si on a :

1. Toutes les variables calculées du système admettent une valeur dès que leur horloge est présente. Quelque soient $V \in \mathcal{V}_N$ et $d \in 0..c$, on a :

$$(\vdash_N clock(V) = H \wedge \Gamma \vdash_N H : present_c) \Rightarrow \exists x \in \mathcal{V}alues_D,\ \Gamma \vdash_N V =_d x$$

2. Toutes les assertions sont vérifiés lorque leur horloges sont présentes. Quelque soient $a \in A$ et $d \in 0..c$, on a :

$$(\vdash_N clock(a) = H \wedge \Gamma \vdash_N H : present_c) \Rightarrow \Gamma \vdash_N a =_d true$$

2.2.3.10 Sous-langage mono-horloge

Nous définissons ici un sous-ensemble de $\mathcal{L}_+$ qui correspond à une vision simplifiée du langage Lustre sans horloges, types énumérés ou appels de nœuds. Soit $N = (D, I, L, O, A)$ un nœud appartenant à $\mathcal{L}_+$. Alors, par définition, $N \in \mathcal{L}_1$ si et seulement si :

1. D ne définit que le type booléen :

$$D = \{\mathcal{B}ool\}$$

2. L'ensemble des horloges valides dans le nœud est restreint à l'horloge de base :

$$\mathcal{C}locks_N = \{base\}$$

Ainsi, les seules expressions valides du langage sont celles dont l'horloge est l'horloge de base.

2.3 Conclusion

Dans ce chapitre, nous avons rappelé l'approche synchrone Lustre pour la modélisation de systèmes réactifs. Nous avons ensuite défini le langage $\mathcal{L}_+$, qui représente un dialècte de Lustre multi-horloges muni d'extensions intéressantes. Nous avons par ailleurs défini une sémantique d'évaluation pour les modèles du langage, qui nous servira de référence pour les prochains chapitres.

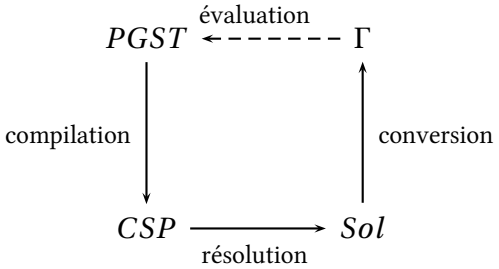
Chapitre 3
GATeL

Dans ce chapitre, nous présentons une méthode existante de génération de tests à partir de systèmes réactifs synchrones décrits dans le langage Lustre, vu au chapitre 2.

La fonction d'évaluation vue précédemment pour le langage $\mathcal{L}_+$ repose sur un ensemble de valuations des flots d'un système dans le temps, que nous avons appelé grille de valeurs. Nous souhaitons générer automatiquement de telles grilles de valeurs étant donné un modèle sous test et un *objectif de test*.

Cette méthode de génération est basée sur la programmation logique par contraintes, que nous avons présenté au chapitre 1. La méthode est mise en œuvre dans l'outil GATeL. Nous donnons ici une formulation nouvelle qui tente de présenter de manière homogène et détaillée les travaux passés autour de GATeL.

Nous nous restreignons ici au sous-ensemble mono-horloge du langage, qui correspond à l'état de GATeL avant nos travaux. Afin de distinguer les versions du logiciel, nous notons « GATeL$_1$ » l'ancienne version, mono-horloge, de GATeL, et « GATeL$_+$ » la nouvelle, qui en prend compte le langage Lustre temporisé $\mathcal{L}_+$. La démarche de GATeL à haut-niveau est représentée par la figure suivante :



Un problème de génération de séquences de tests (PGST), comme nous le détaillons par la suite, est composé d'un modèle Lustre et d'un objectif de test (cf. §3.2). Dans un premier temps, ce PGST est transformé (compilé) en un problème de satisfaction de contraintes (CSP), dédié au problème. En utilisant des techniques de résolution de contraintes, nous obtenons si possible une solution au CSP, à partir de laquelle on peut extraire une grille de valeurs Γ. Celle-ci est censée être évaluable et être une solution

du PGST original, c'est-à-dire qu'elle représente un fonctionnement du système sous test qui réalise l'objectif de test. Or, la méthode de génération peut parfois produire des faux-positifs, c'est-à-dire des séquences qui ne sont pas réellement correctes vis-à-vis de l'évaluation. Cela est dû à un compromis entre l'efficacité de la génération de séquences et sa correction. La fonction d'évaluation que nous avons présenté au chapitre 2 est alors utilisée systématiquement en fin de génération pour tester si la séquence construite est bien une solution ou s'il y a une erreur d'évaluation. Ce filtre permet de donner finalement uniquement les séquences correctes vis-à-vis de l'évaluation.

Nous montrons dans une première section comment le problème de la génération de séquences de test est modélisé sous forme d'un CSP (§3.2) pour le langage Lustre mono-horloge $\mathcal{L}_1$. Nous décrivons ainsi l'approche existante, à savoir GATeL$_1$, à travers une présentation homogène des règles d'évolution du système de contraintes qui permettent de produire des séquences de tests compatibles avec le problème initial (§3.3). Bien que GATeL s'attaque à des problèmes de génération de tests en présence de valeurs flottantes et de contraintes arithmétiques, nous ne détaillons pas ces aspects ici mais nous concentrons sur les contraintes logiques et temporelles.

3.1 Problèmes de génération de séquences de tests

Nous définissons ce qu'est un problème de génération de séquences de test dans le cadre de GATeL, comment celui-ci est fourni à l'outil ainsi que la forme attendue d'une solution.

3.1.1 Environnement de test

Nous souhaitons tester le comportement de systèmes réactifs au cours de certains scénarios. Or, la nature même de ces systèmes fait que les actions à un cycle peuvent se répercuter sur les entrées prochaines du système : il s'agit de la réponse de l'environnement face aux actions du contrôleur.

Exemple le nœud « `deplace` » de la figure 3.1 prend deux entrées entières : « pos », la position courante d'un robot selon un axe et « `cible` », une consigne, c'est-à-dire une position que l'on souhaite atteindre avec ce robot. Il fournit à chaque cycle une commande, appelée « `mvt` », de type `mouvement`. notre cas, le flot `mvt` peut valoir `recule`, `stop` ou `avance`, et on suppose que cette commande suffit à faire bouger le robot sur son axe, de sorte qu'au cycle suivant, la position soit modifiée (respectivement de -1, 0 ou 1).

Ainsi, à chaque cycle d'éxecution, la consigne et la position courante permettent de déterminer l'action à effectuer ; celle-ci doit normalement avoir un impact sur l'environnement extérieur (le robot se déplace), et donc aussi sur les futures entrées du système (sa position change).

```
type mouvement = enum{recule, stop, avance};

node deplace(pos, cible: int)
returns (mvt: mouvement)
let
  mvt = if (cible = pos)
        then stop
        else if (cible > pos)
             then avance
             else recule ;
tel
```

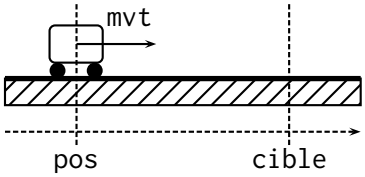


FIGURE 3.1: Contrôle du déplacement du robot sur un axe

Pour tester le contrôleur du robot, on doit pouvoir simuler son éxecution, ce qui nécessite de rendre explicite le changement de position qui s'effectue en fonction de la commande `mvt` courante. Ainsi, on se donne un modèle de l'environnement dans lequel `pos` vaut sa valeur précédente plus -1, 0 ou 1, selon la valeur de `mvt` au cycle précédent ; de plus, on fixe la position initiale du robot. En couplant cet environnement avec le système réactif, on peut simuler son comportement et avoir des traces d'éxecution dans le temps. La figure 3.2 correspond à l'exemple complet pour ce robot.

Le nœud Lustre « `simulation` » de la figure 3.2 prend en entrée la valeur de la consigne (la position cible) et retourne la position courante du robot ainsi que sa commande de déplacement (la variable `mvt` de type `mouvement`). Les équations sont séparées par des commentaires [1] en deux groupes, la partie correspondant au calcul effectué par le contrôleur et celle où l'environnement est simulé. La première équation, celle associée à `mvt`, est la même que celle du nœud précédent « `deplace` ». Dans la simulation de l'environnement, on suppose que : (i) le robot part depuis la position 10, et (ii) la commande de déplacement au cycle courant donne un changement de position de -1, 0 ou 1 au cycle suivant. Ainsi, la position courante est modifiée à chaque cycle, par rapport à la position et à la commande `mvt` du cycle précédent.

La figure 3.3 est un exemple d'éxecution du nœud, avec des valeurs arbitraires pour le flot `cible` en entrée. On constate que la position change progressivement, en respectant à la fois le contrôleur et le modèle de l'environnement.

Spécifications LUSTRE Le langage LUSTRE nous permet de représenter à la fois un contrôleur et son environnement. Un problème de génération de séquences de tests est

1. La syntaxe de LUSTRE autorise l'ajout de commentaires au sein des modèles, délimités par les caractères `(*` et `*)`. Un commentaire permet d'ajouter des informations qui sont ignorées par les outils, mais utiles à la compréhension du modèle. De même, tout texte qui suit les caractères « `--` » est ignoré sur la ligne courante.

```
node simulation(cible: int)
returns (pos: int; mvt: mouvement)
var
    vit: int; -- vitesse
let

    (* CONTROLEUR *)

    mvt = if (cible = pos)
          then stop
          else if (cible > pos)
          then avance
               else recule ;

    (* SIMULATION DE L'ENVIRONNEMENT *)

    vit = if (mvt = stop)
          then 0
          else if (mvt = avance)
               then 1
               else -1 ;

    pos = (10 → ( pre(pos) + pre(vit) );
tel
```

FIGURE 3.2: Simulation du contrôleur de déplacement.

cible	13	13	13	13	214	-241	-477	15
pos	10	11	12	13	13	14	13	12
mvt	avance	avance	avance	stop	avance	recule	recule	avance

FIGURE 3.3: Simulation du nœud `robot`.

donc donné par une spécification Lustre, dont les équations et les invariants peuvent jouer des rôles différents. La simulation est d'autant plus précise que le modèle de l'environnement est proche de la réalité.

3.1.2 Objectif de test

Un problème de génération de séquences de tests (PGST) est donné par une spécification Lustre et un objectif de test. L'objectif définit un cas de test, à savoir un comportement observable du système que l'on cherche à atteindre. Dans cette méthode orienté but *(goal-oriented)*, on cherche à savoir s'il existe une successions d'actions (une séquence d'entrées) capable de réaliser l'objectif. Le système doit pour cela passer par les étapes intermédiaires nécessaires, ce qui nécessite des séquences plus ou moins longues selon les cas : la longueur des séquences est une variable du problème. Dans GATeL, un objectif de test est donné par n'importe quelle expression Lustre qui est évaluable dans le contexte du nœud sous-test, de type booléen. C'est une expression arbitraire : elle peut faire appel à des variables intermédaires, des opérateurs temporels, des calculs, etc.

3.1.3 Annotation des modèles sous test

En plus d'un objectif de test, un PGST peut dépendre de paramètres utilisateurs qui changent les modalités de la génération de test. Par exemple, GATeL peut accepter un paramètre lui indiquant la taille maximale des séquences qu'on lui autorise à générer, ce qui, selon les cas, peut changer le nombre de solutions qui existent. En effet, il est tout à fait possible qu'un objectif de tests nécessite un nombre minimum de cycles d'éxecution du système avant d'être atteint, et suivant qu'on autorise ou non GATeL a créer autant de cycles, celui-ci peut ou non trouver la séquence solution. Par ailleurs, nous verrons par la suite que GATeL admet une directive de sélection de cas de tests à base d'expressions qui raffinent l'objectif de test.

3.1.3.1 Langages étendus

L'objectif de test, ainsi que les paramètres d'éxecution ou les directives de sélection, s'ajoutent à la spécification Lustre. En pratique, GATeL admet une syntaxe enrichie qui permet d'intégrer ces paramètres à une spécification existante. Le modèle devient alors la formalisation d'un PGST. Les directives propres à l'outil sont cependant cachées au sein de commentaires Lustre, ce qui permet à une spécification d'être à la fois interprétable comme modèle éxecutable et comme problème de génération de séquences dans GATeL.

Nous définissons des extensions des langages $\mathcal{L}_1$ et $\mathcal{L}_+$ définis au chapitre 2 pour y intégrer des constructions compréhensibles par GATeL. Les langages ainsi obtenus sont respectivement $\mathcal{L}_1^g$ et $\mathcal{L}_+^g$, où l'exposant dénote le fait que les langages sont interprétables par notre outil. La syntaxe étendue de la figure 3.4 modifie les langages $\mathcal{L}_1$ et $\mathcal{L}_+$ de sorte à autoriser l'ajout d'annotations dans une spécification Lustre mono ou multi-horloges.

Nous modifions la définition d'un nœud pour permettre l'ajout d'annotations en fin de nœud. Cela ne change pas la sémantique d'évaluation d'un modèle.

Node ::= `node` **ID**
`(` *Var* { `;` *Var* } `)`
`returns (` *Var* { `;` *Var* } `);`
[[`var` *Var* { `;` *Var* }]]
`let`
{ *BodyElt* } — Corps du nœud
Annotations — Annotations propres à GATeL
`tel ;`

Ces annotations comportent un objectif de test, des paramètres optionnels et une ou plusieurs directives `split` que l'on détaille par la suite.

Annotations ::= *Objective Params* { *Split* }

Un objectif de test est donné par une clause `reach`, où l'argument est une expression booléenne.

Objective ::= `reach` *Expr* `;`

Les paramètres facultatifs sont plus nombreux en pratique ; nous n'en décrivons qu'un seul dans ce document, qui est la taille maximale des séquences de tests que l'on s'autorise à générer pour le modèle.

Params ::= `maxsize` **INT**
| ϵ

Un découpage fonctionnel est contrôlé par une variable de flot et peut donner lieu à l'introduction de plusieurs sous-objectifs de test.

Split ::= `split` **ID** `with [` *Expr* { `;` *Expr* } `]`

FIGURE 3.4: Modifications de la syntaxe du langage pour y intégrer des annotations nécessaires à la description complète d'un problème de génération de séquences.

3.1.3.2 Applications auxiliaires

Soit N un nœud de $\mathcal{L}_1^g$ (resp. $\mathcal{L}_+^g$). Nous définissons les applications auxiliaires suivantes.

Objectif de test On note $obj(N)$ l'objectif de test associé au nœud N. C'est un flot de données du langage $\mathcal{L}_1$ (resp. $\mathcal{L}_+$) qui respecte les propriétés suivantes :

- $\vdash_N type(obj(N)) = \mathcal{B}ool$
  L'expression est un flot booléen.
- $\vdash_N clock(obj(N)) = H, H \in \mathcal{C}locks_N$
  L'horloge de l'objectif de test est bien définie dans le nœud. En particulier, si $N \in \mathcal{L}_1^g$, on a nécessairement $H = base$.

Longueur de cycle On note $maxsize(N)$ la taille maximale de la séquence générée. Cette borne nous permet de rendre l'espace de recherche fini lors de la génération de séquences de tests.

Directive d'insertion de sous-objectif On note $\mathcal{S}plit(N)$ l'ensemble des termes de la forme *split(Id,S)* défini par le modèle, où Id est un identifiant de flot référençant une variable de $\mathcal{V}_N$ et où S est un ensemble d'expressions qui vérifient les propriétés suivantes ; soit $E \in S$, on a :

- $\vdash_N type(\mathsf{E}) = \mathcal{B}ool$
- $\vdash_N clock(\mathsf{E}) = H_E, H_E \in \mathcal{C}locks_N$

De même que précédemment, l'horloge de l'expression E est l'horloge de base dans le cas de $\mathcal{L}_1^g$. Une directive `split` sert à introduire des découpages fonctionnels en ajoutant à un problème original des objectifs de tests supplémentaires. Nous détaillons la sémantique particulière de cette directive par la suite (§3.3.5.3).

3.1.4 Grille de valeurs solution

Soient $N = (D, I, L, O, A) \in \mathcal{L}_1^g$ et $\Gamma \in \Gamma_N^*$ une grille évaluable de valeurs[2] de longueur $c \in \mathbb{N}$ tels que l'objectif de test *Obj* s'évalue à *true* au cycle c :

$$\Gamma \vdash_N Obj =_c true$$

Alors, Γ est une solution du problème de génération de séquences de tests N, car elle permet d'atteindre un cycle c auquel l'objectif de test est observé, tout en respectant la sémantique du programme.

2. voir §2.2.3.9

Nous cherchons à générer des grilles de valeurs solutions pour tout problème de génération de séquences de tests. Pour cela, nous passons par une modélisation de ces problèmes à l'aide de contraintes.

3.2 Modélisation par un CSP

Nous présentons dans cette section la forme générale des systèmes de contraintes qui nous permettent de générer des séquences de test. Pour cela, nous introduisons aussi les principes généraux de l'approche, comme l'aspect dynamique de la propagation de contraintes (§3.2.1), qui permet l'introduction paresseuse de contraintes (§3.2.2) selon une analyse par chaînage arrière du modèle (§3.2.3). Nous introduisons la compilation d'un PGST vers un système de contraintes à l'aide d'une grammaire attribuée (§3.2.8).

> **Remarque 1.** *À partir de cette section, nous détaillons la modélisation et la génération de séquences de tests dans le cas de Lustre mono-horloge. En particulier, nous ne nous intéressons ni aux types énumérés, ni aux horloges ni à la réinitialisation de flots. Nous présentons la version étendue de GATeL pour horloges multiples au chapitre suivant.*

3.2.1 Exploration dynamique

Dans l'exemple d'introduction de la programmation par contraintes, à savoir « SEND+MORE=MONEY », les ensembles de contraintes et de variables sont statiques : on définit le problème une fois pour toutes et la propagation réduit uniquement les domaines de valeurs. Si on souhaitait produire un CSP statique pour générer des séquences de tests, on produirait une grille de variables correspondant aux valuations des flots dans le temps, et on expimerait les relations entre les flots de données au niveau de ces variables à l'aide de contraintes. Par exemple, on prend le flot s suivant, qui calcule la somme des valeurs successives de l'entrée entière e :

```
s = (e → (pre(s) + e)) ;
```

Si on traduit cette équation de manière systématique à chaque cycle, on obtient un système de contraintes où toutes les relations sont exprimées pour les différents flots, sur cinq cycles d'éxecution. Cependant, dans le cadre d'une génération basée sur objectif de test, on ne sait pas à l'avance combien de cycles sont nécessaires. Par exemple, si l'objectif de test est d'attendre $s = 10$, la taille d'une séquence solution est variable : si e est contraint à valoir soit 1, soit 2, à chaque cycle, alors la taille de la séquence générée pourra aller de 5 cycles à 10 cycles ; dans le cas général, où e peut prendre n'importe quelle valeur positive, négaitve ou nulle, le nombre de cycles n'est pas borné.

En plus du problème de la taille des séquences, il est généralement inefficace d'exprimer toutes les relations de dépendences qui existent entre les flots d'un nœud, notamment si seul un sous-ensemble du système est concerné par un objectif de test : il est inutile d'introduire des contraintes supplémentaires dans le système. En effet, un trop grand nombre de contraintes limite l'efficacité de la recherche de solution en pratique, à plus forte raison sur les gros systèmes. En principe, avec la résolution de contraintes, le nombre de solutions acceptables, d'abord très grand, diminue par raffinnements successifs, jusqu'à ce que l'on atteigne une séquence unique satisfaisante. Or, on ne manipule pas directement l'ensemble des solutions, mais des CSP, qui décrivent indirectement cet ensemble à l'aide de variables et de contraintes.

Avec une approche statique, la taille du CSP ne ferait que décroître, en partant d'un système initial où les domaines de valeurs des variables sont d'abord très grands puis diminuent. La taille initiale serait cependant très importante, même pour des modèles Lustre avec peu de variables. Le dénombrement des variables et des contraintes dans un système augmente en effet beaucoup plus vite que la complexité du modèle sous test, ce que l'on appelle couramment une explosion combinatoire.

Dans GATeL, le CSP initial contient peu de contraintes : implicitement, l'ensemble de solutions admissibles n'est pas contraint : il est là aussi important, mais toutes les relations ne sont pas représentées. Pendant les étapes de propagation et de résolution, de nouvelles contraintes s'ajoutent qui viennent réduire l'espace de recherche en explicitant les relations entre les flots de données du système au cours du temps, ce qui augmente la taille du CSP mais réduit le nombre de solutions admissibles. Finalement, les contraintes peuvent aussi être retirées lorsqu'elles sont trivalement vérifiées. Avec cette approche dynamique où des contraintes peuvent entrer et sortir du système pendant la recherche de solution, on peut diminuer la taille moyenne des CSP que l'on manipule, et ainsi traiter des modèles complexes.

Ainsi, pour faire en sorte que la complexité de la méthode de résolution soit dépendante non pas directement de la taille du modèle, mais de la complexité de l'objectif de test que l'on veut atteindre, GATeL fait évoluer dynamiquement le système de contraintes qui modélise un PGST. Pour cela, il ne faut pas introduire les contraintes trop abondamment dans un CSP, ce qui explique l'approche paresseuse de GATeL.

3.2.2 Évolution paresseuse

L'évolution des CSP dans GATeL est fortement limitée, et ce choix s'exprime à travers les règles de filtrage qui ont l'opportunité d'ajouter des contraintes dans le système. En toute généralité, lorsque l'on cherche à atteindre l'objectif de test, comme déclencher une alarme d'un système, celui-ci peut être accessible par différentes voies : par exemple, celle-ci peut-être déclenchée soit par la détection d'un problème de fonctionnement soit par un bouton d'arrêt d'urgence. Localement, c'est-à-dire au niveau d'une contrainte du système, la propagation doit faire face à un choix. L'approche que nous étudions

Cycle	...	5	4	3	2	1	0
e		1	1	1	2	3	1
s		2	3	4	6	9	10

FIGURE 3.5: Numérotation en arrière

est paresseuse car elle ne s'engage dans un de ces choix et n'introduit les contraintes qui y sont associées que si elle peut déterminer de manière certaine que ce choix est le seul réalisable. En l'occurence, si on a posé, comme invariant de l'environnement, que le bouton d'arrêt d'urgence n'est jamais activé, la génération de test s'intéresse à l'autre cas pour forcer le déclenchement de l'alarme. Lorsqu'il n'est pas possible d'éliminer des choix potentiels d'exploration du modèle, alors GATeL n'ajoute pas de contrainte supplémentaire. Avec une approche systématique, chaque cas possible ferait l'objet d'une traduction vers des contraintes, et la disjonction elle-même serait encodée (réifiée) dans le système, en laissant le soin aux filtrages futurs de détecter quel choix contribue finalement à la réalisation de l'objectif de test. Dans le cas de GATeL, on voit qu'il est nécessaire de déterminer localement si une branche, non traduite sous forme de contrainte, peut être explorée ou non. Nous verrons par la suite (§3.3.3.3) que nous avons recours pour cela à une fonction d'évaluation par sur-approximation.

3.2.3 Numérotation des cycles

GATeL met en œuvre un raisonnement par chaînage arrière, en partant d'un cycle où l'objectif est censé être observé, le cycle final, et en cherchant à déduire l'historique nécessaire des entrées/sorties qui amènent le système à couvrir cet objectif. Nous utilisons une numérotation des cycles adaptée à cette exploration du modèle qui part en arrière, du cycle final vers un cycle initial.

Nous numérotons les cycles en prenant pour origine le cycle de l'objectif de test et en incrémentant vers le passé. Autrement dit, l'objectif de test est vrai au cycle 0 et le cycle 1 correspond au cycle précédent du cycle zéro. Ce choix de numérotation s'accorde avec l'approche en arrière : le point de départ est l'objectif de test, et le nombre de cycles à construire dans le passé n'est pas connu à l'avance mais déterminé au fur et à mesure de la génération de la séquence.

Dans les grilles de valeurs, la direction du temps est préservée : le cycle final, numéroté zéro, est noté en fin de tableau (à droite). Les valeurs antérieures au cycle final sont à gauche, avec un numéro de cycle de plus en plus grand. La figure 3.5 montre une séquence s compatible avec l'objectif de test précédent (p. 74). Le nombre total de cycles de la séquence n'est pas encore déterminé, comme le montrent les points de suspension avant le cycle 5.

3.2.4 Organisation de l'ensemble de contraintes

Un PGST est traduit sous forme d'un ensemble de contraintes, qui jouent des rôles différents. Il existe ainsi, au sein d'un tel CSP :

- une base de faits représentant le modèle sous test (voir 3.2.8)
- des contraintes auxiliaires pour gérer les connaissances globales accumulées pendant la propagation (voir 3.2.5 et 3.2.7)
- des contraintes statiques entre les valeurs des flots (voir 3.2.9)
- des contraintes dynamiques, dont le filtrage peut introduire de nouvelles contraintes (voir 3.2.10)

La base de fait est simplement un ensemble de termes clos, sans filtrage associé, qui encode le PGST que l'on veut résoudre. Elle est nécessaire pour analyser le modèle sous test dynamiquement, étant donné que celui-ci n'est pas complètement traduit à l'avance sous forme de contraintes statiques. Les contraintes auxiliaires manipulent des informations globales, comme le nombre de cycles déjà construits.

3.2.5 Grille de valeurs

La grille des valeurs de flots est représentée par des contraintes de valuations ternaires mettant en relation un identifiant de flot, un cycle et un troisième terme symbolisant la valeur du flot au cycle considéré.

> **Définition 11.** *Valuation d'un flot à un cycle*
>
> Soient $f \in \mathbb{A}$ un atome identifiant un flot Lustre, $c \in \mathbb{N}$ un numéro de cycle et V un terme de $\mathcal{T}$. On note *set*(f, c, V) le fait que V soit la valeur associée au flot f au cycle c. V est généralement une variable, un entier ou un atome, selon le type du flot f. La contrainte admet la notation simplifiée suivante :
>
> $$f \mapsto_c V$$

L'ensemble des contraintes $f \mapsto_c V$ pour les différents flots à travers à travers le temps constitue une représentation de la grille de valeurs produite par la génération de séquences de tests. Ces contraintes sont ajoutées et s'accumulent dans l'ensemble selon les règles d'introductions énoncées plus loin. On note cependant qu'il n'existe qu'au plus une contrainte $f \mapsto_c V$ pour un couple (f, c) donné.

3.2.6 Domaines et types Lustre

Nous notons $D^S_{\texttt{int}}$ le domaine par défaut du type `int`, pour tout ensemble de contraintes S. Celui-ci est dépendant de S car le CSP contient un paramètre décrivant

les bornes minimales et maximales du domaine par défaut. Ainsi, quelque soit $S \subseteq \mathcal{C}_1^g$, il existe i et j deux entiers relatifs tels que le terme *minmax(i,j)* soit présent dans S. Alors, $D^S_{\texttt{int}} = i..j$.

Nous définissons $D_{\texttt{bool}} = [\mathit{false}, \mathit{true}]$ le domaine par défaut associé au type booléen de LUSTRE. Par généralisation, on définit $D^S_{\texttt{bool}} = D_{\texttt{bool}}$ pour tout CSP $S \subseteq \mathcal{C}_1^g$.

3.2.7 Gestion du temps LUSTRE et du cycle initial

Dans la section 3.2.1, nous avons introduit le problème de la génération de tests en utilisant des contraintes statiques. La taille de la séquence à générer n'est facile à estimer étant donnée un objectif de test : en se donnant une fenêtre temporelle trop petite, on peut échouer à générer une séquence de test lorsque l'objectif requiert des séquences relativement longues ; à l'opposé, en donnant une fenêtre trop grande, on introduit beaucoup de variables et de contraintes inutiles dans le problème de satisfaction, au risque de pénaliser l'efficacité de la génération de séquences.

Avec l'approche dynamique par chaînage arrière, la taille des séquences grandit au cours de la propagation, lorsque cela est nécessaire. Par exemple, si une expression fait référence à une valeur de flot au cycle précédent le cycle courant de l'exploration du modèle, on en déduit qu'un cycle passé est nécessaire.

3.2.7.1 Plus grand cycle connu

Dans cette démarche en arrière, les opérateurs d'initialisation « → » posent un problème : à un cycle c donné, faut-il évaluer l'expression portant la flèche comme le membre gauche ou droit de celle-ci ? On rappelle qu'il s'agit du membre gauche uniquement au cycle initial. Si on a déjà exploré un cycle antérieur au cycle c, alors c lui-même ne peut pas correspondre au cycle initial (il existe des valeurs avant dans le temps). En particulier, on peut en déduire immédiatement que la flèche doit s'évaluer selon son membre droit.

Pour permettre ce raisonnement, il suffit de mémoriser le plus grand cycle $mc \in \mathbb{N}$ tel qu'au cours des filtrages précédents, il ait existé une contrainte de la forme $V \Rightarrow_{mc} E$ (avec V et E des termes quelconques de $\mathcal{T}$ et $\mathcal{C}_1^g$, respectivement).

Un tel cycle antérieur a une valeur plus grande que c, étant donnée la numérotation des cycles que l'on a mise en place. Pour se souvenir du plus grand cycle déjà exploré, on introduit la contrainte *cyclemax(mc)* dans l'ensemble de contraintes. Son paramètre intial est zéro, c'est-à-dire le cycle final. Lorsque la génération de test nécessite d'explorer des cycles plus lointains dans le passé, la valeur est incrémentée de manière appropriée. Autrement dit, la contrainte *cyclemax(mc)* existante est remplacée dans l'ensemble de contraintes par *cyclemax(nc)*, avec $nc = mc + 1$.

Ainsi, le filtrage de la contrainte *propage* pour l'opérateur *init*, c'est-à-dire la flèche d'initialisation LUSTRE, peut se référer à la constante contenue dans *cyclemax* et comparer

son propre cycle de propagation pour déterminer quelle branche de l'expression doit être empruntée. La section 3.3.1.5 donne plus précisément les règles de propagation dans ce cas.

3.2.7.2 Statut d'initialité au cycle maximal

Malgré le recours au cycle maximal, on ne peut pas déterminer quelle branche de la flèche explorer lorsque le cycle de propagation correspond exactement au cycle maximal. En effet, la fenêtre temporelle à considérer peut augmenter ou se figer au cycle maximal courant.

Pour représenter le statut du cycle maximal, on introduit un paramètre supplémentaire à *cyclemax*, à savoir une variable dont le domaine par défaut est fini et contient deux valeurs possibles (la contrainte prend donc en fait deux arguments). On note $D_{\texttt{status}} = [initial, noninitial]$ ce domaine.

> **Définition 12.** *Contrainte de statut de cycle*
>
> La contrainte *cyclemax(mc,SI)*, dans un ensemble de contraintes modélisant un problème de génération de séquences de test, n'admet qu'une occurence dans cet ensemble et porte les informations suivantes :
> - *mc* est le plus grand cycle auquel une expression Lustre a été propagée à travers la contrainte dépliable *propage*.
> - *SI* est une variable associée au domaine de valeurs $D_{\texttt{status}}$. Elle indique si le cycle maximal *mc* est le cycle initial ou non. La variable reste non instantiée tant que ce statut est indéterminé.

3.2.8 Arbre syntaxique abstrait

L'approche dynamique de GATeL repose sur le fait que le CSP dispose d'une représentation interne des équations de flots et qu'il peut ainsi accéder à ces équations et les analyser pendant le filtrage des contraintes. Ainsi, par exemple, le flot *s* précédent est traduit en un terme sans variables dans l'ensemble de contraintes. L'expression originale est celle-ci :

```
s = (e -> (pre(s) + e)) ;
```

La traduction dans le langage de contraintes est la suivante :

equa(s, init(e, add(pre(s), e)))

Le terme de haut-niveau *equa(s, Exp)* associe le flot *s* à son équation, le deuxième argument, le terme *Exp*. Ce terme est à son tour une imbrication de sous-termes, chacun correspondant à une sous-expression Lustre de l'équation initale. La flèche est symbolisée par le terme *init* à deux arguments, de même que l'addition par le terme *plus*.

$$\begin{array}{lcll} List\ (L^{\uparrow}) & ::= & Elt\ ,\ List(SL^{\uparrow}) & -\ L = SL + 1 \\ & | & \epsilon & -\ L = 0 \end{array}$$

FIGURE 3.6: Lecture d'une liste et calcul de sa longueur.

Nous présentons rapidement la phase de traduction à l'aide d'une grammaire attribuée [Knu68]. Il s'agit de la même grammaire que précédemment, où cette fois chaque règle de production admet un ou plusieurs attributs. Une règle de production peut synthétiser un attribut a à partir des symboles terminaux lus où en combinant d'autres attributs ; on notera $a^{\uparrow}$ quand un attribut est synthétisé par un règle. Par ailleurs, il est parfois utile de passer un attribut a à des sous-règles, pour que celles-ci aient des informations contextuelles. Dans ce cas, on dit qu'une règle peut hériter d'un attribut, qui est alors noté $a^{\downarrow}$ dans la grammaire. Dans notre traduction, les attributs sont des ensembles de contraintes ou des termes du langage $\mathcal{T}$. On utilise en particulier les opérations d'union sur les ensembles et les notations de manipulation de listes vues précédemment. La grammaire ne traite pas les opérateurs de filtrage et de réinitialisation, qui ne sont pas utilisables dans le langage $\mathcal{L}_1$ que l'on considère.

3.2.8.1 Exemple de grammaire attribuée

Nous donnons dans un premier temps un exemple de grammaire attribuée. Le but d'une grammaire attribuée est double : il s'agit de décrire les arbres syntaxiques que l'on peut construire avec un langage, tout en manipulant des attributs dans les nœuds de cet arbre. La grammaire donne implicitement un algorithme de lecture et d'interprétation d'un langage. Nous notons les attributs entre parenthèses, comme paramètres des symboles non-terminaux.

Une première utilisation des attributs est de synthétiser une information globale lors de la lecture d'un élément. La figure 3.6 illustre ceci : la syntaxe décrit un langage dans lequel on lit une liste d'éléments séparés par des virgules. Le non-terminal *List* est soit la succession d'un élément *Elt* (non précisé ici) et d'une sous-liste, séparés par une virgule, soit le terme vide ϵ. Par ailleurs, ce même non-terminal *List* synthétise un attribut, la longueur L de la liste lue. Cet attribut vaut zéro dans le cas de la liste vide, et $SL + 1$ lorsqu'elle est construite par un élément suivi d'une sous-liste de taille SL.

Les attributs peuvent aussi contraindre l'arbre syntaxique de sorte à n'accepter qu'un sous-ensemble des constructions que l'on pourrait avoir sans ces attributs. Soient $L_1 = \alpha_1, \ldots, \alpha_n$ et $L_2 = \beta_1, \ldots, \beta_n$ deux listes de même taille. Dans le langage d'exemple que l'on définit, on souhaite que l'opération $map(+, L_1, L_2)$ construise la liste suivante, où l'opérateur + est appliqué aux éléments de même rang dans les deux listes :

$$map(+, L_1, L_2) = (\alpha_1 + \beta_1), \ldots, (\alpha_n + \beta_n)$$

$$\begin{array}{rcl} Map & ::= & \texttt{map (} Op \texttt{ , } List(L^{\uparrow}) \texttt{ , } List(L^{\uparrow}) \texttt{)} \\ Op & ::= & \texttt{+} \mid \texttt{-} \end{array}$$

FIGURE 3.7: Lecture d'une expression `map` où les listes ont même longueur.

Dans la figure 3.7, la grammaire accepte la construction des expressions de la forme `map(Op,L1,L2)`, dans laquelle `Op` est une opération arithmétique et les autres arguments sont des listes. Pour s'assurer que ces listes sont bien de même taille, les tailles remontées à travers les attributs de chaque listes doivent être identiques. C'est pour cela qu'elles partagent toutes deux la même variable L. La grammaire rejette ainsi les expression `map` où les listes sont de tailles différentes.

Finalement, les attributs peuvent être hérités, c'est-à-dire passés d'un non-terminal aux éléments qui le définissent dans les règles. Nous utilisons cette technique par la suite pour pouvoir accéder aux variables déclarées lors de la lecture des idenfiants. Nous illustrons ceci dans la gramaire qui suit. Nous définissons un langage dans lequel on évalue une addition en présence de variables. Celles-ci sont données par des identifiants et peuvent être affectées à des valeurs. Toutes les affectations sont regroupées dans un ensemble C avant que l'expression ne soit lue. L'expression est alors évaluée dans le contexte des affectations précédentes, en exploitant le contexte passé en héritage lorsque l'on rencontre un identifiant.

Une évaluation est donnée par un contexte suivi d'une expression qui s'évalue à v dans ce contexte. L'ensemble C est passé par héritage à l'expression pour que celle-ci puisse bénéficier des informations qui y sont contenues.

$$Eval(v^{\uparrow}) \quad ::= \quad Context(C^{\uparrow}) \ : \ Exp(C^{\downarrow}, v^{\uparrow})$$

Un contexte est suite d'affectations entre des identifiants et des valeurs. Si un identifiant est affecté à plusieurs valeurs, la dernière affectation l'emporte.

$$\begin{array}{rcll} Context(C^{\uparrow}) & ::= & Assign((I,v)) \ ; \ Context(SC^{\uparrow}) & -\ C = \begin{cases} SC \text{ si } \exists\ (I,w) \in SC \\ \{(I,v)\} \cup SC \text{ sinon} \end{cases} \\ & \mid & \epsilon & -\ C = \emptyset \\ Assign(A^{\uparrow}) & ::= & Ident(I^{\uparrow}) = Val(V^{\uparrow}) & -\ A = (I,V) \end{array}$$

Une expression est soit une valeur, soit une somme d'expression, et peut faire référence à des identifiants. Si un identifiant renvoit à une valeur dans le contexte, l'expression prend cette valeur. Sinon, la valeur par défaut est zéro.

$$
\begin{array}{lcll}
Exp(C^{\downarrow}, v^{\uparrow}) & ::= & Val(v^{\uparrow}) & \\
 & | & (Exp(C^{\downarrow}, v1^{\uparrow}) + Exp(C^{\downarrow}, v2^{\uparrow})) & -\ v = v1 + v2 \\
 & | & Ident(I^{\uparrow}) & -\ v = \begin{cases} w & si\ \exists\ (I, w) \in C \\ 0 & \text{sinon} \end{cases}
\end{array}
$$

Prenons ainsi l'évaluation suivante :

```
x=3;y=5 : ( x + y )
```

La première affectation produit comme attribut un couple $(x, 3)$, la deuxième le couple $(y, 5)$. L'ensemble des affectation nous donne le contexte $C = \{(x, 3), (y, 5)\}$, qui est passé ensuite dans le non-terminal *Exp* . Il s'agit d'une addition faisant intervenir deux sous-expressions. La première fait référence à x, l'autre à y, et leur valeurs respectives sont remontées depuis le contexte C pour remonter la valeur 8.

3.2.8.2 Compilation

> **Définition 13.** *Langage cible de la traduction*
>
> On note $\mathcal{C}_1^g$ le langage cible de la traduction, c'est-à-dire l'image du langage $\mathcal{L}_1^g$ par application des règles de production de la grammaire attribuée. Ce langage est un sous-ensemble de $\mathcal{C}$ contenant uniquement des termes clos et représentant les constructions du langage $\mathcal{L}_1$.

Nous présentons maintenant la syntaxe attribuée pour la compilation de $\mathcal{L}_1^g$ vers $\mathcal{C}_1^g$. L'ensemble de contraintes synthétisé par la règle *Model* est le CSP issu de la compilation.

Modèle Les déclarations de types ne sont pas encore utiles dans le langage mono-horloge. Un modèle est donc donné directement par un nœud.

$$Model(N^{\uparrow}) \quad ::= \quad Node(N^{\uparrow})$$

Nœud Les déclarations de variables sont traduites et forment un ensemble C qui sert de contexte pour la compilation du corps du nœud. Les équations et les assertions présentes dans ce corps de nœud sont alors remontées, en même temps que le contexte C, sous la forme d'un CSP.

$$
\begin{array}{rcll}
\mathit{Node}(N^{\uparrow}) & ::= & \texttt{node}\ \textbf{ID}\ \texttt{(}\ \mathit{Vars}(\mathit{in}^{\downarrow}, I^{\uparrow})\ \texttt{)} & \\
 & & \texttt{returns (}\ \mathit{Vars}(\mathit{out}^{\downarrow}, O^{\uparrow})\ \texttt{) ;} & \\
 & & \mathit{OptVars}(V^{\uparrow}) & \\
 & & \texttt{let} & \\
 & & \mathit{Body}(C^{\downarrow}, B^{\uparrow}) & \\
 & & \mathit{Extensions}(C^{\downarrow}, E^{\uparrow}) & -\left\{ \begin{array}{lcl} C & = & I \cup O \cup V \\ N & = & C \cup B \cup E \end{array} \right. \\
 & & \texttt{tel ;} &
\end{array}
$$

Cas des variables locales optionnelles :

$$
\begin{array}{rcll}
\mathit{OptVars}(V^{\uparrow}) & ::= & \texttt{var}\ \mathit{Vars}(\mathit{loc}, V^{\uparrow}) & \\
 & | & \epsilon & -\ V = \emptyset
\end{array}
$$

Liste de déclarations de variables :

$$
\begin{array}{rcll}
\mathit{Vars}(E^{\downarrow}, L^{\uparrow}) & ::= & \mathit{Var}(E^{\downarrow}, V^{\uparrow})\ ;\ \mathit{Vars}(E^{\downarrow}, L_0^{\uparrow}) & -\ L = V \cup L_0 \\
 & | & \mathit{Var}(E^{\downarrow}, L^{\uparrow}) &
\end{array}
$$

Déclaration d'une variable L'attribut hérité E permet d'ajouter une étiquette supplémentaire à la variable pour savoir s'il s'agit d'une entrée (in), d'une variable locale (loc) ou d'une sortie (out).

$$
\begin{array}{rcll}
\mathit{Var}(E^{\downarrow}, V^{\uparrow}) & ::= & \textbf{ID}\ :\ \mathit{Type}(T^{\uparrow}) & -\ V = \left\{ \begin{array}{l} \mathit{type}(\textbf{ID}, T), \\ \mathit{io}(\textbf{ID}, E) \end{array} \right\} \\
 & & & \\
\mathit{Type}(T^{\uparrow}) & ::= & \texttt{int} & -\ T = \mathit{int} \\
 & | & \texttt{bool} & -\ T = \mathit{bool}
\end{array}
$$

Corps d'un nœud Les équations et les assertions sont représentées par des contraintes, où les expressions Lustre sont données par des termes clos.

$$
\begin{array}{rcll}
\mathit{Body}(C^{\downarrow}, B^{\uparrow}) & ::= & \mathit{BodyElt}(C^{\downarrow}, E^{\uparrow})\ \mathit{Body}(C^{\downarrow}, B_0^{\uparrow}) & -\ B = \{E\} \cup B_0 \\
 & | & \mathit{BodyElt}(C^{\downarrow}, E^{\uparrow}) & -\ B = \{E\} \\
 & & & \\
\mathit{BodyElt}(C^{\downarrow}, BE^{\uparrow}) & ::= & \textbf{ID}\ =\ \mathit{Expr}(C^{\downarrow}, E^{\uparrow})\ ; & -\ BE = \mathit{equa}(\textbf{ID}, E) \\
 & | & \texttt{assert}\ \mathit{Expr}(C^{\downarrow}, A^{\uparrow})\ ; & -\ BE = \mathit{assert(A)}
\end{array}
$$

Expressions Les règles de production des expressions nécessitent d'avoir les informations présentes dans C, notamment pour savoir si un identifiants fait référence à une valeur d'un type énuméré ou à une variable. Les expressions sont associées à des termes du langage de contraintes, selon les règles suivantes.

$Expr(C^{\downarrow}, E^{\uparrow})$	::=	`(` $Unop(C^{\downarrow}, E^{\uparrow})$ `)`	
	|	`(` $Binop(C^{\downarrow}, E^{\uparrow})$ `)`	
	|	`if (` $Expr(C^{\downarrow}, I^{\uparrow})$ `)`	
		`then` $Expr(C^{\downarrow}, A^{\uparrow})$	
		`else` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = if(I, A, B)$
	|	`true`	$-\ E = true$
	|	`false`	$-\ E = false$
	|	**INT**	$-\ E =$ **INT**
	|	**ID**	$-\ E =$ **ID**
$Unop(C^{\downarrow}, E^{\uparrow})$	::=	`-` $Expr(C^{\downarrow}, F^{\uparrow})$	$-\ E = minus(F)$
	|	`not` $Expr(C^{\downarrow}, F^{\uparrow})$	$-\ E = not(F)$
	|	`pre` $Expr(C^{\downarrow}, F^{\uparrow})$	$-\ E = pre(F)$
$Binop(C^{\downarrow}, E^{\uparrow})$	::=	$Expr(C^{\downarrow}, A^{\uparrow})$ `+` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = add(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `-` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = minus(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `*` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = mult(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `div` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = div(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `mod` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = mod(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) \rightarrow Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = init(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) > Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = gt(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) < Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = lt(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) \geq Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = geq(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) \leq Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = leq(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `and` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = and(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow})$ `or` $Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = or(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) = Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = eq(A, B)$
	|	$Expr(C^{\downarrow}, A^{\uparrow}) \neq Expr(C^{\downarrow}, B^{\uparrow})$	$-\ E = neq(A, B)$

La grammaire contient une règle supplémentaire pour l'extension du modèle selon les directives propres à GATeL, donnée par le non-terminal *Extensions* . Celui-ci est donné dans les règles de la grammaire attribuée qui suivent. En particulier, les extensions GATeL ajoutent au CSP le terme *cyclemax* initial pour que celui-ci soit bien formé. On note ainsi CSP_{def} le CSP contenant les valeurs par défaut de toutes les contraintes auxiliaires. En

l'occurence, on a :

$$CSP_{def} = \{ cyclemax(\text{-}1,noninitial) \}$$

Les informations propres au problème de génération de séquences de tests sont interprétables par GATeL. Ces directives comportent un objectif de test et une ou plusieurs directives que l'on détaille plus bas.

$$\begin{array}{lcll} Extensions(C^{\downarrow}, E^{\uparrow}) & ::= & Objective(C^{\downarrow}, O^{\uparrow}) & \\ & & Params(P^{\uparrow}) & \\ & & Splits\ (C^{\downarrow}, S^{\uparrow}) & -\ E = \{O\} \cup P \cup S \cup CSP_{def} \end{array}$$

Un objectif de test est donné par une clause `reach`, où l'argument est une expression booléenne.

$$Objective(C^{\downarrow}, O^{\uparrow}) \quad ::= \quad \texttt{reach}\ Expr(C^{\downarrow}, E^{\uparrow})\ ; \qquad -\ O = reach(E)$$

Les paramètres du problème : on considère les bornes minimales et maximales des intervalles par défaut des valeurs entières, ainsi que *maxsize*, la plus grande taille de séquence autorisée : lors de l'exploration du modèle, le cycle est compris entre zéro (inclus) et cette valeur (exclue).

$$\begin{array}{lcll} Params(P^{\uparrow}) & ::= & \texttt{maxsize}\ \mathbf{INT}_1\ ; & \\ & & \texttt{minint}\ \mathbf{INT}_2\ ; & -\ P = \left\{ \begin{array}{c} maxsize(\mathbf{INT}_1), \\ minmax(\mathbf{INT}_2, \mathbf{INT}_3) \end{array} \right\} \\ & & \texttt{maxint}\ \mathbf{INT}_3\ ; & \end{array}$$

$$\begin{array}{lcll} Splits(C^{\downarrow}, NS^{\uparrow}) & ::= & Split(C^{\downarrow}, Split^{\uparrow})\ ;\ Splits(C^{\downarrow}, S^{\uparrow}) & -\ NS = \{Split\} \cup S \\ & | & \epsilon & -\ NS = \emptyset \end{array}$$

Une directive `split` est donnée sous la forme suivante :

$$Split(C^{\downarrow}, S^{\uparrow}) \quad ::= \quad \texttt{split}\ \mathbf{ID}\ \texttt{with}\ [\ EList(C^{\downarrow}, L^{\uparrow})\ ] \qquad -\ S = split(\mathbf{ID}, L)$$

$$\begin{array}{lcll} EList(C^{\downarrow}, L^{\uparrow}) & ::= & Expr(C^{\downarrow}, E^{\uparrow})\ ;\ EList(C^{\downarrow}, T^{\uparrow}) & -\ L = [E|T] \\ & | & Expr(C^{\downarrow}, E^{\uparrow}) & -\ L = [E] \end{array}$$

3.2.9 Contraintes statiques

Les contraintes statiques relient les valeurs des entrées et des sorties du système réactif à travers les cycles d'éxecution. Ce sont des contraintes classiques d'opérations arithmétiques et logiques entre des variables à valeurs entières, comme la contrainte

add(R,X,Y) qui représente la relation $R = X + Y$ de sorte que le résultat de l'addition peut réduire les domaines respectifs des opérandes, et inversement. Les contraintes d'appartenance à un domaine et les contraintes d'unification sont des cas particuliers que l'on a présenté au chapitre 1.

3.2.10 Propagation de valeurs à travers des expressions

Dans la section 3.2.8, nous avons dit que l'approche dynamique de propagation de contraintes repose sur le fait qu'il existe une représentation interne du modèle sous test. En effet, le but est de pouvoir parcourir les expressions de définition de ce modèle *pendant* le filtrage des contraintes, afin d'introduire progressivement d'autres contraintes (arithmétiques et logiques) entre les valeurs de flots et à travers les cycles de calcul. Nous présentons ici la contrainte qui met en œuvre cette introduction dynamique.

> **Définition 14.** *Contrainte d'égalité* LUSTRE
>
> Soient E un terme du langage $\mathcal{C}_1^g$, V une variable ou un terme atomique de $\mathcal{T}$ et c un entier naturel positif. On note *propage(V,E,c)* la contrainte d'égalité LUSTRE exprimant le fait que l'expression LUSTRE symbolisée par le terme E a pour valeur V au cycle c. La notation simplifiée de cette contrainte, que l'on utilisera le plus souvent, est la suivante :
>
> $$V \Rightarrow_c E$$

La notation simplifiée représente le fait que la valeur V est propagée à travers l'expression E au cycle c. Au cours de la descente dans l'expression E, des contraintes atomiques sont ajoutées pour raffiner le problème, c'est-à-dire réduire l'ensemble de ses solutions.

3.2.11 Raffinement du système de contraintes

Le principe de la contrainte *propage* consiste à avoir une règle pour chaque type d'opérateur en tête de l'expression E, c'est-à-dire chaque opérateur du langage $\mathcal{L}_1$. Le filtrage selon ces règles permet d'introduire des contraintes sur les sous-termes de E, soit d'autres contraintes dépliables, soit des contraintes statiques entre les valeurs des flots. Par exemple, avec c est un entier naturel et A et B deux termes de $\mathcal{C}_1^g$, le filtrage de « $10 \Rightarrow_c$ *add(A,B)* » retire cette contrainte de l'ensemble et la remplace principalement par trois contraintes :

1. $V_A \Rightarrow_c A$, avec V_A une nouvelle variable
2. $V_B \Rightarrow_c B$, avec V_B une nouvelle variable
3. *add(10,*V_A*,*V_B*)*

La contrainte d'origine associait la valeur 10 à l'addition des flots A et B au cycle c. Elle a été remplacée par une contrainte entre les valeurs ponctuelles de A et B à ce cycle, la troisième contrainte ci-dessus : *add(10, V_A, V_B)* signifie ici que la somme de V_A et V_B vaut 10. Par ailleurs, nous introduisons des contraintes dépliables pour les sous-termes (les deux premières contraintes), si bien que lorsqu'une des variables V_A ou V_B vient à être instanciée, la sous-expression associée, A ou B, peut elle-même être décomposée et produire de nouvelles contraintes. Les règles de propagation de la contrainte d'égalité $V \Rightarrow_c E$ sont détaillées dans la section 3.3.

Le filtrage de la contrainte d'égalité Lustre suit une progression en arrière, depuis le cycle final vers un cycle initial, et du résultat d'une expression vers ses opérandes (jusqu'aux entrées). Des contraintes intermédiaires sont introduites dans cet ordre, tissant ainsi des relations entre les valeurs de flots dans le temps.

3.2.12 Extraction des solutions

L'ensemble de contraintes associé à un problème de génération de séquences évolue de sorte à construire une grille de valeurs évaluable. Nous définissons le prédicat *solution* dans le cas de GATeL, qui nous permet de savoir quand arrêter la procédure de recherche d'une solution. Une fois que l'on a un CSP sous forme résolue, on doit pouvoir extraire une grille de valeurs du CSP et vérifier que si celle-ci est bien évaluable, alors l'objectif de test est couvert par la séquence générée.

