SYSTEM@TIC PARIS-REGION

Usine Logicielle

MoDriVal

Module de vérification MOD-Supélec-1

Tâche 2.1.2

Version 1.0

Diffusable

Historique

Date	Version	Description	Auteur
08/11/2007	0.1	Squelette	Christophe Jacquet
09/11/2007	0.2	Structure générale, reprise des parties déjà rédigées	Dominique Marcadet
13/11/2007	0.3	Chapitres sur l'exemple et sur la syntaxe abstraite (première version)	Dominique Marcadet
20/11/2007	0.4	Chapitre sur la syntaxe concrète (première version)	Dominique Marcadet
26/11/2007	0.5	Chapitre sur la vérification de propriétés	Christophe Jacquet
03/12/2007	0.6	Chapitre l'outil, diverses corrections	Dominique Marcadet
04/12/2007	0.7	Outil de vérification	Christophe Jacquet
05/12/2007	1.0	Dernières corrections : version finale	Dominique Marcadet

Auteurs

Auteur	Partenaire	
Frédéric Boulanger	Supélec	Contexte et objectifs
Ahcene Bouzoualegh	Supélec	Interface avec les composants externes
Christophe Jacquet	Supélec	Vérification de propriétés
Dominique Marcadet	Supélec	Exemple, définition du langage, outil

Table des matières

Module de vérification MOD-Supélec-1

1 Introduction

1.1 Sujet

Dans le cadre de MoDriVal, Supélec a développé une approche méthodologique et un outil pour la conception et la construction d'applications à propriétés vérifiables ; ce document présente ces travaux.

1.2 Périmètre

Ce document est le rapport final sur le travail effectué par Supélec dans le cadre de la tâche 2.1.2 ("Développement des outils") du sous-projet MoDriVal du projet Usine Logicielle du pôle de compétitivité SYSTEM@TIC PARIS-REGION. C'est un élément de la fourniture D2.3. Un autre élément de cette fourniture produit par Supélec est l'outil logiciel réalisé. L'outil décrit dans ce document génère des applications pour la plate-forme réalisée dans le sous-projet Inflexion d'Usine Logicielle.

1.3 Définitions, Acronymes et Abréviations

ADLV est l'acronyme utilisé pour nommer les différents éléments constitutifs de nos travaux (méthode, langage, outil...).

1.3.1 Concepts

Modèle de calcul Règles de combinaison des comportements des composants d'un système permettant de calculer le comportement global du système

Système réactif Système qui réagit à son environnement au rythme de cet environnement (par opposition aux systèmes interactifs qui réagissent à leur propre rythme).

1.3.2 Acronymes, Abréviations

ADL Architecture Description Language

ADLV Architecture Description Language for Verification

CCM CORBA Component Model, Container Component Model

IDL Interface Definition Language

LwCCM Lightweight CORBA Component Model

OMG Object management Group

1.4 Références

[CCM] Object Management Group. *CORBA Component Model Specification, version 4.0*, OMG document number formal/06-04-01. Disponible sur http ://www.omg.org/cgi-bin/doc ?ptc/06-04-01.

[CORBA] Object Management Group. *Common Object Request Broker Architecture, version 3.0.3*, OMG document number formal/04-03-01. Disponible sur http ://www.omg.org/cgi-bin/doc ?ptc/04-03-01.

[SPEC-1] Supélec. *Spécification du module de vérification MOD-Supélec-1*, 13 juillet 2006, disponible sur http ://www.usine-logicielle.org.

[STREAMS] Object Management Group. *Streams for CCM final adopted specification*, OMG document number ptc/05-07-01. Disponible sur http ://www.omg.org/cgi-bin/doc ?ptc/05-07-01.

[Aho 2006] Alfred V. Aho, Ravi Sethi et Jeffrey D. Ullman. *Machine-Independent Optimizations*. In Compilers : Principles, Techniques, and Tools, chapitre 9. Addison-Wesley, 2006.

[Berry 1992] Gérard Berry et Georges Gonthier. *The ESTEREL synchronous programming language : design, semantics, implementation*. Sci. Comput. Program., vol. 19, no. 2, pages 87–152, 1992.

[Chang 1992] Edward Y. Chang, Zohar Manna et Amir Pnueli. *Characterization of Temporal Property Classes*. In ICALP '92 : Proceedings of the 19th International Colloquium on Automata, Languages and Programming, pages 474–486, London, UK, 1992. Springer-Verlag.

[Cocke 1970] John Cocke. *Global common subexpression elimination*. Proceedings of a symposium on Compiler optimization (SIGPLAN), pages 20–24, 1970.

[Gamma 1999] Eric Gamma, Richard Helm, Ralph Johnson et John Vlissides. Design Patterns, Catalogue des modèles de conception réutilisables. Vuibert, 1999.

[Halbwachs 1991] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud. *The synchronous dataflow programming langage Lustre*. Proceedings of the IEEE, vol. 79, no. 9, pages 1305–1320, 1991.

[Halbwachs 1992] Nicolas Halbwachs, Fabienne Lagnier et Christophe Ratel. *Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE*. IEEE Trans. Softw. Eng., vol. 18, no. 9, pages 785–793, 1992.

[Halbwachs 1993] N. Halbwachs, J.C. Fernandez et A. Bouajjanni. *An executable temporal logic to express safety properties and its connection with the language Lustre*. In Sixth International Symposium on Lucid and Intensional Programming, 1993.

[Jagadeesan 1995] Lalita Jategaonkar Jagadeesan, Carlos Puchol et James Von Olnhausen. *Safety Property Verification of Esterel Programs and Applications to Telecommunications Software*. In Proceedings of the 7th International Conference on Computer Aided Verification, pages 127–140. Springer-Verlag London, UK, 1995.

[Laroussinie 1995] F. Laroussinie et Ph. Schnoebelen. *A hierarchy of temporal logics with past*. Theor. Comput. Sci., vol. 148, no. 2, pages 303–324, 1995.

[Manna 1992a] Zohar Manna et Amir Pnueli. *Basic Properties of the Temporal Operators — Monotonicity*. In The Temporal Logic of Reactive and Concurrent Systems Specification, chapitre 3, pages 202–203. Springer-Verlag, 1992.

[Manna 1992b] Zohar Manna et Amir Pnueli. *The Classification of Properties*. In The Temporal Logic of Reactive and Concurrent Systems Specification, chapitre 4, page 282. Springer-Verlag, 1992.