3.2.12.1 CSP sous forme résolue

Un ensemble de contraintes est sous forme résolue lorsqu'il s'agit d'un point fixe par propagation et qu'il ne contient plus aucune contrainte pouvant amener à l'insatisfaisabilité du système. L'ensemble contient alors uniquement la grille de valeurs générée, la représentation statique du PGST et les contraintes de domaines.

Il suffit pour cela que le CSP ne contienne que des contraintes qui portent sur au plus une variable, mais aucune contrainte *propage*. Par ailleurs, il est nécessaire que le cycle initial de la séquence soit connu.

$$\text{GATEL-SOLUTION}\ \frac{\begin{array}{c} \mathit{fail} \notin S \qquad \forall C \in S, |\mathit{vars}(C)| \leq 1 \\ \{V \Rightarrow_c E\} \cap S = \emptyset \\ d \in \mathbb{N} \qquad S \vdash \mathit{status(d)} : \mathit{initial} \end{array}}{\vdash S : \mathit{solution}}$$

Une fois que le CSP satisfait au prédicat *solution* , toutes les variables sont indépendantes les unes des autres et il est alors possible de toutes les instancier aléatoirement dans leur domaines de valeurs respectifs sans aboutir à une insatisfaisabilité.

3.2.12.2 Grille de valeurs évaluable

Grille incomplète À partir d'un CSP solution, on souhaite obtenir une grille de valeurs évaluable (un élément de Γ^*, voir 2.2.3.9). On remarque cependant que seule une partie de la grille de valeurs est générée par GATeL : celle qui est concernée, directement ou indirectement, par l'objectif de test du PGST.

Il est probable qu'il existe des entrées du modèle sous test qui ne sont pas représentées dans la grille que l'on a construite, car la propagation de contraintes n'a pas eu besoin de le faire ; les flots non représentés n'influent pas sur l'objectif de test, que ce soit à travers les dépendances entre les équations de flots ou par l'intermédiaire des invariants du modèle sous test. Or, pour qu'une grille de valeurs soit évaluable, toutes les sorties doivent être évaluables à chaque cycle de calcul, ce qui nécessite de combler les parties manquantes de la grille. Il suffit alors de choisir une valeur dans le domaine par défaut de chaque variable non représentée pour obtenir une grille évaluable.

Instantiation de flots Pour convertir le CSP associé à un PGST P en une grille de valeurs de Γ^*_P, nous récupérons les valuations du CSP qui renvoient à des valeurs atomiques et inversons la numérotation des cycles pour obtenir des valuations de la forme $f\,[c'] \leftarrow v$ que l'on peut interpréter dans le langage $\mathcal{L}_1^g$. Lorsqu'un flot n'est pas représenté dans l'ensemble de contraintes, ou lorsque le terme associé à une valuation ponctuelle d'un flot est un domaine, une valeur arbitraire est prise dans le domaine de valeurs du flot, par l'intermédaire de la fonction *random(D)* ; celle-ci tire aléatoirement une valeur dans le domaine D qu'elle prend en paramètre.

Définitions Nous définissons l'ensemble *inputs(S)* comme étant l'ensemble des identifiants des flots d'entrée d'un programme Lustre représenté par le CSP S. Par définition :

$$inputs(S) = \{f \mid io(f, in) \in S\}$$

Nous définissons aussi la fonction *instanciate(S,f,c)* qui renvoie une valuation pour le flot f au cycle c en fonction de celles présentes ou non dans un CSP. Nous utilisons cette fonction lorsque toutes les variables sont instanciées, donc il n'est pas possible qu'un flot soit affecté à une variable, mais uniquement à une valeur atomique. Ainsi, pour tout CSP S, tout identifiant de flot f, tout terme $T \in \mathcal{C}_1^g$ et tout cycle c :

- Si $S \vdash f \mapsto_c T$, la valuation est déjà présente dans le CSP : *instanciate(S,f,c)* = T
- Sinon, on a $\vdash_s type(\mathtt{f}) = T$. Dans ce cas, le flot n'est pas représenté, et il suffit de prendre une valeur aléatoire dans le domaine par défaut associé au type du flot : *instanciate(S,f,c)* = $random(D_T^S)$

Extraction d'une grille Finalement, nous pouvons définir le prédicat « *seq* » suivant, qui donne une grille de valeurs Γ de longueur d compatible avec un CSP S sous forme résolue : la règle produit un jugemement de la forme « $S \vdash seq : \Gamma$ ».

Le prédicat se décompose en deux règles :

$$\textsc{extract-seq-rec}\ \frac{\begin{array}{ll} S \vdash f \rightarrowtail_c V & V \in \mathbb{V} \qquad S \vdash V \in D \\ w = random(D) & S[V/w] \vdash seq : \Gamma \end{array}}{S \vdash seq : \Gamma}$$

$$\textsc{extract-seq-base}\ \frac{vars\left(\left\{(f \rightarrowtail_c V) \in S\right\}\right) = \emptyset \qquad S \vdash initial(d)}{S \vdash seq : \{f\ [(d-c)] \leftarrow v \mid f \in inputs(S), c \in 0..d, v = instanciate(S,f,c)\}}$$

La première est appliquée tant qu'il reste des variables libres liées à des flots Lustre à travers les cycles. Lorsque c'est le cas, une valeur est tirée dans le domaine de cette variable. On applique la substitution $S[V/w]$, qui remplace toutes les occurences de V par la valeur choisie w. La séquence Γ extraite de CSP intermédiaire est la séquence extraite pour S.

La seconde règle est appliquée quand toutes les valuations renvoient à des valeurs atomiques. Alors, la grille est remplie avec les valeurs présentes dans le CSP et complétée pour les flots non représentés à l'aide du domaine par défaut du flot pour qui une valeur est manquante (par l'intermédiaire de la fonction *instanciate*). Les cycles sont inversés de sorte que le cycle initial soit numéroté 0 et le cycle final d.

La grille obtenue peut ou non être évaluable, car il arrive que GATeL génère des séquences qui ne respectent pas complètement le modèle sous test et ne représentent ainsi pas de véritables solutions (cf. §3.3.3.3). La fonction d'évaluation vue au chapitre 2 nous permet de distinguer les grilles solutions (§3.1.4) des autres.

En réalité, comme GATeL génère les valeurs des variables calculées, l'évaluation consiste en pratique aussi à vérifier que les sorties évaluées sont bien identiques aux valeurs propagées durant la génération de séquence.

3.2.13 Conclusion

Nous avons défini ce qu'est un problème de génération de séquences de tests pour GATeL, donné une représentation d'un tel problème sous la forme d'un ensemble de contraintes et présenté l'approche générale de GATeL. Un tel ensemble de contraintes est produit à partir d'un élément P de $\mathcal{L}_1^g$, et contient alors systématiquement :

1. une représentation statique du problème de génération à l'aide d'un ensemble de termes clos, où les différents termes possibles sont les suivants :
   - *equa(F,E)* : expression de définition E associée à l'identifiant de flot F.
   - *type(F,T)* : type du flot F.
   - *io(F,N)* : nature du flot F, où N peut valoir *in*, *out* ou *loc* selon que F est une entrée, une sortie ou une variable locale du nœud.
   - *assert(A)* : invariant ; A doit être vrai à chaque cycle.
   - *reach(Obj)* : objectif de test.
   - *maxsize(m)* : taille limite des séquences à générer ; les cycles valides sont alors ceux allant de 0 à $m-1$.
   - *minmax(i,j)* : intervalle par défaut pour les valeurs entières.
2. des contraintes auxiliaires, destinées à traiter des informations globales propres au système et pouvant évoluer au cours de la génération de tests ; parmi elles, on a :
   - une contrainte « *cyclemax(mc,SI)* », valant initialement « *cyclemax(-1,noninitial)* », marquant le plus grand cycle mc auquel a déjà été exploré le modèle.
   - une grille de valeurs, initialement vide, représentée par autant de contraintes de la forme $f \mapsto_c V$ que nécessaire.
3. des contraintes statiques (initalement aucune), mettant en relation des variables du langage $\mathcal{C}$ et des valeurs symboliques ; parmi elles, on distingue :
   - les contraintes d'appartenance à un domaine, de la forme « $V \in D$ ».
   - les contraintes d'unification, de la forme « $X \equiv Y$ ».
   - les contraintes plus générales (arithmétiques et logiques), exprimées par des termes composés.
4. des contraintes dynamiques de la forme $V \Rightarrow_c E$, où chacune d'elle associe un terme du langage de contraintes V à une expression Lustre E à un cycle c ; ces contraintes peuvent introduire dynamiquement d'autres contraintes (statiques ou dynamiques) et peuvent aussi, par effet de bord, influencer les contraintes auxiliaires globales.

3.3 Propagation de contraintes

La génération de tests consiste à appliquer la recherche de solution par filtrages successifs de contraintes : nous cherchons à construire un ensemble de contraintes solution qui puisse être interprété comme une grille de valeurs satisfaisant l'objectif de test d'un PGST. Nous présentons le filtrage des contraintes, en particulier la contrainte *propage*, puis les mécanismes de sélection de cas de test et quelques heuristiques de résolutions.

3.3.1 Filtrage des expressions de flots

Nous décrivons les règles de filtrage de contraintes propres à GATeL$_1$, et notamment celles liées à la contrainte *propage*. Nous voyons ici comment le premier terme *propage* est introduit intialement dans le CSP, par l'intermédaire de l'objectif de test, et comment il génère au cours des filtrages successifs de nouvelles contraintes statiques et dynamiques.

3.3.1.1 Objectif de test

Initialement, le CSP fourni par la compilation d'un PGST comporte une contrainte filtrable qui interprète l'objectif de test donné par *reach(E)* et le remplace par la propagation de la valeur de vérité *true* dans l'expression E au cycle zéro, le cycle final.

$$\text{PROP-REACH}\ \frac{}{\vdash S \cup \{reach(E)\} \rightarrow S \cup \{true \Rightarrow_0 E, objective(E)\}}$$

La règle s'assure que le terme original *reach(E)* est remplacé par le terme *objective(E)*, qui lui n'admet pas de règle de propagation mais sert uniquement de marqueur pour se souvenir de l'objectif de test. En laissant la contrainte initiale *reach*, la règle serait toujours applicable. Ici, on s'assure que l'introduction de la contrainte *propage* n'a lieu qu'une fois.

3.3.1.2 Expressions littérales

La contrainte *propage* prend en argument une expression Lustre pour explorer le modèle sous test et introduire les contraintes nécessaires. Parmi ces expressions, le cas de base consiste à propager des valeurs littérales. Dans le cas de Lustre mono-horloge, il s'agit des valeurs numériques entières ou des littéraux booléens.

$$\text{PROP-INT}\ \frac{v \in \mathbb{Z}}{\vdash S \cup \{R \Rightarrow_c v\} \rightarrow S \cup \{R \equiv v\}} \qquad \text{PROP-BOOL}\ \frac{b \in \{false, true\}}{\vdash S \cup \{R \Rightarrow_c b\} \rightarrow S \cup \{R \equiv b\}}$$

3.3.1.3 Expressions arithmétiques

Le filtrage de la contrainte *propage* dans le cas des opérations arithmétiques entre des flots Lustre consiste à représenter chaque sous-terme par une variable et à la propager

dans cette sous-expression au même cycle ; les variables sont alors reliées entre-elles par une contrainte atomique. Par exemple, les règles suivantes montrent la propagation des opérateurs unaires, c'est-à-dire la négation arithmétique « `-E` » et logique « `not E` ».

$$\text{PROP-MINUS-1}\ \frac{S \vdash W : \mathit{fresh}}{\vdash S \cup \{V \Rightarrow_c \mathit{minus}(E)\} \rightarrow S \cup \{W \Rightarrow_c E, \mathit{minus}(W,0,V)\}}$$

$$\text{PROP-NOT}\ \frac{S \vdash W : \mathit{fresh}}{\vdash S \cup \{V \Rightarrow_c \mathit{not}(E)\} \rightarrow S \cup \{W \Rightarrow_c E, \mathit{not}(V,W)\}}$$

Dans les deux cas, la variable V est associée à une variable fraîche W par l'intermédiaire d'une contrainte statique, et la variable W est propagée dans la sous-expression E. Les contraintes *minus(W,0,V)*[3] et *not(V,W)* dans la seconde règle sont celles qui introduisent et réduisent les domaines des valeurs des variables dans l'ensemble de contraintes. De la même manière, le filtrage de la contrainte *propage* dans le cas des opérateurs binaires suit la règle ci-dessous :

$$\text{PROP-BINOP-ARITH}\ \frac{\begin{array}{c} S \vdash V_A, V_B : \mathit{fresh} \\ \mathscr{R} \in \{\mathit{add}, \mathit{minus}, \mathit{mult}, \mathit{div}, \mathit{mod}, \mathit{gt}, \mathit{lt}, \mathit{geq}, \mathit{leq}\} \end{array}}{\vdash S \cup \{R \Rightarrow_c \mathscr{R}(A,B)\} \rightarrow S \cup \{\mathscr{R}(R,V_A,V_B), V_A \Rightarrow_c A, V_B \Rightarrow_c B\}}$$

Les contraintes statiques évoluent ensuite par elle-même selon les domaines de valeurs associés aux variables. Par exemple, dans le cas des opérateurs d'inégalité, le statut de l'inégalité (elle est vraie ou fausse) est donné par le premier argument (R dans la règle précédente) ; si celui-ci est faux, on peut alors remplacer la contrainte par une autre, qui réduit les domaines de valeurs. Ainsi, la règle qui suit correspond à la relation « ≥ » :

$$\text{GEQ3-FALSE}\ \frac{}{\vdash S \cup \{\mathit{geq}(\mathit{false},V,W)\} \rightarrow S \cup \{\mathit{lt}(W,V)\}}$$

Par la suite, le filtrage de la contrainte *lt (lower than)* d'arité 2 peut détecter une insatisfiabilité ou projeter la contrainte sur les domaines de valeurs des arguments. Par exemple, les règles suivantes retirent la contrainte ou mettent en évidence l'insatisfaisabilité du CSP :

$$\text{LT-CHECK}\ \frac{(a,b) \in \mathbb{N}^2 \qquad a < b}{\vdash S \cup \{\mathit{lt}(a,b)\} \rightarrow S} \qquad \text{LT-UNSAT}\ \frac{(a,b) \in \mathbb{N}^2 \qquad a \geq b}{\vdash S \cup \{\mathit{lt}(a,b)\} \rightarrow \{\mathit{false}\}}$$

Les règles de filtrage peuvent aussi intéragir avec les domaines :

$$\text{LT-REDUCE}\ \frac{S \vdash \mathit{lt}(V,W) \qquad v'_h = min(w_h, v_h - 1) \qquad w'_\ell = max(w_\ell, v_\ell + 1)}{\vdash S \cup \{V \in v_\ell..v_h, W \in w_\ell..w_h\} \rightarrow S \cup \{V \in v_\ell..v'_h, W \in w'_\ell..w_h\}}$$

3. *minus(W,0,V)* signifie $W = 0 - V$, c'est-à-dire $W = -V$.

Finalement, les domaines eux-mêmes admettent des règles de propagation. Par exemple, la règle précédente pourrait amener une situation où un domaine est trivialement vide, car la borne maximale est strictement inférieure à la borne minimale d'un intervalle d'entiers ; de même, lorsqu'un domaine ne contient qu'une valeur, on peut instancier la variable concernée par cette valeur :

$$\text{INT-UNSAT}\ \frac{b < a}{\vdash S \cup \{V \in a..b\} \rightarrow \{false\}} \qquad \text{INT-UNIF}\ \frac{}{\vdash S \cup \{V \in a..a\} \rightarrow S \cup \{V \equiv a\}}$$

Nous avons donné ici un aperçu rapide de la complexité liée au traitement des contraintes arithmétiques qui permettent à la propagation de contraintes de filtrer les domaines de valeurs du CSP. Nous ne détaillons pas davantage ces mécanismes car nous nous concentrons sur l'aspect Lustre du problème, à savoir la gestion des flots de données et du temps logique, et non des opérations sur les valeurs portées par ces flots.

3.3.1.4 Borne temporelle

Le traitement des opérateurs temporels dans le cas mono-horloge est réalisé à l'aide de la contrainte « *cyclemax* », qui représente à la fois le cycle maximal déjà exploré et le statut de ce cycle (initial, non-initial ou statut inconnu).

Statut d'un cycle Nous pouvons déterminer à l'aide de la contrainte *cyclemax* le statut d'un cycle quelconque, soit parce que le statut est donné directement par la contrainte au cycle maximal, soit parce que l'on s'intéresse à un cycle c plus petit que le cycle maximal, auquel cas c est non-initial. Nous définissons le prédicat auxiliaire *status(c) : SI* qui indique le statut d'initialité SI associé à un cycle c quelconque, donné par les règles simples suivantes :

$$\text{STATUS-MAX}\ \frac{S \vdash \mathit{cyclemax(mc,SI)}}{S \vdash \mathit{status(mc) : SI}} \qquad \text{STATUS-OTHER}\ \frac{S \vdash \mathit{cyclemax(mc,SI)} \qquad mc > c \geq 0}{S \vdash \mathit{status(c) : noninitial}}$$

Modification du cycle maximal Les contraintes du CSP peuvent modifier la borne donnée par *cyclemax* en agissant sur sa variable de statut SI. Lorsque celle-ci est instanciée à *noninitial*, le cycle maximal est repoussé d'un cran vers le passé et les invariants du système, portés par les déclarations `assert`, sont propagées au nouveau cycle (cf. §2.1.2.3 du chapitre 2). Ainsi, on s'assure que chaque expression invariante est convertie sous forme de contraintes aux cycles où des flots sont explorés, uniquement quand nécessaire, et au plus une fois par cycle. Les règles ci-dessous prennent aussi en compte la limite de cycles donnée par le problème : il est possible que l'on cherche à générer une séquence trop grande pour le paramètre *maxsize* du problème, ce qui se

traduit par un échec de la génération.

$$\textsc{synchro-fail}\ \frac{S \vdash \mathit{maxsize(M)} \qquad nc = c+1 \qquad nc \geq M}{S \cup \{\mathit{cyclemax(c,noninitial)}\} \rightarrow \{\mathit{fail}\}}$$

$$\textsc{synchro-move}\ \frac{\begin{array}{c} S \vdash \mathit{maxsize(M)} \qquad nc = c+1 \qquad nc < M \\ S \vdash SI : \mathit{fresh} \qquad S_A = \{\mathit{true} \Rightarrow_{nc} A \mid \mathit{assert(A)} \in S\} \end{array}}{S \cup \{\mathit{cyclemax(c,noninitial)}\} \rightarrow S \cup S_A \cup \{\mathit{cyclemax(nc,SI)}, SI \in D_{\texttt{status}}\}}$$

Par exemple, on sait que le CSP initial contient la contrainte « *cyclemax(-1,noninitial)* ». Lorsque l'on applique la règle précédente, on introduit les invariants au cycle zéro et on remplace la contrainte initiale par une contrainte *cyclemax(0,SI)*. Autrement dit, le cycle est incrémenté et les invariants propagés au cycle zéro. Lorsque le statut est instancié à la valeur *initial*, cela signifie que l'on considère que le cycle maximal est aussi le cycle initial de la séquence générée.

3.3.1.5 Opérateur d'initialisation

À l'aide du statut de cycle, nous pouvons décrire le filtrage des termes Lustre de la forme « `A → B` ». Ceux-ci sont représentés par des termes *init(A,B)* dans le CSP. Lorsque l'on propage une valeur dans ce terme à un cycle quelconque, on doit déterminer si le cycle est le premier cycle de la séquence ou non. Les règles suivantes exploitent pour cela l'information contenue dans le terme *cyclemax*.

$$\textsc{prop-init-left}\ \frac{S \vdash \mathit{status(c)} : \mathit{initial}}{S \cup \{R \Rightarrow_c \mathit{init(A,B)}\} \rightarrow S \cup \{R \Rightarrow_c A\}}$$

$$\textsc{prop-init-right}\ \frac{S \vdash \mathit{status(c)} : \mathit{noninitial}}{S \cup \{R \Rightarrow_c \mathit{init(A,B)}\} \rightarrow S \cup \{R \Rightarrow_c B\}}$$

Lorsque le statut est variable, aucune contrainte n'est propagée. Nous verrons avec la fonction de sur-approximation qu'il est possible de traiter ce cas où le statut est variable quand une des branches A ou B peut-être ignorée.

3.3.1.6 Opérateurs logiques

Les flots booléens de Lustre sont propagés de manière similaire aux opérateurs arithmétiques. Cependant, ils illustrent l'approche paresseuse de GATeL car les sous-termes ne sont pas systématiquement introduits dans le système.

Nous avons par exemple les règles suivantes pour l'opérateur de branchement conditionnel `if..then..else`. Cet opérateur est représenté par le terme « *if(Cond,Then,Else)* » : le premier terme est la condition à évaluer, et les deux autres termes sont les expressions

alternatives correspondant respectivement au cas où la condition s'évalue à *true* ou *false* à un cycle donné.

Dans un premier temps, la condition est elle-même convertie sous forme de contrainte : une nouvelle variable V est créée et associée à la condition au cycle voulu :

$$\text{PROP-IF-COND}\ \frac{C \notin \mathbb{V} \cup D_{\text{bool}} \qquad S \vdash V : \mathit{fresh}}{S \cup \{R \Rightarrow_c \mathit{if(Cond,Then,Else)}\} \rightarrow S \cup \left\{ \begin{array}{c} V \Rightarrow_c \mathit{Cond}, V \in D_{\text{bool}}, \\ R \Rightarrow_c \mathit{if(V,Then,Else)} \end{array} \right\}}$$

La règle n'est applicable que si la condition n'est pas déjà une variable, où une valeur atomique booléenne. Elle remplace le terme *propage* existant par un nouveau, où cette fois la condition est la variable V.

Le système de contraintes va alors potentiellement appliquer des filtrages sur le domaine de V jusqu'à ce que cette variable soit instanciée par un booléen. Alors, on peut lier le résultat R à une des branches de l'opérateur :

$$\text{PROP-IF-TRUE}\ \frac{}{S \cup \{R \Rightarrow_c \mathit{if(true,E,F)}\} \rightarrow S \cup \{R \Rightarrow_c E\}}$$

$$\text{PROP-IF-FALSE}\ \frac{}{S \cup \{R \Rightarrow_c \mathit{if(false,E,F)}\} \rightarrow S \cup \{R \Rightarrow_c F\}}$$

Les deux règles nécessitent que la condition, le premier élément du terme *if*, soit instancié. Si la condition est variable, on ne sait pas de quelle branche vient le résultat R. La contrainte reste donc bloquée dans le système tant que la condition n'est pas connue. On verra par la suite que certains autres filtrages sont possibles dans ce cas.

3.3.1.7 Décalage temporel

La règle de filtrage pour une contrainte de la forme « $R \Rightarrow_c \mathit{pre(E)}$ » consiste à propager la valeur R dans l'expression E, au cycle qui précède le cycle c, à savoir $c+1$:

$$\text{PROP-PRE}\ \frac{S \vdash \mathit{status(c)} : \mathit{noninitial} \qquad nc = c+1}{\vdash S \cup \{V \Rightarrow_c \mathit{pre(E)}\} \rightarrow S \cup \{V \Rightarrow_{nc} E\}}$$

La première condition d'application de la règle, qui teste si le cycle c est bien un cycle non-initial, est toujours vérifiée implicitement en pratique, car pour passer dans un terme « `pre` », la propagation a nécessairement dû traverser le membre droit d'un opérateur d'initialisation « → ». Cela correspond à l'hypothèse des expressions bien formées du langage vue page 50, et à l'ordonnancement des contraintes introduites dans GATeL. Ici, nous ne détaillons pas l'ordonnancement et nous avons besoin d'ajouter la condition dans les prémisses de la règle.

3.3.1.8 Variables et identifiants

Lorsqu'on décompose les expressions de flots d'un PGST à l'aide de la contrainte *propage*, on arrive finalement soit à des valeurs constantes, soit à des identifiants qui renvoient à d'autres variables. Nous nous intéressons maintenant aux identifiants de variables, présentes dans le langage $\mathcal{C}_1^g$ sous la forme $var(f)$ (référence à un flot f).

Si on accède à la valeur du flot f à un cycle c par l'intermédiaire de *propage*, on la représente par une variable $V \in \mathbb{V}$ du langage de contraintes associée à un domaine, grâce à un terme de la forme $f \mapsto_c V$. L'ensemble des termes de cette forme constitue la grille de valeurs en cours de génération, dont les éléments sont ajoutés selon les cycles visités et les expressions explorées. Si le terme $f \mapsto_c V$ existe déjà dans le CSP courant, la nouvelle valuation est unifiée avec l'ancienne :

$$\text{PROP-VAR-UNIF}\ \frac{S \vdash f \mapsto_c V}{S \cup \{R \Rightarrow_c f\} \rightarrow S \cup \{R \equiv V\}}$$

Ainsi, toutes les références à une même valeur ponctuelle d'un flot sont associées à une même variable, que ce soit pour les variables d'entrées ou les variables calculées. Dans le cas contraire, il n'existe pas de terme $f \mapsto_c V$ dans l'ensemble, et le filtrage de la contrainte *propage* dépend de la nature de la variable visitée (entrée ou flot calculé). Nous définissons tout d'abord le prédicat *undef(f, c)*, qui signifie qu'il n'existe pas de variable associé à f au cycle c :

$$\text{AUX-UNDEF}\ \frac{S \cap \{(f \mapsto_c V) \mid V \in \mathcal{C}_1^g\} = \emptyset}{S \vdash \mathit{undef(f,\ c)}}$$

Lorsque ce prédicat est vérifié et que le flot f est une variable d'entrée de type T, nous introduisons un terme $f \mapsto_c R$, où R est la variable propagée dans l'expression f au cycle c, ainsi qu'une contrainte $R \in D_T$, où D_T est l'intervalle de valeurs par défaut du type T. Nous rappelons que *io(f,in)* désigne le fait que le flot f est un flot d'entrée :

$$\text{PROP-VAR-IN}\ \frac{S \vdash \mathit{undef(f,\ c)} \qquad S \vdash \mathit{type(f,T)} \qquad S \vdash \mathit{io(f,in)}}{S \cup \{R \Rightarrow_c f\} \rightarrow S \cup \{f \mapsto_c R, R \in D_T\}}$$

Dans le cas des flots calculés, la valeur V est de plus propagée dans l'expression de définition de f. Cependant, les règles nous assurent que cette propagation n'est faite qu'au plus une fois par variable et par cycle, pour ne pas dupliquer les contraintes que l'on introduit dynamiquement. En effet, l'introduction de « $R \Rightarrow_c Exp$ » n'est possible que s'il n'existe pas encore d'élément associé à f au cycle c dans la grille de valeurs, mais une fois la propagation réalisée, cette association existe avec la variable R.

$$\text{PROP-VAR-OTHER}\ \frac{S \vdash \mathit{undef(f,\ c)} \qquad S \vdash \mathit{type(f,T)} \qquad S \vdash \mathit{equa(f,Exp)}}{S \cup \{R \Rightarrow_c f\} \rightarrow S \cup \{f \mapsto_c R, R \in D_T, R \Rightarrow_c Exp\}}$$

3.3.2 Encadrement d'une expression de flot

L'approche paresseuse mise en œuvre dans GATeL limite les décisions que l'on peut prendre dans le CSP, comme dans le cas de l'opérateur → (aussi appelé « flèche »), où l'on doit attendre de connaître le statut du cycle de la propagation avant de savoir quelle branche explorer (§3.3.1.5). On s'empêche d'ajouter trop de contraintes dans le système afin de limiter l'explosion combinatoire, mais ceci a un impact sur les informations que l'on peut exploiter.

Or, il n'est pas toujours nécessaire d'introduire de nouvelles contraintes pour déterminer la valeur d'une expression, ni même nécessaire d'avoir la valeur exacte d'un flot à un cycle pour prendre des décisions. Si l'ensemble de contraintes contient déjà suffisamment d'informations pour donner un encadrement d'une expression à un cycle, celui-ci peut suffir à filter les contraintes existantes. Par exemple, si on a la contrainte « $5 \Rightarrow_c$ *init(A,B)* » dans un CSP, et que l'on sait que l'expression A ne peut pas valoir 5 au cycle c, alors on peut en déduire que :

- Le résultat 5 est nécessairement obtenu par le membre droit B
- Le cycle c ne peut pas être le cycle initial

L'évaluation par sur-approximation consiste à donner un encadrement, à un cycle, de la valeur d'une expression Lustre E donnée sous la forme d'un terme de $\mathcal{C}_1^g$. Avec les informations présentes dans le CSP courant, à savoir les valuations de flots déjà existantes et les encadrements qui y sont associés, la sur-approximation retourne un domaine de valeurs possible pour E au cycle c (un intervalle, ou une liste d'atomes). Elle n'introduit pas de contrainte supplémentaire, mais se contente d'observer le CSP courant pour fournir l'encadrement.

> **Définition 15.** *Évaluation partielle par sur-approximation à un cycle*
>
> La fonction d'évaluation est définie pour tout ensemble de contraintes S, toute expression $E \in \mathcal{C}_1^g$ et tout cycle c. Soit D un terme de $\mathcal{T}$ représentant un intervalle d'entiers ou une liste d'atomes (booléens et types énumérés).
> Par définition, on note $S \vdash E \approx_c D$ le fait qu'étant donné le contexte S, l'expression E admette comme encadrement, au cycle c, le domaine D.

Nous décrivons dans les sections qui suivent une partie des règles de calcul pour la sur-approximation des expressions de flots Lustre. Les règles choisies sont représentatives de la méthode de calcul, qui cherche à donner un encadrement précis, lorsque c'est possible, d'une expression à un cycle.

3.3.2.1 Encadrement trivial

En plus de la sur-approximation à un cycle, nous définissons une fonction auxiliaire qui nous donne le domaine associé à une variable de $\mathbb{V}$ ou à une valeur instanciée. On

note « $S \vdash V \approx D$ » le fait que V admette le domaine D comme encadrement dans le contexte S éventuellement vide (ici, aucun cycle n'intervient).

$$\text{APPROX-VAR}\ \frac{S \vdash V \in D}{S \vdash V \approx D}$$

$$\text{APPROX-INT}\ \frac{v \in \mathbb{Z}}{\vdash v \approx v..v} \qquad \text{APPROX-ATOM}\ \frac{a \in \mathbb{A}}{\vdash a \approx [a]}$$

3.3.2.2 Expressions simples

La plupart des opérateurs du langage Lustre admettent des règles simples pour définir la fonction d'évaluation. On considère par exemple l'addition de flots :

$$\approx\text{-PLUS}\ \frac{\begin{array}{cc} S \vdash A \approx_c i..j & S \vdash B \approx_c m..n \\ x = i + m & y = j + n \end{array}}{S \vdash \mathit{add(A,B)} \approx_c x..y}$$

Les domaines donnés par l'évaluation des sous-termes A et B nous donnent un encadrement $x..y$ contenant toutes les valeurs possibles de l'addition. Les autres opérateurs arithmétiques se comportent de la même manière.

3.3.2.3 Opérateur conditionnel

La règle de l'opérateur conditionnel « `if` » admet plusieurs cas, selon que la condition est évaluable de manière exacte ou non. Les deux règles qui suivent prennent en compte respectivement le cas où la condition I admet une évaluation exacte à *true* ou *false* (les intervalles représentent une valeur unique à chaque fois).

$$\approx\text{-IF-THEN}\ \frac{S \vdash I \approx_c [\mathit{true}] \qquad S \vdash T \approx_c D_T}{S \vdash \mathit{if(I,T,E)} \approx_c D_T}$$

$$\approx\text{-IF-ELSE}\ \frac{S \vdash I \approx_c [\mathit{false}] \qquad S \vdash E \approx_c D_E}{S \vdash \mathit{if(I,T,E)} \approx_c D_E}$$

Dans la règle ci-dessous, on considère que la condition n'admet pas d'évaluation exacte, ce qui nous oblige à faire l'union des domaines évalués dans chacune des branches. On rappelle que l'opérateur « $\cup$ » symbolise ici l'union de deux domaines de même nature (§1.4.2 page 17), par exemple deux intervalles d'entiers (pour les flots entiers) ou

deux listes d'atomes (pour des flots de types énumérés).

$$\approx\text{-IF-UNION}\ \frac{\begin{array}{c} S \vdash I \approx_c [\mathit{false}, \mathit{true}] \\ S \vdash T \approx_c D_T \qquad S \vdash E \approx_c D_E \\ D = D_T \cup D_E \end{array}}{S \vdash \mathit{if(I,T,E)} \approx_c D}$$

3.3.2.4 Variables de flots

De même que pour la propagation de contraintes, on distingue l'évaluation d'un identifiant de flot selon que celui-ci est une entrée du système ou une variable calculée, et selon que ce flot admette ou non un représentant dans la grille de valeurs au cycle qui nous intéresse.

Les règles qui suivent s'appliquent lorsqu'il existe un terme « $f \rightarrowtail_c V$ » liant le flot f à la variable V au cycle c. C'est le cas si on a déjà introduit précédement une contrainte *propage* associée à f au cycle c.

Dans la règle qui suit, le terme associé à f est soit une variable, pour qui un domaine existe dans S, soit un entier relatif ou un atome, où l'encadrement est alors trivialement connu. Il existe nécessairement au moins un domaine de valeurs si c'est une variable : la propagation d'une variable introduit au moins un terme « $V \in D_T$ », avec T le type du flot.

$$\approx\text{-VAR}\ \frac{S \vdash V \approx D}{S \cup \{f \rightarrowtail_c V\} \vdash f \approx_c D}$$

Il est possible que d'autres domaines soient associés à V dans le même CSP, ce qui peut modifier le domaine retourné par l'approximation selon que l'on choisit un encadrement ou un autre pour V ; cependant, cela ne change que la précision de l'approximation, qui est toujours un sur-ensemble de toutes les valeurs possibles pour la variable.

Lorsqu'il n'existe pas de terme « $f \rightarrowtail_c V$ » dans l'ensemble de contraintes, on se contente de retourner le domaine par défaut associé au type du flot dont la valeur est sur-approximée au cycle c :

$$\approx\text{-VAR-BOOL}\ \frac{\begin{array}{c} S \vdash \mathit{undef(f,\ c)} \\ S \vdash \mathit{type(f,bool)} \end{array}}{S \vdash f \approx_c D_{\mathtt{bool}}} \qquad \approx\text{-VAR-INT}\ \frac{\begin{array}{c} S \vdash \mathit{undef(f,\ c)} \\ S \vdash \mathit{type(f,int)} \end{array}}{S \vdash f \approx_c D^S_{\mathtt{int}}}$$

3.3.2.5 Opérateur d'initialisation

Dans le cas de l'opérateur d'initialisation, la sur-approximation peut bénéficier des informations de contexte présentes dans S, à savoir le statut de cycle donné par la contrainte *cyclemax*. Selon le cycle ou la sur-approximation a lieu, on peut se contenter de prendre le résultat d'une branche ou d'une autre, ce qui permet d'être plus précis.

3.3.3 Filtrages supplémentaires à l'aide de l'encadrement

Le filtrage des contraintes fait appel à la sur-approximation pour exploiter les informations déjà présentes dans un CSP. L'approximation peut donner un encadrement assez fin d'une expression, plus fin que le domaine par défaut que l'on associe à une valeur de flot quelconque, car elle elle prend en compte les valuations partielles et les domaines déjà présents dans le CSP. En particulier, les encadrement permettent d'éliminer des choix qui se révèlent impossibles à emprunter. Les règles de propagation à l'aide de la sur-approximation se superposent aux règles existantes. Nous présentons une règle générale qui permet de réduire les domaines déjà existants dans un ensemble de contraintes et montrons ensuite comment le filtrage de *propage* peut être étendu dans le cas de l'opérateur d'initialisation et du `if..then..else`, qui sont représentatifs des opérateurs de contrôle où la propagation est paresseuse.

3.3.3.1 Réduction de domaine

La règle suivante montre comment une contrainte « $R \Rightarrow_c Exp$ » peut faire appel à la sur-approximation pour réduire le domaine lié à la variable R.

$$\textsc{prop-}\approx \quad \frac{(R \Rightarrow_c Exp) \in S \qquad S \vdash Exp \approx_c = D_E \qquad D' = D_E \cap D \qquad D' \neq D}{S \cup \{R \in D\} \rightarrow S \cup \{R \in D'\}}$$

Le domaine initial de la variable R, D, est remplacé dans le CSP par un domaine D', l'intersection entre D et l'encadrement D_E de l'expression Exp au cycle c.

Cette règle peut-être utile, par exemple, dans la propagation que l'on a vue dans le cas de l'opérateur `if..then..else`, qui attend que la condition de l'opérateur soit complètement connue avant de propager une ou l'autre des branches.

Lorsque l'on applique la règle PROP-IF-COND vue page 95, on introduit un terme « $V \Rightarrow_c Cond$ » qui associe la variable V à l'expression *Cond* au cycle c ; on peut imaginer que la condition est « `x > y` », où les variables x et y admettent pour encadrements respectifs 5..10 et 0..4 au cycle c. Or, dans ce cas, l'inégalité est forcément toujours vérifiée. L'approximation de *Cond* est exacte et se résume alors à « *true..true* », ce qui finalement déclenche l'unification de V avec la valeur *true*.

3.3.3.2 Exploitation des incompatibilités entre domaines

La sur-approximation nous permet d'avoir un encadrement d'une expression sans la propager. Dans le cas de l'opérateur *if(Cond,Then,Else)*, nous n'avons par encore exploité jusqu'ici les deux expressions alternatives *Then* et *Else* portées par l'opérateur. La règle suivante le fait, lorsqu'une ou l'autre admet un encadrement qui est incompatible avec le résultat retourné par l'opérateur *if*. Dans ce cas, la branche n'est tout simplement pas

compatible avec le résultat, ce qui limite les choix possibles et permet encore une fois de faire progresser le CSP.

$$\textsc{prop-if-not-then}\ \frac{\begin{array}{c} S = S' \cup \{R \Rightarrow_c \mathit{if(C,Then,Else)}\} \\ C \in \mathbb{V} \qquad S \vdash R \approx D_R \qquad S \vdash \mathit{Then} \approx_c D_T \qquad D_R \cap D_T = \emptyset \end{array}}{\vdash S \rightarrow S \cup \{C \equiv \mathit{false}\}}$$

$$\textsc{prop-if-not-else}\ \frac{\begin{array}{c} S = S' \cup \{R \Rightarrow_c \mathit{if(C,Then,Else)}\} \\ C \in \mathbb{V} \qquad S \vdash R \approx D_R \qquad S \vdash \mathit{Else} \approx_c D_E \qquad D_R \cap D_E = \emptyset \end{array}}{\vdash S \rightarrow S \cup \{C \equiv \mathit{true}\}}$$

De même, pour l'opérateur « → », on ajoute des règles pour vérifier si éventuellement une ou l'autre des expressions portées par la flèche est incompatible avec la valeur de l'expression. La règle s'applique quand le statut de cycle est indéterminé :

$$\textsc{prop-init-not-left}\ \frac{\begin{array}{c} S = S' \cup \{R \Rightarrow_c \mathit{init(A,B)}\} \\ S \vdash \mathit{status(c)} : SI \qquad SI \in \mathbb{V} \\ S \vdash R \approx D_R \qquad S \vdash A \approx_c D_A \qquad D_A \cap D_R = \emptyset \end{array}}{\vdash S \rightarrow S \cup \{SI \equiv \mathit{noninitial}\}}$$

$$\textsc{prop-init-not-right}\ \frac{\begin{array}{c} S = S' \cup \{R \Rightarrow_c \mathit{init(A,B)}\} \\ S \vdash \mathit{status(c)} : SI \qquad SI \in \mathbb{V} \\ S \vdash R \approx D_R \qquad S \vdash B \approx_c D_B \qquad D_B \cap D_R = \emptyset \end{array}}{\vdash S \rightarrow S \cup \{SI \equiv \mathit{initial}\}}$$

L'unification avec *initial* rend filtrable la contrainte *cyclemax* par la règle SYNCHRO-MOVE vue page 93, qui décale alors le plus grand cycle connu d'un cran vers le passé.

3.3.3.3 Approximation et séquences générées

Lorsque le filtrage d'une contrainte tire parti d'un encadrement d'une expression, celui-ci repose sur le fait que l'approximation donne un bon encadrement de cette expression. Or, telle qu'elle est présentée ici, l'approximation renvoie toujours un domaine, même dans les cas où l'expression n'est pas évaluable : une fois la séquence générée, on peut obtenir des combinaisons de valeurs qui sont bien dans le domaine retourné initialement par leur encadrements, mais telles que l'expression qui nous intéresse n'admette pas de valeur. Prenons par exemple l'expression LUSTRE suivante :

```
if (false) then (1/0) else 5 ;
```

Cette expression admet pour représentation « *if(false,div(1,0),5)* ». Or, l'encadrement de cette expression est le domaine 5..5, alors qu'elle n'est pas évaluable à cause de la division par zéro selon la sémantique LUSTRE (§2.2.3.4).

Séquences incorrectes Alors que l'on s'efforce d'avoir des règles de propagation de contraintes qui sont correctes vis-à-vis de l'évaluation, au sens où l'on veut ne générer que des séquences évaluables, l'approximation nous donne des encadrements qui peuvent nous amener à générer des séquences incorrectes. Les règles de propagation avec sur-approximation peuvent mener à des CSP sous-contraints : en évitant de propager une branche d'un opérateur, on a potentiellement oublié des contraintes qui auraient assuré la correction de la séquence de test. Dans ce cas, l'étape de vérification que l'on a présenté au début du chapitre est réellement nécessaire, après avoir généré une séquence, pour s'assurer que le test obtenu est bien un test éxécutable.

Ajout des contraintes manquantes Une autre définition de la sur-approximation est possible, qui consiste à remonter, en même temps qu'un encadrement, les contraintes supplémentaires qui permettent de rester compatible avec la sémantique LUSTRE. Par exemple, la sur-approximation d'un terme « *div(A,B)* » peut donner à la fois un encadrement du résultat de la division et une contrainte supplémentaire « $B \not\equiv 0$ ». Cette contrainte doit alors être injectée dans le CSP courant si une règle de filtrage dépend de l'encadrement. Dans l'exemple précédent, la contrainte aurait détecté l'insatisfaisabilité car le terme « $0 \not\equiv 0$ » aurait été introduit.

Approche mixte GATeL repose en partie sur ce principe : l'approximation d'une expression E à un cycle donné remonte en plus de l'encadrement un ensemble de couples (f, c) ; chaque élément indique un instant c auquel le flot f a contribué à l'encadrement, mais tel qu'il n'existe pas encore de valuation dans la grille de valeurs pour le couple (f, c). Lorsque l'encadrement est utilisé par le filtrage du CSP pour prendre une décision, la grille de valeurs est complétée pour ces flots aux instants nécessaires. Pour cela, on ajoute des contraintes « $V_f \Rightarrow_c f$ » (la variable V_f est alors une variable libre) dans le CSP, ce qui en particulier permet d'introduire les invariants donnés par les *assert* aux cycles où ils ne l'étaient pas encore. En introduisant les *assert* aux cycles traversés par la sur-approximation, on s'assure que l'encadrement donné par celle-ci respecte bien les invariants du modèle. Il existe cependant des cas où des expressions intermédiaires, non liées à des identifiants de flots, peuvent reposer sur des hypothèses que l'on ne garantit pas par la suite. Quoiqu'il en soit, la séquence générée par GATeL est systématiquement vérifiée par évaluation afin de retirer les mauvaises solutions.

3.3.4 Génération de séquences de test

Dans le chapitre **??**, nous avons vu que le filtrage de contraintes est utilisé pour réduire l'espace de recherche d'une procédure plus générale de résolution par essais/erreurs. La génération de séquences est ainsi une succession de valuations aléatoires suivies de propagations de contraintes, où chaque propagation applique toutes les règles de filtrages

possibles jusqu'à atteindre un point fixe. Les valuations peuvent conduire à des échecs ou mener à des solutions.

La résolution de contraintes applique des heuristiques pour réaliser les valuations aléatoires. Comme dans le cas général vu page 22, la résolution de contrainte dépend d'un prédicat ayant la forme suivante :

$$S \vdash \mathit{heuristics}(V, w)$$

Il indique la variable V et la valeur w, prise dans le domaine de V, choisies par l'heuristique pour faire progresser la recherche de solution. On explore alors les branches $V \equiv w$ et $V \not\equiv w$. La gestion de l'arbre de recherche n'est pas détaillée ici, mais dans le cas de GATeL, on cherche en priorité à faire avancer la partie paresseuse de la propagation avant de s'intéresser à la partie arithmétique. La partie arithmétique de la procédure est assez rarement appelée. Le choix de la variable et de la valeur tentée pour un système de contraintes donné obéit au raisonnement suivant :

1. Récupérer les variables booléennes qui ont le plus de contraintes en attente, et parmi celles-ci, en choisir une au plus grand numéro de cycle.
2. Comparer ce nombre de contraintes avec le nombre de contraintes en attente sur la variable de statut de cycle St telle que $S \vdash \mathit{cyclemax(c,St)}$.
3. Si ce deuxième nombre est plus grand, chercher à fermer le passé en instanciant le statut à *initial*. Si cette instanciation échoue, le statut est instancié à *noninitial* et la procédure recommencera après le point fixe de propagation qui suivra.
4. Sinon, instancier la variable booléenne de manière aléatoire dans son domaine.
5. Si aucune variable booléenne ni de statut n'est présente dans les contraintes, la procédure choisit une variable entière correspondant à une entrée et l'instancie de manière aléatoire dans son domaine.

3.3.5 Sélection de cas de test

Dans GATeL, la sélection de cas de test est un mécanisme proposé à l'utilisateur pour raffiner un PGST en lui permettant de réaliser des raisonnements par cas sur celui-ci. De la même manière que lors de la résolution de contraintes, des hypothèses supplémentaires sont ajoutées à un point fixe de la propagation pour raffiner celui-ci. Cependant, les hypothèses introduites sont différentes de celles données par l'heuristique de résolution, car elles tiennent compte des opérateurs du langage Lustre. Une autre différence, en pratique, est que la sélection de cas de test fait intervenir l'utilisateur, alors que la recherche de solution est automatisée. Dans une certaine mesure, il est possible d'automatiser la sélection de cas de test, à partir d'un critère de couverture ou d'un découpage prédéfini. Dans une utilisation courante de GATeL, l'utilisateur définit un objectif de test, raffine le problème en un ou plusieurs sous-problème par sélection de cas de tests, puis lance la génération de séquences pour les différents cas de tests.

3.3.5.1 Règles de sélection

Nous définissons par la suite les hypothèses que l'on peut introduire par sélection de cas de test. Pour cela, nous définissons le prédicat suivant, où S, H_1, …, H_n sont tous des CSP de $\mathcal{C}_1^g$:

$$S \vdash select : \{H_1, \ldots, H_n\}$$

Le prédicat signifie que S admet un découpage en n sous-cas ($n > 0$). Chaque ensemble $S \cup H_i$, avec $0 \leq i \leq n$, est un sous-cas de test de l'ensemble initial S si les hypothèses faites dans H_i décrivent bien un sous-ensemble de solutions de S.

Si on peux trouver une solution pour $S \cup H_i$, elle est aussi une solution de S :

$$\text{RESOL-SUBCASE}\ \frac{S \vdash select : \mathscr{S} \qquad H \in \mathscr{S} \qquad \vdash S \cup H \rightsquigarrow Sol}{\vdash S \rightsquigarrow Sol}$$

L'intérêt de la sélection est d'avoir un contrôle plus précis sur les séquences générées, en forçant la génération de tests à considérer des cas rares à produire en pratique, et ainsi augmenter la couverture structurelle ou fonctionnelle du modèle. Le découpage d'un PGST en différents sous-cas de test est aussi appelé dépliage ; on distingue le dépliage structurel du dépliage fonctionnel.

3.3.5.2 Dépliage structurel

Le dépliage structurel découpe le cas de test courant en différents sous-cas en se basant sur la structure des opérateurs Lustre. Prenons par exemple le cas de l'opérateur « or ». Soit S un CSP contenant la contrainte *propage* suivante :

$$R \Rightarrow_c or(A,B)$$

Bien que les règles ne l'imposent pas, la sélection est réalisée sur des points fixes par propagation, et dans notre cas, on suppose que la contrainte ne peut pas être filtrée. Pour cet opérateur, GATeL propose un dépliage en deux cas de tests :

$$\text{SELECT-OR2}\ \frac{}{S \cup \{R \Rightarrow_c or(A,B)\} \vdash select \left\{ \begin{array}{l} \{R \equiv true, true \Rightarrow_c A\}, \\ \{R \Rightarrow_c B, false \Rightarrow_c A\} \end{array} \right\}}$$

Les deux sous-cas correspondent aux cas où l'expression A est respectivement vraie ou fausse. Dans le premier cas, le résultat est nécessairement lui-aussi vrai, alors que dans le second, le résultat de la disjonction est donné par l'expression B. Chaque CSP obtenu en ajoutant un ensemble d'hypothèses ou l'autre constitue un nouveau CSP, que l'on peut traiter indépendamment de l'autre, et lui-même raffiner encore en différents sous-cas. En itérant le dépliage structurel des opérateurs, on peut ainsi améliorer la couverture

structurelle du modèle, mais aussi mieux comprendre le fonctionnement du système sous test lors de sa conception.

Le découpage en deux sous-cas de tests est un choix arbitraire, et il existe en réalité deux autres découpages possibles, en 3 ou 4 cas :

$$\textsc{select-or3}\ \frac{}{S \cup \{R \Rightarrow_c or(A,B)\} \vdash select : \left\{ \begin{array}{c} \{R \equiv false\}, \\ \{R \equiv true, true \Rightarrow_c A\}, \\ \{R \equiv true, true \Rightarrow_c B\} \end{array} \right\}}$$

$$\textsc{select-or4}\ \frac{}{S \cup \{R \Rightarrow_c or(A,B)\} \vdash select : \left\{ \begin{array}{llll} \{true \Rightarrow_c A, & true \Rightarrow_c B, & R \equiv true\}, \\ \{true \Rightarrow_c A, & false \Rightarrow_c B, & R \equiv true\}, \\ \{false \Rightarrow_c A, & true \Rightarrow_c B, & R \equiv true\}, \\ \{false \Rightarrow_c A, & false \Rightarrow_c B, & R \equiv false\} \end{array} \right\}}$$

Le découpage en 3 sous-cas est dit paresseux, car lorsque la disjonction est supposée vraie, au plus une des deux branches A ou B est explicitement explorée. La sélection en 4 cas de test correspond à la table de vérité de l'opérateur : il s'agit d'un découpage exhaustif du problème selon les quatres combinaisons possibles.

Sélection du cycle initial Il existe des règles de sélection pour les différentes constructions du langage. Nous nous intéressons en particulier à l'opérateur d'initialisation, car il sera amené à être modifié par la suite (§**??**). La sélection basée sur un découpage structurel d'un opérateur « → » est applicable à tous les opérateurs dont le filtrage est bloqué par le statut du cycle maximal. En fait, la règle qui concerne ces opérateurs ne fait intervenir que la contrainte *cyclemax*, qui agit indirectement sur les opérateurs qui nous intéressent :

$$\textsc{select-init}\ \frac{St \in \mathbb{V}}{S \cup \{cyclemax(c,St)\} \vdash select : \{\{St \equiv initial\}, \{St \equiv noninitial\}\}}$$

3.3.5.3 Découpage fonctionnel

Il arrive que le découpage structurel soit inefficace ou insuffisant pour explorer les cas d'utilisations d'un système. En effet, les seuls découpages prédéfinis sont ceux qui suivent la description du modèle, à travers les expressions de définitions des flots et éventuellement les invariants du système. Le découpage fonctionnel permet au testeur d'introduire de nouvelles expressions Lustre qui traduiront les hypothèses faites dans chaque sous-cas de test. Chacune de ces hypothèses, formulée par une expression booléenne, agit comme un sous-objectif de test que l'on peut introduire dans le CSP. De même que l'objectif de test, chaque cas du découpage fonctionnel est alors indépendant de la structure des flots de données qui mettent en œuvre le système et peut servir à décrire un scénario de haut-niveau que l'on souaite provoquer.

La syntaxe LUSTRE enrichie admise par GATeL comporte les directives `split` que l'on a déjà présentées brièvement. Il s'agit d'une directive de découpage générique qui permet au testeur de définir un dépliage en n sous-cas de tests ($n > 0$), en fonction d'un flot de contrôle. Dans la règle ci-dessous, le flot de contrôle est f :

$$\text{SELECT-SPLIT}\ \frac{L = [C_1, \ldots, C_n] \qquad n \in \mathbb{N}^* \qquad c \in \mathbb{N} \qquad V \in \mathcal{T}}{S \cup \{f \mapsto_c V, split(f,L)\} \vdash select : \{\{true \Rightarrow_c C_i\} \mid 0 \leq i \leq n\}}$$

La règle signifie que lorsque la variable f est présente dans la grille de valeurs à un cycle c, et qu'elle y est associée à une variable (et non une valeur), alors le cas de test courant peut-être découpé en n cas. Chaque nouveau CSP sélectionné introduit une contrainte *propage* où l'expression C_i correspondante est présumée vraie au cycle c. Ainsi, on contraint davantage le système pour faire en sorte que le sous-objectif soit bien vérifié.

Contrairement au découpage structurel ou les sous-cas sont prédéfinis, on ne peut pas garantir ici que les sous-cas recouvrent bien le CSP original (des solutions peuvent être perdues), ni que les différents cas sont disjoints les uns des autres (des solutions identiques peuvent être produites dans différents sous-problèmes). En revanche, la directive permet d'exprimer n'importe quel critère de couverture fonctionnel qui admet une description sous forme de flots LUSTRE. Nous verrons comment cette directive a été modifiée et comment nous l'utilisons pour définir de nouveaux critères de couverture dans les chapitres qui suivent.

3.3.6 Exemple de génération

Nous reprenons l'exemple du robot de la page 70, en le simplifiant légèrement pour ne pas traiter les types énumérés. Pour cela, nous regroupons la simulation et le contrôleur en nous passant de la variable intermédiaire « `mvt` » de type `mouvement`. La position initiale est toujours 10, et cette fois nous exprimons directement la vitesse de déplacement du robot en fonction de la différence entre sa position et la consigne. À la différence du modèle original, nous ajoutons ici une précondition sur la variable d'entrée `cible`, à l'aide d'une directive `assert`. L'expression indique que la position à atteindre est toujours strictement positive, ce qui nous servira uniquement pour illustrer le mécanisme de propagation des invariants. La spécification LUSTRE mono-horloge est donnée à la figure 3.8.

3.3.6.1 Problème de génération de séquences de test

Afin d'avoir un PGST pour le nœud `simulation`, on doit ajouter un objectif de test et les paramètres de la génération. Dans notre exemple, nous fixons l'intervalle par défaut des entiers à $-100..100$; de même, la borne temporelle est fixée à 20 cycles de calculs au maximum. L'objectif du test est de produire une séquence où le robot atteint la position 12 et s'y arrête. Finalement, nous ajoutons une directive `split` qui propose un découpage

```
node simulation(cible: int)
returns (pos: int; vit: int)
let
    vit = if (cible = pos)
          then 0
          else if (cible > pos)
               then 1
               else -1 ;

    pos = (10 → ( pre(pos) + pre(vit) );
    assert (cible > 0);
tel
```

FIGURE 3.8: Modèle sous test.

fonctionnel du modèle : aux cycles où la cible du robot est propagée dans le modèle, on peut découper le cas de test courant en deux, selon que celle-ci a changé ou non par rapport au cycle précédent [4]. Ce découpage est bien un découpage fonctionnel, car il n'est pas lié à l'implémentation du contrôleur, mais basé sur des cas d'utilisation que l'on veut distinguer. La figure 3.9 récapitule les directives GATeL ajoutées au modèle pour que celui-ci soit un PGST valide.

3.3.6.2 Problème de satisfaction de contraintes initial

Notre problème précédent est transformé en un CSP, selon la fonction de traduction vue à la section §3.2.8. Soit S_0 le CSP associé à notre problème. On y retrouve l'ensemble des déclarations sous la forme de termes du langage $\mathcal{C}_1^g$:

4. sauf au cycle initial, où la directive n'introduit pas de sous-objectif intéressant.

```
reach (pos = 12) and (vit = 0);
maxsize     20 ;
minint    -100 ;
maxint     100 ;
split cible with [ true → (pre(cible) = cible) ;
                   true → (pre(cible) ≠ cible) ];
```

FIGURE 3.9: Extensions du modèle pour décrire le PGST qui nous intéresse.

maxsize(20),
minmax(−100,100),
type(cible, int), type(pos, int), type(vit, int),
io(cible, in), io(pos, out), io(vit, out),
equa(pos, init(10, add(pre(pos), pre(vit)))),
equa(vit, if(eq(cible,pos), 0, if(gt(cible,pos), 1, −1))),
assert(gt(cible,0)),
split(cible, [init(true, eq(pre(cible),cible)),
init(true, neq(pre(cible),cible))]),
reach(and(eq(pos,12), eq(vit,0))),
cyclemax(−1,noninitial)

Nous présentons la propagation de contraintes réalisée à partir du CSP initial S_0 et jusqu'à atteindre un premier point fixe. Nous verrons ensuite un exemple de sélection de cas de test structurelle, puis fonctionnelle, avant de produire une séquence de test. Dans les réécritures, nous noterons les CSP sous la forme $S \cup \Delta_i$ avec i un entier où S est la partie du CSP que l'on souhaite réécrire et Δ_i contient implicitement l'ensemble des autres contraintes du système.

3.3.6.3 Première propagation

L'ensemble S_0 admet deux contraintes filtrables, *cyclemax* et *reach*. La première des deux fait avancer le cycle maximal et introduit les invariants du modèle au cycle final.

Contrainte de cycle

$$
\begin{array}{rcl}
S_0 & = & \{\, \mathit{cyclemax(-1,noninitial)} \,\} \cup \Delta_0 \\
 & \xrightarrow{*} & \{\, \mathit{cyclemax(0,}ST_0\mathit{)}, ST_0 \in D_{\texttt{status}}, \mathit{true} \Rightarrow_0 \mathit{gt(cible,0)} \,\} \cup \Delta_0 \\
 & = & \{\, \mathit{true} \Rightarrow_0 \mathit{gt(cible,0)} \,\} \cup \Delta_1 \\
 & \xrightarrow{*} & \{\, V_0 \Rightarrow_0 \mathit{cible}, V_0 \in -100..100, \mathit{gt(}V_0\mathit{,0)} \,\} \cup \Delta_1 \\
 & \xrightarrow{*} & \{\, \mathit{cible} \mapsto_0 V_0, V_0 \in 1..100 \,\} \cup \Delta_1 \\
 & = & S_1
\end{array}
$$

La première réécriture introduit la variable de statut ST_0 et lui affecte le domaine par défaut, à savoir $D_{\texttt{status}} = [\mathit{noninitial}, \mathit{initial}]$. Par ailleurs, une nouvelle contrainte *propage* apparaît, qui cherche à s'assurer que l'invariant « `cible > 0` » est bien vérifié au cycle final. Le filtrage de la contrainte propage chaque membre de l'inégalité, à savoir l'identifiant *cible* et la valeur zéro : la contrainte « $\mathit{true} \Rightarrow_0 \mathit{gt(cible,0)}$ » est remplacée par la contrainte statique « $\mathit{gt(}V_0\mathit{,0)}$ », où V_0 la variable associée au membre gauche de l'inégalité. En propageant V_0 dans ce sous-terme, le filtrage introduit la variable V_0 dans

la grille de valeurs pour l'entrée `cible` au cycle zéro. Finalement, la contrainte statique d'inégalité réduit le domaine de valeurs à 1..100. On note S_1 le CSP résultant.

Objectif de test La contrainte *reach(Exp)* est présente dans l'ensemble S_1, avec :

$$Exp = and(eq(pos,12),\ eq(vit,0))$$

On la remplace d'abord par une contrainte *objective(Exp)*, pour mémoriser l'objectif de test sans le réintroduire dans le système, ainsi qu'un contrainte *propage* qui s'assure que l'objectif est réalisé au cycle zéro.

$$\begin{array}{rcl} S_1 & = & \{ reach(Exp) \} \cup \Delta_2 \\ & \xrightarrow{*} & \{ objective(Exp), true \Rightarrow_0 Exp \} \cup \Delta_2 \\ & = & \{ true \Rightarrow_0 and(eq(pos,12),eq(vit,0)) \} \cup \Delta_3 \\ & \xrightarrow{*} & \{ true \Rightarrow_0 eq(pos,12), true \Rightarrow_0 eq(vit,0) \} \cup \Delta_3 \\ & \xrightarrow{*} & \{ 12 \Rightarrow_0 pos, 0 \Rightarrow_0 vit \} \cup \Delta_3 \\ & = & S_2 \end{array}$$

On obtient ainsi l'ensemble S_2. La propagation dans l'opérateur *and* introduit deux contraintes dynamiques, car les deux sous-termes doivent nécessairement être vrais pour que l'objectif le soit. La propagation de *true* dans les deux égalités sont transformées en deux contraintes *propage* :

$$0 \Rightarrow_0 vit \qquad\qquad 12 \Rightarrow_0 pos$$

Nous nous intéressons maintenant à la propagation de ces deux contraintes, jusqu'à atteindre un point fixe.

Vitesse nulle La contrainte « $0 \Rightarrow_0 vit$ » impose que la vitesse de déplacement du robot soit nulle au cycle 0. Elle agit sur un identifiant de flot ce qui introduit la valeur zéro dans la grille de valeurs, pour le flot `vit` au cycle 0. De plus, comme on vient d'introduire la valeur dans la grille et que la variable admet une équation de définition, que l'on note E_v, cette même valeur zéro est propagée dans l'expression. Celle-ci consiste en deux opérateurs `if` imbriqués, et nous propageons une variable libre V_2 dans la condition de celui de plus haut-niveau ; cette variable remplace le terme de la condition du `if` :

$$\begin{array}{rcl} S_2 & = & \{ 0 \Rightarrow_0 vit \} \cup \Delta_4 \\ & \xrightarrow{*} & \{ vit \ \mapsto_0 0, 0 \Rightarrow_0 E_v \} \cup \Delta_4 \\ & = & \{ 0 \Rightarrow_0 if(eq(cible,pos),0,if(gt(cible,pos),\ 1,\ -1)) \} \cup \Delta_5 \\ & \xrightarrow{*} & \{ V_2 \Rightarrow_0 eq(cible,pos), 0 \Rightarrow_0 if(V_2,0,if(gt(cible,pos),\ 1,\ -1)) \} \cup \Delta_5 \end{array}$$

On rappelle que le filtrage de l'opérateur `if` est paresseux, au sens où on n'introduit pas les branches alternatives tant que la condition n'est pas connue. Nous suivons

maintenant le terme « $V_2 \Rightarrow_0 eq(cible,pos)$ » :

$$\begin{array}{rcl} S_2 & \xrightarrow{*} & \{V_3 \Rightarrow_0 cible, V_4 \Rightarrow_0 pos, eq(V_2,V_3,V_4)\} \cup \Delta_6 \\ & \xrightarrow{*} & \{V_3 \equiv V_0, V_4 \equiv 12, eq(V_2,V_3,V_4)\} \cup \Delta_6 \\ & \xrightarrow{*} & \{eq(V_2,V_0,12)\} \cup \Delta_6 \\ & = & S_3 \end{array}$$

Chaque sous-terme est associé à une variable fraîche au cycle 0 et une contrainte statique « *eq* » à trois arguments les relie. Les sous-termes sont *cible* et *pos*, qui renvoient à des flots qui sont déjà représentés dans la grille de valeurs au cycle 0, le premier par la propagation de l'invariant et le deuxième par celle de l'objectif de test. On peut donc unifier les variables nouvellement créées avec les termes associés à ces flots et éviter d'insérer de nouvelles contraintes dynamiques dans le CSP. La contrainte d'égalité reste quant-à-elle non filtrée pour le moment.

Ainsi, l'exploration de la condition ne nous permet pas de déterminer quelle branche choisir dans la contrainte *propage* associée à l'opérateur « `if` » précédent. On essaye alors de réaliser une sur-approximation des sous-termes de cette expression :

$$if(V_2,0,if(gt(cible,pos),\ 1,\ -1))$$

Cependant, les encadrements respectifs de 0 et « *if(gt(cible,pos), 1, -1)* » sont les domaines 0..0 et $-1..1$, qui sont tous les deux compatibles avec la valeur 0 (on ne peut pas en éliminer un). Le domaine $-1..1$ est construit comme une union des domaines des cas -1 et 1, et bien que 0 ne soit pas une valeur possible dans la branche *else*, la représentation des domaines ne permet pas de s'en rendre compte. Ainsi, on ne peut pas filtrer davantage à partir de la contrainte « $0 \Rightarrow_0 vit$ ». Nous nous intéressons ensuite à l'autre contrainte introduite par l'objectif.

Position finale La position finale vaut 12 d'après la contrainte « $12 \Rightarrow_0 pos$ ». Comme la variable `pos` admet une équation de définition et n'est pas encore liée dans la grille de valeurs au cycle 0, on propage cette valeur 12 dans l'expression. On réalise en même temps l'affectation dans la grille de valeurs.

$$\begin{array}{rcl} S_3 & = & \{12 \Rightarrow_0 pos\} \cup \Delta_7 \\ & \xrightarrow{*} & \{pos \mapsto_0 12, 12 \Rightarrow_0 init(10,\ add(pre(pos),\ pre(vit)))\} \cup \Delta_7 \end{array}$$

Malheureusement, le statut ST_0 n'est toujours pas connu, ce qui limite la propagation de contraintes. En effet, dans le cas de l'opérateur d'initialisation, on s'empêche d'introduire les contraintes propres à chacune des branches possibles tant qu'on ne sait pas si le cycle étudié est ou non le cycle initial.

Cependant, on peut réaliser une sur-approximation de ces branches en évaluant les deux termes. On a alors, dans le cas de la sur-approximation du membre gauche (cas du

cycle initial) l'encadrement $10 \approx_0 10..10$, alors que le résultat propagé lui est la constante 12 pour qui l'encadrement trivial est $12 \approx 12..12$. Or, l'intersection $10..10 \cap 12..12$ est vide, ce qui signifie que le membre gauche est incompatible avec le résultat. Cela est naturel, car on sait que la position de départ du robot est 10, alors que l'on souhaite atteindre la position 12 dans notre objectif de test.

On élimine donc l'incertitude de la propagation de contraintes en introduisant l'unification « $ST_0 \equiv noninitial$ », ce qui provoque le décalage du cycle contenu dans *cyclemax*, crée une nouvelle variable de statut ST_1 et ajoute l'invariant « `cible > 0` » au cycle 1 (comme précédemement). Ainsi, le statut est connu au cycle zéro (il est non-initial) et la valeur 12 peut descendre dans le membre droit de l'opérateur « → ». Ceci introduit une contrainte *add* statique et explore les valeurs précédentes de *pos* et *vit*.

$$\begin{array}{rcl} S_3 & \xrightarrow{*} & \{ true \Rightarrow_0 add(pre(pos),pre(vit)) \} \cup \Delta_7 \\ & \xrightarrow{*} & \{ V_5 \Rightarrow_0 pre(pos), V_6 \Rightarrow_0 pre(vit), add(12,V_5,V_6) \} \cup \Delta_7 \\ & \xrightarrow{*} & \{ V_5 \Rightarrow_1 pos, V_6 \Rightarrow_1 vit, add(12,V_5,V_6) \} \cup \Delta_7 \end{array}$$

On réalise la propagation au cycle 1 de V_5 dans l'équation E_p définissant « `pos` » et de V_6 dans celle du flot « `vit` » :

$$\begin{array}{rcl} S_3 & = & \{ V_5 \Rightarrow_1 pos, V_6 \Rightarrow_1 vit \} \cup \Delta_8 \\ & \xrightarrow{*} & \left\{ \begin{array}{l} pos \mapsto_1 V_5, V_5 \in -100..100, V_5 \Rightarrow_1 E_p, \\ vit \mapsto_1 V_6, V_6 \in -100..100, V_6 \Rightarrow_1 E_v \end{array} \right\} \cup \Delta_8 \\ & = & S_4 \end{array}$$

Or, on chosit de décomposer pour le moment ce dernier ensemble S_4 ainsi :

$$S_4 = \{ add(12,V_5,V_6), V_5 \in -100..100, V_6 \in -100..100 \} \cup \Delta_9$$

En effet, bien que la propagation des identifiants introduit des domaines par défaut pour les variables V_5 et V_6, à savoir le domaine $-100..100$, on constate par sur-approximation de l'expression E_v que le domaine de V_6 peut-être réduit à $-1..1$. On applique la règle « PROP-$\approx$ » vue page 100. Alors, la réduction de domaine à $-1..1$ pour V_6 est projetée sur le domaine de V_5. En effet, on a $V_5 = 12 - V_6$ et $V_6 \in -1..1$, ce qui nous donne : $12 - 1 \leq 12 - V_6 \leq 12 + 1$. Autrement dit, $V_5 \in 11..13$.

$$\begin{array}{rcl} S_4 & \xrightarrow{*} & \{ add(12,V_5,V_6), V_6 \in -1..1, V_5 \in 11..13 \} \cup \Delta_9 \\ & = & S_5 \end{array}$$

La contrainte « $V_6 \Rightarrow_0 vit$ » ne donne pas d'autre information. Comme au cycle 0, la propagation de contrainte est bloquée sur la définition du flot « `vit` ».

Position au cycle 1 La définition du flot pos est maintenant explorée au cycle 1. Comme précédemment, le statut est pour le moment indéterminé à ce cycle et la contrainte *propage* est bloquée sur l'opérateur « → ». Cette fois, l'encadrement du résultat attendu est celui de V_5, à savoir 11..13. Or, ce domaine n'est toujours pas compatible avec la valeur 10 du membre gauche. On peut donc encore une fois propager V_5 dans le membre droit, après avoir mis à jour le statut de cycle ; on a alors *cyclemax(2,*ST_2*)* dans le CSP.

On rappelle que le membre droit est *add(pre(pos),pre(vit))*. On obtient une nouvelle contrainte « *add* » statique à trois arguments, où cette fois le résultat est V_5. On sait par ailleurs que la variable représentant *pre(vit)* a pour domaine $-1..1$, ce qui nous permet après filtrage des contraintes de déduire le domaine 10..14 pour la valeur de pos au cycle 2.

Position au cycle 2 Le domaine de valeurs est cette fois compatible avec les deux branches de l'opérateur « → ». En effet, le cas où la position vaut 10 peut correspondre aussi bien au cycle initial qu'à un cycle quelconque de calcul.

Point fixe Une fois toutes ces contraintes propagées, nous avons atteint un premier point fixe, que l'on note S_6. Il reste des contraintes non filtrables, notamment celles concernant la valeur de `vit` aux différents cycles et qui reflètent les relations entre la consigne `cible` et la position courante pos.

Nous obtenons finalement les relations suivantes, où l'on représente la grille de valeurs et les domaines des variables qui s'y trouvent :

$$S_6 = \left\{ \begin{array}{lll} cible \mapsto_2 V_8, & cible \mapsto_1 V_7, & cible \mapsto_0 V_0, \\ V_8 \in 1..100 & V_7 \in 1..100 & V_0 \in 1..100, \\ pos \mapsto_2 V_9, & pos \mapsto_1 V_5, & pos \mapsto_0 12, \\ V_9 \in 10..14, & V_5 \in 11..13, & \\ vit \mapsto_2 V_{10}, & vit \mapsto_1 V_6, & vit \mapsto_0 0, \\ V_{10} \in -1..1, & V_6 \in -1..1 & \end{array} \right\} \cup \Delta_{10}$$

Les variables V_7 à V_{10} n'avaient pas été nommées jusqu'ici, elles correspondent aux variables du CSP que l'on introduites lors des différents filtrages.

Il existe différentes contraintes *propage* qui ne peuvent pas être filtrées. Au cycle 2, le filtrage est bloqué car dans l'expression E_p qui définit *pos*, on ne peut pas déterminer si le cycle est ou non initial :

$$V_9 \Rightarrow_2 init(10,\ add(pre(pos),\ pre(vit)))$$

Il existe aussi trois contraintes associées à E_v, issues des propagations dans l'équation de

« vit » aux différents cycles.

$$V_{10} \Rightarrow_2 \textit{if}(V_{12},0,\textit{if}(\textit{gt}(\textit{cible},\textit{pos}),\ 1,\ -1))$$

$$V_6 \Rightarrow_1 \textit{if}(V_{11},0,\textit{if}(\textit{gt}(\textit{cible},\textit{pos}),\ 1,\ -1))$$

$$0 \Rightarrow_0 \textit{if}(V_2,0,\textit{if}(\textit{gt}(\textit{cible},\textit{pos}),\ 1,\ -1))$$

Dans les trois cas, la valeur de la condition du `if` a été remplacée par une variable, et celle-ci a elle même été propagée dans l'expression booléene respective. Finalement, ces trois propagations de la condition du `if` ont donné trois contraintes d'égalité qui elles aussi ne peuvent pas être filtrées :

$$\textit{eq}(V_{12},\ V_8,\ V_9) \qquad \textit{eq}(V_{11},\ V_7,\ V_5) \qquad \textit{eq}(V_2,\ V_0\ ,12)$$

Les trois variables V_{12}, V_{11} et V_2 représentent des valeurs booléennes :

$$V_{12} \in D_{\text{bool}} \qquad V_{11} \in D_{\text{bool}} \qquad V_2 \in D_{\text{bool}}$$

Enfin, il reste deux contraintes arithmétiques :

- Position au cycle 1 : $\textit{add}(V_5,\ V_9,\ V_{10})$
- Position au cycle 0 : $\textit{add}(12,\ V_5,\ V_6)$

3.3.6.4 Dépliage structurel : cycle initial

Nous allons générer une séquence dans le cas où le cycle 2 est le cycle initial. En effet, l'ensemble S_6 contient une contrainte $\textit{cyclemax}(2,ST_2)$ avec ST_2 une variable de statut dont la valeur est indéterminée. Le CSP permet de réaliser un découpage structurel du PGST courant en deux sous-cas :

$$S_6 \vdash \textit{select} : \{\{ST_2 \equiv \textit{initial}\}, \{ST_2 \equiv \textit{noninitial}\}\}$$

On définit alors S_7 comme étant le cas de test qui nous intéresse :

$$S_7 = S_6 \cup \{ST_2 \equiv \textit{initial}\}$$

Une fois cette hypothèse supplémentaire faite, on constate que de nouvelles propagations peuvent s'enchaîner.

Contraintes d'addition Tout d'abord, la valeur de pos au cycle 2 est connue car la position initiale vaut 10 :

$$\begin{array}{rcl} S_7 & = & \{V_9 \Rightarrow_2 \textit{init}(10,\ \textit{add}(\textit{pre}(\textit{pos}),\ \textit{pre}(\textit{vit})))\} \cup \Delta_{11} \\ & \xrightarrow{*} & \{V_9 \Rightarrow_2 10)\} \cup \Delta_{11} \\ & \xrightarrow{*} & \{V_9 \equiv 10\} \cup \Delta_{11} \\ & = & S_8 \end{array}$$

L'unification à la valeur 10 se répercute dans la première contrainte d'addition :

$$S_8 \xrightarrow{*} \{add(V_5, 10, V_{10}), V_{10} \in -1..1, V_5 \in 11..13\} \cup \Delta_{12}$$

Or, d'après les domaines déjà présents, $V_{10} + 10$ est forcément contenu dans l'intervalle 9..11. Le seul cas possible pour que V_5, dont le domaine est 11..13, soit égale à ce résultat est de lui donner la valeur 11, ce qui implique aussi que V_{10} soit égale à 1.

$$\begin{array}{rcl} S_8 & \xrightarrow{*} & \{add(V_5, 10, V_{10}), V_{10} \in -1..1, V_5 \equiv 11\} \cup \Delta_{12} \\ & \xrightarrow{*} & \{add(11, 10, V_{10}), V_{10} \in -1..1\} \cup \Delta_{13} \\ & \xrightarrow{*} & \{V_{10} \equiv 1\} \cup \Delta_{14} \end{array}$$

Maintenant que l'on sait que V_5 vaut 11, le même raisonnement est appliqué à la deuxième contrainte d'addition, qui force la variable `vit` à valoir 1 au cycle 1. Cette valeur était représentée jusqu'ici par V_6 :

$$\begin{array}{rcl} S_8 & \xrightarrow{*} & \{add(12, 11, V_6), V_6 \in -1..1\} \cup \Delta_{15} \\ & \xrightarrow{*} & \{V_6 \equiv 1\} \cup \Delta_{16} \\ & = & S_9 \end{array}$$

Opérateurs conditionnels Maintenant que la vitesse est connue aux cycle initial (2) et au cycle suivant (1), les propagations qui étaient bloquées au niveau des opérateurs conditionnels peuvent se poursuivre. On avait précédemment :

$$S_7 = \left\{ \begin{array}{l} 1 \Rightarrow_2 if(V_{12},0,if(gt(cible,pos), 1, -1)), eq(V_{12}, V_8, V_9) \\ 1 \Rightarrow_1 if(V_{11},0,if(gt(cible,pos), 1, -1)), eq(V_{11}, V_7, V_5) \end{array} \right\} \cup \Delta_{17}$$

Nous avons regroupé ci-dessus les deux propagations dans les opérateurs `if` ainsi que les contraintes d'égalités. Or, cette fois, le résultat 1 est incompatible avec la sous-expression empruntée quand la condition est vraie. Les variables V_{11} et V_{12} peuvent donc être unifiées à *false*, et les deux contraintes *propage* sont remplacées par des contraintes qui propagent la valeur 1 dans le deuxième sous-terme (le cas *else*) :

$$\begin{array}{rcl} S_9 & \xrightarrow{*} & \left\{ \begin{array}{l} 1 \Rightarrow_2 if(false,0,if(gt(cible,pos), 1, -1)), \\ 1 \Rightarrow_1 if(false,0,if(gt(cible,pos), 1, -1)) \end{array} \right\} \cup \Delta_{18} \\ & \xrightarrow{*} & \left\{ \begin{array}{l} 1 \Rightarrow_2 if(gt(cible,pos), 1, -1), \\ 1 \Rightarrow_1 if(gt(cible,pos), 1, -1) \end{array} \right\} \cup \Delta_{18} \end{array}$$

On traite alors les opérateur `if` imbriqués, ce qui introduit des variables pour représenter les conditions respectives :

$$S_9 \xrightarrow{*} \left\{ \begin{array}{l} 1 \Rightarrow_2 if(V_{13}, 1, -1), V_{13} \Rightarrow_2 gt(cible,pos), \\ 1 \Rightarrow_1 if(V_{14}, 1, -1), V_{14} \Rightarrow_1 gt(cible,pos) \end{array} \right\} \cup \Delta_{18}$$

Ces variables sont directement unifiées à *true* car cette fois encore, la sur-approximation nous aide : dans les deux cas, seule la branche *then* est compatible avec la valeur 1. Finalement, les contraintes d'inégalités sont introduites entres les termes qui représentent `cible` et `pos`, aux cycles 1 et 2, et aboutissent à une réduction de domaine pour `cible` pour ces deux cycles.

$$\begin{array}{rcl} S_9 & \xrightarrow{*} & \left\{ \begin{array}{l} gt(V_8,\ 10), V_8 \in 1..100, \\ gt(V_7,\ 11), V_7 \in 1..100 \end{array} \right\} \cup \Delta_{19} \\ & \xrightarrow{*} & \{V_8 \in 11..100, V_7 \in ..12..100\} \cup \Delta_{19} \\ & = & S_{10} \end{array}$$

Par ailleurs, les contraintes précédentes d'égalité entre `cible` et `pos` sont retirées du CSP grâce aux domaines de valeurs que l'on vient de produire : aux deux premiers cycles de la séquence, la valeur de `cible` est supérieure à celle de `pos`, et ne sont donc pas égales.

Le CSP obtenu, S_{10}, est le point fixe par propagation obtenu après la sélection du cas de test. L'hypothèse que l'on a ajoutée concernant le statut du cycle 2 a permis de retrouver les seules valeurs possibles pour la position au cours du temps, et donc aussi la direction du déplacement (flot `vit`).

Il reste encore quelques contraintes. Parmi les domaines de valeurs, on a :

$$V_8 \in 11..100 \qquad V_7 \in ..12..100 \qquad V_0 \in 1..100 \qquad V_2 \in D_{\texttt{bool}}$$

On rappelle que V_7 et V_8 sont les valeurs de `cible` aux cycles 1 et 2, respectivement. La variable V_0 est la valeur de `cible` au cycle zéro, qui n'est toujours pas connue bien qu'elle soit nécessairement égale à zéro (afin que `vit` soit nulle). La variable V_2 contient le résultat du test d'égalité entre `cible` et `pos`. Les deux contraintes associées à ces variables sont :

$$0 \Rightarrow_0 if(V_2,0,if(gt(cible,pos),\ 1,\ -1)) \qquad eq(V_2,\ V_0,\ 12)$$

Nous montrons un exemple de découpage fonctionnel puis montrons comment une séquence est finalement générée à partir du CSP courant.

3.3.6.5 Découpage fonctionnel

L'ensemble S_{10} contient entre autres les termes suivants :

$$cible \mapsto_1 V_7 \qquad split(cible,\ [\ C_1,\ C_2\ ])$$

… où les deux cas possibles sont :

$$\begin{array}{rcl} C_1 & = & init(true,\ eq(pre(cible),cible)) \\ C_2 & = & init(true,\ neq(pre(cible),cible)) \end{array}$$

D'après la règle SELECT-SPLIT vue page 115 :

$$S_{10} \vdash \mathit{select} : \{\{\mathit{true} \Rightarrow_1 C_1\}, \{\mathit{true} \Rightarrow_1 C_2\}\}$$

On choisit de sélectionner le premier cas, qui signifie que la consigne du robot n'a pas changé entre les cycles 2 et 1. Soit :

$$S_{11} = S_{10} \cup \{\mathit{true} \Rightarrow_1 C_1\}$$

Le CSP résultant admet de nouveaux filtrages. Nous ne les détaillons pas, mais finalement on réalise l'unification des variables V_7 et V_8 : toutes les occurences de V_8 sont remplacées par la variable V_7, ce qui en particulier retire la valeur 11 comme valeur acceptable pour la cible aux cycles 1 et 2 :

$$\begin{array}{rcl} S_{11} & \xrightarrow{*} & \{V_8 \equiv V_7, V_8 \in 11..100, V_7 \in 12..100\} \cup \Delta_{20} \\ & \xrightarrow{*} & \{V_7 \in 11..100, V_7 \in 12..100\} \cup \Delta_{21} \\ & \xrightarrow{*} & \{V_7 \in 12..100\} \cup \Delta_{21} \\ & = & S_{12} \end{array}$$

On a une fois encore atteint un point fixe. Le découpage fonctionnel nous a permis de décrire facilement un cas de test qui aurait été difficile à réaliser simplement par des valuations aléatoires de V_7 et V_8.

3.3.6.6 Résolution

Le point fixe S_{12} n'est pas sous forme résolue car il reste des contraintes à satisfaire :

$$S_{12} = \{0 \Rightarrow_0 \mathit{if}(V_2, 0, \mathit{if}(\mathit{gt}(\mathit{cible}, \mathit{pos}), 1, -1)), \mathit{eq}(V_2, V_0, 12)\} \cup \Delta_{22}$$

Nous allons instancier aléatoirement une des variables du système de sorte à atteindre un CSP sous forme résolue. Les variables disponibles sont les suivantes :

$$V_7 \in 12..100, V_0 \in 1..100, V_2 \in D_{\mathtt{bool}}$$

D'après l'heuristique de choix que l'on a présentée page 102, on tente d'instancier en priorité les variables booléenes, c'est-à-dire les variables associées au domaine $D_{\mathtt{bool}}$. Ici, la seule variable disponible est V_2, et on peut constater qu'il est préférable de tenter d'instancier cette variable plutôt qu'une autre.

En effet, V_7 n'est plus liée à aucune contrainte, et donc n'importe quelle valeur convient parmi celles de son domaine. L'autre variable numérique, V_0, ne conduit à une solution que dans le cas où la valeur prise est 12, ce qui permettrait au flot `vit` de valoir 0 comme l'objectif de test le spécifie. Or ce cas est rare à trouver de manière aléatoire. La valuation d'une variable avec un domaine de valeurs très petit, comme V_2, permet de converger plus rapidement vers une solution.

Première valuation On suppose que l'on choisit la variable V_2 et que l'on tente de l'instancier avec la valeur *false* :

$$S_{12} \vdash \mathit{heuristics}(V_2, \mathit{false})$$

On définit alors :

$$S_{13} = S_{12} \cup \{V_2 \equiv \mathit{false}\}$$

On obtient alors les contraintes suivantes :

$$0 \Rightarrow_0 \mathit{if(false,0,if(gt(cible,pos),\ 1,\ -1))} \qquad \mathit{eq(false,\ V_0,\ 12)}$$

La contrainte *propage* peut descendre dans le membre correspondant à la valeur de vérité *false*. L'encadrement par sur-approximation dans les sous-termes de l'opérateur conditionnel que l'on rencontre nous indique que la première alternative est incompatible, car on a « $0 \neq 1$ ». On unifie alors le résultat 0 avec la seule branche restante, qui elle vaut -1. L'unification échoue, et l'on obtient finalement l'ensemble trivialement insatisfaisable $\{\mathit{fail}\}$.

Nous avons vu ici comment la propagation peut aboutir à un CSP insatisfaisable. C'est ce qui permet en pratique à la procédure de recherche de revenir en arrière pour tenter d'autres instanciations.

Seconde valuation On définit maintenant :

$$S_{14} = S_{12} \cup \{V_2 \not\equiv \mathit{false}\}$$

La contrainte retire la valeur *false* du domaine $D_{\texttt{bool}}$ associé à la variable. Comme il ne reste plus qu'une valeur, à savoir *true*, on l'unifie avec la variable V_2. L'unification se propage dans le terme *eq(V_2, V_0, 12)*, qui unifie V_0 à la valeur 12 et permet à la contrainte *propage* associé au `if` de disparaître du CSP.

L'ensemble qui résulte de la valuation est finalement un point fixe sous forme résolue, que l'on appelle S_{15}. La seule contrainte restante est $V_7 \in 12..100$.

Séquence générée Finalement, on peut extraire une séquence évaluable (cf. §3.2.12.2) :

$$S_{15} \vdash \mathit{seq} : \{\mathit{cible}\ [0] \leftarrow 80, \mathit{cible}\ [1] \leftarrow 80, \mathit{cible}\ [2] \leftarrow 12\}$$

L'ensemble résultant donne une valuation pour l'unique flot d'entrée `cible` à travers le temps, où cette fois les cycles sont numérotés vers l'avant, avec le cycle initial valant zéro. La valeur 80 est prise aléatoirement dans le domaine de V_7. Une fois choisie, elle a été substituée à tous les endroits où elle apparaissait. La séquence respecte à la fois l'objectif de test et les deux sélections que l'on a réalisées. Il s'agit bien d'une séquence solution pour le CSP original S_0.

3.4 Conclusion

Chapitre 4

GATeL pour flots multi-horloges

Nous présentons maintenant les évolutions de GATeL pour la prise en compte des flots temporisés. Notre but est de pouvoir traiter des systèmes réactifs décrits avec des horloges et de les traiter à ce niveau d'abstraction, pour bénéficier des relations de parenté qui existent entre les flots par l'intermédiaire de leur horloges.

Changement du noyau Les changements apportés à GATeL consistent à pouvoir interpréter le langage $\mathcal{L}_+^g$ à plusieurs niveaux, comme la compilation, la sur-approximation des flots mais surtout la propagation de contraintes. En ce qui concerne la contrainte *propage*, elle doit être enrichie pour traiter les opérateurs qui n'existaient pas avant et être adaptée dans le cas des anciens opérateurs. Par exemple, l'opérateur d'initialisation doit à la fois être adapté pour traiter différents cycles initiaux et prendre en compte les flots de redémarrage asynchrones (opérateur « $\rightarrow *$ »).

On rappelle en effet que l'exploration en arrière, à partir d'un objectif de test, fait qu'on ne sait pas à l'avance sur combien de cycles la séquence à générer doit s'étendre. Dans le cas de $\mathcal{L}_1^g$, une contrainte particulière notée *cyclemax* représentait la taille des séquences de test. Avec plusieurs horloges, les flots sont intialisés à des cycles différents, ce qui rend la notion de cycle intial relative aux horloges. Nous verrons que le lien entre les horloges et les statuts de cycles initiaux admettent des propriétés intéressantes (§4.1.1) que l'on peut exploiter dans les contraintes. Pour mettre en place ces modifications dans GATeL il faut d'abord revoir la modélisation d'un PGST sous forme d'un système de contraintes.

Nous verrons à la section §4.1 (p. 120) les changements principaux apportés à la compilation d'un PGST vers $\mathcal{C}_+^g$. Ils concernent à la fois les types énumérés, les horloges et le traitement des statuts de cycles initiaux. Pour ce dernier point, nous avons généralisé la contrainte *cyclemax* pour qu'elle soit paramétrée par une horloge. Nous maintenons donc un couple formé d'un statut et d'un cycle pour chaque horloge du système ; l'ensemble des couples ainsi crées évoluent selon des règles issues des propriétés entre horloges et flots (p. 135). Dans la suite du chapitre, nous présentons les modifications pour l'opérateur de décalage `pre` à la section §4.3 (p. 151) et l'opérateur d'initialisation à la section §4.4 (p. 161). Ces deux sections sont représentatives des modifications effectuées dans l'implémentation de GATeL$_+$.

4.1 Modélisation d'un PGST avec horloges

Dans cette section, nous nous intéressons aux différences de modélisation entre les PGST mono et multi-horloges. En particulier, il existe des propriétés intéressantes que l'on souhaite exploiter entre les valeurs prises par les flots à travers le temps et la hiérarchie d'horloges (p. 120). Nous modélisons alors dans les ensembles de contraintes de $\mathcal{C}_+^g$ un ensemble de termes particulier appelé « table d'horloges » qui fait le lien entre les cycles initiaux, les flots et les horloges (p. 123). Nous donnons ensuite la grammaire attribuée construisant un ensemble de contraintes initial de $\mathcal{C}_+^g$ à partir d'un modèle du langage $\mathcal{L}_+$ (p. 125). Par ailleurs, nous introduisons quelques prédicats auxiliaires (p. 131) ainsi qu'une représentation visuelle des grilles de valeurs et de la table d'horloges (p. 132). Nous redéfinissons la contrainte *propage* ainsi que la sur-approximation des flots pour qu'elles prennent un argument supplémentaire, à savoir l'horloge (p. 133).

4.1.1 Relations entre cycles initiaux, flots et horloges

Nous revenons aux langage $\mathcal{L}_+^g$ pour voir les liens entre les horloges du langage, les flots supports et le cycle initial d'un flot dans langage. Nous exploitons ces relations pour structurer le système de contraintes associé à un PGST multi-horloges et pour construire les règles de propagations.

Soit $N \in \mathcal{L}_+$ et Γ_N une séquence évaluable dans N, f un identifiant de flot et $c \in \mathbb{N}^*$ un cycle. Si f est présent au cycle c, on note $v_{f,c}$ la valeur associée à f à ce cycle.

$$\Gamma \vdash f\,[c] \leftarrow v_{f,c}$$

Soit h une horloge de $\mathcal{C}locks_N$.

4.1.1.1 Cycle initial d'un flot

Le cycle initial d'un flot est le premier cycle auquel celui-ci admet une valeur. Or, un flot admet une valeur exactement aux cycles auxquels son horloge est présente, par définition de l'évaluation d'un flot temporisé (cf. §2.2.3.3). Ainsi, le cycle initial d'un flot correspond aussi au premier cycle auquel son horloge est présente.

Plutôt que de parler des cycles initiaux des flots, on peut raisonner sur les cycles initiaux des horloges, que l'on définit comme suit.

> **Remarque 2.** *Numérotation en avant : pour exprimer les propriétés d'horloges dans cette section, les cycles sont numérotés de manière croissante à partir du cycle initial.*

On note i_h le cycle initial associé à l'horloge h :

$$i_h = min\{c \mid \Gamma \vdash_N h : present_c\}$$

Il s'agit du plus petit cycle auquel l'horloge est présente. Ainsi, tous les flots qui partagent la même horloge h partagent aussi le même cycle initial i_h. En particulier, on a $i_{base} = 0$: Ceci a un impact sur la modélisation du problème.

4.1.1.2 Flots supports

Étant donnée la définition des horloges et du cycle initial, il y a un lien direct entre les horloges relatives et les flots supports qui les définissent. Tout d'abord, une propriété évidente est qu'un flot est évaluable à un cycle si et seulement si son horloge est présente. Par exemple, lorsqu'on propage l'objectif de test, on s'assure aussi que son horloge est présente. Dans le cas des invariants décrits par l'opérateur `assert`, l'expression invariante n'a besoin d'être vérifiée qu'aux cycles où son horloge est vraie.

Deuxièmement, si le flot support f prend la valeur de présence v au cycle c, alors le cycle initial de l'horloge (f, v) est nécessairement antérieur ou égal au cycle c (la réciproque est fausse en général) :

$$\left(v_{f,c} = v\right) \quad \Rightarrow \quad \left(i_{(f,v)} \leq c\right) \tag{4.1}$$

Nous pouvons par exemple utiliser cette propriété lors de la propagation pour faire reculer le cycle initial potentiel dans le passé dès que l'on sait que la valeur de f vaut v. La contraposé nous dit par ailleurs que si le cycle initial est supérieur à c, le flot f ne peut pas valoir v. Encore une fois, lors de la propagation, on peut utiliser ce résultat pour retirer v du domaine de f aux cycles qui précèdent le cycle $i_{(f,v)}$.

4.1.1.3 Horloges mères et filles

À travers les flots supports, les horloges sont liées entre-elles par une relation de parenté (horloges mères/filles) qui nous donnent différentes propriétés exploitables.

Présence d'une horloge mère Pour avoir (f, v) présente au cycle c, il faut par définition que l'évaluation de f au cycle c soit égal à v :

$$\Gamma \vdash_{\mathrm{N}} f =_c v$$

Cela, à son tour, nécessite que l'horloge de f soit elle-même présente, d'après la définition de la fonction d'évaluation :

$$\Gamma \vdash_{\mathrm{N}} h(f) : present_c$$

On sait donc que si une horloge est présente à un cycle, son horloge mère y est aussi présente. Cette propriété s'écrit comme suit, où on rappelle que $m(h)$ fait référence à l'horloge mère de h :

$$\left(\Gamma \vdash_{\mathrm{N}} h : present_c\right) \Rightarrow \left(\Gamma \vdash_{\mathrm{N}} m(h) : present_c\right) \tag{4.2}$$

Ordre des cycles initiaux Deuxièmement, on sait aussi que l'horloge mère de h est liée à un cycle initial qui est nécessairement antérieur au cycle initial de h. Autrement dit :

$$i_{m(h)} \leq i_h \tag{4.3}$$

En effet, par définition, h est présente au cycle i_h :

$$\Gamma \vdash_{\mathrm{N}} h : present_{i_h}$$

D'après la propriété 4.2 précédente, l'horloge mère est aussi présente à ce cycle :

$$\Gamma \vdash_{\mathrm{N}} m(h) : present_{i_h}$$

Or, par définition, $i_{m(h)}$ est le cycle *minimal* auquel l'horloge mère de h est présente. En tant que tel, il est inférieur à tout autre cycle de présence de $m(h)$, en particulier i_h.

Par exemple, si le cycle maximal d'une horloge h dépasse le cycle maximal de son horloge mère, on peut aussi repousser le cycle maximal de $m(h)$.

Types énumérés Soit un type énuméré T portant n valeurs t_1 à t_n et un flot f de type T et d'horloge quelconque h. De plus, soient n horloges $h_i = (f, t_i)$. Alors, au cycle initial i_h, f admet une des valeurs t_1 à t_n et ainsi, nécessairement, il existe $j \in 1..n$ tel que $i_{h_i} = i_h$. Autrement dit, lorsque toutes les valeurs possibles d'un flot support servent à définir une horloge, il existe une horloge (f, t_j) dont l'instant initial coïncide avec l'instant initial de l'horloge de h.

Nous exploitons cette propriété à la section §4.2.6.1 lorsqu'on cherche à propager le statut initial d'une horloge mère vers ses horloges filles. Comme l'horloge mère est présente, il y a des chances pour qu'une horloge fille soit présente au même cycle.

Dans le cas des automates de modes que l'on traite au chapitre suivant, une partie de la hiérarchie d'horloges est générée par compilation pour encoder une structure d'automates. Dans ce sous-ensemble d'horloges, par construction, toutes les valeurs des types énumérés définissent des horloges : il existe toujours une horloge fille présente au même cycle que l'horloge mère, qui peut même être déterminée statiquement avant de commencer la génération de tests. Il s'agit donc en pratique d'un cas que l'on peut traiter couramment et qui apporte directement une information supplémentaire au système.

4.1.1.4 Existence du cycle initial

Un des invariants essentiels lié aux horloges est qu'il existe nécessairement un cycle initial associé à chaque horloge. Dans le cas mono-horloge, la résolution de contrainte tentait d'instancier le statut de *cyclemax(,à) initial*, et en cas d'échec, introduisait de nouveaux cycles dans le passé. Cependant, en présence d'une hiérarchie d'horloges qui dépendent les unes des autres, il faut pouvoir garantir que horloge admet bien un cycle initial. Par exemple, il serait possible d'avoir un statut *initial* pour l'horloge *base* à un

cycle c mais que les relations entre les flots empêchent les horloges filles d'avoir un statut *initial* à un cycle ultérieur.

Cependant, cet invariant ne concerne que les horloges qui sont réellement utiles à la réalisation de l'objectif de test du PGST. Ainsi, tant qu'une horloge n'a jamais été introduite dans le système, on peut éviter de lui trouver un cycle intial, ce qui désactive de fait le sous-ensemble de flots qui sont cadencés sur cette horloge.

4.1.1.5 Invariants temporisés

Une des caractéristiques du langage avec horloges est que les invariants donnés par les différentes expressions `assert` d'un modèle peuvent être cadencés par des horloges différentes. La signification d'un invariant avec horloge est qu'il doit être vrai aux cycles où son horloge est présente.

Dans le cas de l'horloge de base, nous introduisons les assertions comme auparavant, lorsque le cycle associé qui lui est associé est incrémenté. Pour les horloges relatives, nous verrons que l'on peut propager les invariants à chaque cycle où l'instanciation d'un flot support d'une horloge montre que celle-ci est présente. Nous faisons en sorte de ne propager un invariant au plus une fois par cycle et par horloge (§4.2.3).

4.1.2 Modélisation de la table d'horloges

On rappelle que les horloges du langage $\mathcal{L}_+$ peuvent être soit *base* soit un couple (f, v). Dans le langage $\mathcal{C}_+^g$, nous les représentons de manière identique, c'est-à-dire soit par un atome *base* soit par un terme de la forme *(f,v)*.

4.1.2.1 Contrainte *cmax(H, c, St)*

Dans le cas mono-horloge vu au chapitre 3, le statut *initial* ou *noninitial* était porté par une contrainte *cyclemax*. Nous définissons la contrainte « *cmax(H, c, St)* », où H est une horloge, c le cycle maximal connu pour cette horloge et *St* le statut de H à ce cycle. Comme auparavant, le statut peut-être *initial* ou *noninitial*.

Il existe autant de contraintes *cmax(H, c, St)* dans un CSP que d'horloges déclarées. L'ensemble de ces contraintes *cmax* est appelé *table d'horloges*, car elles constituent un groupe cohérent de termes (et aussi parce que l'implémentation utilise un tableau). Le nombre de ces contraintes est fixe au cours des propagations, mais le contenu change pour s'adapter aux informations déduites par la génération de test.

Actions à l'instanciation du statut Les contraintes *cmax* manipulent des statuts qui peuvent être observé et/ou modifiés par d'autres contraintes. On souhaite mettre à jour la table d'horloges quand le statut est modifié. Cependant, nous avons aussi besoin, par la suite, de déclencher d'autres traitements quand le statut devient connu (notamment ajouter des contraintes *propage*).

Or, la contrainte *cmax* est retirée du CSP à la même étape de réécriture que celle où le statut est instancié : ce mécanisme est nécessaire pour empêcher une règle d'être applicable à la fois avant et après la réécriture (ce qui peut introduire trop de contraintes dans le système).

À chaque fois que l'on modifie une contrainte *cmax*, on ajoute une autre contrainte de la forme *watch(H, c, St)*. Celle-ci ne disparaît pas immédiatement du CSP est peut réaliser des traitements quand le statut est instancié.

Fonction *setcmax* Nous passons systématiquement par la fonction intermédiaire *setcmax* pour introduire les deux contraintes précédentes. Lorsqu'il existe déjà un statut pour H au cycle c, on se contente d'unifier le statut désiré avec le statut existant. Sinon, on ajoute à la fois la contrainte *cmax* et la contrainte *watch*.

$$setcmax(S, H, c, St) = \begin{cases} \{OSt \equiv NSt\} \text{ si } S \vdash cmax(H, c, OSt) \\ \{cmax(H, c, St), watch(H, c, St)\} \text{ sinon} \end{cases}$$

Celle-ci admet une version simplifié sans la variable de statut en argument. Soit S un CSP et St une variable tels que St soit libre dans S :

$$S \vdash St : fresh$$

On définit alors :

$$setcmax(S, H, c) = setcmax(S, H, c, St) \cup \{St \in D_{\texttt{status}}\}$$

Les règles pour *cmax* et *watch* sont données aux sections §4.2.5 (p. 141) et §4.2.6 (p. 144).

4.1.2.2 Table d'horloges initiale

Dans la grammaire qui suit, nous introduisons l'ensemble $CSP_{def}(C)$ qui contient les déclarations initiales que l'on doit ajouter à un CSP C. Dans le chapitre précédent (§3.2.8.2), l'ensemble CSP_{def} initialisait la contrainte *cyclemax*. Nous faisons de même ici, à ceci près qu'il existe plusieurs contraintes *cmax* à initialiser, une par horloge.

Les cycles maximaux sont tous initialisés à -1, comme précédemment. De même, le statut lié à l'horloge de base vaut directement « *noninitial* », ce qui rend la contrainte filtrable. Les autres statuts sont quant à eux laissés libres, car on n'est pas sûr à priori que les horloges soient présentes au cycle zéro (final).

Ensemble des horloges Pour un ensemble de contraintes S, on note $clocks(S)$ l'ensemble des horloges déclarées dans ce système :

$$clocks(S) = \{H \mid S \vdash clock(x,H), x \in \mathbb{A}\}$$

Ensemble des flots supports De même, nous définissons l'ensemble $ckflows(S)$ qui contient l'ensemble des flots du système qui servent à définir une horloge :

$$ckflows(S) = \{ f \mid (f, x) \in clocks(S), x \in \mathbb{A} \}$$

Ensemble $CSP_{def}(S)$ Soient un ensemble S de contraintes et $H = \{H_1, \ldots, H_n\}$ l'ensemble des horloges relatives de c, avec $n \in \mathbb{N}$:

$$H = clocks(S) \setminus \{base\}$$

On se donne aussi un ensemble $V = \{V_1, \ldots, V_n\} \subseteq \mathbb{V}$ de variables libres dans S :

$$S \vdash V_1, \ldots, V_n : fresh$$

Alors, nous définissons l'ensemble $CSP_{def}(S)$ ainsi :

$$\begin{aligned} CSP_{def}(S) &= setcmax(S,\ base,\ -1, noninitial) \\ &\cup \bigcup_{i=1}^{n} \left(setcmax(S,\ H_i,\ -1, V_i) \cup \{ V_i \in D_{\texttt{status}} \} \right) \end{aligned}$$

Celui contient un statut *noninitial* pour l'horloge de base et un statut variable pour chacune des autres horloges modélisées. Le statut est variable car il est possible qu'il n'y ait pas de cycle initial pour l'horloge dans la séquence générée : le cycle -1 représente le fait que le cycle initial peut se produire après le cycle final de la séquence générée.

4.1.3 Grammaire attribuée

Nous présentons les modifications de la grammaire attribuée vue au chapitre 3 (§3.2.8.2). Nous introduisons les types énumérés, en y intégrant les flots booléens, et ajoutons des informations d'horloges dans le CSP résultant. En particuler, les invariants sont liés à des horloges, de même que l'objectif de test et les directives `split`.

Les différents changements d'organisation qui sont mis en place au sein du CSP par la compilation d'un programme Lustre multi-horloges sont résumés ci-dessous. Nous avons donné à la section §**??** du chapitre précédent un récapitulatif de la structure d'un CSP après compilation depuis le langage mono-horloge $\mathcal{L}_1$. Un élément de $\mathcal{L}_+^g$ nous donne un CSP ayant la structure décrite auparavant, avec les modifications suivantes :

1. Représentation statique
   - *clock(F,H)* : ce nouveau terme associe un flot à son horloge.
   - *enum(T,L)* : dans ce terme clos, l'identifiant de type T est associé à la liste des valeurs symboliques L, constituée d'atomes distincts.
   - *assert(A,H)* : invariant temporisé ; A doit être vrai à chaque cycle où H est présent.