1.5 Résumé

Le chapitre 1 constitue l'introduction de ce document, il présente en particulier la structure du document.

Le chapitre 2 présente le contexte et les objectifs des travaux réalisés.

Le chapitre 3 introduit un exemple d'utilisation de notre approche, exemple qui sera repris dans les chapitres suivants afin d'illustrer les éléments présentés.

Le chapitre 4 présente une syntaxe concrète textuelle de notre langage.

Le chapitre 5 présente le modèle des applications susceptibles d'être décrites par notre langage et donc traitées par notre outil pour la génération vers Inflexion et la vérification des propriétés. Ce modèle constitue la syntaxe abstraite de notre langage.

Le chapitre 6 présente l'aspect vérification de propriétés.

Le chapitre 7 présente l'outil.

2 Contexte et objectifs

2.1 Contexte

La conception des systèmes logiciels embarqués peut être analysée sous deux aspects : la conception des traitements, c'est-à-dire des calculs effectués sur les entrées du système pour produire ses sorties, et la conception du contrôle, qui détermine quels traitements sont effectués, et selon quels paramètres. Chacun de ces aspects peut lui-même être traité selon différentes approches ou modèles de calcul. Les modèles de calcul utilisés pour la conception des traitements s'appuient généralement sur la notion de flot de données circulant entre des opérateurs (modèles de type Matlab/Simulink). Les modèles de calcul utilisés pour le contrôle sont les modèles adaptés aux systèmes réactifs : automates et langages synchrones.

L'intérêt de la conception séparée du contrôle et des traitements est qu'elle permet de rendre le contrôle explicite et vérifiable. Lorsqu'un opérateur à flots de données a plusieurs modes de fonctionnement, on peut soit coder ces modes dans le comportement de l'opérateur, soit les coder dans le contrôle de l'application qui activera des opérateurs différents ou changera les paramètres de l'opérateur selon le mode de fonctionnement souhaité. Si on adopte la première approche, le contrôle est enfoui au sein du code de l'opérateur et n'est pas accessible à des outils de validation tels que les model-checkers. De plus, puisque l'opérateur contient une partie du contrôle de l'application, il a peu de chance d'être réutilisable dans un autre contexte. D'autre part, le contrôle d'un système embarqué est modifié plus souvent que les traitements au cours de la vie du système. Il est donc important de découpler contrôle et traitements afin de limiter l'impact des modifications du contrôle.

Toutefois, pour obtenir une implémentation du système, il faut intégrer le contrôle et les traitements afin que ces derniers soient activés et paramétrés correctement, et que le contrôle reçoive les informations qui lui permettent de choisir la configuration des traitements. Du point de vue de la validation, les propriétés qui doivent être vérifiées sont des propriétés globales de l'application, or les outils de validation formelle ne s'appliquent qu'à un modèle de calcul, par exemple le modèle réactif synchrone. L'intégration du contrôle et des traitements possède donc aussi un aspect « validation » en plus de l'aspect « implémentation ».

2.2 Objectifs

Dans ce contexte, nous proposons de décrire une application sous la forme de composants de traitements qui communiquent par des flots de données et qui sont activés et interconnectés sous la supervision d'un composant de contrôle. Cette description sera utilisée à la fois pour produire l'implémentation de l'application et pour traduire des propriétés de l'application en propriétés vérifiables sur le contrôle. Il ne s'agit pas de réaliser un outil de vérification formelle. L'objectif est uniquement de s'appuyer sur la description de la structure de l'application pour transformer des propriétés globales du système en propriétés exprimées sous une forme reconnue par les outils de vérification disponibles dans le formalisme utilisé pour la spécification du contrôle. L'intérêt de cette approche est qu'elle permet d'exprimer des propriétés selon une sémantique intuitive dans le contexte de l'application (par exemple « le composant C n'est jamais actif lorsque l'entrée E est supérieure à une valeur V »), et de les traduire dans la sémantique événementielle du contrôle (« l'événement Y n'est jamais produit entre une occurrence de X et une occurrence de Z »), cela en s'appuyant sur la même description de l'application qui est utilisée pour produire son implémentation. On a ainsi l'assurance

que la transformation de la propriété à vérifier est cohérente avec la manière dont le code de l'application est généré.

Le but de cette tâche est donc de définir un langage de description d'applications qui permette de décrire :
– l'interface des composants de traitement ;
– l'interface du composant de contrôle ;
– l'activation et la désactivation des composants de traitement ;
– la transformation de données des traitements en événements pour le contrôle ;
– les connexions entre composants de traitement et leur modification ;
– les propriétés à vérifier sur l'application.