- *reach(Obj,H)* : objectif de test temporisé, où H est l'horloge de Obj.

2. Contraintes auxiliaires
   - Nous introduisons la contrainte « *cmax(H, mc, St)* » pour chaque horloge H du système, qui remplace la contrainte *cyclemax* précédente.
3. La contrainte *propage* est modifiée, elle admet un argument supplémentaire (l'horloge) et traite les nouveaux opérateurs du langage.

De même que précédemment, nous décrivons et commentons les différentes règles de production de la grammaire modifiée. Nous ne reprenons pas les règles qui sont identiques à celles de la compilation vers $\mathcal{C}_1^g$.

Modèle Nous prenons en compte maintenant les types énumérés. Les traductions des déclarations de types sont regroupées dans l'ensemble T ; celui-ci est passé en paramètre à la traduction du nœud. Celle-ci synthétise remonte l'ensemble de termes N. Finalement, la traduction M du modèle tout entier correspond à l'union de T et N.

$$Model\,(M^{\uparrow}) \quad ::= \quad Types\,(T^{\uparrow})\ Node\,(T^{\downarrow}, N^{\uparrow}) \qquad -\ M = T \cup N$$

Déclarations de types Une déclaration de type contient l'ensemble des types énumérés d'un modèle, sous la forme d'une relation entre l'identifiant du type et la liste de ses éléments. Comme le type booléen est un type énuméré particulier prédéfini, on ajoute systématiquement le terme *enum* qui lui est associé dans le résultat final.
Soit E_{bool} = *enum(bool,[false,true])* :

$$\begin{array}{rcll} Types\,(T^{\uparrow}) & ::= & Enum\,(E^{\uparrow})\ Types\,(T_0^{\uparrow}) & -\ T = \{E\} \cup T_0 \\ & | & \epsilon & -\ T = \{E_{bool}\} \end{array}$$

Type énuméré La déclaration d'un type énuméré d'identifiant **ID** , à n valeurs symboliques v_1 à v_n, donne lieu à la construction d'un terme $enum(\mathbf{ID}, [v_1, \ldots, v_n])$ associant l'identifiant à la liste ordonnée des valeurs.

$$Enum\,(E^{\uparrow}) \quad ::= \quad \texttt{type}\ \mathbf{ID}\ \texttt{= enum \{}\ EnumList\,(L^{\uparrow})\ \texttt{\}} \qquad -\ E = enum(\mathbf{ID}, L)$$

$$\begin{array}{rcll} EnumList\,(L^{\uparrow}) & ::= & \mathbf{ID}\ \texttt{,}\ EnumList\,(L_0^{\uparrow}) & -\ L = [\mathbf{ID}|L_0] \\ & | & \mathbf{ID} & -\ L = [\mathbf{ID}] \end{array}$$

Nœud Maintenant, la règle admet un argument supplémentaire, la liste des types énumérés. Celle-ci est ajoutée aux déclarations de variables pour constituer le contexte C passé à la traduction du corps du nœud et des extensions propres à GATeL.

$$
\begin{array}{lll}
\mathit{Node}(T^{\downarrow}, N^{\uparrow}) & ::= & \texttt{node}\ \mathbf{ID}\ \texttt{(}\ \mathit{Vars}(\mathit{in}^{\downarrow}, I^{\uparrow})\ \texttt{)} \\
& & \texttt{returns (}\ \mathit{Vars}(\mathit{out}^{\downarrow}, O^{\uparrow})\ \texttt{) ;} \\
& & \mathit{OptVars}(V^{\uparrow}) \\
& & \texttt{let} \\
& & \mathit{Body}(C^{\downarrow}, B^{\uparrow}) \\
& & \mathit{Extensions}(C^{\downarrow}, E^{\uparrow}) \qquad - \begin{cases} C = T \cup I \cup O \cup V \\ N = C \cup B \cup E \end{cases} \\
& & \texttt{tel ;}
\end{array}
$$

Déclaration d'une variable Les règles de production pour les déclarations de variables sont presque identiques à celles vues précédemment. Chaque déclaration définit maintenant, en plus du type d'une variable, son horloge.

$$
\mathit{Var}(E^{\downarrow}, V^{\uparrow}) \quad ::= \quad \mathbf{ID}\ \texttt{:}\ \mathit{Type}(T^{\uparrow})\ \mathit{Clk}(C^{\uparrow}) \qquad - V = \left\{ \begin{array}{l} \mathit{type}(\mathbf{ID}, T), \\ \mathit{io}(\mathbf{ID}, E), \\ \mathit{clock}(\mathbf{ID}, C) \end{array} \right\}
$$

Un type peut être donné par *int*, *bool* ou un identifiant de type énuméré.

$$
\begin{array}{lll}
\mathit{Type}(T^{\uparrow}) & ::= & \texttt{int} \qquad - T = \mathit{int} \\
& | & \texttt{bool} \qquad - T = \mathit{bool} \\
& | & \mathbf{ID} \qquad - T = \mathbf{ID}
\end{array}
$$

Déclaration d'horloge La déclaration d'une horloge suit la même syntaxe que le filtrage avec l'opérateur `when`. Les règles définissent 4 déclarations possibles : le cas général où l'identifiant du flot support et la valeur de présence sont donnés, deux notations simples dans le cas où l'identifiant est un flot booléen et le cas où le flot est cadencé sur l'horloge de base.

$$
\begin{array}{lll}
\mathit{Clk}(C^{\uparrow}) & ::= & \texttt{when}\ \mathbf{ID}_1\ \texttt{match}\ \mathbf{ID}_2 \qquad - C = \mathit{ck}(\mathbf{ID}_1, \mathbf{ID}_2) \\
& | & \texttt{when}\ \mathbf{ID} \qquad - C = \mathit{ck}(\mathbf{ID}, \mathit{true}) \\
& | & \texttt{when not}\ \mathbf{ID} \qquad - C = \mathit{ck}(\mathbf{ID}, \mathit{false}) \\
& | & \epsilon \qquad - C = \mathit{base}
\end{array}
$$

Corps d'un nœud Les équations et les assertions sont semblables aux règles précédentes. Cependant, les assertions portent maintenant une information supplémentaire, à savoir l'horloge de l'expression invariante. La signification d'un invariant exprimé par un flot temporisé est la suivante : à chaque cycle où l'invariant est évaluable, sa valeur doit être *true*.

$$
\begin{array}{lcll}
\mathit{BodyElt}(C^{\downarrow}, BE^{\uparrow}) & ::= & \textbf{ID} = \mathit{Expr}(C^{\downarrow}, E^{\uparrow}, ECk^{\uparrow}) \ ; & -\ BE = \mathit{equa}(\textbf{ID}, E) \\
& | & \texttt{assert}\ \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, ACk^{\uparrow}) \ ; & -\ BE = \mathit{assert}(A, ACk)
\end{array}
$$

Expressions Les règles propres aux expressions remontent maintenant aussi une information d'horloge. Les littéraux booléens sont regroupés dans le cas général des valeurs de types énumérés. Lorsque l'on rencontre un identifiant de flot, l'horloge déclarée de ce flot est retournée comme horloge de l'expression.

$$
\begin{array}{lcll}
\mathit{Expr}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow}) & ::= & (\ \mathit{Unop}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow})\ ) & \\
& | & (\ \mathit{Binop}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow})\ ) & \\
& | & \texttt{if}\ (\ \mathit{Expr}(C^{\downarrow}, I^{\uparrow}, H^{\uparrow})\ ) & \\
& & \texttt{then}\ \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) & \\
& & \texttt{else}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -\ E = \mathit{if}(I, A, B) \\
& | & (\ \mathit{Expr}(C^{\downarrow}, F^{\uparrow}, FH^{\uparrow})\ \mathit{When}(H^{\uparrow})\ ) & -\ E = \mathit{when}(F, FH) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) \rightarrow \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -\ E = \mathit{init}(A, B, []) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) & \\
& & \rightarrow * (\ \mathit{EList}(C^{\downarrow}, L^{\uparrow})\ ) & \\
& & \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -\ E = \mathit{init}(A, B, L) \\
& | & \texttt{merge}\ (\ \textbf{ID}\ ;\ \mathit{EList}(C^{\downarrow}, L^{\uparrow}, HL^{\uparrow}) & -\ E = \mathit{merge}(\textbf{ID}, L) \\
& & & \quad C \vdash \mathit{clock}(\textbf{ID}, H) \\
& | & \mathit{Ident}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow}) & \\
& | & \textbf{INT} & -\ E = \textbf{INT} \\
& & & \quad H = \mathit{base}
\end{array}
$$

Un identifiant peut faire référence soit à un flot, soit à une valeur d'un type énuméré. Lorsqu'il s'agit d'un flot, il existe un terme de foncteur *clock* dans C associé à l'identifiant, ce qui n'est pas le cas lorsqu'il s'agit d'une valeur symbolique.

$$
\begin{array}{lcll}
\mathit{Ident}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow}) & ::= & \textbf{ID} & -\ E = \mathit{var}(\textbf{ID}) \text{ si } C \vdash \mathit{clock}(\textbf{ID}, H) \\
& & & \quad (E, H) = (\textbf{ID}, \mathit{base}) \text{ sinon}
\end{array}
$$

Le filtrage d'un flot correspond aux trois cas que l'on vus précédemment, à savoir le cas général avec les types énumérés et ceux où le flot support est booléen.

$$
\begin{array}{lcll}
\mathit{When}(H^{\uparrow}) & ::= & \texttt{when}\ \textbf{ID}_1\ \texttt{match}\ \textbf{ID}_2 & -\ H = \mathit{ck}(\textbf{ID}_1, \textbf{ID}_2) \\
& | & \texttt{when}\ \textbf{ID} & -\ H = \mathit{ck}(\textbf{ID}, \mathit{true}) \\
& | & \texttt{when not}\ \textbf{ID} & -\ H = \mathit{ck}(\textbf{ID}, \mathit{false})
\end{array}
$$

Listes d'expressions Les listes d'expressions que l'on avait définies précédemment sont modifiées et n'acceptent plus que des identifiants de flots. En effet, une telle liste contient différentes expressions, où l'horloge de chacune d'entre elles est quelconque : elles ne sont pas forcément égales les unes avec les autres. Il faudrait se souvenir de ces horloges, par exemple pour l'opérateur de réinitialisation. Or, on mémorise les horloges associées à chaque flot du nœud. En n'autorisant que des identifiants de flots dans les listes d'expressions, on sait que l'on pourra retrouver l'horloge de chaque expression car chacune est associé à un flot nommé. Cela est important pour les flots de redémarrage, qui sont asynchrones, ou encore la directive `split`. Mais recourir aux identifiants est utile aussi dans le cas de l'opérateur `merge`, où les horloges sont pourtant toujours connues.

En effet, cet opérateur `merge` fait référence à des suites d'expressions dont les horloges ne sont pas forcément *déclarées* dans le nœud : prenons par exemple un flot booléen `b` qui est utilisé uniquement dans l' expression suivante : `merge(b,E when b, F when not b)`. Les flots `E` et `F` ont pour horloges respectives $(b, true)$ et $(b, false)$, mais on ne sait pas, après traduction en un CSP, que ces deux horloges existent. Cela pose problème dans la gestion des horloges et des cycles initiaux, que l'on détaillera par la suite. Le fait d'avoir un identifiant de flot au lieu des expressions « `E when b` » et « `F when not b` » nous assure qu'il existe des termes de la forme *clock(f,(b,true))* et *clock(g,(b,false))* dans le CSP résultant, avec f et g deux identifiants.

Comme l'introduction systématique des identifiants peut nuire à la lisibilité, les exemples que nous proposons par la suite montrent des listes d'expressions et non des listes d'identifiants. Il est implcite dans ce cas qu'il existe des flots intermédiaires associés à ces expressions. Nous évitons ainsi de devoir trop modifier la fonction de traduction.

$$\begin{array}{lcll} \mathit{EList}(C^{\downarrow}, EL^{\uparrow}) & ::= & \textsc{id} \ ; \ \mathit{EList}(C^{\downarrow}, ET^{\uparrow}) & -- \ EL = [\textsc{id}|ET] \\ & | & \textsc{id} & -- \ EL = [\textsc{id}] \end{array}$$

Nous réécrivons ici les règles des opérateurs unaires et binaires restants pour y ajouter le paramètre d'horloge aux non-terminaux. Dans les opérations binaires qui suivent, les deux membres doivent avoir la même horloge.

$$\begin{array}{lcll} \mathit{Unop}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow}) & ::= & \texttt{-} \ \mathit{Expr}(C^{\downarrow}, F^{\uparrow}, H^{\uparrow}) & -- \ E = \mathit{minus(F)} \\ & | & \texttt{not} \ \mathit{Expr}(C^{\downarrow}, F^{\uparrow}, H^{\uparrow}) & -- \ E = \mathit{not(F)} \\ & | & \texttt{pre} \ \mathit{Expr}(C^{\downarrow}, F^{\uparrow}, H^{\uparrow}) & -- \ E = \mathit{pre(F)} \end{array}$$

$$
\begin{array}{lcll}
\mathit{Binop}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow}) & ::= & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{+}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{add}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{-}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{minus}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{*}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{mult}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{div}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{div}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{mod}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{mod}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) > \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{gt}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) < \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{lt}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) \geq \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{geq}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) \leq \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{leq}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{and}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{and}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow})\ \texttt{or}\ \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{or}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) = \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{eq}(A,B) \\
& | & \mathit{Expr}(C^{\downarrow}, A^{\uparrow}, H^{\uparrow}) \neq \mathit{Expr}(C^{\downarrow}, B^{\uparrow}, H^{\uparrow}) & -- E = \mathit{neq}(A,B)
\end{array}
$$

Directives de GATeL$_+$ Les extensions de GATeL sont modifiées pour prendre en compte les horloges. Nous rappelons que dans la règle qui suit, l'ensemble résultat comporte des contraintes par défaut présentes dans le système initial. Ici, on doit mettre en place les contraintes d'horloges et l'ensemble en question est noté $CSP_{def}(C)$, car contrairement au cas mono-horloge, il dépend ici du contexte C. Nous avons défini cet ensemble à la section §4.1.2.2.

$$
\begin{array}{lcll}
\mathit{Extensions}(C^{\downarrow}, E^{\uparrow}) & ::= & \mathit{Objective}(C^{\downarrow}, O^{\uparrow}) & -- E = \{O\} \cup P \cup \\
& & \mathit{Params}(P^{\uparrow}) & \qquad S \cup CSP_{def}(C) \\
& & \mathit{Splits}\ (C^{\downarrow}, S^{\uparrow}) &
\end{array}
$$

L'objectif de test est représenté cette fois par un terme qui abrite, en plus de l'expression Lustre que l'on souhaite observer, l'horloge de celle-ci. Dans ce cas, l'objectif signifie à la fois que l'horloge est présente et que l'expression booléenne est vraie.

$$
\mathit{Objective}(C^{\downarrow}, O^{\uparrow}) \quad ::= \quad \texttt{reach}\ \mathit{Expr}(C^{\downarrow}, E^{\uparrow}, H^{\uparrow})\ \texttt{;} \qquad -- O = \mathit{reach(E,H)}
$$

Les paramètres du problème sont les mêmes que précédemment. Par ailleurs, en changeant la définition de *EList* précédemment, nous avons modifié implicitement les directives `split`. On rappelle que la directive s'exprime ainsi :

$$
\mathit{Split}(C^{\downarrow}, S^{\uparrow}) \quad ::= \quad \texttt{split}\ \textsc{id}\ \texttt{with [}\ \mathit{EList}(C^{\downarrow}, L^{\uparrow})\ \texttt{]} \qquad -- S = \mathit{split}(\textsc{id}, L)
$$

Or, comme chaque sous-cas de test représenté dans la liste d'expression est donné par un identifiant qui peut avoir son horloge propre, indépendante de l'horloge du flot de

contrôle et des horloges des autres cas de test. Lors de l'extension de GATeL aux horloges multiples, les directives `split` ont d'abord été étendues de sorte à ce que les expressions partagent toutes la même horloge. Par la suite, nous avons modifié les règles de calcul d'horloges pour la directive afin que les horloges soient indépendantes.

Cela est consistent avec le traitement de l'objectif de test principal dont l'horloge n'est pas restreinte à priori. De même, ici, chaque sous-cas peut activer une partie différente du système, sans nécessairement avoir à être synchronisé avec l'expression de définition d'un autre cas de test, qui elle est introduite dans un CSP différent lors d'un découpage fonctionnel. Cette modification nous a été aidé lors de la mise en œuvre de la couverture structurelle des automates de modes, que l'on décrit au chapitre suivant.

4.1.4 Relations entre flots et horloges

On rappelle que les relations $\hbar$ et m décrites dans le cadre du langage $\mathcal{L}_+^g$ (fonction d'évaluation) nous donnent respectivement l'horloge d'un flot et l'horloge mère d'une horloge. Elles ont un équivalent dans le langage $\mathcal{C}_+^g$ à travers les prédicats *clock* et *mother*.

4.1.4.1 Horloge d'un flot

La première est donnée directement par les termes de foncteur *clock* que l'on obtient par compilation d'un PGST multi-horloges vers un ensemble de contraintes, à savoir *clock(f,H)* pour tout identifiant de flot f avec H le terme représentant l'horloge de f. Cependant, on la notera de préférence *clock(f)=H* pour différencier les rôles des arguments.

4.1.4.2 Horloge mère d'une horloge

La seconde relation est déduite de la première à l'aide du prédicat *mother* suivant :

$$\text{MOTHER-BASE}\ \frac{}{S \vdash \mathit{mother(base)=base}} \qquad \text{MOTHER}\ \frac{H = (f, v) \qquad S \vdash \mathit{clock(f)=M}}{S \vdash \mathit{mother(H)=M}}$$

4.1.4.3 Présence d'une horloge

Nous définissons les prédicats *present*, *undefclock* et *absent* relatifs aux horloges. Ces prédicats adaptent les définitions d'une horloge présente ou absente vues au chapitre 2 en tenant compte du fait que la grille de valeurs peut-être partiellement construite.

Horloge présente Lorsque l'horloge est un couple (f, v), on dit qu'elle est présente au cycle c si f est représenté par l'atome v. Il faut donc qu'il existe un terme $f \mapsto_c v$ dans le CSP pour que ce soit le cas. L'horloge de base est quant-à-elle présente à tous les cycles.

$$\text{PRESENT-BASE}\ \frac{}{S \vdash \mathit{present(base,c)}} \qquad \text{PRESENT-REL}\ \frac{S \vdash f \mapsto_c v}{S \vdash \mathit{present((f,v),c)}}$$

Horloge indéfinie Lorsqu'il n'existe pas de tel terme $f \mapsto_c v$ dans la grille, c'est-à-dire quand on a *undef(f, c)* dans le CSP courant (voir page 96), l'horloge est indéfinie. Alors, par définition :

$$\textsc{undef-clock}\ \frac{S \vdash \mathit{undef(f,\ c)}}{S \vdash \mathit{undefclock((f,v),c)}}$$

Horloge absente Nous avons aussi parfois besoin de savoir si une horloge est absente à un cycle, de manière certaine. La grille de valeurs doit pour cela contenir suffisamment d'informations. D'une part, une horloge H est absente si son horloge mère l'est elle-même.

$$\textsc{absent-mother}\ \frac{S \vdash \mathit{mother(H)=M} \qquad S \vdash \mathit{absent(M,c)}}{S \vdash \mathit{absent(H,c)}}$$

D'autre part, si l'horloge mère est présente, H n'est absente au cycle c que s'il existe un terme $f \mapsto_c W$ avec W une variable ou un terme non-unifiable avec v.

$$\textsc{absent-clock}\ \frac{\begin{array}{ccccc} & & S \vdash \mathit{mother(H)=M} & & S \vdash \mathit{present(M,c)} \\ H = (f, v) & & S \vdash f \mapsto_c W & & S \vdash W \approx D_W \qquad v \notin D_W \end{array}}{S \vdash \mathit{absent(H,c)}}$$

Si la valeur v est dans le domaine ou si f n'est pas représenté au cycle c, on ne peut pas être sûr que l'horloge est absente.

4.1.5 Visualisation des flots et des statuts de cycle

Dans nos exemples, nous aurons besoin de représenter à la fois les statuts associés aux différentes horloges, ainsi que les valeurs ou variables déjà propagées dans le système, en même temps que leurs domaines respectifs. La figure 4.1 illustre l'ensemble des notations que nous utilisons. La première colonne est une représentation de la hiérarchie entre les flots, où sont mélangés alternativement les horloges et les flots temporisés.

- Sur une ligne où est placée une flot, par exemple f, on représente les valeurs de ce flot dans le temps, telle que donnée par la contrainte $f \mapsto_c V$. Lorsque l'élément V est une variable, on peut indiquer à côté son domaine de valeurs.
- Sur une ligne où est représentée une horloge H, on place les différentes valeurs prises par le statut *cmax(H, c, St)* dans le temps. Pour ne pas surcharger la représentation, on associe les symboles suivants aux différentes valeurs possibles du statut, pour un cycle c donné :

  « ‖ » S vaut *initial*.

  « ∘ » S est indéterminé au cycle c ; on peut aussi avoir une variable à la place.

  « × » Le statut au cycle c est, ou a été, *noninitial*.

Cycles	5	4	3	2	1	0
base	‖	×	×	×	×	×
→ f	u	$F_4 : [u, v]$	v	w	u	v
→ (f,u)	‖				×	
→ x	5				6	
→ (f,v)			∘			×
→ y			Y_3			9
→ (f,w)				‖		
→ z				$Z_2 : 0..10$		
→ m	5	M_4	Y_3	Z_2	6	9

FIGURE 4.1: Grille arbitraire illustrant la représentation du CSP

La table contient deux flots `f` et `m` cadencés sur l'horloge de base. Le flot `f` admet trois valeurs possibles, symboliques, à savoir `u`, `v` et `w`. Les flots entiers `x`, `y` et `z` sont d'horloges respectifes (f, u), (f, v) et (f, w). Le flot `m` est le résultat de l'application d'un `merge` sur ces trois flots : `m = merge(f ; x ; y ; z) ;`

On peut voir ainsi que l'horloge de base est initiale au cycle 5. Par ailleurs, au cycle 4, le domaine associé à F_4, qui représente `f` au cycle 4, exclut la valeur `w`. Comme la première occurence de `f = w` a lieu au cycle 2, on sait aussi que l'horloge (f, w) est initiale au cycle 2. En revanche, le statut de (f, v) est indéterminé au cycle 3 : selon la valeur de `f` au cycle 4, la première occurence de `v` peut-être présente soit au cycle 3 soit au cycle 4.

4.1.6 Génération de tests avec horloges

La contrainte *propage* admet un paramètre supplémentaire dans GATeL$_+$. Elle est de la forme suivante, où R est un terme ou une variable de $\mathcal{C}_+^g$, Exp une expression de flot, h l'horloge de cette expression et c un cycle.

$$propage(R, Exp, h, c)$$

Comme auparavant, on notera cette contrainte sous une forme plus compacte, à savoir :

$$R \Rightarrow_{(h,c)} Exp$$

Nous aurons besoin de faire appel à propage où le résultat propagé est une variable libre. Pour simplifier les règles à venir, nous définissons la contrainte auxiliaire *propage(f,c)*,

où f représente nécessairement un identifiant de flot et qui admet l'unique règle suivante (D_T est l'intervalle de valeurs par défaut du type T) :

$$\text{PROPAGE-2}\ \frac{S \vdash \mathit{clock(f)=H} \qquad S \vdash \mathit{type(f,T)} \qquad S \vdash V : \mathit{fresh}}{\vdash S \cup \{\mathit{propage(f,c)}\} \rightarrow S \cup \{V \Rightarrow_{(c,H)} f, V \in D_T\}}$$

4.1.6.1 Sur-approximation

En plus des contraintes *propage*, nous avons modifié la définition de la sur-approximation : La fonction de sur-approximation prend maintenant un argument supplémentaire, l'horloge du flot évalué. On note ainsi le fait que l'expression E d'horloge H admette le domaine D comme sur-approximation au cycle c :

$$S \vdash E \approx_{(c,H)} D$$

Nous utilisons cette notation par la suite.

4.1.6.2 Changements simples de *propage*

Nous ne détaillons les modifications apportées aux règles relatives à la contrainte *propage* où il suffit d'ajouter l'argument supplémentaire (l'horloge) sans que la sémantique ne soit directement influencée. Par exemple, la propagation des opérateurs binaires de Lustre est pratiquement identique à celle vue page 92 :

$$\text{PROP-BINOP-ARITH}\ \frac{\begin{array}{c} S \vdash V_A, V_B : \mathit{fresh} \\ \mathscr{R} \in \{\mathit{add}, \mathit{minus}, \mathit{mult}, \mathit{div}, \mathit{mod}, \mathit{gt}, \mathit{lt}, \mathit{geq}, \mathit{leq}\} \end{array}}{\begin{array}{l} \vdash S \cup \{R \Rightarrow_{(c,H)} \mathscr{R}(A,B)\} \\ \rightarrow S \cup \{\mathscr{R}(R,V_A,V_B), V_A \Rightarrow_{(c,H)} A, V_B \Rightarrow_{(c,H)} B\} \end{array}}$$

Cependant, une différence existe dans le traitement des flots booléens, car dans le langage $\mathcal{C}_+^g$, le type booléen de Lustre est un pris comme un type énuméré prédéfini et traité comme tel dans le filtrage. Ci-dessous, si le résultat R est associé au terme *val(v)* ou *enum(v)*, alors ce résultat doit s'unifier avec la valeur v contenue dans le terme. Elle remplace les règles précédentes qui traitaient les entiers et les booléens différemment.

$$\text{PROP-VAL-ENUM}\ \frac{\mathscr{U} \in \{\mathit{val}, \mathit{enum}\}}{S \cup \{R \Rightarrow_{(c,H)} \mathscr{U}(v)\} \rightarrow S \cup \{R \equiv v\}}$$

Il existe ainsi quelques modifications mineures entre les deux langages. Nous décrivons rapidement les traitement dans *propage* pour les nouveaux opérateurs du langage `when` et `merge`. Dans la suite du chapitre, nous nous penchons sur les filtrages plus importants qui doivent être modifiés pour supporter la sémantique du langage multi-horloges, comme `pre` et $\rightarrow *$.

Filtrage La propagation dans un terme de la forme *when(E,H_E)* s'effectue directement dans le terme E avec l'horloge H_E. Ici, on rappelle que l'horloge H_E est celle de E, qui est plus haut dans la hiérarchie que H : en effet, l'expression E d'horloge H_E est ralentie sur l'horloge H. Quand on passe du résultat filtré au flot E non filtré, on reprend l'horloge plus rapide H_E conservée dans le terme.

$$\text{PROP-WHEN}\ \frac{}{\vdash S \cup \{R \Rightarrow_{(c,H)} \mathit{when(E,H_E)}\} \rightarrow S \cup \{R \Rightarrow_{(c,H_E)} E\}}$$

Fusion de flots Dans le cas de l'opérateur `merge`, la contrainte filtrée est de la forme :

$$R \Rightarrow_{(c,H)} \mathit{merge(f,E_1,\ldots,E_n)}$$

Dans le cas trivial où la valeur de f est connue au cycle c et vaut v_i, la propagation redescend dans l'expression E_i associée à la valeur v_i ; l'horloge associée à *propage* est alors (f, v_i).

$$\text{PROP-MERGE-INST}\ \frac{S \vdash f \mapsto_c v_i \qquad v_i \in \mathbb{A}}{\vdash S \cup \{R \Rightarrow_{(c,H)} \mathit{merge(f,E_1,\ldots,E_n)}\} \rightarrow S \cup \{R \Rightarrow_{(c,(f,v_i))} E_i\}}$$

Le traitement est un peu plus complexe si f n'est pas évaluable directement. Le domaine de R peut-être incompatible avec une des sur-approximations de E_i, auquel cas on retire la valeur v_i du domaine potentiel de f au cycle c. Par ailleurs, le domaine du résultat R peut être intersecté avec l'union des sur-approximations des E_i. Nous ne détaillons pas ces règles ici.

4.1.6.3 Sélection de cas de tests

La directive `split` fonctionne comme précédemment, en ajoutant chaque nouvelle hypothèse dans un nouveau CSP (§3.3.5.3). Ici, on force les hypothèses à être portées par des identifiants de flots car ainsi ils leur horloges est accessible dans le modèle à base de contraintes (par *clock(f)=H*). Chaque cas de test fonctionnel se comporte alors comme un nouveau sous-objectif de test où l'on propage d'une part son horloge et d'autre par l'hypothèse supplémentaire comme valant *true*.

Dans les autres sections, nous nous intéressons à la sélection structurelle des opérateurs `pre` et $\rightarrow*$ aux pages respectives 160 et 175.

4.2 Cohérence de la table d'horloges

Dans cette partie, nous décrivons les règles liées à la cohérence de la table d'horloges. Ces règles consistent à s'assurer que les invariants énoncés à la section §4.1.1 sont respectés.

Cycles	9	8	7	6	5	4	3	2	1	0
base	‖	×	×	×	×	×	×	×	×	×
→ **x**	0	1	2	3	4	5	6	7	8	9
→ **h**	*H*	a	a	a	b	a	b	b	a	a
→ (h,a)		○	×	×		×			×	×
→ **xa**		1	2	3		5			8	9
→ (h,b)					○		×	×		
→ **xb**					4		6	7		

FIGURE 4.2: Grille initiale

Nous donnons tout d'abord des exemples de propagations que l'on souhaite voir réalisées, puis détaillons les règles propres aux horloges, aux statuts et aux flots supports, notamment l'insertion des invariants portés par expressions `assert`. Nous redéfinissons le filtrage de *propage* dans le cas des identifiants de flots.

4.2.1 Présentation

La table de la figure 4.2 correspond à une grille en partie générée. Elle montre un flot `x` entier, le flot `h` qui sert de support à deux horloges (h, a) et (h, b), où a et b sont les deux éléments d'un type énuméré. Les flots `xa` et `xb` sont les filtrages `x` selon les horloges respectives (h, a) et (h, b). Le cycle 9 est le cycle initial de l'horloge de base. Les statuts des deux autres horloges du système sont encore inconnus, car ils dépendent tous deux de la valeur symbolisée par H. La variable H est la représentante du flot `h` au cycle 9.

4.2.1.1 Instanciation de H

Nous choisissons d'instancier la variable H avec la valeur a. Dans la figure 4.3, ceci correspond à la lettre a encadrée en noir. Comme le flot `h` est un flot support pour l'horloge (h, a), on peut mettre à jour le statut précédent, au cycle 8, car le nouveau cycle initial *potentiel* devient le cycle 9. La croix en gris au cycle 9 montre ce changement de statut.

En mettant à jour le statut au cycle 9, on constate que l'horloge mère est initiale au même cycle. Or, on sait que l'instant initial d'une horloge mère est antérieur (ou égal) à l'instant initial de ses horloges filles. Par ailleurs le cycle c présent dans une contrainte *cmax(Ck, c, St)* est le plus grand cycle déjà propagé qui est succeptible d'être l'instant initial de l'horloge Ck. L'instant initial d'une horloge fille est donc encadré en permanence entre le cycle initial de son horloge mère et la valeur courante du cycle c. Ici, les deux sont identiques, et nécessairement, le statut de (h, a) est *initial* au cycle 9.

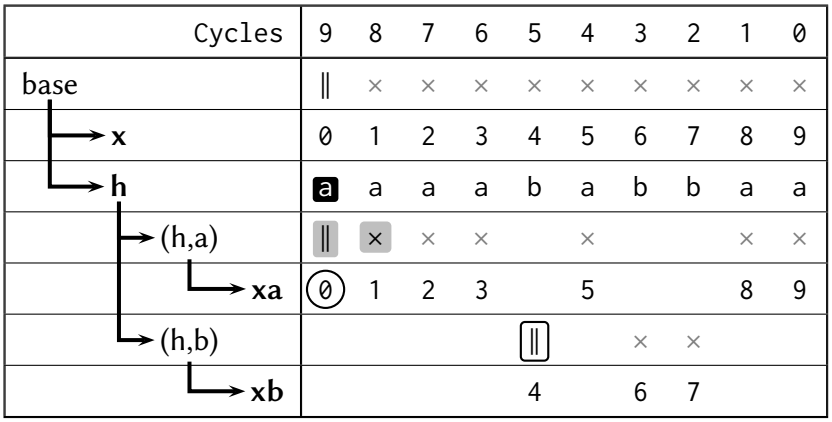


FIGURE 4.3: Instanciation de H

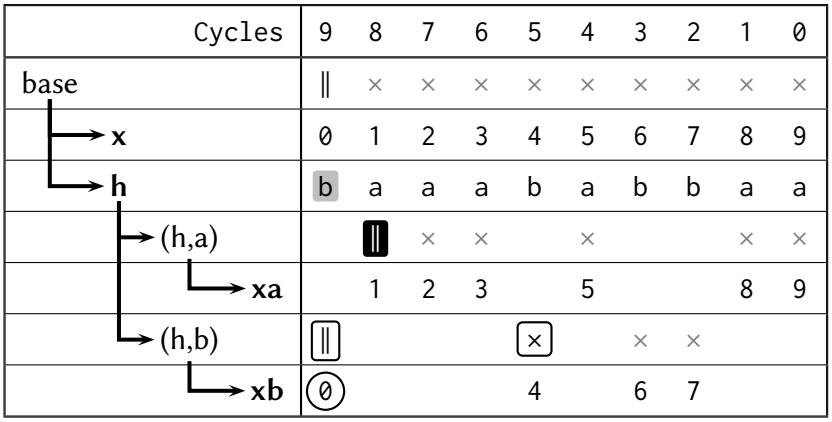


FIGURE 4.4: Instanciation du statut de (h, a) à *initial*

Le même raisonnement s'applique à l'horloge (h, b). L'unification de H avec a nous permet de déterminer que le statut de cette horloge au cycle 5 doit être *initial* (encadré en blanc). Comme *base* est initiale au cycle 9 et qu'il n'existe que des valeurs a pour le flot h entre les cycles 9 et 5, ce cycle 5 est le cycle initial de (h, b).

Finalement, xa admet une valeur au cycle 9, car son horloge est présente (zéro encerclé). En propageant son expression de définition à cette étape, on peut introduire plus de contraintes dans le système et ainsi détecter plus tôt des insatisfaisabilités. De même, s'il existe des invariants dont l'horloge est (h, a), on les propage au cycle 9.

4.2.1.2 Instanciation du statut de (h, a)

Dans la figure 4.4, on suppose cette fois que le statut variable de (h, a) au cycle est instancié à *initial*. Dans ce cas, lorsque le statut est instancié, il n'est plus possible que h

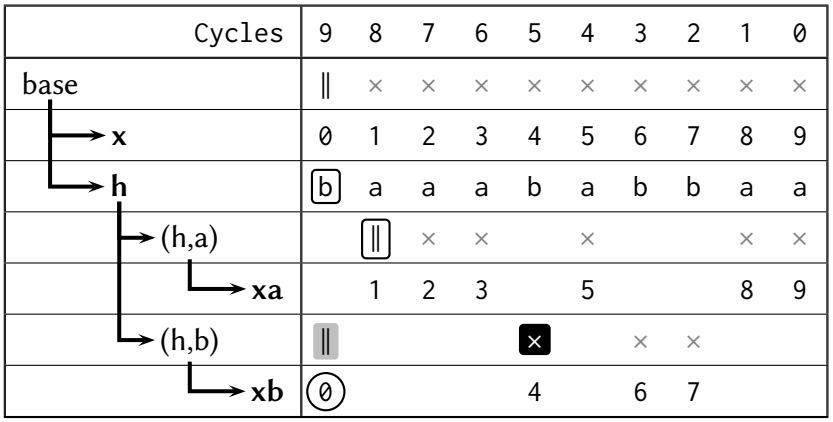


FIGURE 4.5: Instanciation du statut de (h, b) à *noninitial*

prenne la valeur a à un cycle antérieur au cycle 8. C'est pourquoi a est retiré du domaine de H, ce qui ne laisse plus à cette variable que la valeur b. L'instanciation de H à b est similaire au cas précédent, mais cette fois on met à jour le statut de (h, b) qui devient *initial* au cycle 9. Finalement, on propage le flot xb au cycle 9.

4.2.1.3 Instanciation du statut de (h, b)

Dans notre dernier exemple, à la figure 4.5, on unifie cette fois le statut de (h, b) au cycle 5. Dans un premier temps, on cherche, à partir du cycle 5, le dernier instant où le flot h pouvait valoir b. On parcourt donc les valeurs précédentes de h jusqu'à arriver au cycle 9 où la valuation est donnée par H (dont le domaine contient b). Comme le cycle obtenu est le même que le cycle initial de l'horloge parente, le statut de (h, b) y est nécessairement aussi *initial*, ce qui provoque l'unification de H avec b et la propagation consécutive du statut *initial* pour (h, a) au cycle 8.

4.2.2 Statut d'une horloge à un cycle

Nous redéfinissons la fonction *status* vue page 93 pour traiter les horloges multiples. Celle-ci nous donne donc le statut associé à une horloge à un cycle donné. En particulier, le statut retourné est *noninitial* quand le cycle en argument est supérieur (postérieur) au cycle lié à l'horloge par la contrainte « *cmax* ».

$$\text{STATUS-MAX}\ \frac{S \vdash \mathit{cmax}(H, mc, St)}{S \vdash \mathit{status}(H, mc) : St} \qquad \text{STATUS-OTHER}\ \frac{S \vdash \mathit{cmax}(H, mc, St) \qquad mc > c \geq 0}{S \vdash \mathit{status}(H, c) : \mathit{noninitial}}$$

4.2.3 Assertions portées par une horloge

Nous définissons la fonction *asserts(S,H,c)* suivante qui construit, pour un CSP donné $S \in \mathcal{L}_+^g$, l'ensemble de contraintes *propage* qui forcent les invariants d'horloge H à être vrais au cycle c.

$$asserts(S,\ H,\ c) = \left\{ true \Rightarrow_{(c,H)} A \mid assert(A,H) \in S \right\}$$

Nous utilisons cette fonction dans les sections qui suivent. L'introduction des invariants est gérée par deux mécanismes :

- Dans le cas de l'horloge de base, par la contrainte *cmax* introduite précédemment (§??), dont les règles sont détaillées à la section §4.2.5.
- Dans le cas des horloges relatives, par la contrainte *ckflow* définie à la section §4.2.7.

La contrainte *ckflow* nous permet de surveiller l'instanciation d'un flot support : on introduit les invariants quand la valuation d'un flot support à un cycle correspond à sa valeur de présence. Pour l'horloge de base, qui n'est pas liée à un flot, on se repose sur la contrainte *cmax* qui augmente la taille des séquences de tests.

4.2.4 Propagation d'une horloge

Nous définissons une contrainte *propclk(H,c)*, dont le but est de faire en sorte que l'horloge H soit présente au cycle c dans l'ensemble de contraintes. La contrainte est ajoutée car il est possible que l'information selon laquelle l'horloge H en paramètre de *propage* est nécessairement présente ne figure pas encore dans le CSP. Il y a plusieurs raisons à cela, à commencer par la propagation de l'objectif de test.

4.2.4.1 Objectif de test

Par exemple, lorsqu'on débute la génération de test à partir d'un objectif temporisé donné par *reach(E,H)*, son horloge n'est pas forcément déjà propagée. On ajoute donc la contrainte ici :

$$\text{PROP-REACH}\ \frac{}{\vdash S \cup \left\{ reach(E,H) \right\} \rightarrow S \cup \left\{ propclk(H,c),\ true \Rightarrow_{(0,H)} E,\ objective(E,H) \right\}}$$

De même qu'au chapitre précédent, l'objectif est conservé dans un terme neutre n'admettant pas de règle de filtrage.

4.2.4.2 Découpage fonctionnel

De manière similaire à l'objectif de test, la directive `split` propage des hypothèses supplémentaires cadencées sur des horloges quelconques. Nous avons fait en sorte que la directive n'accepte que des identifiants de flots en arguments, car les horloges

de ces identifiants sont conservées dans le modèle, et donc accessibles facilement. La propagation d'un `split` est donc analogue à la propagation dans le cas mono-horloge. En revanche, il est nécessaire que la contrainte *propage* force la présence des horloges dans le cas des identifiants de flots. Nous verrons ainsi à la section §4.2.8 qu'une contrainte *propclk* est introduite lorsqu'on traite les identifiants de variables LUSTRE.

4.2.4.3 Hiérarchie d'horloges

Nous venons d'introduire le fait que la propagation d'un identifiant à un cycle force la propagation simultanée de son horloge, qui à son tour s'assure que son flot support est bien défini au même cycle, et ce jusqu'à ce que l'on propage l'horloge de base.

Le mécanisme récursif de *propclk* selon la hiérarchie d'horloges nous assure que les contraintes *cmax* sont bien mises à jour, et en particulier que les cycles associés à une horloge mère est toujours au moins égale à ceux associés à ses horloges filles (voir propriétés de la section §4.1.1.3).

4.2.4.4 Horloge de base

On sait que l'horloge de *base* est toujours présente, ce qui rend à priori la contrainte *propclk* inutile. Cependant, elle nous permet de repousser le cycle porté par *cmax* au cycle courant. En particulier, elle introduit les assertions cadencées sur l'horloge de base. Si on cherche à propager l'horloge à un cycle égal ou inférieur au plus grand cycle déjà propagé pour cette horloge, la contrainte est effectivement inutile et peut-être retirée :

$$\text{PROPCK-BASE-DISCARD}\ \frac{oc \geq c \qquad S \vdash cmax(base,\ oc,\ St)}{\vdash S \cup \{\, propclk(base,c) \,\} \rightarrow S}$$

Si au contraire le nouveau cycle est supérieur, le statut existant est unifié avec *noninitial* :

$$\text{PROPCK-BASE-INCREMENT}\ \frac{S \vdash maxsize(mc) \qquad bc > c > oc}{\begin{array}{c}\vdash S \cup \{\, propclk(base,c),\ cmax(base,\ oc,\ St) \,\} \\ \rightarrow S \cup \{\, propclk(base,c),\ St \equiv noninitial \,\}\end{array}}$$

Cela suffit pour déclencher dans les propagations successives la création d'un nouveau *cmax* pour l'horloge de base au cycle $oc + 1$. On voit que la contrainte *propclk(base,c)* demeure dans le CSP tant que le terme *cmax* n'a pas atteint le cycle c. Cela nous permet de faire avancer le cycle maximal jusqu'à ce qu'il atteigne le cycle c. Une fois que cela est fait, la contrainte *propclk* est retirée (d'après la règle PROPCK-BASE-DISCARD). L'instanciation à *noninitial* est gérée par la contrainte *watch(base, oc, noninitial)* qui accompagne la contrainte *cmax* ; nous la détaillons à la section §4.2.6 (p. 144).

4.2.4.5 Horloges relatives

Dans le cas des horloges relatives, la contrainte *propclk* propage le flot support à sa valeur de présence. Cela suffit à mettre à jour le statut de cycle comme on le verra pour le filtrage de *cmax* à la section §4.2.5 (p. 141) :

$$\textsc{propck-rel}\ \frac{S \vdash clock(f){=}H}{\vdash S \cup \{propclk((f,v),c)\} \rightarrow S \cup \{v \Rightarrow_{(c,H)} f\}}$$

4.2.5 Filtrage de *cmax*

Les règles impliquant directement la contrainte *cmax* visent à tirer parti de la table d'horloges et des propriétés vues précédemment entre celle-ci et les flots de données.

4.2.5.1 Incrément du cycle de *cmax(base, c, St)*

Nous venons de voir la règle PROPCK-BASE-INCREMENT qui repose sur le filtrage de *cmax* : quand le nouveau cycle auquel l'horloge de *base* doit être présente dépasse le cycle maximal connu, on instancie les statuts de chaque nouveau *cmax* à *noninitial* jusqu'à ce qu'il y ait une contrainte *cmax* positionnée au cycle désiré. Cette règle suppose qu'une fois le statut vaut *noninitial*, on remplace le terme *cmax* existant par un nouveau à un cycle de décalage vers le passé. Cette transformation est réalisée par la règle qui suit :

$$\textsc{cmax-base-noninitial}\ \frac{\begin{array}{c} S \vdash maxsize(mc) \\ nc = c+1 \qquad nc < mc \qquad S_A = asserts(base,nc) \end{array}}{\vdash S \cup \{cmax(base,\ c,\ noninitial)\} \rightarrow S \cup S_A \cup setcmax(S,\ base,\ nc)}$$

On incrémente le cycle de sorte à positionner un terme *cmax* à l'instant $nc = c+1$. De plus, à ce nouveau cycle, on propage à *true* les expressions invariantes cadencées sur l'horloge de base. Ce traitement est réalisé si le nouveau cycle est bien dans l'intervalle de cycle autorisé par la borne maximale mc.

4.2.5.2 Échec de l'incrément

Lorsque, justement, le nouveau cycle est trop grand par rapport à la limite portée par *maxsize*, on est face à un échec de la génération de tests :

$$\textsc{cmax-base-fail}\ \frac{S \vdash maxsize(mc) \qquad c+1 \geq mc}{\vdash S \cup \{cmax(base,\ c,\ noninitial)\} \rightarrow \{fail\}}$$

4.2.5.3 Cycle des horloges relatives

Le cycle c d'une contrainte *cmax(H, c, St)* est associé à un statut St. On rappelle que le cycle c est le plus grand cycle auquel l'horloge H est connue comme étant présente.

Lorsqu'on trouve un nouveau cycle, plus loin dans le passé, auquel l'horloge est présente, on peut décaler la contrainte *cmax* pour qu'elle référence ce nouveau cycle.

$$\textsc{cmax-rel-move}\ \frac{S \vdash f \rightarrowtail_{nc} v \qquad S \vdash \mathit{maxsize(mc)} \qquad mc > nc > oc}{\begin{array}{c}\vdash S \cup \{\mathit{cmax((f,v),\, oc,\, St)}\} \rightarrow \\ S \cup \{St \equiv \mathit{noninitial}\} \cup \mathit{setcmax}(S \cup \{St\},\, \mathit{(f,v)},\, nc)\end{array}}$$

Dans la règle précédente, il existe une valuation au cycle $nc > oc$ de f à v, ce qui nous permet de savoir que l'ancien statut St est *noninitial*, et que le plus grand cycle auquel (f, v) est présent est nc. Nous plaçons ainsi une nouvelle contrainte *cmax* dans le CSP (par le biais de *setcmax*, où on s'assure que la variable créée est différente de St). La figure 4.6 illustre cette situation.

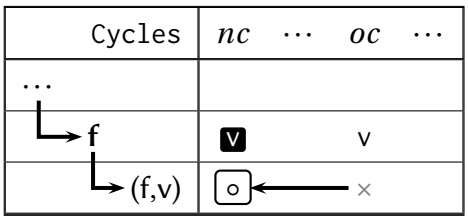


FIGURE 4.6: Décalage du cycle de (f, v) s'il existe une valuation antérieure de f à v.

4.2.5.4 Réduction de domaine

La règle suivante se base sur la propriété de la page 121 et fait en sorte que les valuations de f aux cycles antérieurs au cycle initial d'une horloge (f, v) soient différents de v.

$$\textsc{cmax-reduce-initial}\ \frac{\begin{array}{c}S \vdash \mathit{cmax((f,v),\, ic,\, initial)} \qquad S \vdash f \rightarrowtail_c W \qquad c > ic \\ S \vdash W \approx D_W \qquad v \in D_W \qquad (W \not\equiv v) \notin S\end{array}}{\vdash S \rightarrow S \cup \{W \not\equiv v\}}$$

La situation est représentée à la figure 4.7 : le cycle initial ic est connu pour (f, v), ce qui limite les valeurs possibles pour f aux cycle $c > ic$.

4.2.5.5 Déduction du cycle initial

Il est possible de propager le statut *initial* d'une horloge mère M vers une de ses horloges filles $H = (f, v)$ quand il n'existe qu'une seul cycle possible auquel H peut être elle aussi initiale. En effet, le cycle initial est toujours compris entre le cycle initial de l'horloge mère et le cycle maximal porté par *cmax*. Nous notons respectivement c_M et c_H ces cycles.

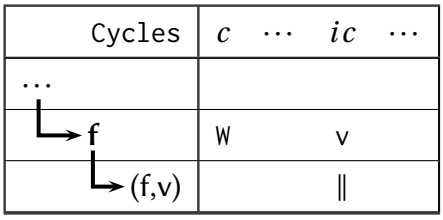


FIGURE 4.7: Réduction de domaine pour le flot support aux cycles antérieurs au cycle initial : ici W ne peut valoir v, car sinon le cycle ic ne serait pas le cycle intial de (f, v).

Dans l'intervalle $c_M..c_H$, il existe un sous-ensemble de cycles auxquels on sait avec certitude que l'horloge H est absente : son flot support f admet par exemple une valeur différente de la valeur de présence v, où l'horloge M elle-même est absente. L'ensemble de ces cycles d'absences est noté A.

Lorsqu'on retire de l'intervalle $c_M..c_H$ les cycles de A, on obtient l'ensemble C des cycles auxquels : (i) le flot f est lié à un domaine trop imprécis ou bien (ii) on ne sait pas si l'horloge M (ou une de ses ancêtres) est elle-même présente ou non. Les cycles retenus sont ceux auxquels H est potentiellement initiale, mais pour lesquels il manque des informations dans le grille de valeurs.

Alors, si l'ensemble C se réduit à un élément c, on peut en déduire que H vaut *initial* au cycle c : c'est l'unique cas possible. On ne perd donc pas de solution pour le PGST en positionnant la contrainte *cmax* au cycle c et au statut *initial* :

$$\text{CMAX-FIND-INITIAL}\ \frac{\begin{array}{c} H = (f, v) \qquad S \vdash \mathit{mother(H)=M} \\ S \vdash \mathit{cmax(M,\, c_M,\, initial)} \\ S \vdash \mathit{cmax(H,\, c_H,\, St)} \qquad \mathit{St} \neq \mathit{initial} \\ A = \{\, x \in c_M..c_H \mid S \vdash \mathit{absent(H,x)} \,\} \qquad C = c_M..c_H \setminus A = \{c\} \\ (\mathit{cmax(H,\, c,\, initial)}) \notin S \qquad (\mathit{St} \equiv \mathit{initial}) \notin S \end{array}}{\vdash S \rightarrow S \cup \mathit{setcmax(S,\, H,\, c, initial)}}$$

Les deux derniers tests de la forme « $\cdots \notin S$ » évitent que la règle ne soit applicable après la réecriture, en tenant compte des deux valeurs possibles données par *setcmax*. Cette contrainte est utile lorsqu'on tente de générer les instants initiaux des séquences de tests, car les statuts sont déduits les uns à partir des autres en cascade lorsqu'il y a suffisamment d'informations.

La figure 4.5 de la page 138 illustre cette règle. Le fait d'instancier le statut existant de (h, b) à *noninitial* fait que le seul cycle potentiel comme cycle initial est le cycle 9. Entre les cycles 9 et 5 inclus, l'horloge (h, b) est toujours absente. Si au moins une des valeurs de h était inconnue dans cet intervalle de cycles, il n'aurait pas été possible de déduire le cycle initial de (h, b).

4.2.5.6 Cycle initial inexistant

Dans la règle précédente, l'ensemble C se réduisait à un seul élément. Si maintenant l'ensemble C est vide, cela signifie que l'horloge H est toujours absente dans l'intervalle $c_M..c_H$, ce qui rend la table incohérente avec les propriétés précédentes. Il n'existe donc pas de solution au PGST modélisé :

$$\textsc{cmax-fail-initial}\ \frac{\begin{array}{c} H = (f, v) \qquad S \vdash \mathit{mother(H)=M} \\ S \vdash \mathit{cmax(M,\ c_M,\ initial\ )} \\ S \vdash \mathit{cmax(H,\ c_H,\ St)} \qquad \mathit{St} \neq \mathit{initial} \\ \forall x \in c_M..c_H, S \vdash \mathit{absent(H,x)} \end{array}}{\vdash S \rightarrow \{\mathit{fail}\}}$$

4.2.6 Contrainte *watch*

L'intérêt de la contrainte *watch(H, c, St)* est qu'elle est filtrable une seule fois dans un CSP puis est retirée de l'ensemble de contraintes, alors que *cmax* ne peut pas être retirée (on en a besoin pour connaître le cycle maximal). Elle nous permet donc d'introduire des contraintes de mise à jour de la table au plus une fois par cycle et par horloge.