2.3 Principes de mise en œuvre

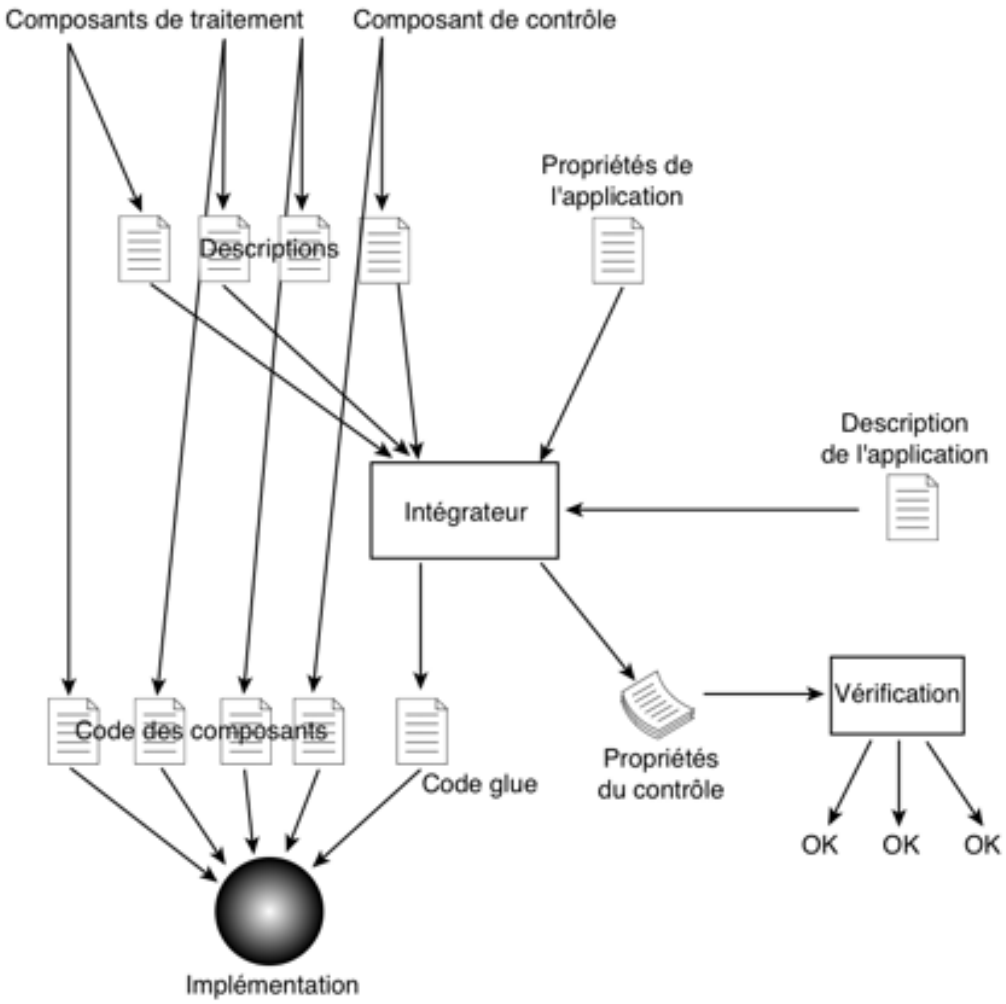

FIG. 1 – Schéma général de la mise en œuvre de l'outil d'intégration contrôle/traitements

Les composants de l'application sont développés en utilisant les outils propres à l'approche utilisée pour leur conception. Notre outil d'intégration n'utilise que la description de leur interface (signaux d'entrée et de sortie) dans notre langage de description. Cette description peut être faite manuellement ou générée

automatiquement à partir d'un modèle du composant. Afin de faciliter la transformation des propriétés à vérifier et leur expression sous une forme utilisable par les outils de vérification formelle, le composant de contrôle ne peut communiquer que par événements purs, sans valeur associée. Une partie du langage de description est donc dédiée à la transformation de conditions sur les données en événements, afin que le contrôle puisse réagir à certaines conditions calculées par les composants de traitement. Ces transformations sont effectuées par des composants qui font partie de la description de l'application, mais qui utilisent un modèle de calcul interne à notre outil.

À partir de la description d'une application, l'outil génère le code des composants qui utilisent le modèle de calcul interne, ainsi qu'un code « glue » d'interface avec le code généré pour les composants de traitement et le composant de contrôle. La nature de cette glue dépend des générateurs de code qui produisent les composants. La description d'un composant dans notre langage doit donc comporter une indication de l'outil utilisé pour produire son code.

De façon similaire, les propriétés à vérifier sur l'application sont transformées en propriétés à vérifier sur le contrôle, et sont exprimées dans le formalisme adapté à l'outil de vérification. Ce formalisme doit donc être indiqué dans la description du composant de contrôle. Le schéma général de mise en œuvre de notre outil d'intégration est représenté sur la figure 1.

3 Exemple applicatif : un régulateur de vitesse

Afin de faciliter la présentation détaillée de la méthodologie et de l'outil, un exemple est présenté ici, et servira de support aux explications données dans les chapitres suivants.

3.1 Interface externe

Nous avons retenu ici un exemple classique (et très simplifié) du régulateur de vitesse. Celui-ci fournit une commande au système d'injection qui est soit directement la commande issue de la pédale d'accélérateur, quand la régulation n'est pas active, soit une commande calculée pour maintenir une certaine vitesse quand la régulation est active.

Nous souhaitons que ce système respecte les règles de sécurité classiques telles que rendre la régulation inactive si le conducteur appuie sur la pédale de frein, interdire l'activation de la régulation si la vitesse n'est pas dans une plage autorisée...

L'interface externe de notre module est ainsi définie :

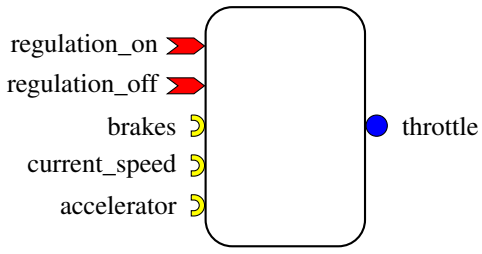

FIG. 2 – Exemple applicatif : interface externe

Les conventions de représentation des ports d'entrée/sortie sont :
- puits d'événements
- source d'événements

- entrée d'un flot de données
- sortie d'un flot de données

Pour notre exemple, nous considérons donc que la demande de mise en route et d'arrêt de la régulation (commandes données par le conducteur) sont des événements alors que les autres entrées sont des flots : un flot de booléens pour indiquer si la pédale de frein est appuyée ou non, un flot de flottants pour donner la vitesse courante et un flot d'entiers (entre 0 et 100) pour indiquer la position actuelle de la pédale d'accélérateur. Notre module fournit un flot d'entiers (aussi entre 0 et 100) à la commande d'injection.

3.2 Contraintes méthodologiques

Dans l'exemple présenté, on remarque que les demandes de mise en route et d'arrêt de la régulation sont modélisées par 2 événements distincts. Ce choix est imposé par notre approche : la communication entre composants se fait soit par l'intermédiaire d'événements purs (sans information associée), soit par l'intermédiaire de flots de données (sans aucune sémantique associée au fait qu'une donnée arrive sur un port : seule la valeur de cette donnée est significative).

Ces contraintes fortes ont pour objectif d'isoler la partie contrôle du système (qui, par hypothèse, ne travaille qu'avec des événements) de la partie traitement de ce même système.

3.3 Architecture à base de composants

Nous retenons une approche classique d'architecture à base de composants.

3.3.1 Un seul contrôleur

Comme indiqué ci-dessus, la partie contrôle est purement événementielle ; une contrainte supplémentaire de notre approche et que ce contrôle est réalisé par un et un seul composant.

Voici l'interface de ce composant contrôleur pour notre régulateur de vitesse :

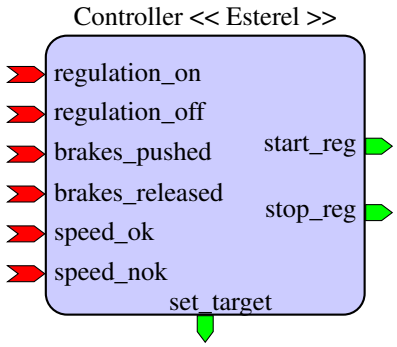

FIG. 3 – Exemple applicatif : le contrôleur

Nous ne fournissons pas d'outil pour réaliser ce contrôleur ; au contraire, nous laissons à l'équipe de développement le choix de cet outil, choix dicté par ses propres contraintes ou habitudes, et nous proposons des mécanismes permettant d'intégrer le code C ou C++ généré par l'outil choisi. Dans l'exemple présenté, nous supposons que le contrôleur est réalisé en Esterel.

3.3.2 Plusieurs composants de traitement

Pour notre exemple simplifié de régulateur de vitesse, nous avons besoin d'un composant de traitement, celui qui calcule la bonne commande d'injection quand la régulation est active.

Comme pour le composant contrôleur, nous laissons à l'équipe de conception le choix de l'outil pour réaliser ce composant de traitement ; contrairement au contrôleur, il n'y a pas de contrainte sur le nombre de composants de traitement dans une application ADLV.

Voici l'interface de notre régulateur, ici supposé réalisé en Simulink :

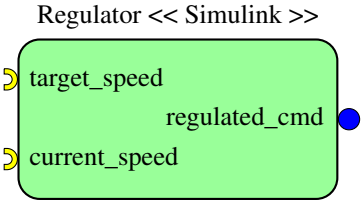

FIG. 4 – Exemple applicatif : le régulateur

Ce composant ne travaille que sur des flots de données. Cette contrainte n'est pas ici imposée par notre méthodologie (contrairement au choix d'un contrôleur purement événementiel), mais facilite l'interfaçage du code généré par notre outil avec le code généré par l'outil choisi pour réaliser le composant de traitement.

Le composant contrôleur et les composants de traitement sont qualifiés de composants externes car leurs comportements sont définis à l'extérieur de notre outil.

3.3.3 Composants internes

Nous avons vu les contraintes imposées aux composants de contrôle et de traitement, et donc à l'équipe réalisant la conception du système avec notre approche ADLV. En contre-partie de celles-ci, nous offrons la possibilité de définir des petits composants chargés d'émettre des événements selon certaines conditions portant sur des valeurs, ou d'émettre des flots de données. Pour l'instant, il y a 2 sortes de tels composants[1] :

– un composant émettant un événement quand une expression booléenne (pouvant faire référence à des flots de données entrantes) passe de faux à vrai ;
– un composant émettant la valeur d'un attribut comme flot de données, valeur calculée par une expression sur réception d'un événement.

On trouve comme exemples de composants de la première sorte un composant chargé d'indiquer les changements d'état de la pédale de frein :

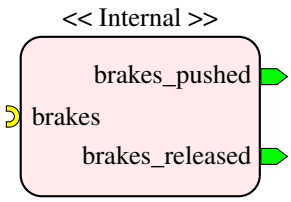

FIG. 5 – Exemple applicatif : changement d'état de la pédale de frein

[1] Ce sont bien sûr les 2 sortes dont nous avons besoin pour notre exemple, mais ils sont conçus pour faire des choses plus complexes que ce qui est montré ici

et un composant chargé d'isoler les périodes où la vitesse permet l'activation de la régulation :

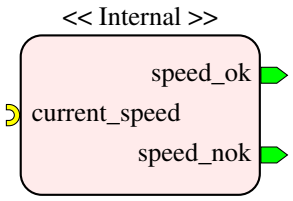

FIG. 6 – Exemple applicatif : vitesse correcte pour activer la régulation

Comme exemple de composant de la seconde sorte, on trouve le composant fournissant la vitesse de consigne au régulateur :

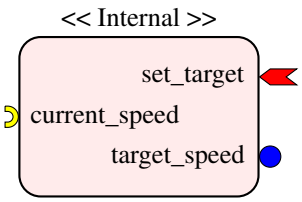

FIG. 7 – Exemple applicatif : mémorisation de la vitesse de consigne

3.3.4 Configuration dynamique

Le contrôleur décide quand activer/désactiver la régulation, l'accélérateur fournit la commande non régulée, le régulateur fournit la commande régulée, il nous reste à lier ces éléments pour déterminer la sortie de notre module régulateur de vitesse. Ceci est réalisé grâce à la possibilité de définir des connexions dynamiques (établies sur réception d'un événement) entre 2 ports d'entrée/sortie de l'application ou des composants :

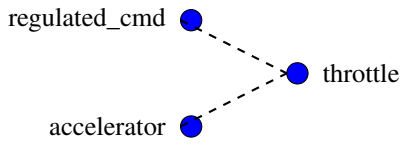

FIG. 8 – Exemple applicatif : configuration dynamique

3.4 L'exemple complet

Voici le schéma complet de notre exemple applicatif :