4.2.6.1 Horloge initiale

Au moment où une horloge H voit sont statut devenir *initial*, on peut favoriser la détection d'une insatisfaisabilité en propageant ses horloges filles si elles ne l'ont pas encore été. Ici, « propager les horloges filles » signifie qu'on introduit des contraintes *propage* pour tous les flots supports qui ont pour horloge H.

Comme on l'a vu à la section §4.1.1.3, le cas est intéressant en pratique lorsqu'on a un type énuméré dont toutes les valeurs sont utilisées pour définir une horloge. En propageant les flots f d'horloge H alors que l'horloge H est initiale, on aura facilement une valeur pour f (la branche gauche de l'opérateur « → » est généralement trivialement évaluable) et cette valeur nous donnera l'instant initial d'une horloge fille.

Dans la règle qui suit, la fonction *children* renvoie l'ensemble des contraintes *propage* nécessaires, à savoir celles portant sur des flots f d'horloge H, où f appartient à *ckflows(S)*. On rappelle que cet ensemble regroupe les flots de S ayant un rôle de flot support pour au moins une horloge (cf. §4.1.2.2). La fonction est définie ainsi :

$$\mathit{children(S,H,c)} = \{\, \mathit{propage(f,c)} \mid f \in \mathit{ckflows(S)}\ , S \vdash \mathit{clock(f)=H} \,\}$$

On se repose sur la contrainte *propage* à deux arguments qui se charge de créer de nouvelles variables libres à propager dans les expressions de définition des flots. La règle de filtrage lorsque le statut est initial est la suivante :

$$\textsc{watch-initial}\ \frac{}{\vdash S \cup \{\, \mathit{watch(H,\ c,\ initial\ )} \,\} \rightarrow S \cup \mathit{children(S,H,c)} \cup \{\, \mathit{propclk(H,c)} \,\}}$$

On peut remarquer qu'on ajoute aussi une contrainte *propclk*. Elle est utile notamment pour les horloges relatives car elle force le flot support de H à prendre sa valeur de présence.

Par exemple, si on déduit qu'un cycle c est le cycle initial d'une horloge par la règle CMAX-FIND-INITIAL de la section §4.2.5.5, on force l'horloge et ses ancêtres à être toutes présentes à ce cycle c. Cela est intéressant car justement, quand on applique CMAX-FIND-INITIAL, le cycle c fait partie des cycles pour lesquels on ne peut pas dire si l'horloge est ou non présente seulement en observant la grille de valeurs. La règle WATCH-INITIAL que l'on vient de voir complète donc les informations manquantes dans le CSP.

4.2.6.2 Horloge non-initiale

Si le statut est *noninitial*, on cherche à garantir qu'il existe bien un cycle précédent à l'horloge au cycle qui nous intéresse. En effet, on sait qu'il existe nécessairement un cycle précédent à l'horloge, puisqu'un cycle initial doit exister et que le cycle courant a le statut *noninitial*.

Pour cela, nous nous reposons sur une contrainte *precycle(H,c)* à deux arguments qui propage les contraintes nécessaires de sorte que l'horloge H admette un cycle précédent au cycle c. En particulier, elle ne change pas le CSP si cela déjà le cas. Nous ne détaillons cette contrainte que dans la section §4.3.4 (p. 158) où l'on s'intéresse à l'opérateur `pre`. Nous pouvons cependant déjà y faire appel ici :

$$\text{WATCH-REL-NONINITIAL}\ \frac{}{\vdash S \cup \{\, watch((f,v),\, c,\, noninitial\,)\,\} \rightarrow S \cup \{\, precycle((f,v),c)\,\}}$$

4.2.6.3 Instanciation à la limite

Comme la taille des séquences est bornée par le paramètre mc donné par *maxsize(mc)*, on peut instancier le statut St d'une contrainte *cmax(H, c, St)* à *initial* lorsque c vaut $mc-1$. En effet, il n'est pas possible que le statut soit égal à *noninitial* : il faudrait alors repousser la contrainte au moins jusqu'au cycle mc, ce qui entraînerait un échec de la génération de test. Dans la règle qui suit, on change le statut connu mais on ne retire pas *watch* car on souhaite aussi pouvoir appliquer l'une des règles précédentes :

$$\text{WATCH-LIMIT}\ \frac{St \neq initial \qquad S \vdash maxsize(mc) \qquad c = mc - 1}{\vdash S \cup \{\, watch(H,\, c,\, St)\,\} \rightarrow S \cup \{\, watch(H,\, c,\, initial\,), St \equiv initial\,\}}$$

4.2.7 Contrainte *ckflow*

Jusqu'à présent, on observait principalement les modifications du statut ou du cycle attachés à une horloge. Afin de mettre à jour les relations entre les horloges, on souhaite aussi surveiller les valuations des flots supports lors de la propagation de contraintes.

Par exemple, soit $V_{f,c}$ la variable liée au flot f à l'instant c : on a $f \mapsto_c V_{f,c}$. De plus, f est un flot support d'une horloge $H = (f, v)$. On souhaite pouvoir observer les réductions de domaines ou l'instanciation de $V_{f,c}$ pour ensuite en déduire des filtrages dans la table d'horloges : on pourrait ainsi repousser le cycle associé à H dans la contrainte « *cmax* ».

On ajoute donc une contrainte *ckflow(V, f, c)* à chaque cycle c où l'identifiant f est exploré par la contrainte *propage*. On ajoute ce terme uniquement si f est bien un flot support d'une horloge définie dans le système sous test (sinon, ce terme est inutile). La section §4.2.8 détaille comment *ckflow* est ajoutée dans le système lors du filtrage de *propage*.

La variable $V_{f,c}$ associée précédemment au cycle c et au flot support f peut subir des réductions de domaines ou une instanciation. Lorsque la variable est instanciée à la valeur de présence v, nous pouvons :

- Faire avancer le cycle maximal lié à l'horloge jusqu'au cycle courant c, si c est bien supérieur à mc. Il nous suffit de passer par *propclk*, qui filtre les cas où le cycle est inférieur ou supérieur à la borne courante.
- Introduire les contraintes *propage* au cycle c pour les invariants portés par les `assert` cadencés sur (f, v). Nous faisons ce traitement ici car on peut s'assurer que chaque invariant n'est propagé qu'au plus une fois. La contrainte *ckflow* est en effet nécessairement unique pour un flot et un cycle donnés (voir §4.2.8).

Le cas où la variable est instancié à une autre valeur ne nous intéresse pas ici. La règle est donc la suivante :

$$\textsc{ckflow-inst}\ \frac{S_A = asserts(S,(f,v),c)}{\vdash S \cup \{ ckflow(v, f, c) \} \rightarrow S \cup S_A \cup \{ propclk((f,v),c) \}}$$

Avec cette contrainte *ckflow*, nous pouvons propager des informations des flots vers les horloges, alors que les autres contraintes projettent d'un statut à un autre ou d'un statut vers un flot. L'ajout de *ckflow* est fait au niveau de la contrainte *propage*, dans le cas des identifiants de flots, que nous détaillons maintenant.

4.2.8 Propagation des identifiants de flots

Nous donnons les règles de réécriture pour $R \Rightarrow_{(c,H)} f$ lorsque f est un identifiant de flot Lustre. Elles sont basée sur la règle vue à la section §3.3.1.8 du chapitre 3.

4.2.8.1 Flot déjà propagé

La première règle est identique à celle que l'on avait définie alors : lorsqu'il existe déjà un terme $f \mapsto_c V$ dans le CSP, la valeur propagée R est unifiée avec le terme V existant :

$$\textsc{prop-var-unif}\ \frac{S \vdash f \mapsto_c V}{S \cup \{ R \Rightarrow_{(c,H)} f \} \rightarrow S \cup \{ R \equiv V \}}$$

4.2.8.2 Flot indéfini

Dans le cas contraire, f est indéfini au cycle c. On ajoute les contraintes propres à f ainsi qu'une relation $f \mapsto_c R$ qui nous garantit que la règle n'est applicable qu'une fois.

Nous propageons par ailleurs l'horloge H sur laquelle est cadencé le flot f, à travers *propageclock* : en plus de s'assurer que l'horloge est bien présente à ce cycle (la grille de valeurs n'a pas forcément cette information), on repousse indirectement les statuts des horloges mères à travers la hiérarchie d'horloges. Quand le flot est calculé par une expression, on propage le résultat R dans cette expression. Si le flot est un flot support, on ajoute une contrainte *ckflow* :

$$\text{PROP-VAR-UNDEF}\ \frac{\begin{array}{ccc} & S \vdash \mathit{undef(f,\ c)} & \\ S \vdash \mathit{type(f,T)} & \mathcal{E} = \mathit{Equa(S,R,f,c,H)} & \mathcal{C} = \mathit{CkFlow(S,R,f,c)} \end{array}}{\vdash S \cup \{R \Rightarrow_{(c,H)} f\} \quad \rightarrow S \cup \mathcal{E} \cup \mathcal{C} \cup \{f \mapsto_c R, R \in D_T, \mathit{propclk(H,c)}\}}$$

Les différentes contraintes ajoutées sont construites par des fonctions auxiliaires, dans des ensembles que l'on joint. Nous passons par des ensembles de contraintes car la règle traite différentes combinaisons de cas particuliers, selon qu'un identifiant renvoie à une entrée du système ou admet une équation, ou bien qu'il s'agisse d'un flot support ou non. Les fonctions auxiliaires sont les suivantes :

Equa Propage l'équation de définition de f si elle existe.

$$\mathit{Equa(S,R,f,c,H)} = \begin{cases} \{R \Rightarrow_{(c,H)} \mathit{Exp}\} & \text{si } S \vdash \mathit{equa(f,Exp)} \\ \emptyset & \text{sinon} \end{cases}$$

CkFlow Propage la contrainte *ckflow* si f est un flot support pour au moins une horloge.

$$\mathit{CkFlow(S,V,f,c)} = \begin{cases} \{\mathit{ckflow(V,\ f,\ c)}\} & \text{si } \exists\ (g, w) \text{ tq. } S \vdash \mathit{clock(g,(f,w))} \\ \emptyset & \text{sinon} \end{cases}$$

4.2.9 Exemple

Nous reprenons les deux premiers exemples de la section §4.2.1 pour illustrer la propagation de contraintes au niveau de la table d'horloges. Nous définissons l'ensemble S_0 comme étant l'ensemble de contraintes correspondant à la grille initiale donnée par la figure 4.2 page 136. Celle-ci se décompose de la manière suivante, où Δ_1 correspond à l'ensemble des autres contraintes du système, qui ne nous intéressent pas ici :

$$S_0 = \left\{ \begin{array}{l} \mathit{cmax(base,}\ 9,\ \mathit{initial}\ ) \\ \mathit{cmax((h,a),}\ 8,\ \mathit{St}_1), \mathit{watch((h,a),\ 8,\ St}_1) \\ \mathit{cmax((h,b),}\ 5,\ \mathit{St}_2), \mathit{watch((h,b),\ 5,\ St}_2) \\ h \mapsto_9 H \\ \mathit{ckflow(H,\ h,\ 9)} \end{array} \right\} \cup \Delta_1$$

On retrouve trois contraintes *cmax*, une pour chaque horloge, avec St_1 et St_2 les variables de statut associées aux horloges *(h,a)* et *(h,b)*. Pour ces deux dernières, où le statut n'est pas encore connu, il existe deux contraintes *watch* qui sont complémentaires aux contraintes *cmax* et qui ont pour but de contrôler la mise à jour de la table d'horloges symbolisée par *cmax*.

Il y a en plus, dans S_0, une relation $h \mapsto_9 H$ qui associe la variable H (de domaine $[a, b]$) au flot h et au cycle 9. Finalement, il existe une contrainte *ckflow* qui relie entre-elles ces mêmes termes. L'information contenue dans *ckflow* est redondante avec celle de la grille de valeurs (contrainte $h \mapsto_9 H$), mais les deux relations sont différentes. Avec les règles données, le terme *ckflow* n'est filtrable qu'une seule fois, car on le retire quand il n'est plus utile, ce qui n'est pas le cas pour la relation $\mapsto$.

4.2.9.1 Instanciation de H

De même que précédemment (p. 137), on souhaite maintenant instancier la variable H à la valeur a. On ajoute donc au système la contrainte « $H \equiv a$ » pour obtenir le système S_1 :

$$S_1 = S_0 \cup \{H \equiv a\}$$

L'unification provoque la substitution de H par a dans S_1 et aboutit à $S_2 = S_0[H/a]$:

$$\vdash S_1 \xrightarrow{*} S_2$$

Ce système est alors le suivant :

$$\begin{aligned} S_2 &= \left\{ \begin{array}{l} cmax(base, 9, initial) \\ cmax((h,a), 8, St_1), watch((h,a), 8, St_1) \\ cmax((h,b), 5, St_2), watch((h,b), 5, St_2) \\ h \mapsto_9 a \\ ckflow(a, h, 9) \end{array} \right\} \cup \Delta_2 \\ &= \{ckflow(a, h, 9)\} \cup \Delta_3 \end{aligned}$$

Nous avons décomposé S_2 en un singleton et l'ensemble Δ_3 pour montrer le filtrage de la contrainte *ckflow* selon la règle CKFLOW-INST (p. 146). On a alors :

$$\vdash \{ckflow(a, h, 9)\} \cup \Delta_3 \rightarrow \{propclk((h,a),9)\} \cup \Delta_3$$

La contrainte a été remplacée par *propclk*, qui s'assure que l'horloge *(h,a)* est bien présente au cycle 9. S'il y avait eu des assertions d'horloge *(h,a)*, on les aurait introduites durant cette étape.

La propagation pour *propclk* passe par la règle PROPCK-REL vue page 141. On sait que l'horloge de h est *base*, ce qui nous donne :

$$\vdash \{propclk((h,a),9)\} \cup \Delta_3 \rightarrow \{a \Rightarrow_{(9,base)} h\} \cup \Delta_3$$

On note S_3 le CSP à droite de la réecriture, avec $\vdash S_2 \xrightarrow{*} S_3$ d'après les deux règles précédentes. La propagation de *ckflow* termine cependant ici car la contrainte *propage* que l'on vient d'introduire est redondante avec l'information déjà connue, à savoir : $h \mapsto_9 a$. Dans ce cas précis, *propclk* n'a pas été utile, et on peut retirer la contrainte *propage* :

$$\vdash S_3 \rightarrow \Delta_3$$

Ce dernier ensemble est décomposable ainsi :

$$\Delta_3 = \{ cmax((h,a), 8, St_1), h \mapsto_9 a \} \cup \Delta_4$$

D'après la règle `cmax-rel-move` de la page 142, on peut décaler le cycle maximal de la contrainte *cmax* (la limite *maxsize* est supérieure à 9) :

$$\vdash \Delta_4 \cup \{ cmax((h,a), 9, St_1) \} \rightarrow S_4$$

avec S_4 l'ensemble suivant et St_3 une variable libre dans Δ_4 différente de St_1 :

$$\begin{aligned} S_4 &= \Delta_4 \cup \{ St_1 \equiv noninitial \} \cup setcmax(S \cup \{ St_1 \}, (h,a), 9) \\ &= \Delta_4 \cup \left\{ \begin{array}{l} St_1 \equiv noninitial, St_3 \in D_{\texttt{status}} \\ cmax((h,a), 9, St_3), watch((h,a), 9, St_3) \end{array} \right\} \\ &= S_5 \end{aligned}$$

Dans le système résultant S_5, on retrouve trois contraintes à propager :

1. L'unification de St_1 se répercute dans la contrainte *watch* au cycle 8 qui devient alors *watch((h,a), 8, noninitial)*. Cette contrainte est filtrable selon `watch-rel-noninitial` vue page 145. La contrainte ajoute un terme *precycle((h,a),8)* que nous ne détaillons pas ici, car on sait déjà qu'il existe un cycle précédent pour *(h,a)* au cycle 8.
2. Les deux contraintes *cmax((h,a), 9, St_3)* et *cmax(base, 9, initial)* nous permettent de savoir que St_3 vaut nécessairement *initial*, d'après la règle `cmax-find-initial` (p. 143). On peut ainsi réaliser la propagation du statut initial depuis l'horloge mère *base* vers sa fille *(h,a)*.
3. De même, on peut propager le statut *initial* depuis *base* vers *(h,b)* selon la même règle. Nous détaillons ce point ci-dessous.

Pour la troisième réecriture possible, on décompose S_5 ainsi :

$$S_5 = \{ cmax(base, 9, initial), cmax((h,b), 5, St_2) \} \cup \Delta_5$$

Pour appliquer la règle, on s'intéresse à l'intervalle de cycles 5..9, et en particulier aux instants où *(h,b)* est absente. La figure de la page 137 nous rappelle que le flot `h` vaut `a` aux

cycles 6..9, laissant comme seule possibilité le cycle $c = 5$ comme cycle *initial*. On peut alors ajouter *setcmax(S, (h,b),* 5*,initial)* à l'ensemble S_5, qui dans notre cas va ajouter l'unification $St_2 \equiv$ *initial* d'après la définition vue page 124.

Les unifications des statuts laissent des contraintes résiduelles *watch* qu'on peut alors filtrer à leur tour. Ici, elles n'ont pas d'impact significatif sur le CSP, même si dans le cas générales elles pourraient introduire des contraintes *propage* pour les horloges filles. Nous avons donc propagé les différentes contraintes qui s'assurent de la cohérence de la table d'horloges pour obtenir la situation de la figure 4.3, qui se décompose ainsi, où Δ_6 ne contient plus de contrainte filtrable liée aux horloges :

$$\left\{ \begin{array}{l} \mathit{cmax(base,}\ 9,\ \mathit{initial}\,) \\ \mathit{cmax((h,a),}\ 9,\ \mathit{initial}\,) \\ \mathit{cmax((h,b),}\ 5,\ \mathit{initial}\,) \\ h\ \mapsto_9 a \end{array} \right\} \cup \Delta_6$$

4.2.9.2 Instanciation du statut de *(h,a)*

Nous ne détaillons pas complètement l'exemple de la figure 4.4 page 137. Cependant, on remarque qu'une fois que le statut de *(h,a)* est instancié à *initial* au cycle 8, on peut appliquer la règle `cmax-reduce-initial` de la section §4.2.5.4 (p. 142). On note S_6 le CSP S_0 dans lequel on a instancié le statut de *(h,a)*. On a alors :

$$S_6 \vdash \mathit{cmax((h,a),}\ 8,\ \mathit{initial}\,)$$

De plus, on a toujours la relation suivante issue de la grille initiale S_0 :

$$S_6 \vdash h\ \mapsto_9 H$$

On a bien $9 > 8$, ainsi que $S_6 \vdash H \approx a..b$ avec $a \in a..b$. Alors, on peut appliquer la règle `cmax-reduce-initial` et ajouter la contrainte $H \not\equiv a$ au système S_6. De cette manière, on projette l'unification d'un statut (ici *initial*) vers le domaine du flot support `h` au cycle 9. Après cette réecriture, on déduit le cycle initial associé à *(h,b)* grâce aux règles `cmax-rel-move` et `cmax-find-initial` vues précédemment.

4.2.10 Conclusion

L'ensemble des mécanismes de cette section nous permettent de faire communiquer les contraintes existantes avec les contraintes de la table d'horloges tout en exploitant les propriétés temporelles du langage.

Les relations *cmax* et *watch* font évoluer les statuts et les cycles potentiellement initiaux liés aux horloges. Par ailleurs, la contrainte *propclk* s'assure que des horloges sont présentes en forçant les flots à valoir certaines valeurs. Ces différentes règles mettent en relation les statuts des horloges à différents niveaux de la hiérarchie à travers la relation

de parenté qui les unit. D'autre part, la valeur d'un flot de type énuméré peut influencer un statut d'horloge à travers *ckflow*. Enfin, nous traitons les assertions temporisées en garantissant qu'un invariant n'est introduit qu'au plus une fois par cycle et seulement quand son horloge est présente.

4.3 Retard unitaire

Nous nous intéressons maintenant à l'opérateur de décalage unitaire « `pre` ». Pour cela, nous commençons par définir à la section §4.3.1 une fonction *findcycle* qui tente de trouver le cycle précédent d'un flot étant donnée une grille de valeurs partiellement générée. Cette fonction peut être bloquée par manque d'informations. Nous donnons alors une contrainte nommée *precycle* qui cherche à combler les éléments inconnus du CSP courant en introduisant de nouvelles contraintes selon le résultat partiel donné par *findcycle* (§4.3.4 p. 158). Nous pouvons ensuite nous intéresser à la règle de *propage* dans le cas de l'opérateur `pre`, qui n'a plus qu'à se reposer sur cette contrainte *precycle* (§4.3.5 p. 160). Nous passons par une fonction et une contrainte intermédiaires plutôt que de réaliser tout le traitement dans le filtrage de « `pre` » car : (i) celles-ci seront réutilisables dans la section suivante et (ii) la séparation des rôles entre les deux facilite la présentation de l'approche. À la section §4.3.6, nous nous intéressons finalement à la sélection de cas de test. En effet, l'opérateur `pre` devient dépliable dans le langage multi-horloges, par le biais de la contrainte *precycle*.

4.3.1 Recherche du cycle précédent

Le cycle précédent d'un flot à un cycle dépend des instants de présence de son horloge, et à travers elle des instants de présence des horloges de plus haut niveau. Nous avons montré la fonction d'évaluation *precycle* dans le chapitre 2, qui détermine le cycle précédent d'une horloge lors de l'évaluation des séquences. Nous définissons ici une fonction analogue, à la différence que les séquences sont partiellement générées et qu'elles ne contiennent ainsi pas forcément suffisamment de valeurs pour déterminer quel est le cycle précédent recherché.

Nous définissons la fonction *findcycle* qui cherche le cycle précédent d'une horloge H à partir d'un cycle c, dans le contexte d'un ensemble de contraintes S. On note alors R le résultat de la recherche :

$$S \vdash \mathit{findcycle}(H,c) = R$$

La fonction *findcycle* admet une seule règle de définition, car elle passe par une fonction auxiliaire nommée *fcwp* qui met en œuvre la recherche :

$$\textsc{findcycle}\ \frac{S \vdash \mathit{fcwp}(H,c+1) = R}{S \vdash \mathit{findcycle}(H,c) = R}$$

Les initiales de *fcwp* signifient *find cycle when present*. Dans cette fonction *fcwp*, le cycle $d = c + 1$ en argument est potentiellement la solution recherchée, à la différence du cycle c de *findcycle* qui est le cycle d'origine. Autrement dit, l'appel à *fcwp* cherche le cycle précédent en commençant par $c + 1$.

Les fonctions *fcwp* et *findcycle* retournent un entier quand le cycle peut-être trouvé grâce aux informations déjà présentes dans l'ensemble de contraintes. Dans le cas contraire, le résultat indique pourquoi il n'est pas possible de trouver un cycle précédent. Les différents cas sont les suivants :

entier La valeur de retour est un entier si le cycle précédent peut être calculé.

fail Le terme retourné vaut *fail* si le cycle précédent ne peut pas exister avec la configuration courante de la grille de valeurs.
Cela est possible par exemple lors de la résolution de contraintes, si on prend l'hypothèse qu'un cycle est le cycle initial de la séquence de test, mais que cette décision empêche un opérateur `pre` d'accéder à une valeur précédente. Lors de l'évaluation du cycle précédent, la fonction *fcwp* se rend compte de l'insatisfaisabilité du système de contraintes.

undef(c, f) Il est possible aussi que pour trouver le cycle précédent d'une horloge, on doive connaître le cycle précédent d'une horloge ancêtre, mais que le flot support de celle-ci n'a jamais été propagée en tant que contraintes dans le système. Dans ce cas, il n'existe pas de valuation pour le flot support dans le système : la valeur de retour est *undef(c, f)*, où f est le flot support bloquant et c le cycle auquel on a besoin de connaître sa valeur.

wait(c, f, W) Par ailleurs, si le flot support f d'une horloge bloquante est propagé mais que la variable W qu'on lui associe dans le CSP a un domaine trop imprécis, on ne peut pas savoir si l'horloge est présente ou non. La valeur de retour de *fcwp* est alors *wait(c,f,W)*, où W est la variable sur laquelle on est en attente, associée au flot support f de l'horloge au cycle c.

status(c, St) Finalement, la fonction *fcwp* peut ne pas conclure car on bloque sur le statut d'initialité de l'horloge de base. En effet, dans certains cas, cette horloge n'existe pas sur suffisamment de cycles, ce qui limite la recherche d'un cycle précédent pour une horloge fille. Dans ce cas, le terme retourné est *status(c, St)*, où St est le statut bloquant associé à l'horloge de base au cycle c. Pour faire progresser la recherche, on doit alors instancier ce statut à *noninitial*.

4.3.2 Exemples

Les figures 4.8 à 4.11 illustrent l'utilisation de *findcycle* pour propager le fait qu'une horloge possède un instant précédent.

Notre exemple repose sur deux flots `f` et `g`.

- `f` est défini sur l'horloge de base et peut prendre à chaque cycle une des valeurs `a`, `b` ou `c`.
- Le flot `g` a pour horloge (f, a) et peut prendre une des valeurs `u`, `v` ou `w`.
- On note H l'horloge (g, u).

On pose $c \in \mathbb{N}$ un cycle et S l'ensemble de contraintes associé à l'exemple. En particulier, la figure 4.8 illustre l'état de la grille de valeurs dans S. On cherche à trouver le cycle précédent de H à partir du cycle c.

4.3.2.1 Appels récursifs

On cherche à trouver le terme R_1 telle que :

$$S \vdash \mathit{findcycle(H,c)} = R_1$$

Pour cela, on appelle *fcwp* à partir du cycle $c+1$, qui se charge de déterminer le résultat :

$$S \vdash \mathit{fcwp(H,c+1)} = R_1$$

Au cycle $c+1$, l'horloge mère de H, (f, a) est absente : on sait en effet que la valeur de `f` est `b` à ce cycle. On appelle récursivement *fcwp* pour trouver le cycle précédent de (f, a) à partir de $c+1$. Or, à ce même cycle, l'horloge mère de (f, a), l'horloge de base, est présente, mais la valeur prise par `f` est différente de `a`. Le calcul du cycle précédent de (f, a) continue au cycle $c+2$, où l'horloge (f, a) est bien présente. On en déduit le jugement suivant, à savoir que le cycle précédent de (f, a) à partir de $c+1$ est $c+2$:

$$S \vdash \mathit{fcwp((f,a),c+1)} = c+2$$

Alors, on peut reprendre le calul pour $H = (g, u)$ à partir du cycle $c+2$, qui doit nous donner le résultat R_1 attendu :

$$S \vdash \mathit{fcwp(H,c+2)} = R_1$$

Finalement, comme `g` n'a jamais été associé à un terme dans la grille de valeurs au cycle $c+2$, le calcul est bloqué, et on a :

$$R_1 = \mathit{undef}(c+2,\ g)$$

Le résultat n'est pas un cycle, mais marque l'interruption de la recherche du cycle précédent de H à partir de c, à cause du flot `g` qui n'admet pas de valeurs au cycle $c+2$.

Cycles	$c+2$	$c+1$	c
base	∘	×	×
→ f	a	b	a
→ (f,a)	∘		×
→ g	?		u
→ (g,u)			∘

FIGURE 4.8: Recherche du cycle précédent de (g, u) au cycle c. La fonction *findcycle* est bloquée au cycle $c+2$ par le flot support g qui n'a jamais été propagé à ce cycle.

Cycles	$c+2$	$c+1$	c
base	∘	×	×
→ f	a	b	a
→ (f,a)	×		×
→ g	$W : [u, v, w]$		u
→ (g,u)			∘

FIGURE 4.9: La variable W est associée au flot g au cycle $c+2$. La valeur de *findcycle* reste indéterminée tant que le domaine de W n'est pas suffisamment réduit.

4.3.2.2 Progression du calcul par propagation

La fonction *fcwp* ne fait qu'observer l'ensemble de contraintes pour déterminer s'il est possible ou non de trouver un cycle précédent. Lorsqu'on propage les contraintes du système, on cherche à s'assurer que *fcwp* finit par donner un cycle comme résultat, en débloquant les situations qui empêchent la recherche d'aboutir.

Dans notre exemple, on ajoute une nouvelle variable W dans S que l'on associe à g au cycle $c+2$. On ajoute pour cela la contrainte suivante :

$$W \Rightarrow_{(c+2,(f,a))} g$$

On obtient après filtrages successifs la grille de valeurs de la figure 4.9, où cette fois apparaît la variable W ainsi que son domaine. L'appel à *fcwp* retourne cette fois la valeur *wait*$(c+2, g, W)$, car ce domaine ne permet pas de savoir si (g, u) est ou non présente au cycle $c+2$.

4.3.2.3 Bloquage par l'horloge de base

Nous instancions W avec la valeur v. Cette fois, on sait que (g, u) est absente au cycle $c+2$ et on incrémente le cycle candidat pour chercher cette fois à $c+3$. À ce cycle, on dépasse le cycle maximal associé à l'horloge de base. Précédemment, ce cycle maximal était $c+2$. Le résultat retourné par *fcwp* était alors *status(c+3, St)*, avec St le statut associé à l'horloge de base. En instanciant le statut à *noninitial*, on obtient la grille de la figure 4.10. Finalement, l'ensemble de contraintes S_1 obtenu est tel que :

$$S_1 \vdash fcwp(H,c) = undef(c+3, f)$$

Ici, on bloque sur le flot f qui est indéfini au cycle $c+3$.

Cycles	$c+3$	$c+2$	$c+1$	c
base	∘	×	×	×
→ f		a	b	a
→ (f,a)		∘		×
→ g		v		u
→ (g,u)				∘

FIGURE 4.10: On sait que g vaut v au cycle $c+2$; la recherche du cycle précédent doit continuer au cycle $c+3$, ce qui n'est possible que si on repousse le cycle maximal associé à l'horloge de base.

4.3.2.4 Grille après propagations

On propage f puis g au cycle $c+3$ de sorte que (g, u) soit présente au cycle $c+3$. Le CSP obtenu est noté S_2. Cette fois, le résultat pour *fcwp* est bien $c+3$, ce qui nous permet de conclure pour l'appel initial.

$$S_2 \vdash findcycle(H,c) = c+3$$

Nous décrivons dans la section qui suit les règles qui nous permettent d'évaluer *fcwp* dans le contexte d'un CSP. La fonction observe des séquences partiellement générées : nous devons ajouter des contraintes dans le CSP pour faire en sorte que le résultat soit bien le cycle cherché. Par la suite, nous montrerons comment la contrainte *precycle* introduit ces contraintes en fonctions des valeurs symboliques retournées par *fcwp*.

Cycles	$c+3$	$c+2$	$c+1$	c
base	∘	×	×	×
→ f	a	a	b	a
→ (f,a)	∘	×		×
→ g	u	v		u
→ (g,u)	∘			×

FIGURE 4.11: Après d'autres propagations intermédiaires, le cycle précédent de (g, u) au cycle c est finalement trouvé et vaut $c+3$.

4.3.3 Sémantique de *fcwp*

4.3.3.1 Fin de la recherche

Lorsque l'horloge H qui nous intéresse est présente au cycle c, ce cycle c est le résultat que l'on cherche :

$$\text{FCWP-FOUND}\ \frac{S \vdash \mathit{present(H,c)}}{S \vdash \mathit{fcwp(H,c)} = c}$$

4.3.3.2 Flot support indéfini

Il est possible que l'horloge H ne soit pas définie au cycle c. Dans ce cas, il s'agit d'une horloge relative $H = (f, v)$. On rappelle que l'horloge est indéfinie lorsque son flot support n'est pas encore présent dans la grille de valeurs générée. Dans ce cas, on doit observer l'horloge mère de H, à savoir l'horloge de f, que l'on note M. Il y a deux cas possibles, à savoir :

1. L'horloge M elle même est indéfinie : on cherche le cycle de présence le plus proche pour M avant de continuer la recherche avec H.
2. L'horloge M est définie : l'horloge bloquante est H.

Le second cas est représenté par la règle ci-dessous, où l'horloge de f est présente, mais f est indéfini. Dans ce cas, on interrompt la recherche avec le terme *undef* qui indique la raison pour laquelle le cycle précédent ne peut pas être trouvé : le flot f doit être propagé avant de pouvoir réappliquer *fcwp*.

$$\text{FCWP-UNDEF}\ \frac{S \vdash \mathit{clock(f)=M} \qquad S \vdash \mathit{present(M,c)} \qquad S \vdash \mathit{undef(f,\ c)}}{S \vdash \mathit{fcwp((f,v),c)} = \mathit{undef(c,\ f)}}$$

Le premier cas nous demande de traiter les horloges mères et de pouvoir ensuite revenir au traitement de l'horloge courante. Nous détaillons ceci dans le paragraphe qui suit.

4.3.3.3 Appel récursif vers les horloges mères

Si l'horloge mère M de H n'est elle-même pas définie au cycle courant, on cherche d'abord à trouver un cycle où M est définie. Pour cela, on itère sur les cycles à l'aide de *fcwp* à partir du cycle courant c en considérant cette fois l'horloge M. Une fois qu'un cycle précédent d sera trouvé pour M, on pourra y continuer la recherche du cycle précédent pour l'horloge fille $H = (f, v)$ (première règle ci-dessous). Cependant, si le calcul du cycle précédent de l'horloge mère est interrompu et retourne le terme R, on interrompt aussi celui de H avec le même résultat R (seconde règle).

$$\textsc{fcwp-mother-ok}\ \frac{M = clock(f){=}M \qquad S \vdash undefclock(M,c) \qquad S \vdash fcwp(M,c) = d \qquad d \in \mathbb{N} \qquad S \vdash fcwp((f,v),d) = R}{S \vdash fcwp((f,v),c) = R}$$

$$\textsc{fcwp-mother-ko}\ \frac{M = clock(f){=}M \qquad S \vdash undefclock(M,c) \qquad S \vdash fcwp(M,c) = R \qquad R \notin \mathbb{N}}{S \vdash fcwp((f,v),c) = R}$$

Ces deux règles traitent les cas des horloges mères intermédiaires. Il nous reste encore à voir les cas où le flot support qui nous intéresse est bien défini.

4.3.3.4 Valeur du flot support

Le flot support f d'une horloge relative $H = (f, v)$ admet maintenant une valuation dans la grille générée. Cette valuation est notée W et admet pour sur-approximation le domaine D_W. Lorsque la valeur de présence v n'est *pas* dans le domaine D_W, alors l'horloge H ne peut pas être présence au cycle c. Dans ce cas, on continue la recherche au cycle $c + 1$:

$$\textsc{fcwp-neq}\ \frac{S \vdash f \rightarrowtail_c W \qquad S \vdash W \approx D_W \qquad v \notin D_W \qquad S \vdash fcwp((f,v),c{+}1) = R}{S \vdash fcwp((f,v),c) = R}$$

Le cas où le flot vaut exactement v est couvert par la règle FCWP-FOUND précédente. Il est possible aussi que le domaine soit trop imprécis pour déterminer si H est ou non présente au cycle c. On doit alors attendre une réduction de domaine pour W avant de poursuivre, ce qui est symbolisé par le terme de retour *wait* suivant :

$$\textsc{fcwp-wait}\ \frac{S \vdash f \rightarrowtail_c W \qquad S \vdash W \approx D_W \qquad v \in D_W \qquad D_W \setminus \{v\} \neq \emptyset}{S \vdash fcwp((f,v),c) = wait(c,\ f,\ W)}$$

4.3.3.5 Statut bloquant

Le dernier cas où *fcwp* peut être interrompue (sans échec) se produit lorsque le nombre de cycles déjà générés dans la grilles de valeurs est insuffisant. Cela se manifeste au niveau du statut lié à l'horloge de base ; si le paramètre c de *fcwp* est supérieur au cycle maximal associé à *base* (et s'il est bien inférieur à la borne maximale prédéfinie), alors *fcwp* retourne le terme *status(St)* qui indique le blocage :

$$\text{FCWP-STATUS}\ \frac{S \vdash cmax(base, sc, St) \qquad S \vdash maxsize(mc) \qquad mc > c > sc}{S \vdash fcwp(base,c) = status(c, St)}$$

4.3.3.6 Cas d'erreurs

Finalement, il existe des cas où il n'est tout simplement pas possible de trouver un cycle précédent, non pas parce que l'on manque d'informations mais parce que le CSP est insatisfaisable. La règle ci-dessous nous permet de détecter une insatisfaisabilité lorsque le cycle en argument de *fcwp* est égal ou supérieur à la borne maximale.

$$\text{FCWP-FAIL-1}\ \frac{S \vdash maxsize(mc) \qquad c \geq mc}{S \vdash fcwp(H,c) = fail}$$

On peut aussi arrêter de chercher le cycle précédent d'une horloge H si on sait qu'il n'existe pas de cycle précédent à l'instant c qui nous intéresse. C'est le cas si le statut de l'horloge est *initial* et que l'on a dépassé son cycle initial.

$$\text{FCWP-FAIL-2}\ \frac{S \vdash cmax(H, hc, initial) \qquad c > hc}{S \vdash fcwp(H,c) = fail}$$

4.3.4 Construction des cycles précédents

La fonction *findcycle* se contente d'observer le CSP courant. Selon que les séquences contiennent ou non suffisamment d'informations, le calcul d'un cycle précédent peut aboutir ou être interrompu. Nous définissons maintenant une contrainte de la forme *precycle(H,c)*. Celle-ci est introduite lorsqu'on souhaite qu'au cycle c, l'horloge H admette un cycle précédent d où celle-ci est présente.

Pour cela, la contrainte fait en sorte que lorsque l'on appelle *findcycle* pour H et c, le résultat soit un entier, le cycle précédent. Elle réalise cela en débloquant les situations qui empêchent à *findcycle* de progresser (flot indéfini, domaine trop imprécis ou statut de l'horloge de base bloquant). Une fois que cela est fait, la contrainte *precycle* disparaît du système de contraintes.

Nous devons prendre quelques précautions pour éviter que les règles ne soient applicables plusieurs fois de suite. Pour cela, nous passons par une contrainte *precycle*

à trois arguments :

$$\textsc{precycle-start}\ \frac{S \vdash \mathit{findcycle(H,c)} = R}{\vdash S \cup \{\mathit{precycle(H,c)}\} \rightarrow S \cup \{\mathit{precycle(H,c,unblock(R))}\}}$$

Dans la règle ci-dessus, le résultat R retourné par *findcycle* est encaspulé dans un terme *unblock*, lui-même placé comme troisième élément de la nouvelle contrainte *precycle*.

Le terme *unblock* nous sert à garantir que les règles de déblocages ci-dessous ne sont applicables qu'une fois. Les deux première règles prennent en compte le cas trivial où le résultat est un cycle et celui-où on détecte une insatisfaisabilité. Dans le premier cas, la contrainte disparaît du CSP. Dans le second, le CSP devient triviallement insatisfaisable à l'aide de la contrainte *fail*.

$$\textsc{unblock-done}\ \frac{d \in \mathbb{N}}{S \cup \{\mathit{precycle(H,c,unblock(d))}\} \rightarrow S}$$

$$\textsc{unblock-fail}\ \frac{}{S \cup \{\mathit{precycle(H,c,unblock(fail))}\} \rightarrow \{\mathit{fail}\}}$$

Les deux règles qui suivent contraignent le CSP pour faire en sorte que *findcycle* surmonte le blocage décrit par le troisième argument. La première règle introduit une contrainte *propage* pour les flots qui sont indéfinis dans la grille de valeurs :

$$\textsc{unblock-undef}\ \frac{R = \mathit{undef(g,\ d)}}{S \cup \{\mathit{precycle(H,c,unblock(R))}\} \rightarrow S \cup \{\mathit{precycle(H,c,R)}, \mathit{propage(g,d)}\}}$$

La deuxième fait en sorte que le statut bloquant associé à l'horloge de base soit unifié avec *noninitial* :

$$\textsc{unblock-status}\ \frac{R = \mathit{status(d,\ St)}}{S \cup \{\mathit{precycle(H,c,unblock(R))}\} \rightarrow S \cup \{\mathit{precycle(H,c,R)}, \mathit{St} \equiv \mathit{noninitial}\}}$$

Finalement, lorsque le terme de retour est de la forme *wait(c,f,W)*, on ne peut pas débloquer localement la situation. On doit attendre qu'une réduction de domaine ait lieu dans le CSP courant. Nous verrons à la section §4.3.6 que la sélection de cas de test peut débloquer la situation en instanciant W. Pour le moment, on se contente ici de retirer le terme *unblock* :

$$\textsc{unblock-wait}\ \frac{}{S \cup \{\mathit{precycle(H,c,unblock(wait(c,f,W)))}\} \rightarrow S \cup \{\mathit{precycle(H,c,wait(c,f,W))}\}}$$

Dans les trois règles précédentes, le terme *unblock* est retiré dans le CSP après filtrage. Cela évite par exemple d'introduire plusieurs contraintes *propage* à la suite,

ou d'unifier plusieurs fois le statut avec *noninitial* : sans connaître à priori la stratégie d'application des règles, on ne peut pas se fier au fait que les nouvelles contraintes soit filtrées immédiatement. Cependant, une fois qu'elles seront propagées, le résultat de *findcycle* sera modifié. Par exemple, au lieu d'être interrompu sur le flot indéfini g donné par *undef(g, d)*, la fonction pourra avoir plus informations sur la valeur de g au cycle d et se rapprocher d'un résultat entier. Pour mettre en œuvre cette progression, la contrainte *precycle* est modifiée à chaque fois que l'on peut appliquer *findcycle* et obtenir un résultat différent du dernier résultat obtenu :

$$\textsc{precycle-step}\ \frac{S \vdash \mathit{findcycle(H,c)} = R_2 \qquad R_1 \neq R_2}{S \cup \{\mathit{precycle(H,c,}R_1\mathit{)}\} \rightarrow S \cup \{\mathit{precycle(H,c,unblock(}R_2\mathit{)\,)}\}}$$

À chaque étape, le nouveau résultat R_2 est différent du résultat R_1 précédent. Le nouveau résultat est placé dans un terme *unblock*, pour laisser les contraintes de débloquages agir. Les différentes règles que l'on vient de définir nous donnent ainsi une contrainte *precycle(H,c)* qui fait évoluer un CSP jusqu'à ce qu'il y ait une insatisfaisabilité ou que l'on trouve un cycle d tel que *findcycle(H,c)* = d dans le contexte du nouveau système.

4.3.5 Propagation de `pre`

Nous pouvons maintenant traiter l'opérateur `pre` temporisé. Par définition du langage $\mathcal{L}_+$, un opérateur `pre` est toujours sous la portée d'un opérateur d'initialisation de plus haut-niveau. Or, nous verrons que la propagation du membre droit de celui-ci conduit à instancier le statut de l'horloge H à *noninitial* (cf. §4.4.1 p. 162), auquel cas on on ajoute une contrainte *precycle* (cf. §4.2.6.2) : celle-ci se charge de garantir l'existence d'un cycle précédent pour H au cycle c. La propagation d'un `pre` n'a donc pas besoin d'ajouter elle-même une contrainte *precycle* et peut se contenter d'attendre que la fonction *findcycle* soit évaluable vers un entier :

$$\textsc{propage-pre}\ \frac{S \vdash \mathit{findcycle(H,c)} = nc \qquad nc \in \mathbb{N}}{\vdash S \cup \{R \Rightarrow_{(c,H)} \mathit{pre(E)}\} \rightarrow S \cup \{R \Rightarrow_{(nc,H)} E\}}$$

4.3.6 Sélection de cas de test

Le seul cas où la contrainte *precycle* précédente ne peut pas faire en sorte que *findcycle* puisse progresser est celui où la recherche bloque sur une variable W dont le domaine de valeurs est trop imprécis pour déterminer la valeur du flot support qu'elle représente, et ainsi déterminer quelle horloge définie sur ce flot est active. Le cas qui nous intéresse donc est celui où la fonction *findcycle* retourne un terme *wait(c,f,W)*. On peut remarquer que f est forcément un flot de type énuméré, puisqu'il s'agit toujours d'un flot support d'une horloge.

Nous proposons donc une règle pour la sélection de cas de test présentée au chapitre précédent (p. 103) qui va construire autant de cas de tests qu'il existe de valeurs possibles dans le type énuméré de f (le nombre de cas ainsi générés n'est jamais très grand en pratique). Dans chacun des sous-problèmes obtenus, W est unifié avec une valeur du type, permettant à la contrainte *precycle* de progresser. Nous associons cette règle de sélection à la contrainte *propage* dans le cas de l'opérateur `pre`, tel que cela est réalisé dans GATeL$_+$:

$$\text{SELECT-PRE}\ \frac{\begin{array}{c} S \vdash \mathit{findcycle(H,c)} = \mathit{wait(d,f,W)} \\ S \vdash \mathit{type(f,T)} \qquad S \vdash \mathit{enum(T,[t_1, \ldots, t_n])} \end{array}}{S \cup \left\{ R \Rightarrow_{(c,H)} \mathit{pre(E)} \right\} \vdash \mathit{select} \left\{ \begin{array}{c} \{W \equiv t_1\}, \\ \vdots \\ \{W \equiv t_n\} \end{array} \right\}}$$

4.4 Initialisation, réinitialisation

L'opérateur d'initialisation « → » a été modifié entre le langage mono-horloge $\mathcal{L}_1$ et le langage $\mathcal{L}_+$. D'une part, la présence d'horloges nécessite de tenir compte des statuts associés aux horloges des flots qui nous intéressent. Mais surtout, l'opérateur admet maintenant une forme étendue qui permet de réinitialiser l'opérateur en fonction de zéro, un ou plusieurs flots de redémarrage. Le cas où ce nombre de flots est nul est ainsi équivalent à la flèche simple.

Dans la suite de cette partie, nous nous intéressons aux mécanismes qui traitent de l'opérateur « → ∗ ». On rappelle que cet opérateur est transformé par la compilation en un terme T = *init(A,B,RList)* dans le langage de contraintes. Les deux premiers arguments sont les sous-expressions de part et d'autre de la flèche. Le troisième est une liste d'idenfitiants de flots booléens. On appelle *horloge principale* du terme T l'horloge commune à A et à B. On appelle *flots de redémarrage* les flots contenus dans la liste *RList*. Ceux-ci ont chacun une horloge propre indépendante de l'horloge principale du terme. L'horloge principale peut ou non admettre au cycle c de la propagation un cycle précédent d. La fenêtre de réinitialisation est donnée par l'intervalle de cycles $(d+1)..c$; on rappelle qu'au moins une occurrence de la valeur *true* pour n'importe quel flot de redémarrage dans cette fenêtre de temps indique une réinitialisation (ou rédemarrage) des flots, signifiant que la branche à évaluer devient nécessairement la branche de gauche au cycle c.

Il existe une relation entre le statut de l'horloge principale au cycle c, la branche qui est propagée et le fait qu'une réinitialisation ait lieu ou non. Dans la section §4.4.1 ci-après, nous explicitons cette relation à travers la contrainte *init*. Nous donnons à la section §4.4.2 (p. 164) la règle de filtrage pour *propage* dans le cas de l'opérateur « →∗ ». Comme l'opérateur est complexe, le traitement de la contrainte est délegué à deux autres

contraintes *propage* portant sur deux nouveaux termes abstraits *branch* et *reset* que nous décrivons respectivement dans les sections §4.4.3 et §4.4.4 (p. 165).

Les règles que nous définissons ici diffèrent de l'implémentation actuelle réalisée dans GATeL$_+$ car on cherche avant tout à décrire l'approche et non les mécanismes exacts qui ont lieu. C'est notamment le cas à la dernière sous-section où, par exemple, la réinitialisation est traitée à l'aide de contraintes multiples au-lieu de recourir à des parcours de listes.

4.4.1 Contrainte *init*

Lorsqu'on associe un résultat R à l'expression « a $\rightarrow *$ $(r_1, \ldots, r_n)$ b », on doit considérer plusieurs alternatives :

- Le résultat peut provenir du membre gauche *(left)* ou droit *(right)* de la flèche.
- Le cycle de la propagation peut être le cycle *initial* ou un cycle ultérieur (*noninitial*).
- L'opérateur peut ou non être réinitialisé (*reset* ou *nonreset*), selon la valeur prise par les flots de redémarrage dans la fenêtre temporelle où ceux-ci ont une influence sur le résultat (entre le cycle précédent exclus et le cycle courant inclus).

Parmi les $2^3 = 8$ combinaisons possibles, 4 correspondent à des situations correctes vis-à-vis de la sémantique de l'opérateur $\rightarrow *$; les 4 autres sont des cas d'erreurs : si elles se produisent dans le CSP, on détecte l'insatisfaisabilité du problème. Nous résumons les différents cas de figure dans la table ci-dessous, où les cas possibles sont marqués d'un 1 et les cas incorrects d'un 0 :

	left		*right*	
	reset	*nonreset*	*reset*	*nonreset*
initial	1	1	0	0
noninitial	1	0	0	1

Par exemple, il n'est pas possible que le résultat soit unifiable uniquement avec le membre droit de la flèche alors que le cycle courant est *initial*. Nous définissons alors une contrainte *init* à trois arguments et différentes règles de filtrages qui modélisent la relation vue dans la table précédente. Une contrainte *init(Status,Branch,Reset)* porte trois arguments dont les domaines respectifs sont les suivants :

$$\begin{aligned} D_{\texttt{status}} &= \{noninitial, initial\} \\ D_{\texttt{branch}} &= \{left, right\} \\ D_{\texttt{reset}} &= \{nonreset, reset\} \end{aligned}$$

Nous prenons en compte tous les cas possibles pour les 3 variables : le tableau suivant montre quelle règle est appliquée dans les différents cas possibles, en suivant

la numérotation des règles ci-après. Les points d'interrogation représentent les cas où les variables ne sont pas instanciées. Certaines combinaisons d'arguments ne permettent pas de déduire de filtrage car il n'y a pas suffisamment d'information. Elles sont représentée par des croix (×) dans le tableau.

	left			*right*			*?*		
	reset	*nonreset*	*?*	*reset*	*nonreset*	*?*	*reset*	*nonreset*	*?*
initial	1.a	1.a	1.a	1.a	1.a	1.a	1.a	1.a	1.a
noninitial	2.a	2.a	2.a	2.a	2.a	2.a	2.b	2.b	×
?	3.a	3.a	3.a	3.a	3.a	3.a	3.b	×	×

Les règles sont les suivantes :

1. Statut initial

$$\vdash S \cup \{ init(initial,Branch,Reset) \} \rightarrow S \cup \{ Branch \equiv left \}$$

2. Statut non-initial
   (a) Branche connue

$$\vdash S \cup \{ init(noninitial,left,Reset) \} \rightarrow S \cup \{ Reset \equiv reset \}$$
$$\vdash S \cup \{ init(noninitial,right,Reset) \} \rightarrow S \cup \{ Reset \equiv nonreset \}$$

   (b) Branche inconnue

$$\frac{Branch \in \mathbb{V}}{\begin{array}{c} \vdash S \cup \{ init(noninitial,Branch,reset) \} \rightarrow S \cup \{ Branch \equiv left \} \\ \vdash S \cup \{ init(noninitial,Branch,nonreset) \} \rightarrow S \cup \{ Branch \equiv right \} \end{array}}$$

3. Statut inconnu
   (a) Branche connue

$$\frac{Status \in \mathbb{V}}{\begin{array}{c} \vdash S \cup \{ init(Status,left,nonreset) \} \rightarrow S \cup \{ Status \equiv initial \} \\ \vdash S \cup \{ init(Status,right,Reset) \} \rightarrow S \cup \{ Status \equiv noninitial, Reset \equiv nonreset \} \end{array}}$$

   (b) Branche inconnue

$$\frac{Status \in \mathbb{V} \qquad Branch \in \mathbb{V}}{\vdash S \cup \{ init(Status,Branch,reset) \} \rightarrow S \cup \{ Branch \equiv left \}}$$

La contrainte *init* contrôle le filtrage de *propage* dans le cas de l'opérateur $\rightarrow *$.