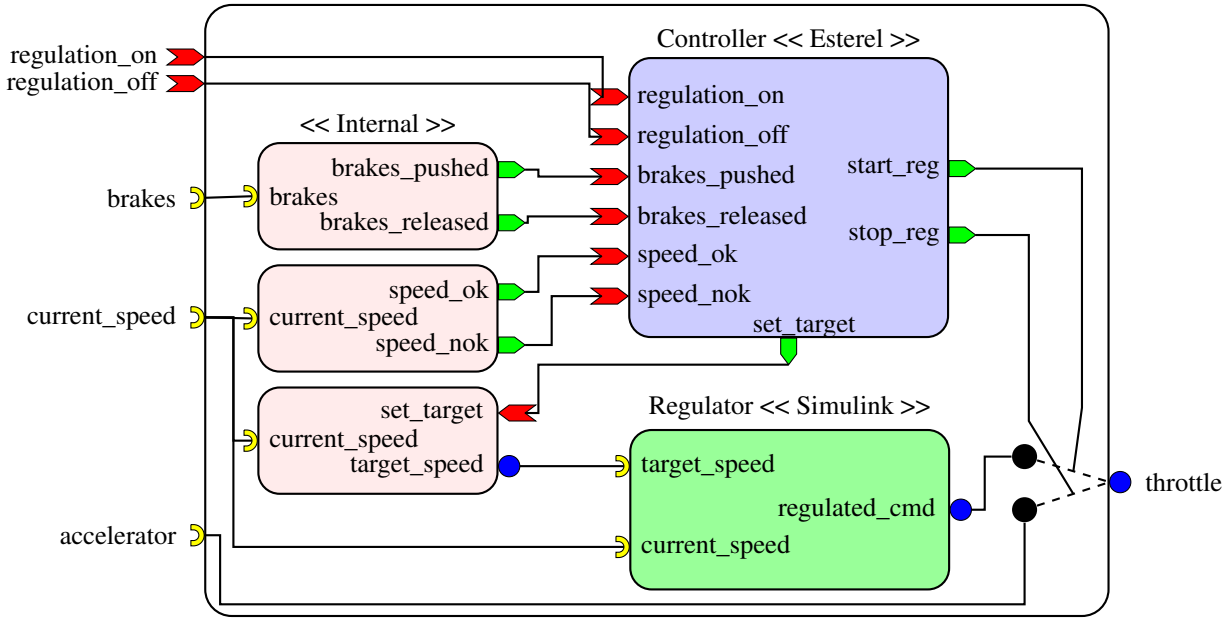

FIG. 9 – Exemple applicatif : application complète

3.5 Vérification de propriétés

Sur notre démonstrateur applicatif, voici quelques exemples de propriétés, exprimées ici en langage naturel, que l'on souhaite vérifier :

- la régulation n'est jamais active si les freins sont appuyés,
- on ne mémorise pas la vitesse de consigne si cette dernière n'est pas dans l'intervalle autorisé,
- si le conducteur appuie sur les freins, alors l'injection sera commandée par l'accélérateur tant que les freins n'auront pas été relâchés et que le conducteur n'aura pas redemandé l'activation de la régulation.

3.6 Génération vers Inflexion

L'exemple donné ici sera explicité dans le chapitre suivant dans le langage textuel ADLV que nous avons défini, afin de pouvoir être traité par nos outils. Ceux-ci vont permettre :

- de générer les différents fichiers pour la plate-forme Inflexion,
- de générer les observateurs permettant de vérifier les propriétés.

La génération vers Inflexion va ainsi produire les composants correspondant au schéma suivant :

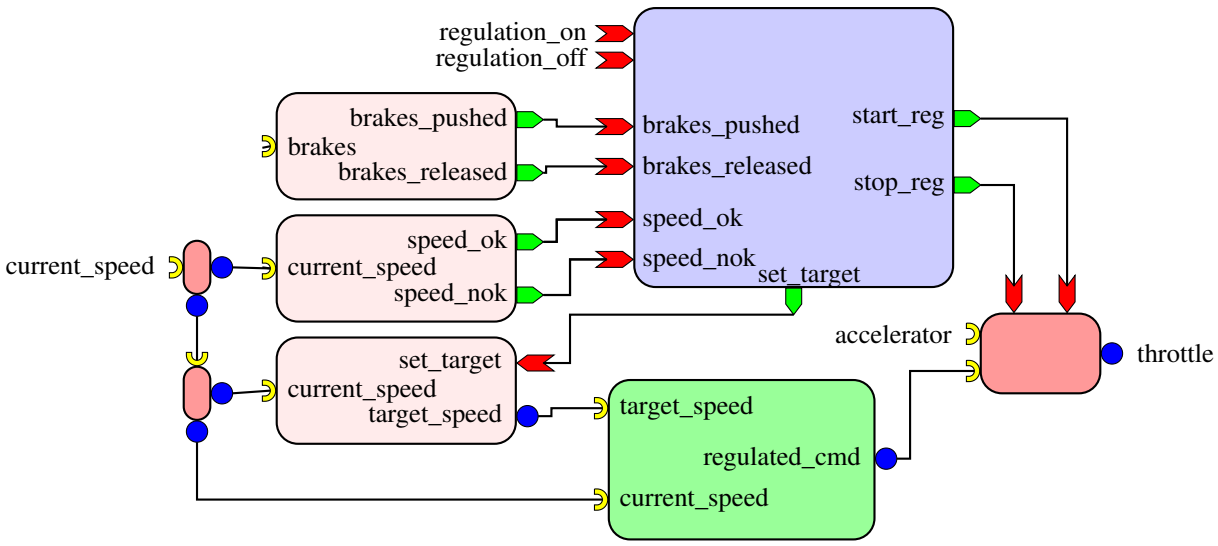

FIG. 10 – Exemple applicatif : composants générés

On voit que les composants internes et externes ont été conservés en tant que composants Inflexion, mais aussi que la gestion de la configuration dynamique est devenue un composant, et que des duplicateurs de flots ont été ajoutés (eux aussi en tant que composants Inflexion). Un dernier composant, qui n'apparaît pas sur ce schéma, a été créé : il s'agit d'un composant "miroir" de l'application comportant tous les ports de l'application mais avec une inversion des rôles producteur-consommateur. Ce dernier composant peut être complété à la main pour réaliser par exemple une simulation.

4 Syntaxe concrète

Nous allons, dans ce chapitre, présenter la syntaxe concrète de notre langage en utilisant comme base l'exemple présenté dans le chapitre précédent.

La définition complète de cette syntaxe concrète figure dans la dernière section de ce chapitre.

Il existe déjà de nombreux langages de description d'architecture. Aucun ne répond précisément à nos besoins, mais pour minimiser la phase d'apprentissage de notre langage, nous avons décidé d'utiliser un langage existant et de lui ajouter les extensions nécessaires à notre approche.

Le langage IDL [CORBA] a été choisi comme base car il peut être qualifié de standard et il est utilisé par Inflexion, plate-forme cible de notre outil.

4.1 Définitions dans un module

Nous imposons que toute définition soit faite dans un module. La syntaxe est exactement celle d'IDL :

```
module SpeedRegulatorModule {
    // Tous les extraits présentés par la suite sont supposés être
    // à l'intérieur de ce module
```

```
    // ...
};
```

4.2 Définition des types d'événements

Les types des événements sont définis avec la syntaxe IDL :

```
// Le conducteur demande, ou arrête, la régulation de vitesse
eventtype RegulationOn {};
eventtype RegulationOff {};

// Autres définitions non montrées ici
```

On remarque que les accolades de la syntaxe IDL ont été conservées, même si elles ne servent à rien ici puisque nous n'autorisons que des événements purs.

4.3 Définition des types de flots de données

Nous avons repris la syntaxe IDL définie dans la spécification "Streams for CCM Specification" [STREAMS], à l'aide du mot clef **streamtype** :

```
// La vitesse courante et de consigne
streamtype< double > CurrentSpeed;
streamtype< double > TargetSpeed;

// Vrai si le conducteur appui sur le frein
streamtype< boolean > BrakesOn;

// La position de l'accélérateur [0..100]
streamtype< long > AcceleratorPosition;

// Autres définitions non montrées ici
```

On remarque que, dans notre exemple, chaque flot de données a son propre type. Ce n'est pas imposé par notre méthodologie, et on aurait pu, par exemple, se contenter de :

```
// La vitesse
streamtype< double > Speed;
```

ou même utiliser directement **streamtype**< **double** > aux endroits adéquats. Le choix de typage fort permet, ici comme avec d'autres langages, de détecter des erreurs plus tôt dans le cycle de développement. Cette remarque s'applique aussi aux définitions des types d'événements.

4.4 Définition des types de composants

Nous avons ici ajouté de nouveaux mots-clefs pour distinguer nos différentes sortes de composants : **control**, **processing** et **internal**.

4.4.1 Définition des composants externes

Les composants externes doivent de plus préciser avec quel outil ils ont été réalisés, avec le mot-clef **`tool`**. Les autres aspects syntaxiques sont conformes à IDL, avec les mots-clefs **`consumes`**, **`publishes`**, **`sink`** et **`source`**.

Voici par exemple la définition du contrôleur :

```
// Un module de contrôle est purement événementiel
control component Controller {
    tool Esterel;

    consumes RegulationOn regulation_on;
    consumes RegulationOff regulation_off;

    consumes SpeedCorrect speed_correct;
    consumes SpeedIncorrect speed_incorrect;

    consumes BrakesPushed brakes_pushed;
    consumes BrakesReleased brakes_released;

    publishes StartSpeedRegulation start_speed_regulation;
    publishes StopSpeedRegulation stop_speed_regulation;
    publishes SetTargetSpeed set_target_speed;
};
```

et celle du régulateur :

```
// Un module de traitement est purement à flots
processing component Regulator {
    tool Simulink;

    sink CurrentSpeed current_speed;
    sink TargetSpeed target_speed;
    source RegulatedInjectionCommand regulated_injection_command;
};
```

4.4.2 Définition des composants internes

Le comportement des composants internes est écrit dans notre langage.

Voici par exemple le composant émettant des événements pour chaque changement de l'état de la pédale de frein :

```
internal component BrakesCheck {
    sink BrakesOn brakes_on;

    // Syntaxe : when <expression-booléenne>
    // Un événement est émis quand l'expression du when passe de false à true
    publishes BrakesPushed brakes_pushed {
        when   brakes_on;
    }

    publishes BrakesReleased brakes_released {
        when ! brakes_on;
    }
};
```

On voit que l'expression booléenne fait référence au nom d'un port de flot d'entrée ; ce composant va donc examiner les valeurs booléennes arrivant sur le port d'entrée `brakes_on` et émettre un événement de type `BrakesPushed` sur le port `brakes_pushed` quand il détectera une valeur vraie alors que la précédente était fausse. De même, il émettra un événement de type `BrakesReleased` sur le port `brakes_released` quand il détectera une valeur fausse alors que la précédente était vraie.

L'expression du **`when`** suit la syntaxe du langage C++, et peut donc faire intervenir des opérateurs logiques et de comparaison. Voici ainsi le composant interne détectant les vitesses correctes pour activer la régulation :

```
internal component SpeedCheck {
    sink CurrentSpeed current_speed;

    publishes SpeedCorrect speed_correct {
        when current_speed >= 40
          && current_speed <= 130;
    }

    publishes SpeedIncorrect speed_incorrect {
        when current_speed < 40
          || current_speed > 130;
    }
};
```

L'exemple du régulateur de vitesse utilise une seconde sorte de composant interne : un composant émettant la valeur d'un attribut comme flot de données, valeur calculée par une expression sur réception d'un événement. Voici sa définition :

```
internal component TargetSpeedSet {
    attribute double stored_target_speed;

    sink CurrentSpeed current_speed;

    consumes SetTargetSpeed set_target_speed {
        stored_target_speed = current_speed;
    }

    source TargetSpeed target_speed {
         output stored_target_speed;
    }
};
```

Sur réception d'un événement sur le port `set_target_speed`, ce composant va affecter la valeur courante du flot d'entrée du port `current_speed` (une expression plus complexe pourrait être utilisée) à l'attribut `stored_target_speed`, attribut dont la valeur est émise comme flot de sortie sur le port `target_speed`.

4.5 Définition des applications

4.5.1 Définition de la structure d'une application

Une application est un conteneur pour composants, fonction identifiée par le mot clef **`application`**, mais aussi un composant dans le sens où elle a ses propres ports. Ces derniers sont déclarés de la même façon que pour les composants :

```
application SpeedRegulatorApplication {
    consumes RegulationOn regulation_on;
    consumes RegulationOff regulation_off;

    sink CurrentSpeed current_speed;
    sink BrakesOn brakes_on;
    sink AcceleratorPosition accelerator_position;

    source InjectionCommand injection_command;

    // suite ci-dessous
```

Les composants qui constituent l'application sont introduits par le mot clef **instance** :

```
    // suite de ci-dessus

    instance Controller controller;
    instance SpeedCheck speed_checker;
    instance BrakesCheck brakes_checker;
    instance TargetSpeedSet target_speed_setter;
    instance Regulator regulator;

    // suite ci-dessous
```

On précise donc le type du composant, et un nom pour l'identifier. Il est bien sûr possible d'avoir plusieurs composants d'un même type.

4.5.2 Définition de la configuration d'une application

C'est au niveau de l'application que l'on indique quels sont les connexions entre ports ainsi que les composants actifs. Ce dernier point provient du fait que notre méthodologie permet de modifier dynamiquement les connexions mais aussi de désactiver ou activer un composant dynamiquement. Ces deux informations (connexions et composants actifs) sont distinguées selon que la configuration est permanente ou dynamique.