4.4.2 Contrainte *propage* pour l'opérateur de réinitialisation

Nous répartissons différents aspects du traitement de l'opérateur de réinitialisation dans différentes contraintes. La règle pour *propage* pour cet opérateur est ainsi la suivante :

$$\textsc{propage-init}\ \frac{S \vdash \mathit{status(H,c)} : \mathit{Status} \qquad S \cup \{\mathit{Value}\} \vdash \mathit{Branch}, \mathit{Reset} : \mathit{fresh}}{\vdash S \cup \{\mathit{Value} \Rightarrow_{(c,H)} \mathit{init(A,B,RList)}\} \rightarrow S \cup \left\{ \begin{array}{l} \mathit{Branch} \in D_{\texttt{branch}} \\ \mathit{Reset} \in D_{\texttt{reset}} \\ \mathit{init(Status,Branch,Reset)} \\ \mathit{Value} \Rightarrow_{(c,H)} \mathit{branch(A,B,Branch)} \\ \mathit{Reset} \Rightarrow_{(c,H)} \mathit{reset(RList)} \end{array} \right\}}$$

4.4.2.1 Contrainte *init*

Nous ajoutons les variables *Branch* et *Reset* dans le CSP en leur affectant respectivement les domaines $D_{\texttt{branch}}$ et $D_{\texttt{reset}}$ vus dans la section précédente. Alors, nous introduisons la contrainte *init* qui va observer ces deux variables ainsi que le statut d'horloge au cycle c : lorsque par ailleurs les variables seront instanciées, *init* projettera les réductions de domaines d'une variable à une autre selon les règles vues à la section §4.4.1.

4.4.2.2 Branchement

Nous introduisons un nouveau terme abstrait *branch* à trois arguments filtrable pour la contrainte *propage*. La contrainte cherche alors à déterminer par sur-approximation si une des branches A (gauche) ou B (droite) est incompatible avec le terme résultat *Value*, et ainsi instancier la variable *Branch*. Une fois la valeur de *Branch* connue, elle propage *Value* dans la sous-expression correspondante au cycle c et avec l'horloge H. Le filtrage de ce terme est présenté à la section §4.4.3.

4.4.2.3 Réinitialisation

Nous définissons aussi le terme *reset(RList)* comme sous-expression valide de *propage*, qui se charge alors de savoir s'il existe ou non une réinitialisation, étant donnés la liste des flots de redémarrage *RList*, l'horloge H et le cycle courant. Par ailleurs, si la variable *Reset* est un atome (*reset* ou *nonreset*), la contrainte fait en sorte que les flots présents dans *RList* reflètent cette valeur. Nous détaillons le filtrage de *reset* à la section §4.4.4 (p. 165).

4.4.2.4 Conclusion

La gestion du terme $\rightarrow *$ est complexe car elle fait intervenir des choix concernant la sous-expression devant être propagée, le statut de l'horloge H et l'existence ou non

d'une réinitialisation. Nous avons exprimé cette relation à l'aide de la contrainte *init*, à travers laquelle on fait communiquer des contraintes dédiées à chaque composante. Par exemple, on sait déjà que l'instanciation de la variable de statut peut déclencher ou provenir d'autres contraintes de la table d'horloges. Les sections qui suivent détaillent les deux autres aspects, à savoir la propagation dans les sous-expressions et la gestion des flots de redémarrage.

4.4.3 Branchement dans une sous-expression

Nous nous intéressons au filtrage de *propage* pour le terme *branch* à trois arguments. Les deux premières sont respectivement les expressions à gauche et à droite de la flèche de l'opérateur d'initialisation, alors que le troisième indique laquelle de ces branche s'unifie avec le résultat propagé. Cet argument peut valoir *left* ou *right*. Les deux règles triviales pour filtrer le terme *branch* sont celles où le troisième argument est connu :

$$\textsc{left}\ \frac{}{\begin{array}{c}\vdash S \cup \{ r \Rightarrow_{(c,H)} \mathit{branch(A,B,left)} \} \\ \rightarrow S \cup \{ r \Rightarrow_{(c,H)} A \}\end{array}} \qquad \textsc{right}\ \frac{}{\begin{array}{c}\vdash S \cup \{ r \Rightarrow_{(c,H)} \mathit{branch(A,B,right)} \} \\ \rightarrow S \cup \{ r \Rightarrow_{(c,H)} B \}\end{array}}$$

Cependant, on peut aussi filtrer la contrainte lorsque le troisième argument est variable, s'il est possible d'exclure une des branches par sur-approximation. Dans les deux règles qui suivent, le domaine donné par la sur-approximation est incompatible avec celui du résultat R, ce qui a pour effet d'unifier la variable BR et de propager le résultat dans la branche opposée.

$$\neg\textsc{left}\ \frac{\begin{array}{c} BR \in \mathbb{V} \\ r \approx D_R \qquad A \approx_{(c,H)} D_A \\ D_A \cap D_R = \emptyset \end{array}}{\begin{array}{c}\vdash S \cup \{ r \Rightarrow_{(c,H)} \mathit{branch(A,B,BR)} \} \\ \rightarrow S \cup \{ BR \equiv \mathit{right}, r \Rightarrow_{(c,H)} B \}\end{array}} \qquad \neg\textsc{right}\ \frac{\begin{array}{c} BR \in \mathbb{V} \\ r \approx D_R \qquad B \approx_{(c,H)} D_B \\ D_B \cap D_R = \emptyset \end{array}}{\begin{array}{c}\vdash S \cup \{ r \Rightarrow_{(c,H)} \mathit{branch(A,B,BR)} \} \\ \rightarrow S \cup \{ BR \equiv \mathit{left}, r \Rightarrow_{(c,H)} A \}\end{array}}$$

4.4.4 Traitement des flots de redémarrage

Nous donnons ici les règles de filtrage, pour *propage* dans le cas des termes de la forme *reset(RList)* ainsi que pour d'autres contraintes, qui nous permettent de déterminer si une réinitialisation a lieu pour un opérateur $\rightarrow *$. Cela dépend des valeurs prises par les flots de redémarrages de *RList* pendant une période de temps comprise entre le cycle précédent de l'horloge principal H et le cycle courant c.

4.4.4.1 Liste vide

Le cas trivial où la list est vide nous donne un statut *nonreset* :

$$\textsc{propage-reset-empty}\ \frac{}{\vdash S \cup \{ \mathit{Reset} \Rightarrow_{(c,H)} \mathit{reset([])} \} \rightarrow S \cup \{ \mathit{Reset} \equiv \mathit{nonreset} \}}$$

4.4.4.2 Une contrainte *rst* par élément

Si maintenant *RList* est non-vide, on sait que chacun des flots de redémarrage est indépendant des autres et admet une horloge quelconque. Nous choisissons donc de placer une contrainte par flot : chacune indique si le flot qu'elle gère provoque ou non la réinitialisation de l'opérateur.

$$\textsc{propage-reset-list}\ \frac{\begin{array}{c} RList = [r_1,\ldots,r_n] \qquad n>0 \qquad S \vdash Rst : fresh \\ R = \left\{ Rst \Rightarrow_{(c,H)} rst(r_i,c) \mid 1 \le i \le n \right\} \end{array}}{\begin{array}{r} \vdash S \cup \left\{ Reset \Rightarrow_{(c,H)} reset(RList) \right\} \\ \rightarrow S \cup \left\{ Reset \Rightarrow_{(c,H)} reset(Rst,RList)\,, Rst \in \{reset, cancel\} \right\} \cup R \end{array}}$$

Dans la règle que l'on vient de donner, plusieurs actions ont lieu. Tout d'abord, on transforme le terme courant *reset(RList)* en un terme *reset(Rst,RList)* où *Rst* est une variable libre. Ensuite, pour chaque flot r_i présent dans *RList*, on construit une contrainte de la forme suivante :

$$Rst \Rightarrow_{(c,H)} rst(r_i,d)$$

Il s'agit d'une contrainte *propage* dédiée au traitement d'un flot de redémarrage r_i par l'intermédiaire du terme *rst* : cette contrainte observe le flot r_i dans la fenêtre temporelle qui nous intéresse, au cycle d, en le propageant quand nécessaire. Le cycle d vaut initialement c. Les valeurs de c et de H sont conservées à travers *propage* car elles mémorisent l'instant auquel l'opérateur $\rightarrow *$ est propagé ainsi que son horloge (l'horloge principale). La variable *Rst* est introduite et partagée uniquement entre les contraintes *rst* issues du même filtrage.

4.4.4.3 Observation de la réinitialisation à travers *Rst*

La variable *Rst* est dans le domaine $D_{\texttt{rst}} = \{reset, cancel\}$. Cela est dû au fonctionnement de *rst*. Le cas particulier où à sa valeur est *cancel* est détaillé à la section §4.4.4.6. Mis à part ce cas, le principe général de *rst* est le suivant : chaque terme *rst* peut unifier la variable à *reset* dès que le flot r_i qui lui est associé est évaluable à *true*. Sinon, si ce flot est absent ou vaut *false*, on cherche à décaler le cycle d'observation d de *rst*, toujours en restant dans les limites de temps qui ont influence sur la réinitialisation. Lorsqu'il n'est plus possible de décaler le terme *rst* dans le passé, à cause de la limite fixée par la fenêtre de réinitialisation où parce que le flot r_i a atteint son cycle initial, la contrainte qui le porte est retirée du CSP courant. C'est seulement quand toutes les contraintes *rst* sont retirées que l'on peut en déduire qu'il n'y a pas de réinitialisation. À ce moment, on peut unifier *Reset* avec *nonreset* :

$$\textsc{reset-no-more-rst}\ \frac{\begin{array}{c} RList = [r_1,\ldots,r_n] \qquad n>0 \qquad Rst \in \mathbb{V} \\ \left\{ Rst \Rightarrow_{(c,H)} rst(r,d) \in S \mid (r,d) \in \{r_1,\ldots,r_n\} \times \mathbb{N} \right\} = \emptyset \end{array}}{\vdash S \cup \left\{ Reset \Rightarrow_{(c,H)} reset(Rst,RList) \right\} \rightarrow S \cup \{Reset \equiv nonreset\}}$$

L'intérêt de la variable *Rst* est qu'elle représente l'état observé de la réinitialisation, à la différence de *Reset* qui représente l'état souhaité : il est en effet possible que *Reset* soit déjà instancié au moment où on propage les flots r_i, alors que *Rst* est nécessairement libre jusqu'à ce qu'un flot vaille *true*. Alors, quand *Rst* est instancié à *reset* on peut l'unifier avec *Reset* pour voir si cela correspond au statut souhaité :

$$\text{RESET-IS-RESET}\ \frac{}{\vdash S \cup \{ Reset \Rightarrow_{(c,H)} reset(reset, RList) \} \rightarrow S \cup \{ Reset \equiv reset \}}$$

En parallèle, les autres contraintes *rst* peuvent être retirées du CSP, car elles ne changeront plus la valeur finale de *Reset* :

$$\text{RST-REMOVE-ATOMIC}\ \frac{Rst \in \mathbb{A}}{\vdash S \cup \{ Rst \Rightarrow_{(c,H)} rst(r,d) \} \rightarrow S}$$

On note aussi qu'à la différence des autres variables du CSP, les variables *Rst* ne sont pas prévues pour être instanciées autrement que par les règles de cette section. En particulier, si à l'étape de résolution on choisit cette variable et qu'on l'instancie arbitrairement, la propagation de contraintes peut avoir un comportement inattendu (on considère qu'il s'agit d'une variable locale à la contrainte *reset*). Nous nous intéressons maintenant aux différents filtrages possibles pour *rst*.

4.4.4.4 Exploration de la fenêtre de réinitialisation avec *rst*

Dans les contraintes qui suivent, le cycle d associé une contrainte *rst* est toujours compris dans la fenêtre de réinitialisation de l'opérateur que l'on traite. Cet intervalle est compris entre le cycle précédent nc (exclus) de l'horloge principale H et le cycle c (inclus). Aux cycles qui précèdent nc où qui suivent c, les flots n'ont aucune influence sur la variable *Reset*. Il est donc important de ne pas propager de contraintes en dehors de l'intervalle de cycle compris entre nc et c.

Fenêtre temporelle partielle Si on connaît exactement le cycle précédent, il est facile de vérifier que d est bien compris dans l'intervalle souhaité. Cependant, la grille de valeurs comprise dans le CSP est partiellement construite et ne nous permet pas toujours de calculer nc. Il s'agit du cas ou la fonction *findcycle(H,c)* = R retourne un résultat R qui n'est pas un cycle. Plutôt que d'attendre que le cycle soit connu avant de commencer à propager les contraintes *rst*, nous commençons à propager directement chacune des contraintes *rst* et nous nous basons sur le résultat partiel renvoyé par *findcycle* pour avoir un minorant du cycle précédent nc. Ce minorant est donné par la fonction *cycle(R)* dans la définition 16 suivante.

Définition 16. *Cycle maximal atteint par findcycle et fcwp*

Soient $c \in \mathbb{N}$, X et Y deux termes de $\mathcal{C}_+^g$. Nous définissons alors la fonction *cycle* qui nous donne le sous-terme associé au cycle dans la valeur de retour des fonctions *findcycle* et *fcwp* (elle est indéfinie pour *fail*) :

$$\begin{array}{ll} cycle(undef(c,X)) = c & cycle(wait(c,X,Y)) = c \\ cycle(status(c,X)) = c & cycle(c) = c \end{array}$$

Alors, nous définissons le prédicat *checkcycle* qui nous indique si un cycle d est bien compris entre le cycle de départ c et le cycle du résultat de *findcycle*.

$$\textsc{checkcycle}\ \frac{S \vdash findcycle(H,c) = Result \qquad cycle(Result) > d \geq c}{S \vdash checkcycle(H,c,d)}$$

Nous choisissons cette approche pour pouvoir tirer parti des flots de redémarrages, qui peuvent par exemple être déjà connus et valoir *true*, sans avoir à attendre de connaître la valeur définitive de nc. On limite alors la propagation des contraintes *rst* aux cycles compris entre c et la borne temporaire donnée par *cycle(Result)*.

Décalage de *rst* La règle de propagation suivante traite le cas où le cycle d d'un terme *rst* est modifié :

$$\textsc{rst-change-cycle}\ \frac{\begin{array}{c} S \vdash clock(r){=}H_r \qquad S \vdash changecycle(r, H_r, d) \\ S \vdash findcycle(H_r, d) = nd \qquad nd \in \mathbb{N} \qquad S \vdash checkcycle(H,c,d) \end{array}}{\vdash S \cup \{Rst \Rightarrow_{(c,H)} rst(r,d)\} \rightarrow \{Rst \Rightarrow_{(c,H)} rst(r,nd)\}}$$

On récupère tout d'abord l'horloge H_r du flot booléen r. Une première condition pour appliquer cette règle est qu'il faut être dans une situation où le cycle courant n'est certainement pas un cycle où r vaut *true*. Cette information est donnée par le prédicat *changecycle*, qui admet deux cas : soit l'horloge du flot r est absente au cycle d, soit le flot r y vaut *false*.

$$\textsc{cc-absent}\ \frac{S \vdash absent(H_r,d)}{S \vdash changecycle(r,H_r,d)} \qquad \textsc{cc-false}\ \frac{S \vdash r \mapsto_d false}{S \vdash changecycle(r,H_r,d)}$$

Alors, la contrainte RST-CHANGE-CYCLE cherche le cycle précédent de H_r au cycle d à l'aide d'un premier appel à *findcycle*. Le résultat nd doit être un entier : il s'agit du nouveau cycle où le flot r est présent. Cependant, avant de modifier le cycle du terme *rst* dans la réécriture, on vérifie que nd est bien dans l'intervalle de cycles que l'on sait déjà valides grâce à *checkcycle*.

En décalant successivement le terme *rst* vers le passé, il arrive un moment où celui-ci a parcouru tous les cycles auxquels son flot booléen était présent dans la fenêtre temporelle. De plus, ce flot a toujours valu *false* dans cet intervalle de temps, ce qui signifie que le flot ne contribue pas à réinitialiser l'opérateur et qu'il peut être retiré du CSP. La section suivante s'intéresse à tous les cas où la contrainte *rst* peut ainsi disparaître d'un ensemble de contraintes.

4.4.4.5 Retrait des termes *rst* inutiles

Comme nous venons de le voir à le section précédente, on cherche à décaler le cycle d associé à chaque terme *rst* à chaque fois que le flot r qu'il porte est absent ou vaut *false*. Si on atteint un cycle d où on souhaiterait décaler la contrainte d'après le prédicat *changecycle* mais que le cycle résultant est antérieur ou égal au cycle précédent de l'horloge principale H, cela signifie qu'on a exploré toutes les occurences du flot r dans l'intervalle de temps qu'on veut parcourir. On retire donc la contrainte du système.

$$\text{RST-REMOVE-LIMIT}\ \frac{\begin{array}{c} S \vdash clock(r){=}H_r \qquad S \vdash changecycle(r,\, H_r, d) \\ S \vdash findcycle(H_r, d) = nd \\ S \vdash findcycle(H, c) = nc \qquad nd \in \mathbb{N} \qquad nc \in \mathbb{N} \qquad nd \geq nc \end{array}}{\vdash S \cup \{ Rst \Rightarrow_{(c,H)} rst(r,d) \} \rightarrow S}$$

On rappelle qu'une fois que toutes les contraintes sont retirées, on peut déduire qu'il n'y a pas de réinitialiation. Il y a d'autres cas où on retire un terme *rst*. Tout d'abord, il est possible que le cycle initial de r soit compris dans l'intervalle de recherche. Autrement dit, on atteint un cycle d auquel l'horloge de r est liée au statut *initial* dans la table d'horloges. Si à ce cycle d le flot r ne vaut toujours pas *false*, il n'existe pas de cycle antérieur où il le vaudra, et on peut retirer la contrainte :

$$\text{RST-REMOVE-INITIAL-FALSE}\ \frac{S \vdash cmax(H_r,\, d,\, initial) \qquad S \vdash r \mapsto_d false}{\vdash S \cup \{ Rst \Rightarrow_{(c,H)} rst(r,d) \} \rightarrow S}$$

D'autre part, il est possible aussi que le cycle d de la recherche soit déjà antérieur au cycle initial du flot r. C'est possible par exemple si le cycle c est lui-même antérieur à l'instant initial ic du flot r. Cette fois aussi, on peut retirer la contrainte *propage* inutile :

$$\text{RST-REMOVE-INITIAL-BEYOND}\ \frac{S \vdash cmax(H_r,\, ic,\, initial) \qquad d > ic}{\vdash S \cup \{ Rst \Rightarrow_{(c,H)} rst(r,d) \} \rightarrow S}$$

4.4.4.6 Annulation des contraintes au cycle initial

Nous avons vu à la section §4.4.4.4 que les contraintes *rst* sont introduites dès que l'on propage $\rightarrow *$, pour bénéficier du fait que les flots de redémarrages puissent être

déjà connus et valoir *true*. Cela est vrai même quand le statut de l'horloge principale H est indéterminé au cycle c de la propagation. Les règles concernant la fenêtre de réinitialisation sont toujours applicables : comme on ne sait pas si H admet un cycle précédent au cycle c, on peut seulement explorer les flots de redémarrages aux cycle c (d'après *checkcycle*) ; si l'un des flots booléens vaut *true*, on peut déjà en déduire que l'opérateur $\rightarrow\!*$ a la valeur de son membre gauche.

Un problème potentiel vient du fait que le statut de H peut-être instancié à *initial* avant que les contraintes *rst* ne déduisent un *reset*. Dans ce cas, il est inutile de continuer à les propager. On unifie la variable *Rst* qui les unit à la valeur *cancel* pour qu'elles soient retirées du CSP. En effet, la règle RST-REMOVE-ATOMIC retire les termes *rst* dès que la variable est instanciée, quelque soit la valeur (*reset* ou *cancel*). La règle qui réalise l'annulation est la suivante [1] :

$$\text{RST-CANCEL}\ \frac{S \vdash \mathit{status(H,c)} : \mathit{initial}}{\vdash S \cup \left\{ \mathit{Reset} \Rightarrow_{(c,H)} \mathit{reset(Rst,RList)} \right\} \rightarrow S \cup \{ \mathit{Rst} \equiv \mathit{cancel} \}}$$

4.4.4.7 Détection d'une réinitialisation

Le cas où une contrainte unifie la variable *Rst* se produit quand le flot *Rst* est initialement variable et que le flot r porté par la contrainte *rst* vaut *true* au cycle auquel on l'observe.

$$\text{RST-UNIFY-RESET}\ \frac{\mathit{Rst} \in \mathbb{V} \qquad S \vdash r \ \mapsto_d \mathit{true} \qquad S \vdash \mathit{Rst} \Rightarrow_{(c,H)} \mathit{rst(r,d)} \qquad (\mathit{Rst} \equiv \mathit{reset}) \notin S}{\vdash S \rightarrow S \cup \{ \mathit{Rst} \equiv \mathit{reset} \}}$$

4.4.4.8 Propagation du flot de redémarrage

Si jamais l'horloge d'un flot r de redémarrage est présente à un cycle d sans que le flot lui-même n'ait été propagé, nous pouvons le propager car sa valeur nous intéresse. On introduit alors une contrainte *propage* qui affecter une variable à r au cycle d. De plus, si le statut désiré et connu et vaut *nonreset*, la valeur propagée est nécessairement *false*, on passe par une fonction intermédiaire *resetval*.

Valeur propagée Les deux règles qui suivent prennent en compte les cas simples où la valeur de *Reset* est soit *nonreset* soit indéfinie. On rappelle que *Reset* est

1. Nous pourrions aussi ajouter la condition que *Reset* soit variable pour ignorer la règle dans le cas contraire et forcer les flots de redémarrage à refléter le status donné par *Reset*. Cela pourrait être utile par exemple si on souhaite faire la couverture structurelle exhaustive d'un opérateur $\rightarrow\!*$ en générant, au cycle initial, d'une part le cas *Reset*=*nonreset* et d'autre part le cas *Reset*=*reset*, bien que le résultat de l'opérateur soit identique dans les deux cas.

accessible à partir de la variable *Rst* car il reste une contrainte *propage* portant un terme *reset(Rst,RList)* en argument :

$$\text{RV-FALSE}\ \frac{Rst \in \mathbb{V} \qquad S \vdash nonreset \Rightarrow_{(c,H)} reset(Rst,RList)}{S \vdash resetval(Rst,false)} \qquad \text{RV-VAR}\ \frac{Rst \in \mathbb{V} \qquad S \vdash Reset \Rightarrow_{(c,H)} reset(Rst,RList) \qquad Reset \approx D_{\texttt{reset}} \qquad S \vdash V : fresh}{S \vdash resetval(Rst,V)}$$

Dans le cas où *Reset* vaut *reset*, on peut déduire que la valeur à propager est *true* uniquement s'il n'existe plus qu'une seule contrainte *rst* et que celle-ci n'admette plus qu'un seul cycle de présence dans la fenêtre temporelle.

$$\text{RV-TRUE}\ \frac{\begin{array}{c} Rst \in \mathbb{V} \qquad S \vdash findcycle(H,c) = nc \qquad nc \in \mathbb{N} \\ S \vdash reset \Rightarrow_{(c,H)} reset(Rst,[r_1,\ldots,r_n]) \\ \left\{(r_i,d_i) \mid Rst \Rightarrow_{(c,H)} rst(r_i,d_i) \in S, (r_i,d_i) \in \{r_1,\ldots,r_n\} \times \mathbb{N}\right\} = \{(r,d)\} \\ S \vdash clock(r){=}H_r \qquad S \vdash findcycle(H_r,d) = R \qquad R \notin \mathbb{N} \qquad cycle(R) \geq nc \end{array}}{S \vdash resetval(Rst,true)}$$

Sinon, la valeur retournée est aussi une variable libre (nous ne détaillons pas la règle).

Propagation du flot Alors, la propagation du flot indéfini r obéit la règle suivante :

$$\text{RST-PROPAGE-WAIT}\ \frac{\begin{array}{c} S \vdash clock(r){=}H_r \qquad S \vdash present(H_r,d) \\ S \vdash undef(r,\ d) \qquad S \vdash resetval(Rst,V) \end{array}}{\begin{array}{l} \vdash S \cup \left\{Rst \Rightarrow_{(c,H)} rst(r,d)\right\} \\ \rightarrow S \cup \left\{V \Rightarrow_{(d,H_r)} r, Rst \Rightarrow_{(c,H)} wait(r,d)\right\} \end{array}}$$

Le terme *rst* est modifié temporairement dans le CSP en un terme de foncteur *wait* jusqu'à ce que le nouveau terme *propage* soit filtré et introduise un terme $r \mapsto_d W$ (avec W quelconque) dans le système. Alors, la valeur W pourra être éventuellement modifiée par la suite et nous donner soit *true* soit *false*, pour lesquels on peut appliquer les règles précédentes.

$$\text{RST-PROPAGE-DONE}\ \frac{S \vdash r \mapsto_d W}{S \cup \left\{Rst \Rightarrow_{(c,H)} wait(r,d)\right\} \rightarrow \left\{Rst \Rightarrow_{(c,H)} rst(r,d)\right\}}$$

4.4.4.9 Propagation de l'horloge du flot

Chaque flot de redémarrage r d'un opérateur $\rightarrow *$ est cadencé sur une horloge H_r indépendante de l'horloge principale H. Alors, lorsqu'on a un terme *rst(r,d)* porté par une contrainte *propage*, on ne peut pas faire l'hypothèse que H_r est bien présente au cycle d. Il est même possible que le flot support de H_r, si ce n'est pas l'horloge de base, n'ait jamais

été propagé dans le CSP. Cependant, on sait qu'il existe au moins une horloge ancêtre H_A dans la hiérarchie d'horloges, entre H_r et l'horloge de base, qui est nécessairement présente au cycle d.

On cherche alors la première horloge (f, v) d'horloge H_A telle que H_A est présente au cycle d mais (f, v) indéfinie, et on y propage f : autrement dit, on cherche à générer une valeur pour f au cycle d de sorte à savoir si (f, v) est présente ou non. On continue cette approche, en essayant de redescendre dans la hiérarchie et en reculant si nécessaire dans les cycles jusqu'à atteindre un instant de présence nd de l'horloge H_r. Ainsi, sans jamais propager d'hypothèse non vérifiée dans le système, on favorise la propagation des termes *rst* en cherchant les instants de présence du flot r.

Contrainte *precycle* dédiée Cette démarche est en fait celle réalisée par la contrainte *precycle* que l'on a décrite à la section 4.3.4, lorsque le cycle de départ de la contrainte n'est pas un cycle de présence connu. Nous pourrions donc reprendre la contrainte pour qu'elle cherche un cycle de présence pour H_r et propage quand nécessaire les horloges intermédiaires bloquantes. Cependant, nous avons besoin de plus de contrôle sur la contrainte lorsqu'on traite l'opérateur $\rightarrow *$. On souhaite pouvoir retirer la contrainte dès qu'elle devient inutile (une autre contrainte *rst* trouve le statut *reset* en parallèle), et surtout on souhaite qu'elle ne propage jamais d'information au-delà de la limite temporelle donnée par l'opérateur.

Nous défnissons une nouvelle contrainte à travers le terme *rstfind* traité par *propage*, qui se substitue à *rst* le temps de trouver un cycle de présence nd pour r à partir du cycle d *inclus* : en effet, le cycle nd que l'on trouve n'est pas forcément un cycle précédent, il peut s'agir de d si les horloges ancêtres s'avèrent y être présentes.

Introduction et retrait de *rstfind* La première règle que nous présentons remplace le terme *rst* à deux arguments par un terme *rstfind(r,H_r,d,unblock(R))*. Nous pouvons remarquer que nous reprenons la mise en œuvre de *precycle* en introduisant un terme *unblock*.

$$\textsc{rstfind-intro}\ \frac{\begin{array}{c} S \vdash clock(r){=}H_r \qquad S \vdash undefclock(H_r,d) \\ S \vdash undef(r,\ d) \qquad Rst \in \mathbb{V} \qquad S \vdash fcwp(H_r,d) = R \qquad R \notin \mathbb{N} \end{array}}{\begin{array}{c} \vdash S \cup \{ Rst \Rightarrow_{(c,H)} rst(r,d) \} \\ \rightarrow S \cup \{ Rst \Rightarrow_{(c,H)} rstfind(r,H_r,d,unblock(R)) \} \end{array}}$$

La règle s'applique quand l'horloge H_r de r est indéfinie au cycle d et que *fcwp* ne peut pas trouver de cycle auquel propager *rst* : son résultat R n'est pas un entier. Ce cas est disjoint des règles précédentes où on savait avec certitude que H_r était absente ou r faux au cycle d. Alors, on introduit le nouveau terme *rstfind* où le résultat R est placé dans un terme *unblock*. On rappelle que celui-ci sert de marqueur dans le CSP pour ne traiter la situation bloquante symbolisée par R qu'une seule fois. On a alors une règle similaire

à PRECYCLE-STEP vue page 160, qui fait progresser *rstfind* dès que le résultat R_2 retourné par *fcwp* change du résultat précédemment bloquant R_1.

$$\text{RSTFIND-STEP}\ \frac{Rst \in \mathbb{V} \qquad S \vdash fcwp(H_r,d) = R_2 \qquad R_1 \neq R_2}{\begin{array}{c}\vdash S \cup \{Rst \Rightarrow_{(c,H)} rstfind(r,H_r,d,R_1)\} \\ \rightarrow S \cup \{Rst \Rightarrow_{(c,H)} rstfind(r,H_r,d,unblock(R_2))\}\end{array}}$$

À la différence de *precycle-step*, on a recours ici à *fcwp* qui commence l'exploration au cycle d plutôt que $d+1$ comme c'est le cas pour *findcycle* (cf. p. 151). De plus, on évite d'appliquer la règle quand *Rst* est instancié. En effet, lorsque c'est le cas, il suffit de retirer la contrainte car il n'est plus nécessaire de chercher un cycle de présence pour r : un autre flot de redémarrage s'évalue à *true* :

$$\text{RSTFIND-REMOVE}\ \frac{}{\vdash S \cup \{reset \Rightarrow_{(c,H)} rstfind(r,H_r,d,R)\} \rightarrow S}$$

Par ailleurs, on peut complètement retirer le terme dès que le cycle précédent de l'horloge principale H est connu et que le résultat R est supérieur ou égal à ce cycle : le flot r n'admet plus d'instant de présence dans la fenêtre de réinitialisation.

$$\text{RSTFIND-BEYOND}\ \frac{S \vdash findcycle(H,c) = nc \qquad nc \in \mathbb{N} \qquad cycle(R) \geq nc}{\vdash S \cup \{Rst \Rightarrow_{(c,H)} rstfind(r,H_r,d,R)\} \rightarrow S}$$

Pour cela, nous surchargeons *cycle* de sorte que :

$$cycle(unblock(R)) = cycle(R)$$

Nous présentons maintenant les traitements des termes *unblock* et les différences que l'on y trouve par rapport à ceux de *precycle*.

Fin de la recherche Le premier cas que l'on traite est celui où *fcwp* finit par trouver un cycle précédent pour H_r au cycle d. Le nouveau cycle nd est donc un instant de présence pour H_r auquel on peut placer le terme *rst*. On vérifie cependant que le cycle est bien dans la fenêtre autorisée.

$$\text{RSTFIND-DONE}\ \frac{S \vdash checkcycle(H,c,d) \qquad d \in \mathbb{N}}{\begin{array}{c}S \cup \{Rst \Rightarrow_{(c,H)} rstfind(r,H,c,unblock(d))\} \\ \rightarrow S \cup \{Rst \Rightarrow_{(c,H)} rst(r,d)\}\end{array}}$$

Par la suite, si r lui-même est indéfini, il sera propagé au cycle nd par les règles RST-PROPAGE-WAIT et RST-PROPAGE-DONE vues à la section précédente.

Flot intermédiaire indéfini Lors de l'appel à *fcwp*, on obtient un cycle temporaire td auquel un flot g, qui est nécessairement présent, n'est pas propagé. Le cycle td est bien dans la fenêtre de réinitialisation que l'on connaît jusqu'à présent, et on peut donc propager g au cycle td. Si au contraire td n'est pas dans la fenêtre, la règle reste inapplicable jusqu'à ce que ce soit le cas. Si jamais le cycle td venait à dépasser le cycle précédent de H une fois que celui-ci est connu, alors la contrainte est retirée par RSTFIND-BEYOND.

$$\text{RSTFIND-UNDEF}\ \frac{R = \mathit{undef}(g,\ td) \qquad S \vdash \mathit{checkcycle}(H,c,td)}{\begin{array}{r} S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,\mathit{unblock}(R))\} \\ \rightarrow S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,R), \mathit{propage}(g,td)\} \end{array}}$$

Statut d'une horloge Une autre situation bloquante est celle où le statut St lié à l'horloge de base est une variable qui nous empêche d'explorer les cycles passés. On a donc un terme de la forme *status(td, St)* à un cycle temporaire td. Il y a deux cas de figure possibles lié au statut de l'horloge principale H au cycle c auquel l'opérateur $\rightarrow *$ est propagé :

1. Le statut de H au cycle c est connu et vaut *noninitial*.
   Dans ce cas, H admet nécessairement un cycle précédent nc et s'il n'est pas encore connu, il existe une contrainte *precycle* qui le cherche. Or, comme on vérifie *checkcycle(H,c,td)*, le cycle td bloquant est aussi le cycle bloquant de cette contrainte *precycle*, qui va propager le statut à *noninitial* d'après la règle UNBLOCK-STATUS de la page 159. Il n'y a donc ici rien à faire, si ce n'est retirer le terme englobant *unblock*.
2. Le statut de H au cycle c est variable ou *initial*.
   Le cycle td est alors égal au cycle c d'après *checkcyle*. On ne peut pas forcer l'existence d'un cycle précédent pour l'horloge de base ici en unifiant St à *noninitial*, car il se pourrait aussi que le cycle td soit le cycle initial à la fois de *base* et de H.

Dans les deux cas, il nous suffit d'appliquer la règle suivante :

$$\text{RSTFIND-STATUS}\ \frac{R = \mathit{status}(td,\ St)}{\begin{array}{r} \vdash S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,\mathit{unblock}(R))\} \\ \rightarrow S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,R)\} \end{array}}$$

Attente d'une valeur Dans le cas où un flot est propagé mais que sa valeur est inconnu, on attent qu'une réduction de domaine ait lieu :

$$\text{RSTFIND-WAIT}\ \frac{}{\begin{array}{r} S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,\mathit{unblock}(\mathit{wait}(c,f,W)))\} \\ \rightarrow S \cup \{\mathit{Rst} \Rightarrow_{(c,H)} \mathit{rstfind}(r,H,c,\mathit{wait}(c,f,W))\} \end{array}}$$

Échec de la recherche La recherche faite par *fcwp* peut échouer, s'il n'est pas possible de trouver un cycle précédent pour H_r au cycle d. Cependant, il ne s'agit pas nécessairement d'un insatisfaisabilité. Nous retirons la contrainte du système, car il n'existe pas de cycle de présence pour H_r :

$$\textsc{rstfind-fail}\ \frac{}{S \cup \{\, Rst \Rightarrow_{(c,H)} rstfind(r,H,c,unblock(fail)) \,\} \rightarrow S}$$

En effet, il y a deux raisons pour qu'un échec se produise dans *fcwp* (§4.3.3.6). Le premier se produit quand r a déjà son cycle initial à un cycle ultérieur à la position d du terme *rst*, ce qui nous empêche de trouver un cycle précédent. Or, quand c'est le cas, cela signifie simplement que le flot ne provoque pas un redémarrage. En effet, on ne cherche pas à garantir que le flot r est bien présent, mais seulement à tenter d'observer sa valeur. Il s'agit du même cas que celui traité par RST-REMOVE-INITIAL-BEYOND à la page 169, où on retire la contrainte. Le second cas est celui où on bloque sur *maxsize*, le paramètre global qui limite la taille des séquences. Là aussi, il n'est pas nécessaire qu'un cycle précédent existe pour r. Cela serait nécessaire si on était sûr que le statut au cycle d est *noninitial* pour H_r. Or, si c'est le cas, il existe une contrainte *precycle* qui se charge de vérifier l'existence d'un cycle précédent comme on la vu à la section §4.2.6.2 (p. 145). Il n'y a donc aucun traitement à faire ici.

Ordonnancement Il y a une différence majeure entre la contrainte *precycle* et *rstfind*. Lorsque *precycle* est propagé, on sait qu'il doit exister un cycle précédent pour les arguments qu'on lui donne, alors que *rstfind* cherche un instant de présence d'un flot de redémarrage sans savoir si celui-ci existe. Nous n'avons pas détaillé jusqu'ici les problèmes d'ordonnancement de contraintes dans ce chapitre, mais dans l'implémentation actuelle, on propage systématiquement les contraintes nécessaires lorsque *fcwp* nous donne un résultat partiel dans *rstfind*. Il y a alors un risque de propager trop de contraintes avec cette approche.

Par exemple, si on sait que le statut de la réinitialisation est *reset*, on peut propager chaque flot séquentiellement, en examinant d'abord ceux dont l'horloge est présente (comme *base*), puis celles qui sont moins connues, de sorte à trouver à moindre coût un instant où on peut avoir une valeur *true*. Cela est intéressant car la propagation des flots de redémarrage s'interrompt dès que l'on trouve cette valeur *true*, évitant ainsi de propager potentiellement beaucoup de contraintes. Le cas où la propagation systématique est nécessaire est celui où le statut désiré est *nonreset*. Si cela s'avère utile, on pourrait ainsi chosir de limiter ou d'ordonner les termes *unblock* à traiter dans la propagation.

4.4.5 Sélection de cas de test

Nous proposons ici plusieurs découpages possibles pour l'opérateur $\rightarrow *$. Ceux-ci manipulent la contrainte *init* vue à la page 164. Tout d'abord, nous pouvons faire la

sélection sur la branche à propager, gauche ou droite, ce qui nous donne deux sous-cas de tests :

$$\textsc{select-init-2}\ \frac{}{S \cup \{\, init(Status,Branch,Reset) \,\} \vdash select \left\{ \begin{array}{c} \{Branch \equiv left\} \\ \{Branch \equiv right\} \end{array} \right\}}$$

Un autre découpage naturel peut-être de donner directement 3 sous-cas, où l'on traite respectivement le cas du cycle initial, d'une réinitialisation à un cycle non-initial et le cas non-initial sans réinitialisation :

$$\textsc{select-init-3}\ \frac{}{S \cup \{\, init(St,Br,Res) \,\} \vdash select \left\{ \begin{array}{c} \{St \equiv initial\} \\ \{St \equiv noninitial, Res \equiv reset\} \\ \{St \equiv noninitial, Res \equiv nonreset\} \end{array} \right\}}$$

Enfin, la sélection actuellement mise en œuvre consiste à proposer d'abord le découpage selon le statut, s'il est inconnu, puis l'instanciation de *Reset*. Il y a donc deux règles :

$$\textsc{select-init-2-status}\ \frac{Status \in \mathbb{V}}{S \cup \{\, init(Status,Branch,Reset) \,\} \vdash select \left\{ \begin{array}{c} \{Status \equiv initial\} \\ \{Status \equiv noninitial\} \end{array} \right\}}$$

$$\textsc{select-init-2-reset}\ \frac{Status \in D_{\texttt{status}}}{S \cup \{\, init(Status,Branch,Reset) \,\} \vdash select \left\{ \begin{array}{c} \{Reset \equiv reset\} \\ \{Reset \equiv nonreset\} \end{array} \right\}}$$

4.5 Conclusion

Dans ce chapitre, nous nous sommes concentrés sur les modifications les plus représentatives du noyau. En particulier, nous avons montré les propriétés exploitables par les contraintes pour lier les horloges et les flots entre eux. Nous avons ainsi décrit les changements de *propage* pour traiter la cohérence des horloges, de l'opérateur de décalage temporel et celui de (ré)initilisation de flots. Nous avons réparti chacun des traitements dans plusieurs contraintes dédiées, communiquant à travers des variables partagées. C'est notamment le cas pour $\rightarrow *$, qui nous servira au chapitre suivant pour encoder les structures de haut-niveau de SCADE 6 vers le langage $\mathcal{L}_+$.

Nous avons présenté en détails quelques aspects des modifications du noyau de GATeL$_1$ pour obtenir GATeL$_+$, dans le but de traiter directement les horloges. Les travaux auxquels j'ai participé concernent en pratique la compilation pour le langage étendu, l'évaluation en avant des flots temporisés, la sur-approximation, la propagation de contraintes, la mise en œuvre de la table d'horloges, une partie de la sélection de cas

de tests et de l'interface graphique. Ces changements ont été apportés par l'équipe toute entière que je remercie ici pour le soutien et les conseils qui m'ont été donnés.

Nous avons testé et mis au point le nouvel outil tout au long de la thèse à l'aide de nombreux cas d'études. En particulier, nous montrons les résultats de nos expérimentations au chapitre suivant, qui comparent notre approche (modification du noyau) avec une approche par interprétation des horloges dans LUSTRE mono-horloge. Nous verrons aussi comment nous exploitons ce nouveau langage pour générer automatiquement des séquences de tests pour les automates de modes, notamment des séquences satisfaisant des critères de couverture structurelle pour ces automates.

Chapitre 5

Automates de modes

Dans ce chapitre, nous nous intéressons aux automates de modes du langage Scade 6, du point de vue de la génération automatique de tests. Nous présentons ce langage puis une fonction de traduction existante qui traduit les automates en flots temporisés du langage $\mathcal{L}_+$, présenté au chapitre 2. Cette étape de transformation nous permet de générer des séquences pour des modèles Scade 6 à travers leur représentation dans le langage $\mathcal{L}_+$.

Afin de faciliter la génération de séquences en présence d'horloges, nous raffinons les règles de propagation de contraintes pour exploiter des propriétés invariantes présentes au niveau des automates de modes mais non explicites dans le langage $\mathcal{L}_+$. Nous pouvons ainsi générer des séquences de tests efficacement pour le langage Scade 6.

Dans une deuxième partie, nous nous intéressons à l'utilisation des techniques de génération de tests pour couvrir structurellement les automates de modes. Pour cela, nous adaptons les critères de couverture disponibles pour les automates à états finis et le langage Statecharts au langage Scade 6. Nous montrons comment nous pouvons exploiter la traduction vers $\mathcal{L}_+$ afin d'exprimer la couverure des critères dans ce langage. Nous définissons alors des ensembles de flots observateurs caractérisant la couverture de certains éléments présents dans les automates, notamment les états et les transitions. En combinant ces flots observateurs avec un objectif de test dans GATeL, nous pouvons ainsi générer des séquences couvrant les critères sélectionnés par un utilisateur.

Ce chapitre s'intéresse donc à l'utilisation de la génération de tests présentée pour le langage $\mathcal{L}_+$ afin de traiter le cas des automates de Scade 6, ainsi que pour couvrir structurellement ces modèles.

5.1 Génération de tests pour automates de modes

Nous avons déjà parlé des automates de modes dans l'introduction générale. Nous rappelons ici que les automates mélangent à la fois des concepts issus des langages flots de données comme Lustre et des formalismes visuels hiérarchiques à base d'état et de transitions. Les automates peuvent être imbriqués et agir en parallèle. Chaque automate calcule alors un mode actif par cycle dans lequel une ou plusieurs variables du système admettent des équations de flots propres à ce mode ; tant que l'automate ne change pas

de mode, l'équation active détermine la sortie des variables liées à l'automate. Nous cherchons à générer des tests automatiquement pour ce formalisme.

Dans une première partie, nous décrivons la sémantique des automates de modes telle qu'elle est définie dans SCADE 6 (§5.1.1 p. 180). Nous montrons ensuite comment cette sémantique est traduite sous forme d'expressions temporisées (§5.1.2 p. 185). L'encodage des automates repose sur des types énumérés et des arbres d'horloges qui reflètent la hiérarchie d'automates. Ensuite, nous voyons comment l'encodage nous permet directement de générer des séquences de tests pour SCADE 6 (§5.1.4 p. 192).

Bien que la génération soit possible, nous voulons faciliter la traçabilité entre le modèle de haut-niveau et le modèle après traduction. Pour cela, nous montrons comment il nous est possible d'extraire certaines informations du modèle haut-niveau à la section §5.1.3 (p. 188), où nous définissons des fonctions de correspondances d'un modèle vers sa traduction.

Cela nous amène à définir une contrainte spécifique aux automates, créée grâce à ces fonctions de correspondances et injectées dans l'ensemble de contraintes pour y rendre explicite les relations structurelles des automates au niveau des variables d'états qui les encodent (§5.1.5 p. 193). Finalement, nous montrons quelques expérimentations autour de notre approche, où nous faisons alors le lien avec la couverture structurelle d'automates (§5.1.6 p. 195).

5.1.1 Sémantique des automates de modes dans Scade 6

Les figures 5.1 et 5.2 sont deux exemples d'automates que nous utiliserons dans les sections suivantes. Un automate est une structure de contrôle composée d'états et de transitions associée à un sous-ensemble des variables calculées du système. Chaque sous-ensemble d'équations, ou bloc, correspond alors à un comportement différent du système pour les variables considérées. Ce même ensemble de variables calculées peut-être découpé et réparti dans des sous-automates, qui évoluent alors en parallèle.

Il est possible de définir des signaux, qui peuvent être émis et interceptés à différents niveaux du modèle. Les automates peuvent définir zéro ou plusieurs états finaux, qui sont des points de rendez-vous permettant de synchroniser des calculs agissant en parallèle au sein d'un automate parent. Par exemple, si un état E contient plusieurs sous-automates parallèles, et que ceux-ci ont tous atteint un état final, alors une transition particulière appelée transition « synchro », sortant de E, est activée (si elle existe). La transition est dite syncrhonisante, car elle n'est franchissable que lorsque les sous-automates ont atteint leurs points de rendez-vous respectifs.

Les transitions entre les modes de calculs sont soit des transitions fortes, soit des transitions faibles. Elles peuvent par ailleurs être soit des transitions à redémarrage (transitions *restart*) ou à mémoire (transitions *resume*).

La syntaxe et la sémantique des automates de modes sont assez complexes. Nous présentons ci-dessous de manière informelle, et incomplète, les automates de modes du

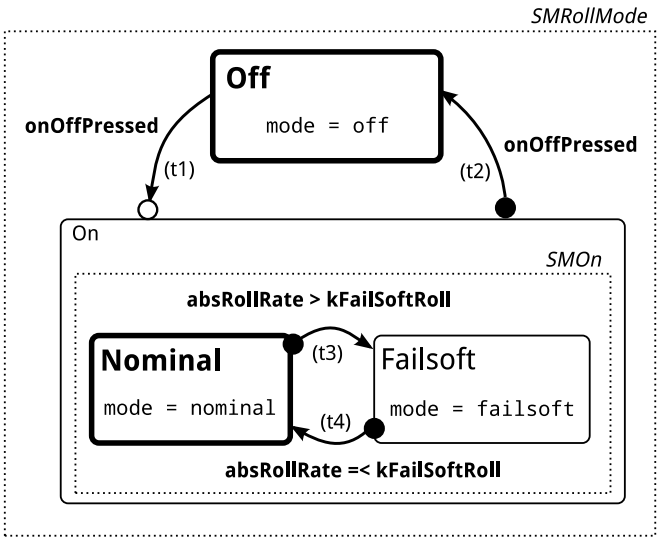


FIGURE 5.1: Les automates SMRollMode et SMOn

langage SCADE 6. Pour cela, nous nous focalisons sur les mécanismes de base du langage, en sachant que le langage tout entier est pris en compte dans nos travaux, comme on le verra dans les sections suivantes.

5.1.1.1 Exemples

RollMode L'automate SMRollMode de la figure 5.1 calcule une sortie, appelée `mode`, dont la valeur est l'une des constantes `off`, `nominal` et `failsoft`. Ces constantes sont définies par l'utilisateur : les flots en entrées du système sont *onOffPressed*, un flot booléen, et *absRollRate*, un flot numérique réel. Selon que le premier est vrai ou faux aux différents cycles, l'automate principal bascule entre les états *Off* et *On*. Une fois dans l'état *On*, il y a deux modes de fonctionnement : si la valeur en entrée dépasse la constante *kFailSoftRoll*, l'automate passe dans l'état *Failsoft* ; si la valeur est en dessous du seuil, on repasse dans le mode *Nominal*. Les différentes transitions sont notées $t1$, $t2$, $t3$ et $t4$.

ABC La figure 5.2 représente l'automate *ABC*, sous sa version textuelle et avec la représentation graphique adoptée ici. Cet exemple nous permet d'illustrer les différentes transitions, fortes et faibles, et notamment les différentes successions de transitions possibles (forte puis faible, etc.). L'exemple utilise de plus des transitions *resume*. L'automate dépend d'une entrée booléenne « e » qui lui sert à passer d'un état à l'autre et produit une sortie entière « z ». Les transitions sont identifiées ici par $u1$, $u2$, $u3$ et $u4$.

```
automaton ABC
  initial state A
  unless if e resume B
  let
     z = 0→(pre(Z)+1) ;
  tel
  until  if (not e) resume C

  state B
  unless if e resume C
  let
     z = 0→(pre(Z)+2) ;
  tel

  state C
  let
     z = 0→(pre(Z)+3) ;
  tel
  until if e resume A
returns z ;
```

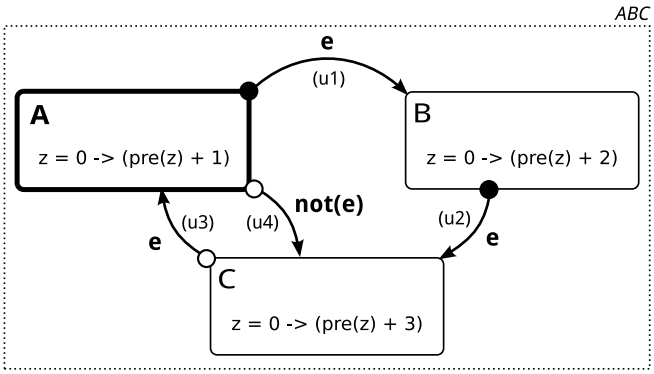


FIGURE 5.2: Automate ABC

5.1.1.2 Modes de calculs

Tous les états d'un automate calculent le même ensemble de variables de flots, selon un bloc d'équations ou à travers des automates imbriqués. Cela forme une hiérarchie d'automates : un automate parent est tel qu'un ou plusieurs de ses états sont définis par des sous-automates, et récursivement par des états feuilles qui contiennent des équations de flots simples. Les calculs sont réalisés selon des modes différents, mais un seul mode est actif par cycle et par automate.

L'ensemble des états feuilles actifs agissant en parallèle à un cycle constituent donc une combinaison d'équations sur les sorties de l'automate. Par exemple, l'automate *ABC* possède trois configurations, celles données par les trois états. Le système SMRollMode possède aussi trois configurations, identifiées par les modes *Off*, *Nominal* et *Failsoft*. Si on place *ABC* et SMRollMode en parallèle dans un système plus complexe regroupant l'ensemble des entrées et sorties de ces deux automates, alors le système contient 9 configurations différentes, c'est-à-dire (A, Off), $(A, Nominal)$, …, $(C, Nominal)$, $(C, Failsoft)$.

5.1.1.3 Transitions

Le mode actif reste implicitement le même tant qu'aucune transition n'est franchissable. Il existe deux niveaux de transitions dans un automate (fortes et faibles), selon que le franchissement s'effectue avant ou après les calculs. Les transitions faibles et fortes sont représentées respectivement par des arcs « $\multimap$ » et « $\bullet\!\!\rightarrow$ ».

À chaque cycle d'éxecution, avant qu'un quelconque calcul ait lieu, automate peut déclencher une transition *forte*, pour sortir de l'état sélectionné courant et rediriger le contrôle sur le mode indiqué par cette transition, qui devient l'état actif. Au contraire, les transitions *faibles* sont évaluées seulement après que les sorties au cycle courant sont produites, et définissent le prochain état sélectionné de l'automate (celui-ci pourra alors éventuellement être modifié par une transition forte).

Par exemple, on sait que SMRollMode est toujours dans l'état *Off* au cycle initial, même si *onOffPressed* est vrai : il s'agit de l'état initial, et ($t1$) est une transition faible, qui est donc évaluée à la fin du cycle, après l'évaluation de « `mode=off` ». On suppose maintenant que *onOffPressed* est vrai au cycle initial (cycle 0) ainsi qu'au cycle 1. Le cycle 1 commence dans l'état « sélectionné » *On*. Cette fois, la transition forte ($t2$) est franchie (sa condition est vraie) avant que les calculs ne se produisent, activant ainsi l'état *Off* dans lequel une fois encore « `mode=off` ». Dans cet exemple, *On* est un état transitoire dans lequel aucun calcul n'a été réalisé : l'état « actif » au cycle 1 est *Off*. La trace d'éxécution correspondante est la suivante :

Cycles	0	1	2	3	4	5	6
e	t	f	t	t	f	t	t
sel	A	B	B	C	A	C	A
act	B	B	C	C	A	C	B
nxt	B	B	C	A	C	A	B
z_A					0		
z_B	0	2					4
z_C			0	3		6	
z	0	2	0	3	0	6	4

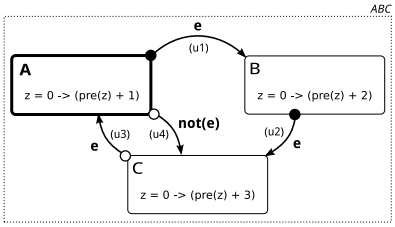


FIGURE 5.3: Trace d'exécution de l'automate *ABC*.