La description de la configuration dynamique inclut les changements dynamiques de configuration : ceux-ci sont déclenchés par des événements, et donc attachés à des ports consommateurs internes à l'application :

```
    // suite de ci-dessus

    consumes StartSpeedRegulation start_speed_regulation {
        activate regulator;
        // Sur un port consommateur, on ne peut brancher qu'un seul port
        // producteur. Donc, s'il y avait déjà une connexion sur le port
        // consommateur, cette dernière est supprimée
        injection_command << regulator.regulated_injection_command;
    }

    consumes StopSpeedRegulation stop_speed_regulation {
        deactivate regulator;
        injection_command << accelerator_position;
    }

    // suite ci-dessous
```

et la configuration initiale :

```
// suite de ci-dessus

init {
    injection_command << accelerator_position;
}

// suite ci-dessous
```

La syntaxe pour indiquer une connexion est donc `port_consommateur << port_producteur`. Si c'est un port de composant, son nom est préfixé par le nom du composant, sinon il s'agit d'un port d'application.

Les configurations permanentes sont indiquées dans un bloc **`static`** :

```
// suite de ci-dessus

static {
    controller.regulation_on << regulation_on;
    controller.regulation_off << regulation_off;

    controller.speed_correct << speed_checker.speed_correct;
    controller.speed_incorrect << speed_checker.speed_incorrect;

    // Autres connexions et activations non montrées ici

    activate brakes_checker;
    activate target_speed_setter;
}

// suite ci-dessous
```

4.5.3 Définition des propriétés à vérifier d'une application

La dernière partie, optionnelle, de la définition d'une application est la partie contenant les propriétés que l'on souhaite vérifier. Ces propriétés sont de deux sortes : celles qui doivent être toujours vraies (clauses **`always`**) et celles qui doivent être toujours fausses (clauses **`never`**). Ces deux notations constituent une *facilité d'écriture*, car nous verrons plus loin qu'elles sont sémantiquement équivalentes. Par exemple, on peut avoir :

```
// suite de ci-dessus

never {
    // le régulateur ne peut pas être actif quand les freins
    // sont appuyés
    brakes_on && active( regulator );

    // on ne mémorise pas la vitesse si cette dernière n'est pas
    // dans l'intervalle prévu
    controller.set_target_speed &&
            ( current_speed < 40 || current_speed > 130 );
}

always {
    // Contrôleur toujours actif
    active( controller );
```

```
        }

        // suite ci-dessous
```

Les formules doivent être exprimées en logique temporelle linéaire (LTL) passée (voir section 6.1). Ces formules présentent un bon degré d'expressivité, et nous verrons au paragraphe 6.1.4 qu'il est possible de passer de certaines expressions « au futur » à des expressions « au passé ».

4.6 Définition formelle de la syntaxe textuelle ADLV

Voici en notation BNF la syntaxe textuelle :

```
<module_definition> ::=
    "module" <identifier> "{"
        <definition>*
        <application_def>?
    "}" ";"

<definition> ::=
      <event_type> ";"
    | <stream_type> ";"
    | <component_type> ";"

<event_type> ::=
    "eventtype" <identifier> "{" "}"

<stream_type> ::=
    "streamtype" "<" <datatype> ">" <identifier> "{" "}"

<datatype> ::=
      "boolean"
    | "double"
    | "long"

<component_type> ::=
      <control_component_type>
    | <processing_component_type>
    | <internal_component_type>

control_component_type ::=
    "control" "component" <identifier> "{"
        "tool" <identifier> ";"
        <control_component_body>*
    "}"

processing_component_type ::=
    "processing" "component" <identifier> "{"
        "tool" <identifier> ";"
        <processing_component_body>*
    "}"

internal_component_type ::=
    "internal" "component" <identifier> "{"
        <internal_component_body>*
```

```
     "}"

application_def ::=
    "application" <identifier> "{"
        <application_body>*
        "init"   "{" ( <configuration_step> ";" )* "}"
        "static" "{" ( <configuration_step> ";" )* "}"
        ( <ltl_clause> ";" )*
    "}"


control_component_body ::=
    ( consumes_dcl | publishes_dcl ) ";"


processing_component_body ::=
    ( source_dcl | sink_dcl ) ";"


internal_component_body ::=
      ( source_dcl    ( ";" | source_def )
    |   sink_dcl        ";"
    | ( consumes_dcl  ( ";" | component_consumes_def )
    | ( publishes_dcl ( ";" | publishes_def )
    | "attribute" <datatype> <identifier> ";"

application_body ::=
      "instance" <identifier> <identifier> ";"
    | ( consumes_dcl  ( ";" | application_consumes_def )
    | publishes_dcl ";"
    | sink_dcl ";"
    | source_dcl ";"
    | init_proc

source_dcl ::=
    "source" <identifier> <identifier>


source_def ::=
    "{" "output" <identifier> ";" "}"

sink_dcl ::=
    "sink" <identifier> <identifier>


consumes_dcl ::=
    "consumes" <identifier> <identifier>

component_consumes_def ::=
    "{" <identifier> "=" <expression> ";" "}"

application_consumes_def ::=
    "{" ( <configuration_step> ";" )* "}"

publishes_dcl ::=
    "publishes" <identifier> <identifier>
```

```
publishes_def ::=
    "{" "when" <expression> ";" "}"

configuration_step ::=
      "activate" <identifier>
    | "deactivate" <identifier>
    | <port_id> "<<" <port_id>

port_id ::=
    <identifier> ( "." <identifier> )?

ltl_clause ::=
    ( "never" | "always" ) "{"
        ( <expression> ";" )*
    "}"

expression ::=
    <binary_expression>
    ( <logical_op> <binary_expression> )*

logical_op ::=
      "&&"
    | "||"
    | "=>"
    | "since"
    | "backto"

binary_expression ::=
    <simple_expression>
    ( <compare_operator> <simple_expression> )?

compare_operator ::=
      "=="
    | "!="
    | "<"
    | "<="
    | ">"
    | ">="
    | "<<"

simple_expression ::=
      "(" <expression> ")"
    | <unary_operator> <simple_expression>
    | <port_id>
    | <litteral>
    | "active" "(" <identifier> ")"

unary_operator ::=
      "!"
    | "always"
    | "once"
    | "prev"
```

5 Syntaxe abstraite, modèle des applications

Si l'utilisation de la syntaxe concrète textuelle présentée dans le chapitre précédent est actuellement le seul moyen[2] de décrire des applications pour notre outil, il est tout à fait envisageable d'utiliser d'autres méthodes comme la transformation de modèles. En effet, nous avons respecté les approches actuelles de développement de nouveaux langages : définition des concepts et de leurs liens sous la forme d'un méta-modèle (instance d'EMOF), c'est la syntaxe abstraite d'un langage, et définitions d'une ou plusieurs syntaxes concrètes[3], textuelles ou graphiques.