Cycle	0	1
onOffPressed	true	true
mode	off	off
sel	Off	On
act	Off	Off
nxt	On	Off

Le formalisme possède cependant une limitation : *à chaque cycle d'exécution, au plus une transition, forte ou faible, peut-être franchie.* Cela signifie que si une transition forte est franchie au début de cycle, aucune transition faible, même si elle franchissable, ne sera empruntée. De même, si une transition forte est franchie, les transitions fortes de l'état cible ne seront pas prises en compte. Cette limitation permet d'éviter les boucles récursives dans l'évaluation de l'état actif. La figure 5.3 est un exemple de toutes les combinaisons de transitions fortes et faibles qui peuvent survenir, pour l'automate *ABC* cette fois.

Les transitions sont évaluées les unes après les autres, selon leur ordre d'apparition dans la spécification, d'abord dans les automates de plus haut-niveau, et ensuite dans les sous-automates. Par ailleurs, elles peuvent déclencher des signaux lorsqu'elles sont franchies, ou servir à synchroniser des automates imbriqués : une transition *synchro* sortant d'un état est activée lorsque tous les sous-automates de cet état sont dans un état final.

Enfin, chaque transition peut être associée à un redémarrage de l'état cible, ou simplement à sa reprise. Une transition *restart* a pour effet de faire basculer, dans l'état cible, tous les automates imbriqués dans leur état initial. De plus, les équations de flots dans ces automates interprètent l'opérateur d'initialisation « → » selon son membre gauche, comme si le système venait de démarrer (mais toujours à partir des valeurs courantes des flots en entrée). En revenche, une transition *resume* va revenir dans un état interrompu auparavant en y retrouvant le même état, c'est-à-dire les mêmes modes de calculs et les mêmes états mémoires que lorsque l'état avait été quitté. Par exemple, si on regarde dans la figure 5.3 la sortie de l'automate *ABC*, dans lequel toutes les transitions sont des transitions *resume*, on voit que les états incrémentent la dernière valeur qu'ils ont

calculées, et non la dernière valeur produite par l'automate. Si on avait eu des transitions *restart*, chaque compteur repartirait à zéro en entrant dans un état.

5.1.2 Interprétation dans le langage $\mathcal{L}_+$

SCADE 6 étend significativement la syntaxe de LUSTRE, mais il est traduit durant la compilation [1] vers un formalisme à mi-chemin entre ces deux langages. Nous avons défini une version simplifiée de ce formalisme à travers le langage $\mathcal{L}_+$ au chapitre 2. Ainsi, les constructions de haut-niveau que sont les automates et les signaux sont encodés par des flots de données munis d'horloges, construites à partir de types énumérés. Les équations font intervenir les nouveaux opérateurs temporels du langage $\mathcal{L}_+$, en particulier « `merge` » et $\rightarrow *$ (la réinitialisation de flots). Le langage intermédiaire bénéficie de l'expressivité suffisante pour simuler efficacement la sémantique des automates à l'aide des horloges. Le recours aux flots temporisés permet d'avoir une compilation efficace du langage SCADE 6, basée sur des activations conditionnelles de sous-systèmes d'un modèle.

Nous présentons ici brièvement la traduction existante [CPP05b] de SCADE 6 vers le langage $\mathcal{L}_+$. Intuitivement, cette traduction consiste à effectuer les calculs des différents modes d'un automate en les cadençant sur des horloges complémentaires. Pour chaque automate, un type énuméré représente ses états et un autre représente ses transitions. Alors, les calculs et les transitions sont effectués à des rythmes différents, puis regroupés sur la même horloge que l'automate à l'aide d'un opérateur « `merge` ». Les transitions *restart* sont encodées à travers des opérateurs de réinitialisation.

La traduction est décrite formellement dans l'article de référence, c'est pourquoi nous préférons la présenter ici à l'aide d'un exemple illustratif. L'exemple ne couvre pas tous les cas de figure mais donne l'idée générale du fonctionnement de la transformation. Il s'agit d'une éxécution réelle de la traduction à partir de l'automate SMRollMode, où les différents noms ont toutefois été simplfiés pour plus de lisibilité.

5.1.2.1 Types énumérés

Tout d'abord, la transformation d'un automate introduit 2 nouveaux types :

1. Un type énuméré représentant tous les états de l'automate.
2. Un type énuméré représentant toutes les transitions, ainsi qu'une valeur particulière signifiant « aucune transition ». Ainsi l'ensemble T des transitions d'un automate est traduit vers un type énuméré possédant $|T| + 1$ éléments.

1. On considère ici le compilateur KCG6 fourni par la plate-forme Scade.

Voici les déclarations de types correspondantes pour notre exemple :

```
type ssm_RM = enum { st_Off, st_On };
type tr_RM = enum {
  RM_notrans,
  tr_Off_1,    (* t1 *)
  tr_On_1      (* t2 *)
};
```

Le type `ssm_RM` correspond à l'ensemble des états de l'automate SMRollMode. De même, le type `tr_RM` contient toutes les transitions définies dans l'automate, ainsi qu'une valeur `RM_notrans` indiquant l'absence de transition. Ce type est utile pour définir des flots qui contrôlent quelles transitions ont potentiellement été franchies à un cycle de calcul. Les noms des valeurs des types sont composés de l'état source de chaque transition ainsi que de son numéro (il peut y avoir plusieurs transitions émanant d'un état).

5.1.2.2 Variables d'état et sorties

La sémantique d'évaluation des automates que l'on a présentée succintement dans la section 5.1.1 consiste à effectuer plusieurs étapes au sein d'un même cycle de calcul :

1. L'automate détermine son état *sélectionné*, que l'on appelle « `sel` ». Celui-ci est donné soit par l'état initial de l'automate, soit par l'état donné par `nxt` au cycle précédent (voir ci-après).
2. L'état *actif*, noté `act`, est calculé :
   - si une transition forte est franchissable à ce cycle, l'état actif est l'état cible de la transition ;
   - sinon, l'état actif est le même que l'état sélectionné.
3. Les sorties et les automates imbriqués sont calculés dans le contexte de l'état actif `act`.
4. Finalement, les transitions faibles sont évaluées, donnant ainsi le prochain état sélectionné, à savoir `nxt` :
   - si une transition forte a déjà été franchie au début du cycle, ou si aucune transition faible n'est applicable au cycle courant, on a « `nxt = act` » ;
   - sinon, l'état `nxt` est l'état cible de la transition faible considérée.

L'extrait ci-dessous montre les variables intéressantes encodant ces différents étapes lors de la traduction de l'automate SMRollMode.

```
-- State variables
sel  = st_Off → pre nxt ;
act  = merge(sel; act_Off ; act_On);
mode = merge(act; mode_Off; mode_On);
```

```
nxt  = merge(act; nxt_Off; nxt_On);
...
-- Computations in state Off
act_Off  = st_Off when (sel match st_Off);
mode_Off = off when (act match st_Off);
...
```

Les variables `sel`, `act` et `nxt` sont du type « `ssm_RM` » (elles representent les états que l'on a décrit précédemment). La première est donnée comme la valeur précédente de `nxt`, ou par `st_Off` au cycle initial : il s'agit de l'état initial. Ensuite, selon la valeur de `sel`, seule une horloge parmi `(sel,st_Off)` et `(sel,st_On)` est active à un cycle. Par exemple, si on a `sel = st_Off`, la valeur courante de `act` est donnée par le flot `act_Off`, c'est-à-dire celui correspondant à l'horloge `(sel,st_Off)` dans l'opérateur `merge` présent dans l'équation de définition de `act`. L'autre flot, `act_On`, n'est pas évalué à ce cycle, grâce à la sémantique des flots temporisés. Une fois la valeur de `act` connue, il est possible de calculer les sorties, ici la variable `mode`, ainsi que le prochain état sélectionné. La définition de `act_Off` retourne la constante `st_Off` cadencée sur l'horloge `(sel,st_Off)`.

Dans le cas général, l'opérateur d'initialisation dans la définition de `sel` peut-être remplacée par une flèche de réinitialisation $\rightarrow *$, afin de faire reprendre à l'automate son état initial dans le cas de transitions *restart*.

5.1.2.3 Transitions

Le modèle intermédiaire évalue les transitions sortant de l'état sélectionné pour aller vers l'état actif, puis, si nécessaire, les transitions faibles sortant de l'état actif au prochain état sélectionné. Ainsi, les conditions de franchissement des transitions sont encodées par des flots cadencés selon des horloges dont le flot support est respectivement `sel` puis `nxt`. Il existe ainsi deux variables auxiliaires cadencées toutes deux sur l'horloge de l'automate, nommées `fired1` et `fired2`. Elles sont de type `tr_RM` et sont définies ainsi pour notre exemple :

```
fired1 = merge(sel; f1_Off  ; f1_On);
fired2 = merge(act; f2_Off  ; f2_On);
```

La première représente la transition forte prise à un cycle donné. La deuxième contient la transition finalement prise par l'automate au même cycle, c'est-à-dire soit la transition forte donnée par `fired1`, soit une transition faible. Les deux peuvent éventuellement valoir `RM_notrans` pour indiquer l'absence de transition. Les opérateurs `merge` répartissent les calculs dans des flots cadencés sur les horloges complémentaires basées sur les flots `sel` et `act`.

Nous nous intéressons aux calculs réalisés dans l'état *Off* de l'automate. Puisqu'il n'y a aucune transition forte sortant de ce mode, `f1_Off` est défini simplement comme suit :

```
f1_Off = RM_notrans when (sel match st_Off);
```

Les extraits suivants nous montrent les équations de flots associés au calcul des transitions faibles depuis l'état *Off*. Elles sont donc toutes cadencées selon l'horloge `(act,st_Off)`.

```
f2_Off = merge(fired_Off; strg_Off; weak_Off);
```

Le flot `f2_Off` est défini à l'aide d'un opérateur `merge` dont le rôle est de différencier les cas où une transition forte a, ou non, déjà été franchie. La variable booléene `fired_Off` est vraie quand la transition forte franchie est différente de `RM_notrans` :

```
fired_Off =    (fired1     when act match st_Off)
            <> (RM_notrans when act match st_Off);
```

Ainsi, suivant la valeur de ce flot, le calul de la transition faible est donnée soit par `strg_Off`, soit par `weak_Off`, respectivement. Le premier de ces deux flots est donné simplement par l'égalité suivante :

```
strg_Off = fired1 when (act match st_Off)
                  when fired_Off;
```

Comme une transition forte a déjà été effectuée dans ce cas, la transition finalement franchie, `fired2`, est égale au cycle courant à la valeur de `fired1`. On s'empêche ainsi d'évaluer les conditions de franchissement des transitions faibles. Dans l'autre cas, la condition de franchissement est traduite en ajoutant systématiquement les opérateurs de filtrage nécessaires pour faire correspondre l'horloge de la variable `weak_Off` avec celle de son expression de définition.

```
weak_Off = if onOffPressed when (act match st_Off)
                           when not fired_Off
           then tr_Off_1 when (act match st_Off)
                         when not fired_Off
           else RM_notrans when (act match st_Off)
                           when not fired_Off ;
```

Les calculs dans les autres états suivent le même schéma de traduction.

5.1.3 Analyse des modèles SCADE 6 et fonctions de correspondances

La traduction des automates nous permet de traiter les spécifications SCADE 6 à travers leur transformation vers le langage $\mathcal{L}_+$, celui-ci étant maintenant supporté par l'outil de génération de tests GATeL$_+$. Cependant, certaines propriétés de haut-niveau

qui apparaissent dans les automates de modes peuvent ne plus être exploitables dans le code traduit. Nous verrons dans la section 3.3 quelles sont ces propriétés et comment nous les traitons. Par ailleurs, nous nous intéressons dans ce chapitre à la couverture structurelle des automates de modes (§5.2), qui elle demande de pouvoir identifier des constructions de haut-niveau et de trouver les équations qui les concernent dans le modèle traduit. Finalement, on peut vouloir remonter les résultats obtenus lors de la génération de séquences de tests jusqu'à au niveau du modèle original, tel qu'un utilisateur l'a conçu. Dans ces différents cas, il est utile d'avoir une vision haut-niveau des modèles SCADE 6, ainsi que des relations entre ces modèles de haut-niveau et les flots de données produits par la traduction. Nous présentons ici ces deux aspects.

5.1.3.1 Analyse syntaxique de modèles SCADE 6

Nous ne sommes pas intéressés par l'intégration des constructions SCADE 6 directement dans les règles de propagation de contraintes, c'est-à-dire dans le noyau de génération de tests de l'outil, car il est suffisant, pour le moment, de passer par la forme intermédiaire d'un automate (celle exprimée dans le langage $\mathcal{L}_+$). Notre but ici est de connaître la structure des automates de modes d'un modèle donné dans GATeL. Nous construisons ainsi une base de faits dans le langage de contraintes qui regroupe les informations structurelles intéressantes contenues dans un automate.

En pratique, le logiciel SCADE maintient les modèles SCADE 6 sous forme de fichiers XML. Nous avons inféré la syntaxe du format de fichier à partir de différents exemples pour écrire une feuille de style XSLT [2]. Notre prototype permet de produire une base de faits Prolog à partir d'un fichier XML provenant de SCADE. Par exemple, pour l'automate SMRollMode (cf. page 181), la base de faits contient les informations suivantes :

- Il existe un unique nœud, que l'on nomme *node1* dans la représentation ; il contient un unique automate, idenfifié par *sm1*. Nous gardons aussi une référence vers le véritable nom de l'automate *rollmode* :
  - *node(node1,node(rollmode,[sm1]))*
- Pour chaque identifiant d'automate on répertorie son véritable nom, le nœud auquel il appartient, l'état et l'automate qui le contiennent (éventuellement *nil*), la liste des sous-automates, la liste des états (l'état initial en tête) et les listes des transitions.
  - *statemachine(sm1,sm(sm_rollmode,node1,nil,nil,[sm2],[st1,st2],[t1,t2]))*
  - *statemachine(sm2,sm(sm_on,node1,st2,sm1,[],[st3,st4],[t3,t4]))*
- Chaque état possède un nom, une référence vers l'automate auquel il appartient, la liste des sous-automates qu'il définit, la liste des transitions entrantes et sortantes,

2. Extensible Stylesheet Language Transformations

et contient deux booléens indiquant respectivement si le nœud est initial et/ou final.
  - *state(st1,state(off,sm1,[],[t2],[t1],true,false))*
  - *state(st2,state(on,sm1,[sm2],[t1],[t2],false,false))*
  - *state(st3,state(nominal,sm2,[],[t4],[t3],true,false))*
  - *state(st4,state(failsoft,sm2,[],[t3],[t4],false,false))*
- Chaque transition référence l'automate qui le contient, les états de départ et d'arrivée, ainsi que sa nature (strong/weak, resume/restart).
  - *trans(t1,trans(sm1,st1,st2,weak,restart))*
  - *trans(t2,trans(sm1,st2,st1,strong,restart))*
  - *trans(t3,trans(sm2,st3,st4,strong,restart))*
  - *trans(t4,trans(sm2,st4,st3,strong,restart))*

Navigation dans un modèle La structure dexs termes *node*, *sm*, *state* et *trans* respectivement associés aux nœuds, automates, états et transitions d'un modèle SCADE 6 est résumée ci-dessous, où les différents champs contenus dans ces structures sont nommés.

- « node » : un nœud contenant un ou plusieurs automates de modes

  name nom du nœud

  smlist liste des automates contenus dans le nœud
- « sm » : un automate *(state-machine)*

  name nom de l'automate

  parent automate parent

  state état englobant

  children automates imbriqués dans les différents nœuds

  states liste des états

  trans liste des transitions
- « st » : état d'un automate

  name nom de l'état

  sm automate qui possède l'état

  smlist automates inclus dans cet état

  incoming transitions entrantes

  outgoing transitions sortantes
- « trans » : transition dans un automate

  sm automate qui contient la transition

  from état source

to état cible

ws weak/strong

rr resume/restart

On note respectivement $\mathcal{C}_{\texttt{node}}$, $\mathcal{C}_{\texttt{sm}}$, $\mathcal{C}_{\texttt{st}}$ et $\mathcal{C}_{\texttt{tr}}$ le sous-ensemble du langage de termes $\mathcal{C}$ représentant ces constructions. L'ensemble $\mathcal{C}_A$ représente le sous-langage encodant les automates. Les champs nommés des structures nous servent à naviguer d'un élément à un autre du modèle, à l'aide d'une notation pointée. Par exemple, pour un automate $A \in \mathcal{C}_{\texttt{sm}}$, $A.states$ est la liste des états contenus dans A. De même, pour une transition T appartenant à $\mathcal{C}_{\texttt{tr}}$, $T.from$ nous donne l'état source de T. Pour la transition $t1$ de SMRollMode, l'état source $t1.from$ est ici l'identifiant $st1$.

Un identifiant nous donne implicitement l'élément référencé. Pour de plus de simplicité, nous assimilons toujours une référence (comme $st1$) à l'élément qu'elle identifie, à savoir le terme St_1 ci-dessou :

$$St_1 = state(off,sm1,[],[t2],[t1],true,false)$$

Si la référence est *nil*, c'est cette valeur *nil* qui est obtenue. Nous pouvons ainsi composer les accès aux références si nécessaire, comme dans l'exemple suivant, où on accède aux états de l'automate parent de A :

$$A.parent.states$$

En réalité, *A.parent* est un identifiant à travers lequel on retrouve un terme auquel on accède au champ *states*…. Nous n'avons pas besoin d'autant de détails ici.

5.1.3.2 Relations de correspondances entre modèles

En se basant sur la fonction de traduction entre SCADE 6 et $\mathcal{L}_+$, nous pouvons définir des relations de correspondance entre notre représentation des modèles SCADE 6 et les flots de données encodant la sémantique d'automates de modes. Soient A un identifiant d'automate de mode (p. ex. *sm1*), E un identifiant d'état (p. ex. *st1*) et T un identifiant de transition de A (p. ex. *t1*).

Types énumérés Nous définissons *sm_states*(A) et *sm_trans(A)* comme étant les types énumérés associés aux états et aux transitions de A dans le modèle traduit. De même, nous définissons la fonction *st_val(E)*, donnant la valeur symbolique correspondant à l'état E dans *sm_states*(A), et *tr_val(T)* la valeur symbolique associée dans *sm_trans(A)* à la transition T. Finalement, nous définissons *notrans(A)* comme étant la valeur constante associée à l'absence de transition dans le type énuméré *sm_trans(A)*. Pour l'automate SMRollMode, on a :

$$\begin{aligned} \textit{sm\_states(sm1)} &= \{\texttt{st\_Off}, \texttt{st\_On}\} \\ \textit{sm\_trans(sm1)} &= \{\texttt{tr\_Off\_1}, \texttt{tr\_On\_1}\} \\ \textit{notrans(sm1)} &= \texttt{RM\_notrans} \end{aligned}$$

$$\begin{aligned} \textit{st\_val}(st1) = \texttt{st\_Off} \quad \textit{st\_val}(st2) = \texttt{st\_On} \\ \textit{tr\_val}(t1) = \texttt{tr\_Off\_1} \quad \textit{tr\_val}(t2) = \texttt{tr\_On\_1} \end{aligned}$$

Variables d'états Nous définissons $sel(A)$ (resp. $act(A)$) comme étant le flot auxiliaire associé à l'état sélectionné (resp. actif) de l'automate A. La valeur retournée est un triplet appartenant à $\mathcal{V}ars_D$, pour la déclaration de types D produite par la traduction. Dans notre exemple, la correspondance est la suivante :

$$\begin{aligned} \textit{sel(sm1)} &= (\texttt{sel}, \textit{sm\_states(sm1)}, \textit{base}) \\ \textit{act(sm1)} &= (\texttt{act}, \textit{sm\_states(sm1)}, \textit{base}) \\ \textit{sel(sm2)} &= (\texttt{sel\_On}, \textit{sm\_states(sm2)}, (\texttt{act}, \texttt{st\_On})) \\ \textit{act(sm2)} &= (\texttt{act\_On}, \textit{sm\_states(sm2)}, (\texttt{act}, \texttt{st\_On})) \end{aligned}$$

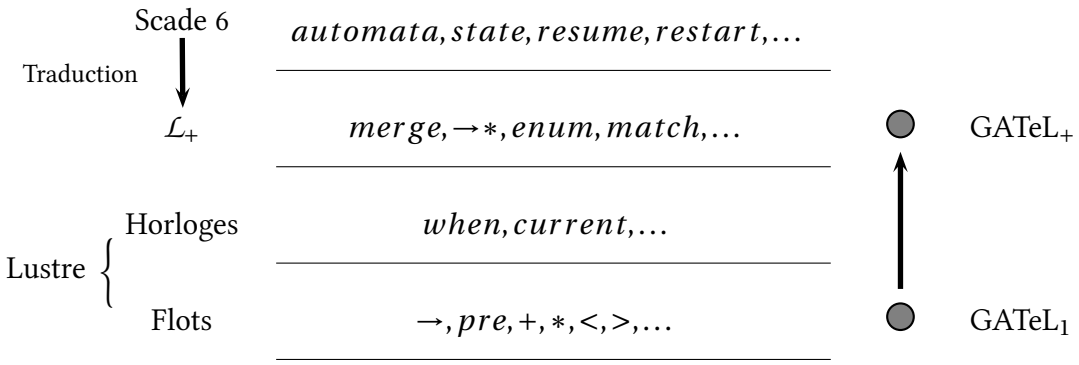
Transitions Nous définissons $\textit{fired}(A)$ comme étant le flot indiquant quelle transition est franchie à un cycle donné. Dans l'exemple que nous avons suivi, il s'agit du flot « `fired2` » (§5.1.2.3) :

$$\textit{fired(sm1)} = (\texttt{fired2}, \textit{sm\_trans(sm1)}, \textit{base})$$

Conclusion À l'aide des informations de haut-niveau présentes dans SCADE 6 et de la connaissance de la fonction de traduction vers $\mathcal{L}_+$, nous avons pû établir des relations de correspondances entre les automates de modes et leur encodage. Nous utilisons ces informations par la suite pour la génération de séquences de tests et pour la couverture structurelle d'automates. Ces informations nous donnent aussi un moyen d'intéragir avec l'utilisateur en lui présentant le modèle de l'automate tel qu'il était conçu dans l'éditeur graphique.

5.1.4 Génération de séquences de tests

Nous avons présenté langage SCADE 6 et de sa sémantique, ainsi qu'une intuition de l'encodage de celui-ci dans le langage $\mathcal{L}_+$. Le tableau suivant réacapitule les différents langages que l'on considère, l'évolution de l'outil GATeL ainsi que la traduction que l'on exploite.



Les dialèctes de la famille Lustre sont étagés selon la richesse de la sémantique. À titre indicatif, les opérateurs représentatifs de chaque sous-langage est représenté. Ainsi, le langage Lustre original décrivant déjà une sémantique avec horloge peut-être découpé en un sous-ensemble mono-horloge (flots de données purs) et une seconde partie introduisant les opérateurs `when` et `current` servant à cadencer les flots à des rythmes différents. Nous découpons ce langage en deux car initialement, GATeL$_1$ ne traitait que la partie mono-horloge du langage, les horloges étant peu rencontrées dans les cas d'étude industriels. Le langage Scade 6 définit une sémantique de haut-niveau qui mélange les flots de données et les automates à états finis pour donner des automates de modes. La compilation de Scade 6 a pour cible intermédiaire le langage $\mathcal{L}_+$ présenté au chapitre 2. Nous avons décrit au chapitre 3 quelques mécanismes qui expliquent comment nous traitons maintenant directement le langage $\mathcal{L}_+$ dans GATeL$_+$. Ainsi, nous pouvons générer des séquences de tests pour les automates de modes de Scade 6 par l'intermédiaire du langage $\mathcal{L}_+$.

5.1.5 Contrainte *automaton*

La traduction nous donne un encodage possible des automates de modes dans $\mathcal{L}_+$. Cependant, la structure de haut-niveau (états/transitions) sont perdues et ne peuvent plus être exploitées après la traduction. Cela signifie que, même si la traduction nous donne la capacité de générer des tests pour Scade 6, il peut y avoir des cas où la compilation masque des informations exploitables lors de la propagation de contraintes.

Nous présentons dans cette section la contrainte *automaton* [BJM$^+$10], qui lie entre-elles les valeurs ponctuelles des variables *sel(A)*, *act(A)* et *nxt(A)* d'un automate *A*. Plus précisément, si on note *Sel*, *Act* et *Nxt* les variables de $\mathbb{V}$ représentant les variables d'états d'un automate au cycle c, alors *automaton(Sel,Act,Nxt)* peut propager des réduction de domaines d'une variable à une autre en se basant pour cela sur le fait qu'au plus une transition est tirée par cycle.

Cela veut dire par exemple que si le domaine de *Sel* est différent du domaine de *Act*, il y a nécessairement une transition forte présente au cycle courant ; comme il s'agit de

la seule transition possible, il n'est pas possible de franchir une transition faible et le prochain état, *Nxt*, est alors égal à l'état *Act*. Il en va de même si le domaine de *Nxt* est différent de celui de *Act*, où cette fois une transition faible est franchie ce qui nous autorise à unifier les variables *Sel* et *Act*.

$$\textsc{auto-strong}\ \frac{S \vdash Sel \approx D_S \qquad S \vdash Act \approx D_A \qquad S \vdash Nxt \approx D_N \qquad D_A \cap D_S = \emptyset}{\vdash S \cup \{automaton(Sel,Act,Nxt)\} \rightarrow S \cup \{Nxt \equiv Act\}}$$

$$\textsc{auto-weak}\ \frac{S \vdash Sel \approx D_S \qquad S \vdash Act \approx D_A \qquad S \vdash Nxt \approx D_N \qquad D_N \cap D_A = \emptyset}{\vdash S \cup \{automaton(Sel,Act,Nxt)\} \rightarrow S \cup \{Sel \equiv Act\}}$$

Autrement dit, la valeur prise par *Act* est dans tous les cas égale à l'une des valeurs des deux autres variables :

- Si aucune transition n'est franchie, elles sont toutes égales.
- Si une transition forte est franchie : $Sel \neq Act$ et $Act = Nxt$.
- Si une transition faible est franchie : $Sel = Act$ et $Act \neq Nxt$.

En plus des règles précédentes, on peut parfois réduire le domaine de *Act* : comme cette variable s'unifiera soit avec *Sel* soit avec *Nxt*, son domaine de valeurs est compris dans l'union des domaines de ces dernières.

$$\textsc{auto-union}\ \frac{S \vdash automaton(Sel,Act,Nxt) \qquad S \vdash Sel \in D_S \qquad S \vdash Nxt \in D_N \qquad D'_A = D_A \cap \{D_S \cup D_N\} \neq D_A}{\vdash S \cup \{Act \in D_A\} \rightarrow S \cup \{Act \in D'_A\}}$$

De cette manière, *Sel*, *Act* et *Nxt* sont traitées localement par *automaton*. Si on la retire, ces variables ne communiquent plus que par l'intermédiaire de la chaîne de contraintes créée par *propage*, qui relie les valuations des flots selon la structure des expressions Lustre. Or, à travers plusieurs contraintes, les dépendances sont moins nettes et les réductions de domaines moins efficaces.

D'autres travaux de cette nature sont en cours pour traduire les informations pertinentes des automates vers des contraintes entre flots de données. Par exemple, le graphe des états et des transitions peut donner des indications utiles sur les séquences de valeurs prises par les variables d'états. Comme les automates de Scade 6 admettent une représentation sous forme de contraintes, il est facile de les analyser depuis GATeL$_+$. En résumé, bien que nous générons des tests pour Scade 6 par le biais d'une traduction intermédiaire, nous pouvons aussi traiter les informations de haut-niveau du langage.

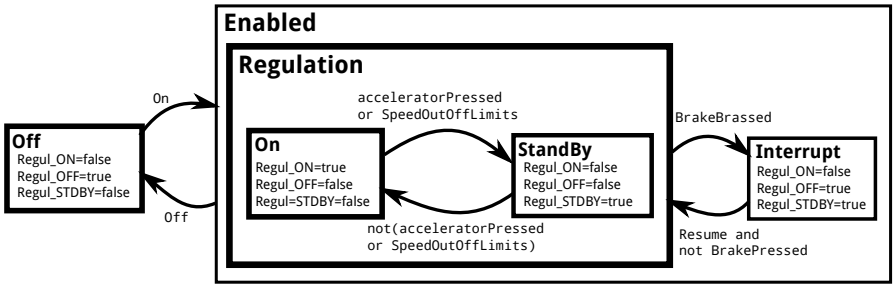


FIGURE 5.4: Automate *CruiseControl*.

5.1.6 Expérimentations

La figure 5.4 montre l'automate *CruiseControl* que nous avons utilisé lors de nos expérimentations.

CruiseControl Nous avons réalisé des mesures pour l'automate *CruiseControl* tel qu'il était modélisé dans SCADE 5. L'automate contrôle un régulateur de vitesse. Il admet deux états principaux, allumé ou éteint, représentés par les états *Enabled* et *Off*. Dans l'état *Enabled*, peut-être soit dans le mode nominal *Regulation*, soit en mode *Interrupt* quand la commande de frein est appuyée suffisamment longtemps et jusqu'à ce qu'on autorise le régulateur à redémarrer. En mode régulation, le contrôleur peut-être actif, dans l'état *On*, où en attente (*StandBy*).

les variables `On`, `Off`, `Brake`, `Resume`, `Accel` et `SpeedLimits` sont des entrées booléennes du système. L'automate produit 3 sorties booléennes appelées `ON`, `OFF` et `STDBY` qui encodent l'état courant du régulateur pour les autres composants du système. Les sorties dans les différents états sont représentées dans le tableau suivant :

États			ON	OFF	STDBY
Off			false	true	false
Enabled	Interrupt		false	true	true
	Regulation	On	true	false	false
		StandBy	false	false	true

On peut voir que l'état d'interruption est encodé par la valeur *true* simultanée pour `OFF` et `STDBY`. Les transitions entre les états sont donnés dans ce tableau :

Source	Cible	Condition de franchissement
Off	Enabled	On
Enabled	Off	Off
Regulation	Interrupt	Brake
Interrupt	Regulation	Resume and (not Brake)
On	StandBy	Accel or SpeedLimits
StandBy	On	not(Accel or SpeedLimits)

Nous avons choisi cet automate car il en existe une version à la fois dans SCADE 6 et dans SCADE 5. L'automate dans SCADE 5 était modélisé dans le langage des *Safe State Machines*, puis compilé vers du LUSTRE mono-horloge. Nous appelons cette version mono-horloge **Cruise I**. La version SCADE 6 est traduite comme on l'a vu précédemment vers le langage $\mathcal{L}_+$, c'est-à-dire à l'aide de flots multi-horloges ; nous appelons cette version **Cruise III**. Finalement, nous avons écrit une version mono-horloge appelée **Cruise II** à partir de Cruise III, où l'on a simulé la sémantique des horloges dans le langage mono-horloge. La simulation cherche à produire des séquences observables identiques à la version multi-horloges, aux instants où les horloges sont présentes. Cela nécessite de passer par un encodage, et notamment celui du retard unitaire.

Simulation Nous reprenons un instant l'exemple vu page 38 dans la figure 5.5. Dans cet exemple, le flot `xb2` temporisé était calculé par un simple décalage dans le temps avec `pre`. Ici, le flot `pxb` simule la valeur de `xb2`, de sorte à ce que les valeurs significatives (lorsque `x` vaut `b`) soient identiques. Pour cela, il faut se souvenir explicitement de la dernière valeur prise par `x` la dernière fois que `s` valait `b` : cette mémorisation est implicite dans le langage multi-horloges, alors qu'elle doit être encodée si on veut se passer des flots temporisés. Le flot `pxb` réalise ce calcul et permet ainsi de simuler le décalage du flot temporisé `xb` que l'on a vu précédemment (le flot `xb2` à la figure **??**) :

```
pxb = 0 → (if (pre(s) = b) then pre(x) else pre(pxb)) ;
```

Les valeurs du flots aux instants où `s` vaut `b` sont encadrées dans la figure 5.5 et corresondent aux valeurs de `xb2`. Nous avons mis en œuvre ce genre de simulation, en remplaçant systématiquement les opérateurs `when`, `pre` et `merge` par des combinaisons de flots à l'aide notamment de `if..then..else` et de `pre` mono-horloges.

Efficacité de la propagation Les systèmes Cruise II et Cruise III reprennent presque le même encodage, à ceci près que les horloges sont simulées par des flots mono-horloges dans le cas de Cruise II. Lorsque nous tentons de générer des tests pour ces deux versions, on peut voir si le fait d'intégrer directement les horloges dans GATeL$_+$ nous permet d'être plus efficace ou s'il est suffisant d'encoder les horloges dans du LUSTRE à horloge unique. Les systèmes Cruise I et Cruise II sont tous les deux mono-horloges, mais diffèrent par

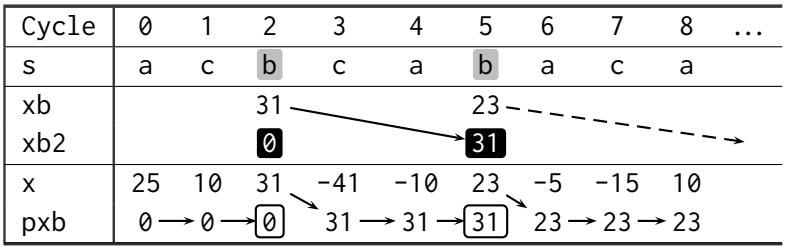


FIGURE 5.5: Simulation du retard unitaire temporisé dans $\mathcal{L}_1$

l'encodage du même automate à l'aide de flots. Le premier test réalisé est représenté dans la graphique de la figure 5.6.

Nous avons cherché à générer une séquence de test avec un objectif de test que l'on sait irréalisable pour l'automate sous test, à savoir atteindre un cycle où `ON` et `STDBY` valent toutes deux *true*, ce qui est impossible d'après la table précédente. Cet objectif oblige GATeL à parcourir tout l'espace de recherche avant de pouvoir réfuter l'existence d'une solution : on cherche à voir si la propagation de contraintes est suffisamment efficace pour détecter des insatisfaisabilités locales et éviter d'explorer justement toutes les combinaisons possibles. Nous avons donc demandé à GATeL$_+$ de générer des séquences de tests à partir de l'objectif demandé pour les 3 versions. Pour cela, nous avons modifié le paramètre *maxsize* du problème d'une génération à une autre, pour voir l'impact de cette variable sur le temps de génération.

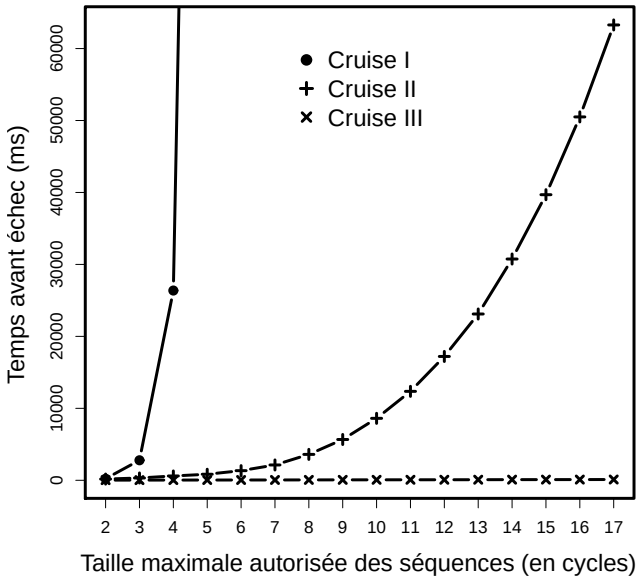


FIGURE 5.6: Comparaison du temps de réfutation quand la taille autorisée des séquences augmente.

À la figure 5.6, nous pouvons voir les courbes pour les trois versions du modèle et l'impact de la variable *maxsize* sur le temps nécessaire pour détecter l'inatteignabilité de l'objectif de test.

Cruise I Les contraintes n'arrivent pas à filtrer suffisamment bien les domaines ce qui oblige la fonction de recherche à tester un grand nombre de combinaison de valeurs pour chaque variable du problème. Une des variable testées est la variable de statut d'horloges, qui est instanciée à *initial* puis à *noninitial* : on génère donc tous les cas pour 1 cycles, puis tous les cas pour 2 cycles, etc. jusqu'à atteindre le cycle maximal autorisé. C'est pourquoi le temps d'exécution augmente très rapidement.

Cruise-II L'encodage choisi est plus favorable ici à GATeL que celui réalisé pour la version I. Néanmoins, le temps nécessaire à la réfutation augmente toujours en fonction de la borne *maxsize* et atteint 1 minute dès 17 cycles.

Cruise-III Le temps avant réfutation ne dépend plus directement de la borne *maxsize* mais est au contraire constant et très faible. Avec la représentation temporisée du modèle que l'on a dans GATeL$_+$, les équations de flots pour les sorties sont simplifiées et apparaissent avec des horloges différentes selon les états actifs des automates. On arrive à éliminer facilement les combinaisons de flots infaisables car les deux valeurs que l'on souhaite observer simultanément sont d'horloges différentes. Le filtrage est suffisamment efficace pour permettre au système de détecter rapidement les combinaisons impossibles de valeurs.

5.2 Couverture structurelle d'automates de modes

Nous souhaitons pouvoir guider la génération de séquences de tests afin de couvrir structurellement les automates de modes. Cela signifie par exemple que l'on aimerait produire des séquences qui activent chacune une transition de l'automate, dans le but de couvrir tous les changements d'états.

Nous présentons de quelle manière nous pouvons couvrir des modèles SCADE 6 à travers des flots observateurs : chaque critère de couverture est exprimé par un ou plusieurs flots booléens temporisés. Une fois que ces flots sont intégrés au modèle sous test, nous exploitons la méthode de génération de tests par objectif de test en cherchant à produire une séquence de tests dans laquelle la couverture considérée est observée. Cela est traduit par un problème de satisfaction de contraintes dynamiques dans GATeL, qui produit si c'est possible un historique de valeurs compatible avec cet objectif. L'approche de couverture structurelle est ainsi intégrée au mécanisme de génération par chaînage arrière, et peut-être combiné à d'autres objectifs de tests, structurels ou fonctionnels.

5.2.1 Adaptation de critères classiques

Le domaine du test d'automates à états finis possède de nombreux critères de couverture structurelle [BT01, OLAA03].

Certaines approches cherchent à s'assurer de la conformité d'une implémentation vis-à-vis d'un modèle à transitions d'états, sans avoir de connaissance préalable de cette implémentation [LY96]. Dans notre cas, nous nous plaçons dans un contexte où l'implémentation est directement produite à partir du modèle original, à l'aide d'un compilateur certifié. Nous ne nous intéressons donc pas au problèmes de conformité du code bas-niveau généré par rapport au modèle SCADE 6, même si c'est une piste de travail intéressante pour les cas où : (i) ce code bas-niveau ne serait pas produit par compilation, ou (ii) on souhaiterait produire des jeux de tests pour s'assurer que le compilateur n'introduit pas de fautes. Nous cherchons à couvrir les modèles SCADE 6 afin de les valider, c'est-à-dire afin de s'assurer qu'ils sont conformes aux besoins. En cherchant à produire des tests couvrant tous les états, toutes les transitions, etc., on peut ainsi se rendre compte qu'un modèle ne modélise pas le *bon* système réactif.

Les automates de modes de SCADE 6 sont cependant différents des automates à états finis, ou FSM (*finite state-machine*), car ils introduisent les notions de composition parallèle et hiérarchique, de signaux et de rendez-vous, ou encore d'états avec mémoire, ce qui les rend proches du formalisme des STATECHARTS. Des travaux portant sur le test et la couverture structurelle de modèles STATECHARTS existent déjà [Bur99, HLSC01], notamment ceux définissant la famille de critères de tests SCCF [3] [dSMFM00], qui cible les particularités de ces modèles. Nous nous inspirons de ces travaux pour adapter la

3. Statecharts Coverage Criteria Family

couverture structurelle d'automates aux modèles SCADE 6. Nous pouvons aussi citer des travaux récents autour de la couverture de spécifications LUSTRE/SCADE ayant la même approche, à l'aide de flots booléens appelés « conditions d'activation » des critères structurels [PMDBP09, LP09]. À notre connaissance, ces travaux s'orientent cependant plutôt sur les flots temporisés et non sur les modèles des automates de modes. Nous définissons ici les critères de couverture suivants :

all-states	Couverture de tous les états actifs.
all-transitions	Couverture de toutes les transitions d'un automate.
all-nested-transitions	Idem, avec les transitions imbriquées.
transition-pair	Couverture d'une paire de transitions.
all-transition-pairs	Couverture de toutes les paires de transitions.
all-configurations	Couverture de toutes les configurations d'états actifs.

5.2.1.1 Couverture de tous les états

Le premier critère de test vise à couvrir l'ensemble des états d'un automate. On considère uniquement les états actifs, c'est-à-dire ceux où les calculs effectifs des sorties ont lieu. Par exemple, pour couvrir ce critère pour l'automate SMRollMode, on doit générer un test qui atteint l'état *On* et un deuxième qui atteint l'état *Off*. Dans le premier cas, il faut activer la transition faible $t1$ depuis l'état initial *Off*, et s'assurer que la transtition forte $t2$ n'est ensuite *pas* franchie ; sinon, l'automate passerait uniquement dans l'état sélectionné *On* de manière transitoire avant de repasser dans l'état *Off*.

5.2.1.2 Couverture de transitions

Nous mettons en œuvre la couverture d'une transition, au sens général, ce qui regroupe les transitions fortes et faibles, avec rédemarrage ou non, ainsi que les transitions *synchro* et les signaux. Nous étudions aussi la couverture d'un ensemble de transitions présentes dans un automate. Ainsi, le critère *all-transitions* est couvert si on peut générer toutes les séquences couvrant chacune une transition différente d'un automate. Cela est différent d'un critère qui viserait à passer par toutes les transitions dans la même séquence.

Étant donné que les automates peuvent être imbriqués, nous définissons aussi le critère *all-nested-transitions*, qui considère les transitions de tous les sous-automates présents dans un automate donné.

5.2.1.3 Couverture des paires de transitions

Le critère *transition-pair* [OLAA03] demande de fournir un test dans lequel deux transitions consécutives sont empruntées, l'une faisant entrer l'automate dans un mode actif, et l'autre le faisant en sortir, après un ou plusieurs cycles de calculs. En effet, pour SCADE 6, on adapte le critère de sorte que le système puisse rester dans le

mode intermédiaire considéré le temps nécessaire à ce que la transition sortante soit franchissable. On rappelle qu'un mode de calcul est composé d'équations de flots qui peuvent influer sur les conditions de franchissement des transitions d'un automate. Il peut arriver qu'il faille attendre plusieurs cycles de calculs avant que la seconde transition du critère ne soit empruntable. Nous étendons aussi celui-ci afin de couvrir l'ensemble des paires de transitions d'un automate.

5.2.1.4 Couverture des configurations

Enfin, nous définissons le critère *all-configurations*, qui vise à s'assurer que toutes les combinaisons d'états simultanément actifs sont visitées, dans le cas où des sous-automates évoluent en parallèle.

Une configuration pour un ensemble d'automates en parallèle est l'ensemble des combinaisons d'états *feuilles* qui peuvent être actifs simultanément. Un état feuille est un état qui n'est plus décomposé en sous-automates : toutes les variables à calculer sont données par une équation. La couverture de critère mène à une séquence par configuration.

Par exemple, afin de couvrir chaque configuration de SMRollMode, il nous faut produire trois séquences pour couvrir respectivement les états *Off*, *Nominal* et *Failsoft*. Si on suppose que la constante `kFailSoftRoll` vaut 10, alors la séquence suivante couvre l'état *Failsoft* :

Cycle	0	1
onOffPressed	`true`	`false`
absRollRate	5	12
mode	`off`	`failsoft`

5.2.1.5 Conclusion

Les critères présentés ici permettent de couvrir un grand de nombre de cas pour les automates de modes. En effet, étant donné l'encodage des automates, le critère *all-(nested)-transition* couvre aussi tous les signaux de l'automate (*all-broadcasting*), toutes les transitions *resume* (*all-history-configurations*), de même que les transitions *synchro* (*all-possible-rendezvous*) telles que définies dans la famille de critères SCCF [dSMFM00]. Il est possible de détailler les critères pour couvrir ces sous-cas de manière isolée, simplement en sélectionnant un sous-ensemble des transitions ou des états à couvrir avec les critères précédents.

5.2.2 Utilisation des mécanismes de sélection de GATeL

Pour un modèle dans le langage $\mathcal{L}_+$ et un objectif de test, GATeL construit un ensemble de contraintes S. Nous avons vu que cet ensemble évolue au fil des propagations et de

la résolution de contraintes. Cependant, GATeL a la possibilité de traiter plusieurs de ces ensembles de contraintes de manière indépendante, qui correspondent chacun à un cas de test propre. Ainsi, lors de l'utilisation intéractive de l'outil, la propagation de contraintes est pilotée par un testeur, qui peut raffiner l'objectif de test en *découpant* un cas de tests en plusieurs sous-cas. Il existe deux mécanismes issus de cette approche, le découpage structurel des flots de données, et une méthode de découpage en sous-objectifs de tests, qui correspond alors une sélection selon des critères fonctionnels. Nous avons vu ces mécanismes respectivement aux pages 104 et 105.

5.2.3 Observation de la couverture des critères

Nous décrivons la couverture structurelle d'automates de modes à travers des expressions booléennes du langage $\mathcal{L}_+$. Ces flots caractérisent la couverture de certains éléments du modèle original SCADE 6, à savoir un état ou une transition. Lorsqu'un tel flot observateur est introduit sous forme de contraintes dans le système à résoudre, et que l'on force l'occurence d'une valeur *true* pour ce flot à un cycle, celui-ci restreint les séquences générées à celles qui sont compatible avec la couverture du critère considéré au cycle choisi.

Nous présentons ici les fonctions associées aux différents critères énoncés à la section **??**. Ces fonctions agissent sur des éléments structurels d'un modèle SCADE 6 quelconque et produisent des expressions dans le langage $\mathcal{L}_+$. Nous exploitons la possibilité de construire des flots temporisés, mais afin de pouvoir le faire tout en respectant la bonne construction des expressions avec horloges, nous avons besoin de la fonction auxiliaire suivante.

5.2.3.1 Fonction auxiliaire *filter*

Dans le langage $\mathcal{L}_+$, les constantes ont pour horloge l'horloge de base du nœud où elles apparaissent. Cela veut dire que si on souhaite comparer un flot f avec une constante, il faut faire en sorte de filtrer la constante jusqu'à ce que celle-ci ait la même horloge que f. La fonction auxiliaire *filter* sert précisément ce but et construit une expression composées d'opérateurs `when` imbriqués autour d'une constante v et en fonction d'une horloge H.

On a vu par exemple dans la présentation de la traduction de modèles SCADE 6, et en particulier à la section 5.1.2.3, que les expressions rencontrées comportent un grand de nombre de ces filtrages de constantes. Le même cas de figure se présente à nous pour la définition des flots observateurs, qui doivent respecter les règles de calcul des horloges.

Définition 17. *Filtrages successifs d'une constante*

Soient $N = (D, I, L, O) \in \mathcal{L}_+$ un nœud, $c \in \mathcal{V}alues_D$ une constante du langage dans ce nœud, et $(id, v) \in \mathcal{C}locks_N$ une horloge bien formée dans N.
Alors, la fonction *filter* construit autour de la constante c l'ensemble des filtrages successifs nécessaires pour obtenir une expression d'horloge (id, v) dont la valeur aux instants de présence est c :

$$\begin{array}{rcl} filter(base, c) & = & c \\ filter((id, v), c) & = & (W \texttt{ when } id \texttt{ match } v) \\ & \text{avec} & W = filter(clock(var_N(id)), c) \end{array}$$

5.2.3.2 Couverture des états

Soit $A \in \mathcal{C}_{\texttt{sm}}$ un automate et $E \in \mathcal{C}_{\texttt{st}}$ un état référencé dans $A.states$, la liste des états de A (nous reprenons les notations vues à la section 5.1.3.1). Soient par ailleurs $Act = act(A)$ la variable de flot représentant l'état actif de A, ainsi que $e = st\_val(E)$ la valeur symbolique associée à l'état E, dans le modèle $\mathcal{L}_+$ issu de la traduction des automates (nous utilisons la fonction de correspondance vue à la section 5.1.3.2). On note finalement $h = clock(Act)$ l'horloge du flot Act (ici, nous utilisons les accesseurs définis à la section 2.2.1.8 dans la définition du langage $\mathcal{L}_+$). Alors, la couverture de l'état actif E s'observe à l'aide du flot observateur $cover\_state(E)$ ainsi défini :

$$cover\_state(E) = \ (\ id(Act) \ = \ filter(h, e)\ )$$

Le flot observateur obtenu n'est vrai qu'aux cycles où l'état actif, représenté par l'identifiant de « Act », est égal à la valeur symbolique représentant l'état E. Le flot observateur résultat respecte le calcul d'horloge graĉe à la fonction auxiliaire *filter* qui fait en sorte de cadencer e sur la même horloge que celle de Act.

Lorsque l'on veut couvrir un état actif dans un automate, il nous suffit désormais d'utiliser l'expression produite par la fonction *cover_state*. Pour pouvoir intégrer cette expression au système de contraintes, on peut l'ajouter à l'objectif de test, ou la rendre activable par l'intermédiaire d'une directive `split` :

$$\texttt{split } id(sel(A)) \texttt{ with } [cover\_state(E)]$$

Dans ce cas, l'expression peut-être introduite par l'utilisateur à tous les cycles où le flot associé à l'état sélectionné de l'automate est propagé dans le système de contraintes. Ici, il n'y a qu'un sous-cas de test ; nous généralisons cette approche pour couvrir un ensemble d'états.

5.2.3.3 Couverture de tous les états

Le critère *all-states* est couvert lorsque chaque état d'un automate est couvert par une séquence de test. Soit $A \in \mathcal{C}_{\texttt{sm}}$ un automate. La sélection de ce critère produit autant de cas de tests qu'il y a d'états dans l'automate A.

Soit $A.states = [s_1, \ldots, s_n]$, avec $n > 0$; nous rappellons que cette liste contient des références vers des états. Nous réutilisons la couverture d'un état décrite auparavant et définissons alors *all_states(A)* comme étant le flot observateur suivant :

```
all_states(A) = split id(sel(A)) with
      [ cover_state(s_1) ;
          ...
        cover_state(s_n) ]
```

5.2.3.4 Couverture d'une transition

La même approche est appliquée pour la couverture d'une transition. Soit $T \in \mathcal{C}_{\texttt{tr}}$ une transition que l'on souhaite voir franchie. On note $A = T.sm$ l'automate qui contient cette transition et F le flot défini par $F = fired(A)$, c'est-à-dire la variable encodant la transition franchie à un cycle. L'expression booléenne $cover\_trans(T)$ est donc ainsi définie :

$$cover\_trans(T) = (id(F) = filter(clock(F), tr\_val(T)))$$

Le fait de comparer le flot F à la valeur symbolique associée à la transition T suffit à savoir si T est franchie.

5.2.3.5 Couverture de toutes les transitions

Le critère *all-transitions* s'applique à tout automate $A \in \mathcal{C}_{\texttt{sm}}$ et couvre chaque transitions de celui-ci ; soit $A.trans = [t_1, \ldots, t_n]$ la liste des transitions de A, avec $n > 0$. Alors, $all\_trans(A)$ est donnée par l'expression *all_trans(A,{t_1,...,t_n})*, elle-même définie ainsi :

```
all_trans(A, {t_1,...,t_n}) = split id(sel(A)) with
                                [ cover_trans(t_1) ;
                                    ...
                                  cover_trans(t_n) ]
```

5.2.3.6 Couverture de toutes les transitions imbriquées

Nous réutilisons la définition précédente pour mettre en œuvre le critère *all-nested-transitions*, dont le but est d'activer toutes les transitions d'un automate, y

compris celles présentes dans les sous-automates de celui-ci, que ce soit directement ou aux niveaux les plus bas. L'expression de couverture est définie comme étant $all\_trans(A, collect\_nested([A]))$, où la fonction auxiliaire *collect_nested* donne l'ensemble de toutes les transitions imbriquées à partir d'une liste d'automates. Pour deux listes L_1 et L_2, avec $L_1 = [\alpha_1, \ldots, \alpha_n]$, on note $L_1 :: L_2$ la concaténation des deux listes, qui est alors la liste contenant tous les éléments de L_1 suivis de ceux de L_2 :

$$L_1 :: L_2 = [\alpha_1, \ldots, \alpha_n | L_2]$$

Soit un automate $A \in \mathcal{C}_{\mathtt{sm}}$. Nous définissons *collect_nested(L)* comme étant la concaténation de toutes les listes de transitions appartenant aux automates et sous-automates de la liste L :

$$\begin{aligned} collect\_nested([]) &= [] \\ collect\_nested([A|L]) &= A.trans :: collect\_nested(A.children) :: collect\_nested(L) \end{aligned}$$

Nous déstructurons la liste donnée en un élément de tête A et une sous-liste L. L'appel à *collect_nested(L)* nous donne récursivement la liste des transitions imbriquées pour la liste L, éventuellement vide si L est vide. De même, l'appel à *collect_nested(A.children)* nous donne l'ensemble des transitions imbriquées dans la liste des automates enfants de A. La liste *A.children* peut-être vide. Nous concaténons ces deux listes et ajoutons en tête l'ensemble des transitions contenues dans A.

5.2.3.7 Couverture des paires de transitions

Le critère de couverture d'une paire de transition s'applique à un état B appartenant à un automate X, une transition entrante T_{AB} et une transition sortante T_{BC}. Il est couvert après la couverture consécutive des deux transitions, la première entre un état A et l'état B, la seconde entre l'état B et un état C. Dans la figure 5.7, on peut distinguer aussi une transition en pointillé sortant de B : elle représente les transitions issues de B différentes de T_{BC}. En effet, le critère demande que les deux transitions soient consécutives, ce qui interdit à toute autre transition d'être empruntée pendant que le système est dans le mode B. Cependant, il est possible que le mode B soit complexe et fasse intervenir des sous-automates, et dans ce cas on autorise le passage dans un ou plusieurs sous-modes de B avant de couvrir le critère.

Nous définissons le flot observateur $tr\_pair(T_{AB}, T_{BC})$ à l'aide d'un nœud Lustre auxiliaire, générique, appelé « `transpair` ». Il nous alors suffit de passer les bons

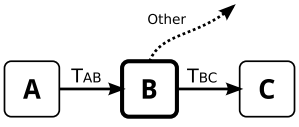


Figure 5.7: Couverture des paires de transitions

arguments à ce nœud pour que celui-ci observe les franchissements attendus. Le nœud `transpair` est défini plus bas.

Nous commençons par définir le flot $Other$, l'expression représentant le fait qu'une transition sortant de B, différente de T_{BC}, est franchie. On note pour cela $F = fired(X)$ le flot associé aux transitions franchies dans l'automate X. On a :

$$Other = id(F) <> filter(clock(F), notrans(X)) \\ \texttt{and } id(F) <> filter(clock(F), tr\_val(T_{BC}))$$

Nous rappelons que $notrans(X)$ est la valeur du type énuméré $sm\_trans(X)$ figurant l'absence de transition dans X. Alors, $Other$ est vrai à tous les cycles où une transition est franchie, mais différente de T_{BC}.

La directive « `split` » associée au critère s'exprime pour le moment sous la forme suivante :

$$tr\_pair(T_{AB}, T_{BC}) = \texttt{split } id(sel(X)) \texttt{ with} \\ \texttt{[transpair(}cover\_trans(T_{AB}), \\ cover\_trans(T_{BC}), \\ Other\texttt{)] ;}$$

Lorsque la directive est appliquée à un cycle, le flot booléen retourné par l'appel au nœud `transpair` doit retourner `true`. Ce nœud prend trois arguments :

```
node transpair(trAB, trBC, other: bool)
returns (pair: bool)
let
    pair = never_since(other, trAB) and trBC ;
tel;
```

Ici, `trAB` (resp. `trBC`) est vrai à un cycle si et seulement si la transition T_{AB} (resp. T_{BC}) y est franchie ; nous réutilisons la fonction *cover_trans* vue précédemment dans l'appel de nœud pour ces deux premiers arguments.

Dans le corps du nœud de `transpair`, l'appel intermédiaire au nœud `never_since` signifie : *aucune transition différente de T_{BC} n'a été empruntée depuis que T_{AB} a été franchie*. Lorsque cette condition est vraie en même temps que `trBC` (T_{BC} est franchie), la sortie `pair` s'évalue à vrai.

Préemptions Avec la couverture que nous venons d'implémenter, lorsque l'on essaye de couvrir la paire de transition allant de `Nominal` à `Failsoft` puis de nouveau à `Nominal` dans l'automate SMRollMode, il nous arrive de générer la séquence de valeurs suivantes pour la sortie `mode` :

Cycle	0	1	2	**3**	4
mode	off	nominal	failsoft	off	nominal

Au cycle 2, la première transition est franchie, produisant `nominal` puis `failsoft`, mais la transition opposée est temporairement préemptée au cycle 3, quand l'automate *parent* se rend dans le mode `off` ; cela est possible car notre observateur s'évalue tout de même à vrai une fois que l'on retourne dans l'état `on`. L'automate parent à la priorité dans l'évaluation des transitions, et comme l'observateur n'est lui-même actif que dans l'état *On*, il ne prend pas en compte les cycles où le système passe par d'autres modes à un plus haut niveau.

Si on veut empêcher ce comportement, on peut remplacer l'expression $Other$ précédente par une expression plus complexe que l'on appelle $AllOther$, définie ci-dessous, qui indique non seulement le franchissement de transitions non désirées dans l'automate courant, comme le faisait $Other$, mais aussi celui d'une transition dans l'automate parent. Soit where $PAct = act(X.parent)$, la variable « `act` » de l'automate parent de X. On note $PStval = st\_val(X.state)$ la valeur associée à l'état, dans cet automate parent, qui abrite l'automate X ; l'état en question dans notre exemple est « *On* » dont la valeur représentative est « `st_On` » dans le langage $\mathcal{L}_+$. On définit alors $AllOther$ ainsi :

$$AllOther = (\texttt{true} \rightarrow *(id(PAct) <> filter(clock(PAct), PStval)); \ldots) \ Other)$$

À l'aide de l'opérateur de réinitialisation, on considère que le paramètre `other` passé à `transpair` est vrai aussi quand le système sort de l'état qui abrite l'automate X. Les points de suspension (…) dans l'équation représentent le fait qu'il peut y avoir encore d'autres niveaux supérieurs prioritaires pouvant interrompre l'observation du critère. Pour s'assurer qu'aucun changement d'état de haut-niveau n'a lieu lors de la couverture du critère dans l'automate X, il faut ainsi écrire autant de conditions de réinitialisation qu'il y a d'automates ancêtres dans la hiérarchie.

Avec la modification que l'on vient d'apporter, la flèche peut-être réinitialisée dès que l'état actif parent sort du mode M qui contient X. Si on retourne par la suite dans ce mode M, le redémarrage force l'évaluation à `true` de l'argument `other`, qui invalide l'observation du critère. Dans l'exemple qui nous intéresse, GATeL génère systématiquement la séquence suivante, où les deux transitions choisies sont tirées à des cycles de calculs consécutifs :

Cycle	0	1	2	3
`mode`	`off`	`nominal`	`failsoft`	`nominal`

En résumé, le fait de passer $Other$ ou bien $AllOther$ au nœud `transpair` nous donne deux variantes différentes du critère *transition-pair*, qui permettent ou non aux automates parents de changer de mode actif lors de la couverture successives des transitions.

5.2.3.8 Couverture de toutes les paires de transitions

Le critère précédent est généralisé à l'ensemble de toutes les paires de transitions présentes au sein d'un automate. Soit $A \in \mathcal{C}_{\mathtt{sm}}$ un automate. Nous réunissons l'ensemble des paires entrantes et sortantes, pour chaque état, dans l'ensemble $\mathcal{T} = \{P_1, \ldots, P_n\}$ de couples de transitions. Chaque paire $P_i = (T_{i,1}, T_{i,2})$, avec $1 \leq i \leq n$, compte alors pour un cas de test dans la directive `split` suivante :

$$
\begin{aligned}
all\_tr\_pair(\mathcal{T}) \ = \ & \texttt{split} \ \ id(sel(A)) \ \ \texttt{with} \\
& [\ \ tr\_pair(T_{1,1}, T_{1,2}) \ \ ; \\
& \ \ \ldots \\
& \ \ \ tr\_pair(T_{n,1}, T_{n,2}) \ \ ]
\end{aligned}
$$

5.2.3.9 All-configurations coverage

Dans SCADE 6, les automates peuvent faire appels à des contrôleurs imbriqués et/ou parallèles. L'objectif du critère *all-configurations* est de couvrir l'ensemble des combinaisons possibles de modes simulaténment actifs, dans un nœud ou un automate. Pour illustrer ce critère, nous ajoutons en parallèle à l'automate SMRollMode un deuxième automate que l'on appelle SMCount, représenté à la figure 5.8. Les configurations de l'automate SMRollMode seul sont les états *Off*, *Nominal* et *Failsoft*, que nous désignons par les lettres O, N et F respectivement par la suite ; l'ensemble des configurations de l'automate est : $\{O, N, F\}$.

De même, l'ensemble des configurations de SMCount est $\{z, o, t\}$, où les lettres représentent les états *Zero*, *One* et *Two*. Alors, les configurations du système tout entier combinant ces deux automates en parallèle est donné par l'ensemble des couples suivants :

$$
\left\{ \begin{array}{c} (O,z),(O,o),(O,t),(N,z),(N,o), \\ (N,t),(F,z),(F,o),(F,t) \end{array} \right\}
$$

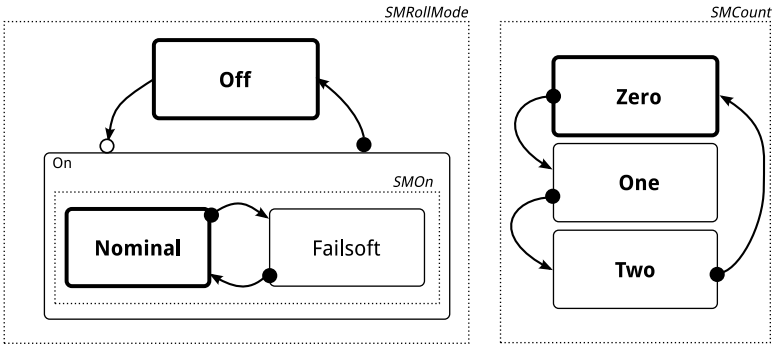


FIGURE 5.8: Automates SMRollMode et SMCount

Autrement dit, un ensemble de configurations est construit à partir des sous-ensembles de configurations agissant en parallèle, par un produit cartésien. L'ensemble final contient toutes les configurations de l'automate, où chaque élément est un tuple des états actifs présents simultanément. Nous obtenons cet ensemble par une énumération récursive des états imbriqués et parallèles du modèle sous-test.

Soit $\mathscr{C} = \{C_1, \ldots, C_n\}$ un ensemble de configuration dans un automate $A \in \mathcal{C}_{\texttt{sm}}$; alors, la couverture de toutes les configurations est exprimé à travers un `split` où chaque cas fonctionnel correspond à la couverture d'une configuration $C \in \mathscr{C}$. La directive est contrôlée par la variable $sel(A)$ de l'automate considéré :

```
split id(sel(A)) with [ cover_conf(C1) ;
                                ...
                        cover_conf(Cn) ]
```

Chaque cas est une expression construite par *cover_conf*, qui observe la couverture d'une configuration. Soit $C = (e_1, \ldots, e_m)$ un tuple de $\mathscr{C}$ avec m états actifs parallèles. Comme le critère demande de couvrir des états, nous réutilisons la fonction *cover_state* définie pour le critère *all_states*. Cependant, la couverture simultanée des états en parallèles fait intervenir des flots qui ont des horloges différentes les uns des autres : l'horloge de l'expression *cover_state* pour l'état *Failsoft* de SMRollMode n'est pas la même que celle pour l'état *Two* de SMCount. Par conséquent, l'expression suivante, que l'on voudrait utiliser pour définir *cover_conf*, n'est pas nécessairement une expression bien formée vis-à-vis des horloges :

```
cover_state(e1) and ... and cover_state(em)
```

Afin de faire coïncider des flots booléens d'horloges indépendantes, nous exploitons la sémantique asynchrone de l'opérateur de réinitialisation $\rightarrow *$. La fonction *cover_conf* est définie ainsi :

```
cover_conf(e1,...,em) =
        (false -> true)
    and (true ->*(cover_state(e1)) false)
              ...
    and (true ->*(cover_state(em)) false) ;
```

Chaque expression *cover_state* a sa propre expression, mais le flot résultant est cependant lui cadencé sur l'horloge de base du nœud englobant.

Avec cette définition, lorsque la sélection du `split` a lieu, chaque sous-cas de test doit couvrir une configuration à un cycle c donné. La sous-expression correspondante, donnée par *cover_conf*, est propagée avec une valeur *true* au cycle c, ce qui ensuite force chaque sous-terme de la conjonction à valoir *true* à son tour. Le premier terme,

« `false → true` », ne peut-être vrai que si c n'est pas le cycle initial de l'instance de nœud. *On sait donc que c n'est pas le cycle initial.*

Les expressions suivantes sont toutes traitées de manière identique : une expression de réinitialisation R est portée par `true →* (R) false`, dont l'évaluation doit valoir vrai au cycle c. Or, cela n'est possible que si : (i) c est le cycle initial, ce qui contredit l'observation précédente, ou (ii) l'expression R est elle aussi vraie entre le cycle c courant et le dernier cycle (exclu) où l'horloge de l'expression réinitialisée était présente. Étant donnée que cette horloge est l'horloge de base, le seul cycle auquel R peut-être vrai est le cycle c. La propagation fait donc en sorte que R soit vraie au cycle c, en activant au passage ses horloges ancêtres.

5.2.4 Implémentation

En pratique, la méthode se greffe autour GATeL et nous permet d'expérimenter des critères de couvertures, en tirant parti de la traduction des automates en flots temporisés et de la traduction des flots Lustre en contraintes.

Nous instrumentons le modèle issu de la traduction en fournissant des emplacements prédéfinis où des équations Lustre peuvent être insérées : chaque critère de test est paramétrable et peut introduire des déclarations globales (des types ou des nœuds auxiliaires), des déclarations de variables locales et leur équations de définition respectives, ou encore de directives de sélection d'objectif de tests propres à GATeL.

Nous insérons pour cela des balises PHP [4] dans le programme original (langage $\mathcal{C}_+^g$), à des endroits prédéfinis, facilement reconnaissables (au début du fichier, dans l'en-tête et à la fin du corps d'un nœud Lustre donné). Un critère est alors un ensemble de fonctions PHP qui injectent des flots intermédiaires. À l'aide des fonctions de correspondance entre modèles, un critère peut faire référence de manière générique aux flots qui encodent l'automate dans le modèle $\mathcal{L}_+$. Ainsi, une fois interprété par PHP, les balises sont remplacées par du nouveau code Lustre en fonction du critère choisi et des paramètres fournis.

Notre approche ici est de pouvoir expérimenter différents critères sans modifier directement le noyau de GATeL. Elle repose sur une chaîne d'outils existants, ce qui m'a permis de réaliser un prototype facilement pour tester l'approche et définir plusieurs critères de tests. Cela n'empêche pas d'intégrer plus étroitement certains critères à l'outil par la suite, si nécessaire. De plus, l'instrumentation peut se mêler à des objectifs fonctionnels et être exploitée par la sélection dynamique de cas de tests.

5.2.5 Mesures de couverture

Nous avons testé les critères sur les exemples fournis dans la plate-forme Scade pour valider la couverture de critères pour automates. De plus, pour les mêmes modèles qu'à

4. PHP : Hypertext Preprocessor

la section §5.1.6 (p. 195), nous avons cherché à couvrir toutes les combinaisons de sorties, toutes les transitions et les paires de transitions (voir §5.2.1.3). Les critères ont été traduits à l'aide d'objectifs de tests et par sélection de cas de tests, de manière propre pour chaque version de *CruiseControl*. La figure 5.9 montre les temps de génération pour ces trois couvertures et pour les trois versions du modèle.

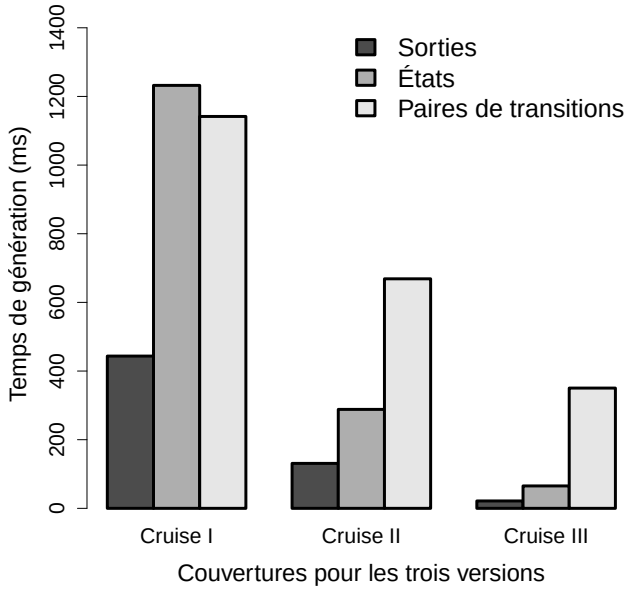


FIGURE 5.9: Comparaison du temps de génération pour différents critères de couvertures selon le modèle sous test.

Nous avons des temps de génération plus court avec la nouvelle version de GATeL et l'encodage à base d'horloge, c'est-à-dire dans le cas de Cruise III.

5.3 Conclusion

Nous avons présenté la génération de séquences de test pour automates de modes de SCADE 6 et la couverture des automates selon des critères structurels.

Génération de tests La génération de séquences de tests sur la traduction des modèles SCADE 6 dans le langage $\mathcal{L}_+$, grâce à un compilateur existant. Ce langage cible est connu de GATeL et nous pouvons ainsi générer les tests désirés à partir du modèle traduit.

En principe, il y aurait un intérêt à prendre une plus grande partie du langage SCADE 6 dans GATeL, car il y a à ce niveau d'abstraction des relations que l'on ne retrouve

pas directement après traduction. Cependant, le langage est principalement un cadre structurant autour d'une utilisation intensive des horloges : les améliorations que l'on a obtenues dans la génération de séquences de tests proviennent avant tout du traitement interne des horloges.

Dans notre approche, nous avons accès aussi en partie au modèle original et pouvons la transformer facilement en une base de faits interprétable dans GATeL. La base de faits nous donne la structure des automates avant traduction : nous pouvons alors renforcer les contraintes du modèle en nous basant sur les structures de haut-niveau, comme dans le cas de la contrainte *automaton*. Cette méthode est plus simple à mettre en œuvre que de celle consistant à modifier GATeL pour traiter directement le langage SCADE 6. Nous dépendons alors d'un compilateur et du format de données de la plate-forme SCADE. Cependant, nous ne traitons que partiellement le format de données. C'est pourquoi une intégration plus étroite avec la plate-forme nous permettrait d'avoir un accès direct au modèle et ainsi traiter plus facilement les phases de traductions.

L'analyse des modèles originaux SCADE 6 et la construction des fonctions de correspondance nous permettent de fournir une meilleure traçabilité des opérations effectuées dans GATeL auprès de l'utilisateur, pour qui le modèle sous test est un automate de mode et non des équations de flots.

Couverture structurelle Par ailleurs, nous avons adapté pour SCADE 6 un ensemble de critères de couverture structurelle. Nous avons montré comment ceux-ci peuvent être couverts dans GATeL : nous traduisons les critères en des expressions observables dans le modèle, et celles-ci contribuent alors à la définition d'un objectif de test ou d'un dépliage fonctionnel.

Nous avons alors implémenté des outils qui servent à instrumenter automatiquement un modèle de $\mathcal{L}_+$ issus de modèles SCADE 6 à l'aide des critères de couvertures. La définition d'un critère fonctionnel passe par la traduction de celui-ci en un flot observable. Lors de la définition, les éléments couverts sont accédés par le biais des fonctions de correspondances. Chaque critère est définie en fonction de ces accesseurs et peut-être appliqué à des modèles quelconques.

Conclusion

Dans ce manuscrit, j'ai présenté mes travaux sur la modification de l'outil de génération de tests GATeL pour les systèmes réactifs synchrones. Ces travaux visaient à intégrer des concepts d'horloges et d'automates pour pouvoir générer efficacement des séquences de test à partir de modèles SCADE 6. Je reviens dans un premier temps sur les contributions de la thèse, pour donner ensuite des perspectives possibles.

Contributions

Les deux principales contributions de la thèse se répartissent, d'une part, en la refonte du noyau de GATeL pour la prise en charge des flots temporisés, et d'autre part, en la gestion des automates et notamment la définition de critères de tests.

Génération de séquences de test temporisées

J'ai repris le noyau de génération de tests existant en le modifiant de manière significative pour traiter les flots temporisés selon l'approche paresseuse originale. Dans un premier temps, les horloges traitées étaient uniquement celles assimilables à des flots booléens. Par la suite, les types énumérés sont venus s'ajouter au langage, pour redéfinir le filtrage et la fusion de flots (`when` et `merge`).

Les horloges de LUSTRE rendent plus complexes certains détails de la génération de séquences de tests, comme l'introduction des invariants portés par les `assert`, les types énumérés, les appels de nœuds ou encore l'instanciation des valeurs qui définissent les horloges (celles-ci doivent rester cohérentes les unes par rapport aux autres). Il fallait ainsi modifier complètement la propagation de certains opérateurs, notamment le retard unitaire et l'initialisation.

J'ai présenté la sémantique opérationnelle de GATeL, à la fois l'ancienne version et la nouvelle. Pour cela, j'ai défini au premier chapitre une notation adaptée au fonctionnement de GATeL. Celle-ci repose sur des règles logiques dont certaines définissent l'évolution de l'ensemble de contraintes à travers des étapes de réecriture. Cela m'a permis de décrire à la fois les contraintes manipulées mais aussi les fonctions intermédiaires comme la sur-approximation d'expressions qui rend effective l'approche paresseuse.

J'ai aussi présenté la compilation d'un système réactif en un arbre syntaxique abstrait, modélisé à l'aide de termes clos du langage de contraintes (pour les deux versions de Lustre présentées). Un tel arbre syntaxique constitue une base de faits accessible lors de la propagation de contraintes, ce qui est nécessaire pour explorer les équations de flots et les convertir à la volée en des contraintes atomiques.

La nouvelle version passe par l'implémentation des contraintes d'horloges dans le système, qui offrent un moyen supplémentaire de faire circuler des réductions de domaines à travers un CSP. En particulier, j'ai défini des règles de filtrage qui projettent ces contraintes des flots vers les statuts liés aux horloges et inversement, en prenant en compte aussi les relations de parenté qui existent entre les horloges.

La démarche présentée repose sur la sémantique multi-horloges définie au chapitre 2. Il s'agit à la fois d'une version simplifiée de Lustre, où il n'y a ni nombre réels ni appels de nœuds, et d'une extension décrivant des nouveaux opérateurs temporels.

La présentation du langage a été pensée pour être symétrique de l'approche de génération. C'est pourquoi l'évaluation dans $\mathcal{L}_+$ est définie dans le contexte d'une grille de valeurs plus ou moins complète : nous générons une telle grille durant la résolution de contraintes. Il est alors facile de convertir le résultat de la génération de séquences de tests pour vérifier par évaluation si la solution trouvée est bien une séquence acceptable.

Il est donc possible de générer des séquences de tests à partir de modèles multi-horloges, en traitant directement les contraintes d'horloges dans le noyau de l'outil. Cela donne de meilleurs résultats qu'en simulant la sémantique de $\mathcal{L}_+$ dans le langage mono-horloge $\mathcal{L}_1$ et m'a permis de traiter directement les automates de modes.

Prise en charge des automates de mode

Le langage Scade 6 introduit les automates de modes dans le langage formel Lustre. Il combine dans un langage unique les machines à états finis et les flots de données. Il se trouve que le langage est défini comme une extension d'un langage multi-horloges intermédiaire, le langage $\mathcal{L}_+$. Comme nous avons accès à la traduction préliminaire de Scade 6 vers $\mathcal{L}_+$, la génération de séquences de tests pour automates peut passer par la traduction du modèle sous test dans $\mathcal{L}_+$.

Un premier avantage de l'approche est qu'elle m'a permis d'intégrer l'ensemble du langage Scade 6 dans GATeL$_+$ à travers le format intermédiaire. Par ailleurs, si le langage Scade évolue par la suite, on peut espérer qu'il sera toujours basé sur des horloges et en particulier le langage $\mathcal{L}_+$, ce qui évite d'être trop dépendant des versions du langage de haut-niveau.

J'ai présenté une contrainte supplémentaire qui lie entre elles les variables qui encodent les états d'un automate, de sorte à renforcer des invariants présents à l'origine dans le modèle de haut-niveau. Plus précisément, chaque automate est encodé à l'aide de trois flots : l'état sélectionné, l'état actif, et le prochain état sélectionné. À un cycle donné, les trois flots définissent un triplet de valeurs, telles que deux de ces valeurs sont

nécessairement égales (l'état actif est égal soit à l'état sélectionné soit au prochain état). Cela est dû à la sémantique des transitions. En introduisant l'invariant au lieu de se reposer uniquement sur la traduction, celui-ci est plus facilement propagé.

Pour ajouter cette contrainte dans le problème sous test, il est nécessaire : (1) d'avoir accès au modèle original SCADE 6 et (2) de savoir quelles sont les variables qui encodent les automates de ce modèle après traduction.

Pour le premier point, je me suis reposé sur le format de données dans lequel sont sauvegardés les modèles SCADE 6. Concrètement, l'abre syntaxique du système est encodé dans un arbre XML. Il suffit donc de réaliser une transformation du modèle vers Prolog pour obtenir une base de faits qui reflète les informations portées par le modèle sous test. Avec cette approche, j'ai obtenu rapidement un prototype de lecture de modèles SCADE 6. Il s'agit d'un compromis entre la difficulté liée au développement et à la maintenance d'un analyseur complet pour SCADE 6 et la perte d'informations liées à la traduction.

J'ai ensuite développé un prototype qui repose sur le modèle de haut-niveau et le modèle traduit pour compiler les relations entre l'un et l'autre. Ainsi, on peut retrouver les variables d'états encodant un automate donné. À l'aide de ces informations, on peut ajouter quand nécessaire les contraintes portant sur les automates, si on a accès au modèle traduit et au modèle original tel que donné par l'éditeur SCADE.

Par ailleurs, j'ai utilisé cette base d'informations pour deux autres aspects, à savoir : (1) produire une interface graphique qui reflète, dans GATeL, le modèle original à base d'automates et (2) mettre en place l'instrumentation du modèle traduit pour observer des critères de couvertures propres aux états et aux transitions.

Ces derniers sont des critères structurels pour les automates de modes, que j'ai adapté d'une famille de critères qui existait déjà pour STATECHARTS. Les critères de couverture définissent des classes de jeux de tests que l'on souhaite générer, comme la couverture de tous les états. La couverture d'un modèle selon un critère peut être rendue observable en ajoutant des variables supplémentaires au modèle : par exemple, la couverture d'un état actif E donné est observée quand le flot qui encode l'état actif prend la valeur correspondant à E.

Ainsi, j'ai proposé une traduction des différents critères sous forme de flots observateurs. Ces flots sont ajoutés au modèle dans une phase d'instrumentation de code. L'avantage de passer par ces flots supplémentaires est qu'ils peuvent contribuer à l'objectif de test : en forçant l'observation d'un de ces flots à un cycle, on ajoute indirectement les contraintes nécessaires pour que la séquence générée, si on en trouve une, couvre un critère particulier.

La traduction des critères exploite le langage $\mathcal{L}_+$ pour encoder efficacement l'observation des critères. J'ai développé une chaîne d'outils qui permet au testeur de sélectionner un critère (e.g. couverture d'un état) et ses paramètres (e.g. l'état en question), et qui instrumente le programme sous test pour y introduire les flots observateurs propres

à ce critère. Alors, une fois analysé dans GATeL$_+$, le critère est sélectionnable et peut guider la génération de séquences de tests.

Perspectives

Étude et amélioration des performances

Bien qu'il y ait eu des tests et des mesures pour valider la nouvelle version de GATeL, il serait intéressant de voir à quel moment les performances de la génération de tests commencent à se dégrader. On pourrait ainsi réaliser des mesures de performance en variant le nombre d'états, de transitions, de niveaux d'imbrications, de signaux, etc. d'un modèle. En particulier, il serait utile de tester des modèles complexes issus de cas industriels.

Cela permettrait aussi de mieux mesurer les problèmes éventuels liés à la traduction depuis SCADE 6 vers $\mathcal{L}_+$. En effet, il y a encore des invariants de haut-niveau non exploités à traduire sous forme de contraintes pour faciliter le traitement des automates. De plus, il est probable que pour des modèles complexes, la traduction produise de nombreux flots et types énumérés pour encoder la sémantique de SCADE 6 dans $\mathcal{L}_+$: cela pourrait gêner la compilation au niveau de GATeL$_+$.

D'autre part, une étude plus approfondie des cas industriels réels permettrait de raffiner les critères de tests présentés. J'ai adapté des critères de tests pour SCADE 6 comme la couverture d'états, de transitions ou de configurations, qui prennent en compte indirectement les différents types de transitions, les états finaux, les signaux, …

Ces critères pourraient être complétés par des critères plus complexes, comme la couverture de chemins dans le graphe d'états/transitions, plus adaptés aux difficultés liées au test de modèles complexes.

Améliorations techniques et intégration

Recompilation des modèles

Le fait de pouvoir spécifier les objectifs de tests et les dépliages fonctionnels dans le même langage que celui du système amène naturellement à un mécanisme d'annotations du système sous test avec les paramétrages de ce test. C'est ainsi que procède GATeL$_+$, en laissant certains commentaires avoir une sémantique interprétable par l'outil.

Or, un des problèmes de performance (et d'utilisabilité) potentiels est la nécessité de recompiler le modèle depuis le langage $\mathcal{L}_+^g$ non seulement quand le système réactif est modifié, mais aussi dès que l'on souhaite changer un des aspects du test, comme par exemple :

- un invariant (`assert`),
- l'objectif de test (`reach`),

- une directive de dépliage fonctionnel (`split`),
- un paramètre de la génération [5] (e.g. taille des séquences).

L'implémentation de la couverture des critères structurels pour automates passe par l'exploitation de ce mécanisme d'annotations. Cela implique de devoir recompiler le modèle à chaque fois qu'on souhaite prendre en compte ou non un critère de couverture (une fois compilé, le critère peut ou non être appliqué selon les actions de l'utilisateur).

Il serait intéressant de pouvoir compiler des expressions isolées, pour modifier seulement un sous-ensemble du système sous test. Si cela était réalisé, les critères seraient exprimés en Lustre puis compilés en arrière-plan pour s'intégrer dans la forme interne traitée par GATeL$_+$. Un tel changement serait alors utile aussi pour modifier un objectif de test ou une directive de dépliage fonctionnel depuis l'interface graphique, qui pourrait proposer l'édition de certaines parties du modèle sous test (les directives, les invariants, …).

Il existe déjà un mécanisme de cache qui raccourcit le temps de compilation d'un modèle et d'un nœud Lustre défini dans ce dernier. Il serait intéressant de poursuivre l'optimisation de la phase de compilation en limitant les traitements réalisés d'une version à une autre d'un modèle sous test. Il s'agit d'un problème difficile à mettre en œuvre.

Interface pour les automates

Pour faciliter l'utilisation de GATeL$_+$ lors du test d'automates de modes, il est important d'offrir au testeur un moyen d'agir au niveau des éléments Scade 6 et non au niveau du modèle intermédiaire, qui lui est étranger. De la même manière que GATeL$_1$ permet de sélectionner des cas de tests à travers des éléments d'interface pour les flots de données, on peut envisager d'avoir une interface propre aux automates.

L'importation des structures de Scade 6 dans GATeL$_+$ accède aussi aux différents positionnements graphiques des états et des transitions tels qu'ils sont mis en place dans la plate-forme Scade. Avec ces informations, on peut produire une interface graphique dans l'outil.

L'interface graphique peut-être utilisée pour représenter l'état d'un automate à un cycle de calcul, pendant la propagation de contraintes ou l'évaluation en avant (simulation). Elle peut aussi servir à la sélection de cas de tests : lorsqu'un état cible peut provenir de plusieurs états sources potentiels, on peut déplier le cas de test courant en autant d'états antérieurs. D'autre part, lors de la sélection d'un critère de couverture, il est nécessaire de définir quels éléments on souhaite couvrir. Cela pourrait se passer au niveau de l'interface dédiée aux automates.

5. Au passage, on peut remarquer qu'il n'est pas possible de lier des paramètres préférés à un nœud dans un modèle, à travers une directive, bien qu'en pratique un même système sous test admette souvent des paramètrages propres.

Intégration avec SCADE

Pour réaliser l'outil actuel qui lit les modèles SCADE 6 et les compile en une base de faits dans GATeL$_+$, j'ai inféré la syntaxe du modèle à partir des exemples que j'avais à ma disposition. Il serait plus simple d'avoir recours aux outils existants dans SCADE 6, que ce soit par le mécanisme de greffons existant, l'appel à un des outils de compilation de SCADE ou simplement en connaissant le format exact manipulé par l'éditeur. Ce travail d'intégration peut nécessiter cependant une bonne connaissance à la fois de la plate-forme SCADE et de GATeL$_+$.

Bibliographie

[A+96] C. André et al. Representation and analysis of reactive behaviors : A synchronous approach. In *Proc. CESA*, volume 96. Citeseer, 1996. vii

[AAB+06] M. Aiguier, A. Arnould, C. Boin, P. Le Gall, and B. Marre. Testing from algebraic specifications : test data set selection by unfolding axioms. *Formal Approaches to Software Testing*, pages 203–217, 2006. x

[AALGL09] M. Aiguier, A. Arnould, P. Le Gall, and D. Longuet. Exhaustive test sets for algebraic specification correctness. *Rapport de recherche, Université Évry-Val d'Essonne*, 2009. x

[AB00] C. ANDRÉ and H. BOUFA
"IED. Execution machine for synchronous languages. *IDPT-2000 (Integrated Design and Process Technology), Dallas (TX)*, 2000. vi

[AD97] L. Apfelbaum and J. Doyle. Model based testing. In *Software Quality Week Conference*, pages 296–300. Citeseer, 1997. ix

[ADS+06] P. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using scade. *Leveraging Applications of Formal Methods*, pages 115–129, 2006. xi

[ALGM96] A. Arnould, P. Le Gall, and B. Marre. Dynamic testing from bounded data type specifications. *Dependable Computing—EDCC-2*, pages 283–302, 1996. x

[And03] C. André. Semantics of ssm (safe state machine). *Esterel Technologies*, 2003. viii

[And05] C. André. Comparaison des styles de programmation de langages synchrones. Technical report, CNRS, UNSA, INRIA, june 2005. Projet AOSTE. vi

[Apt98] Krzysztof R. Apt. A proof theoretic view of constraint programming. *CoRR*, cs.AI/9810018, 1998. 2

[AW75] E.A. Ashcroft and W.W. Wadge. Demystifying program proving : an informal introduction to lucid. Technical report, University of Waterloo, feb 1975. vi

[AW76] E.A. Ashcroft and W.W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3) :336–354, 1976. vi

[Bar01] R. Bartak. Theory and practice of constraint propagation. *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control (CPDC 2001)*, pages 7–14, 2001. 1

[BBD+09] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams. Constraint-Based Software Testing. *1ères journées nationales du GDR-GPL, groupe de travail MTVV, Toulouse, France*, jan 2009. ix, 1

[BBLG97] G. Bernot, L. Bouaziz, and P. Le Gall. A theory of probabilistic functional testing. 1997. ix

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1) :64–83, 2003. 28

[BCH+85] JL Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real time data flow language. 1985. vi

[BCHP08] D. Biernacki, J.L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130. ACM, 2008. viii

[BDK+09] B. Botella, M. Delahaye, N. Kosmatov, P. Mouy, M. Roger, and N. Williams. Automating structural testing of C programs : Experience with PathCrawler. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 70–78. IEEE, 2009. x

[BDL+06] B. Blanc, G. Durrieu, A. Lakehal, O. Laurent, B. Marre, I. Parissis, C. Seguin, and V. Wiels. Automated functional test case generation from data flow specifications using structural coverage criteria. In *3rd European Congress on Embedded Real Time Software (ERTS2006), Toulouse, France*. Citeseer, 2006. x

[Ber86] J.L. Bergerand. *LUSTRE : un langage déclaratif pour le temps réel*. PhD thesis, Institut National Polytechnique de Grenoble, 1986. vi

[Ber89] G. Berry. Real-time programming : General purpose or special-purpose languages. *Information Processing*, 89 :11–17, 1989. 27

[Ber00] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction*, pages 425–454, 2000. vi, 27, 28

[BFG+03] C. Bigot, A. Faivre, J.P. Gallois, A. Lapitre, D. Lugato, J.Y. Pierron, and N. Rapin. Automatic test generation with AGATHA. *Tools and*

Algorithms for the Construction and Analysis of Systems, pages 591–596, 2003. x

[BGM91a] G. Bernot, M.C. Gaudel, and B. Marre. A Formal Approach to Software Testing. In *Proceedings of the Second International Conference on Methodology and Software Technology : Algebraic Methodology and Software Technology*, pages 243–253. Springer-Verlag, 1991. x

[BGM91b] G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405, 1991. x

[BGM+02] B. Botella, A. Gotlieb, C. Michel, M. Rueher, and P. Taillibert. Utilisation des contraintes pour la generation automatique de cas de tests structurels. 2002. ix

[BH08] S. Bardin and P. Herrmann. Structural testing of executables. In *ICST*, pages 22–31. IEEE Computer Society, 2008. x

[BH09] S. Bardin and P. Herrmann. Pruning the Search Space in Path-based Test Generation. In *ICST 2009*. IEEE Computer Society, 2009. x

[BJM+10] B. Blanc, C. Junke, B. Marre, P. Le Gall, and O. Andrieu. Handling State-Machines Specifications with GATeL. In *Proceedings of the Workshop on Model Based Testing 2010*, 2010. x, 193

[BLG90] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the signal language. *Automatic Control, IEEE Transactions on*, 35(5) :535–546, May 1990. vi

[BLP04] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B : A constraint solver to animate a B specification. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2) :143–157, August 2004. x

[BM05] B. Blanc and B. Marre. Test Selection Strategies for Lustre Descriptions in GATeL. *ENTCS*, 111 :93–111, January 2005. x

[BPP99] V. Bertin, M. Poizé, and J. Poulou. Une nouvelle méthode de compilation pour le langage Esterel. *Proceedings GRAISyHM-AAA, Lille France, March*, 1999. vi

[BRG+01] J.R. Beauvais, E. Rutten, T. Gautier, R. Houdebine, et al. Modeling Statecharts and Activitycharts as Signal equations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(4) :397–451, 2001. viii

[BT01] E. Brinksma and J. Tretmans. Testing transition systems : An annotated bibliography. *Modeling and verification of parallel processes*, pages 187–195, 2001. vi, 199

[Bur99] S Burton. Towards automated unit testing of statechart implementations. Technical report, 1999. 199

[CBG09] F. Charreteur, B. Botella, and A. Gotlieb. Modelling dynamic memory management in constraint-based testing. *Journal of Systems and Software*, 82(11) :1755–1766, 2009. ix

[CGHP04] J.L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international Conference on Embedded Software*, page 239. ACM, 2004. 34

[CGP+08] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE : automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2) :1–38, 2008. x

[Cla76] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, pages 215–222, 1976. ix

[Cod95] P. Codognet. Programmation logique avec contraintes : une introduction. *Technique et Sciences Informatiques*, 14(6), 1995. 2

[CP96] P. Caspi and M. Pouzet. Synchronous kahn networks. In *ACM SIGPLAN Notices*, volume 31, pages 226–238. ACM, 1996. 34

[CP03] J.L. Colaco and M. Pouzet. Clocks as first class abstract types. In *Embedded software : third international conference, EMSOFT 2003, Philadelphia, PA, USA, October 13-15, 2003 : proceedings*, page 134. Springer Verlag, 2003. viii

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A Declarative Language for Programming Synchronous Systems. In *POPL*, pages 178–188, 1987. vi, 28, 34

[CPP05a] J.L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. *Proceedings of the 5th ACM international conference on Embedded software*, pages 173–182, 2005. viii, 34

[CPP05b] J. L. Colaço, B. Pagano, and M. Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *EMSOFT*, pages 173–182, 2005. 185

[CR96] A. Colmerauer and P. Roussel. The birth of prolog. In *History of programming languages—II*, pages 331–367. ACM, 1996. 1

[CS94] D. Carrington and P. Stocks. A tale of two paradigms : Formal methods and software testing. In *Z user workshop, Cambridge*, pages 51–68. Citeseer, 1994. ix

[Dio04] B. Dion. Correct-by-construction methods for the development of safety-critical applications. 2004. viii

[DJK+99] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. 1999. ix

[DO91] R.A. DeMilli and A.J. Offutt. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on*, 17(9) :900–910, 1991. ix

[Dor08] F.-X. Dormoy. Scade 6 : a model based solution for safety critical software development. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'08)*, pages 1–9, 2008. viii

[dSMFM00] S.R.S. de Souza, J.C. Maldonado, S.C.P.F. Fabbri, and P.C. Masiero. Statecharts specifications : A family of coverage testing criteria. In *XXVI Conferência Latinoamericana de Informática*. Citeseer, 2000. 199, 201

[Edv99] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999. ix

[EFW01] I.K. El-Far and J.A. Whittaker. Model-based software testing. *Encyclopedia of Software Engineering*, 2001. ix

[Fag96] F. Fages. *Programmation logique par contraintes*. Ellipses, 1996. 1, 2

[Gau01] M.C. Gaudel. Testing from formal specifications, a generic approach. *Reliable SoftwareTechnologies—Ada-Europe 2001*, pages 35–48, 2001. x

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *ACM SIGSOFT Software Engineering Notes*, 23(2) :53–62, 1998. ix

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. *Computational Logic—CL 2000*, pages 399–413, 2000. ix, 1

[GBW06] A. Gotlieb, B. Botella, and M. Watel. Inka : Ten years after the first ideas. In *19th International Conference on Software and Systems Engineering and their Applications (ICSSEA'06)*, 2006. ix

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005. x

[GLL08] A. Gotlieb, N. Lazaar, and Y. Lebbah. Towards Constraing-Based Local Search for Automatic Test Data Generation. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, page 195. IEEE, 2008. ix

[Got09] A. Gotlieb. Euclide : A constraint-based testing framework for critical C programs. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 151–160. IEEE, 2009. ix

[GPSV+] A. Girault, R.D. Powell, A. Sangiovanni-Vincentelli, J.B. Stéfani, E.A. Benveniste, É. Closse, and A. Schiper. Alain GIRAULT, Habilitation à Diriger des Recherches. viii

[GRMB04] J. Gassino, P. Régnier, B. Marre, and B. Blanc. Criteria and Associated Tool for Functional Test Coverage of Safety Critical Software. In *Proceedings of 4th ANS International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC HMIT 2004)*, number 700309, Columbus, USA, September 2004. ANS. x

[Hal98] N. Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16. Springer, 1998. vi, 28

[Hal05] N. Halbwachs. A synchronous language at work : the story of Lustre. In *Formal Methods and Models for Co-Design, 2005. MEMOCODE'05. Proceedings. Third ACM and IEEE International Conference on*, pages 3–11. IEEE, 2005. vi, 28

[Ham94] D. Hamlet. Foundations of software testing : dependability theory. *ACM SIGSOFT Software Engineering Notes*, 19(5) :128–139, 1994. ix

[Har84] D Harel. Statecharts : A Visual Approach to Complex Systems. Technical Report CS84-05, Weizmann Institute of Science, feb 1984. vii

[Har87] David Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of computer programming*, 8(3) :231–274, 1987. vii

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5) :514–530, 1988. vii

[Hea03] S. Heath. *Embedded systems design*. Newnes, 2003. 27

[HGdR88] C. Huizing, R. Gerth, and W. de Roever. Modelling statecharts behaviour in a fully abstract way. *CAAP'88*, pages 271–294, 1988. vii

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *Software Engineering, IEEE Transactions on*, 18(9) :785–793, Sep 1992. xi

[HLSC01] H.S. Hong, I. Lee, O. Sokolsky, and S.D. Cha. Automatic test generation from statecharts using model checking. In *Proceedings of the First Workshop on Formal Approaches to Testing of Software*, pages 15–30. Citeseer, 2001. 199

[HMG06] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1) :79–95, 2006. ix

[HP85] D. Harel and A. Pnueli. *On the development of reactive systems*. Weizmann Institute of Science, Dept. of Computer Science, 1985. 27

[HP95] H.M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4) :309–327, 1995. ix

[HP00] G. Hamon and M. Pouzet. Modular resetting of synchronous data-flow programs. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 289–300. ACM, 2000. 40

[HR04] G. Hamon and J. Rushby. An operational semantics for stateflow. *Fundamental Approaches to Software Engineering*, pages 229–243, 2004. vii

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. 28

[Inc87] D.C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1) :63, 1987. ix

[JB09] C. Junke and B. Blanc. CSP dynamiques pour la génération de tests de systèmes réactifs. *Journées Francophones de Programmation par Contraintes*, 2009. x, 8

[JH05] A. Joshi and M.P.E. Heimdahl. Model-based safety analysis of simulink models using scade design verifier. *Computer Safety, Reliability, and Security*, pages 122–135, 2005. xi

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987. 1

[JL06] E. Jaffuel and B. Legeard. LEIRIOS test generator : Automated test generation from B models. *B 2007 : Formal Specification and Development in B*, pages 277–280, 2006. x

[JLMR94] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, pages 211–218. IEEE, 1994. viii

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming : A survey. *The Journal of Logic Programming*, 19 :503–581, 1994. 1

[JRLN07] E. Jahier, P. Raymond, D. Lesens, and X. Nicollin. Virtual execution of AADL models via a translation into synchronous programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 134–143. ACM Press New York, NY, USA, 2007. viii

[JS05] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. In *Computers and Digital Techniques, IEEE Proceedings-*, volume 152, pages 114–129. IET, 2005. 28

[Kin76] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7) :385–394, 1976. ix

[Knu68] D.E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2) :127–145, 1968. 80

[KoEDoCL73] R. Kowalski and University of Edinburgh. Department of Computational Logic. *Predicate logic as programming language*. Edinburgh University, 1973. 1

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8) :870–879, 1990. ix

[Kow88] R.A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1) :38–43, 1988. 1

[Lay93] G.T. Laycock. *The theory and practice of specification based software testing*. Citeseer, 1993. ix

[LDB05] O. Labbani, J.L. Dekeyser, and P. Boulet. Mode-automata based methodology for scade. *Hybrid Systems : Computation and Control*, pages 386–401, 2005. viii

[LGA96] P. Le Gall and A. Arnould. Formal specifications and test : Correctness and oracle. *Recent Trends in Data Type Specification*, pages 342–358, 1996. ix

[LKYC02] S. Lee, M. Kim, Y. Youm, and W. Chung. Control of a car-like mobile robot for parking problem. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 1–6. IEEE, 2002. v

[LP09] A. Lakehal and I. Parissis. Structural coverage criteria for LUSTRE/SCADE programs. *Software Testing, Verification and Reliability*, 19(2) :133–154, 2009. 200

[LY96] D Lee and M Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings Of The IEEE*, 84 :1090–1123, 1996. 199

[MA00] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, volume 0, page 229, Los Alamitos, CA, USA, 2000. IEEE Computer Society. x, 1

[Mar91] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. In *Logic Programming : Proceedings of the Eighth International Conference*, page 202. The MIT Press, 1991. x

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, pages 550–564. Springer, 1992. vii

[Mar95] B. Marre. Loft : a tool for assisting selection of test data sets from algebraic specifications. *TAPSOFT'95 : Theory and Practice of Software Development*, pages 799–800, 1995. x

[Mat08] A.P. Mathur. *Foundations of Software Testing*. Pearson Education, 2008. ix

[MB05] B. Marre and B. Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111 :93–111, 2005. x

[McM04] P. McMinn. Search-based software test data generation : A survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004. ix

[MH96] F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 72–89. Springer, 1996. viii

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989. v

[MP08] Louis M. and Florence P. Interactive programming of reactive systems. In *Proceedings of Model-driven High-level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008. 27

[MPP10] L. Madani, V. Papailiopoulou, and I. Parissis. Towards a testing methodology for reactive systems : A case study of a landing gear controller. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0 :489–497, 2010. xi

[MR98] F Maraninchi and Y Remond. Mode-automata : About modes and states for reactive systems. In *ESOP*, volume 1381, pages 185–199, 1998. vii

[MR00] F. Maraninchi and Y. Remond. Applying Formal Methods to Industrial Cases : the Language Approach (The Production-Cell and Mode-Automata). In *Proc. of the 5th International Workshop on Formal Methods for Industrial Critical Systems, Berlin*, 2000. vi, vii

[Mye08] G.J. Myers. *The art of software testing*. Wiley-India, 2008. ix

[Nav06] N. Navet, editor. *Systèmes Temps-réel : Techniques de Description et de Vérification–Théorie et Outils, volume 1, chapter Lucid Synchrone, un langage de programmation des systèmes réactifs*, volume 1, chapter 7, pages 217–260. Hermes, 2006. vi

[OLAA03] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1) :25, March 2003. 199, 200

[PBdST05] D. Potop-Butucaru, R. de Simone, and J.P. Talpin. The synchronous hypothesis and synchronous languages. *The Embedded Systems Handbook*, 2005. 28

[PBM+93] J.P. Paris, G. Berry, F. Mignard, P. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, et al. Projet SYNCHRONE : les formats communs des langages synchrones. 1993. viii

[PHR04] G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *Software Tools for Technology Transfer*, 5(2—3), March 2004. ix

[PMDBP09] V. Papailiopoulou, L. Madani, L. Du Bousquet, and I. Parissis. Extending structural test coverage criteria for lustre programs with multi-clock operators. pages 23–36, 2009. xii, 200

[PST03] N. Pernet, Y. Sorel, and O. Team. Optimized implementation of distributed real-time embedded systems mixing control and data processing. In *Proceedings of the ISCA 16th International Conference : Computer Applications in Industry and Engineering (CAINE-2003)*. Citeseer, 2003. viii

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code LUSTRE-V 3.* PhD thesis, Institut National Polytechnique de Grenoble, 1991. 28

[RTFA+24] A. Robisson, P. Thevenod-Fosse, A. Arnould, J. Raguideau, N. Perot-Messaoudi, R. Calippe, B. Marre, B. Berard, M. Bidoit, C. Bodennec, C. Dubois, C. Hote, P. Raymond, and D. Weber. Vérification et génération de tests pour un système de comptage de neutrons. Rapport LAAS 98017, LAAS, 68p., 1998-02-24. x

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium, Madrid, Spain*. Citeseer, 1998. xi

[SA93] S. Schaal and C.G. Atkeson. Open loop stable control strategies for robot juggling. In *IEEE International Conference on Robotics and Automation*, pages 913–913. Citeseer, 1993. v

[Sim01] H. Simonis. Building industrial applications with constraint programming. *Constraints in computational logics*, pages 271–309, 2001. 1

[SMA05] K. Sen, D. Marinov, and G. Agha. CUTE : a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272. ACM New York, NY, USA, 2005. x

[SSC+04] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a safe subset of simulink/stateflow into lustre. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268. ACM, 2004. vii

[Sto93] P.A. Stocks. *Applying formal methods to software testing*. Citeseer, 1993. ix

[Utt06] M. Utting. Position paper : Model-based testing. *Verified Software : Theories, Tools, Experiments (VSTTE)*, 2006. ix

[VH91] P. Van Hentenryck. Constraint logic programming. *Knowledge Engineering Review*, 6(3) :151–194, 1991. 1

[WA85] W.W. Wadge and E.A. Ashcroft. Lucid : The data flow programming language. 1985. vi

[Whi87] L.J. White. Software testing and verification. *Advances in Computers*, 26 :335–391, 1987. ix

[Whi00] J.A. Whittaker. What is software testing ? And why is it so hard ? *IEEE software*, 17(1) :70–79, 2000. ix

[WMM04] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In *19th International Conference on Automated Software Engineering, 2004.*, pages 290–297, 2004. x

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. *Dependable Computing-EDCC 2005*, pages 281–292, 2005. x, 1