Ce chapitre présente donc cette cette syntaxe abstraite, c'est-à-dire les différents concepts utilisés pour décrire les applications décrites en ADLV. Cette présentation utilise des diagrammes UML puisque cette syntaxe abstraite est directement utilisée pour la construction de l'outil (voir le chapitre 7).

5.1 Application

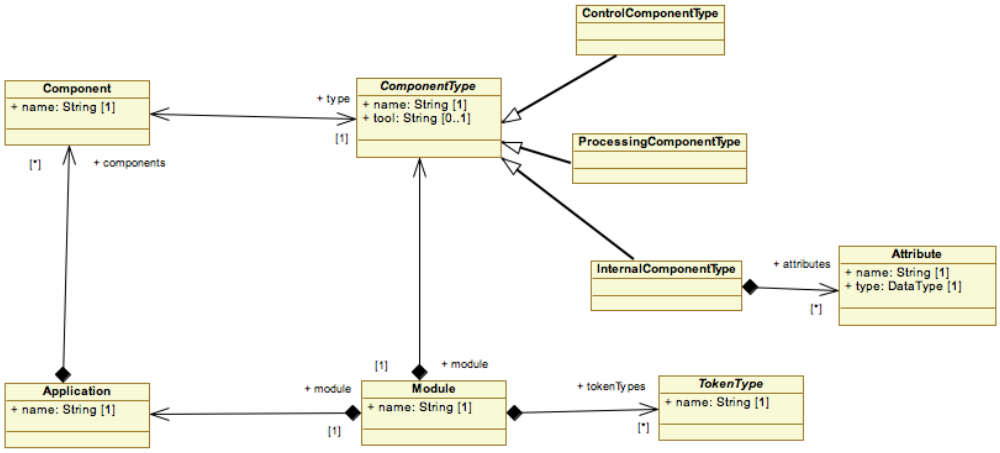

FIG. 11 – Syntaxe abstraite : application

Se reporter à l'extrait de la syntaxe abstraite sur la figure 11.

Une application ($Application$) est classiquement constituée de composants ($Component$) interconnectés. Un composant est instance de son type ($ComponentType$). Il existe 3 sortes de (types de) composants : le[4] composant de contrôle ($ControlComponentType$), les composants de traitement ($ProcessingComponentType$) et les composants internes ($InternalComponentType$). Les différentes sortes de composants internes ne sont pas distinguées au niveau de la syntaxe abstraite ; on modélise cependant l'éventuelle présence d'attributs.

Nous imposons que toute définition (application, type des composants) soit faite dans un module ($Module$) pour permettre une certaine modularité. Les types des données échangées entre composants ($TokenType$) doivent aussi être définis dans un module.

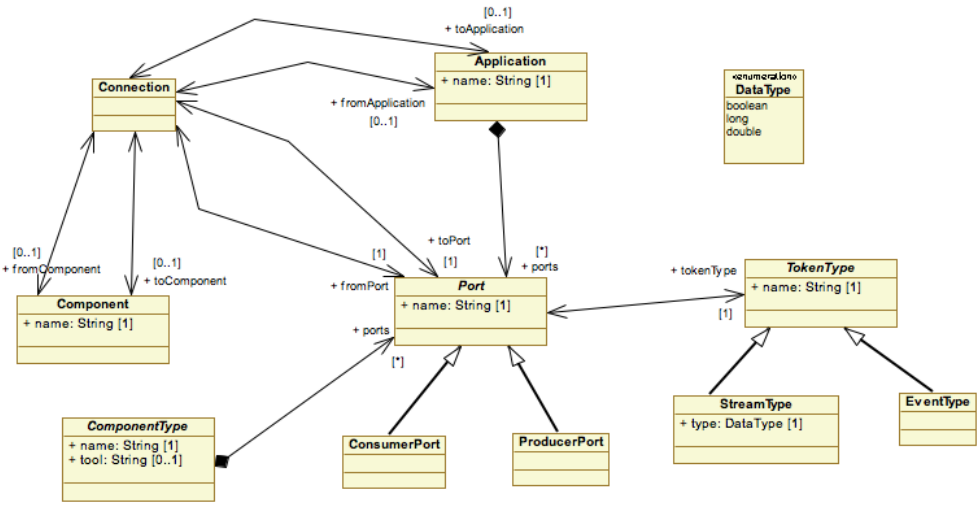

FIG. 12 – Syntaxe abstraite : composants

5.2 Composants

Se reporter à l'extrait de la syntaxe abstraite sur la figure 12.

Dans une architecture à base de composants, on établit des connexions ($Connection$) entre ports d'entrée/-sortie ($Port$). Ici, les ports sont qualifiés de consommateurs ($ConsumerPort$) ou producteurs ($Produ-cerPort$). Un port peut être soit porté par un composant, c'est alors le type du composant qui définit celui-ci, soit par l'application elle-même. C'est pourquoi chaque extrémité d'une connexion fait référence à un port et soit à un composant, soit à une application.

Une connexion est directionnelle : elle transporte des données d'un port ($fromPort$) à un autre ($toPort$). Le port source doit être un $ProducerPort$ si c'est un port de composant et un $ConsumerPort$ si c'est un port d'application. De même, le port de destination doit être un $ConsumerPort$ si c'est un port de composant et un $ProducerPort$ si c'est un port d'application. Les ports sont typés par les données qu'ils consomment ou produisent, et les deux extrémités d'une connexion doivent utiliser le même type. Les ports peuvent être de type événementiels ($EventType$) ou flot de données ($StreamType$) ; dans ce dernier cas, le type de la donnée est soit un booléen, soit un entier, soit un flottant.

5.3 Configuration

Se reporter à l'extrait de la syntaxe abstraite sur la figure 13.

C'est l'application qui définit la configuration de ses composants, c'est-à-dire :
- quels sont les composants actifs et inactifs au lancement de l'application,
- quels sont les connexions établies au lancement de l'application,
- quels composants sont activés ou désactivés lors de l'émission d'un événement particulier,
- quelles connexions sont établies lors de l'émission d'un événement particulier.

[2] ce n'est pas tout-à-fait exact, car notre langage existe sous la forme d'un modèle instance d'eCore, donc EMF sait automatiquement générer un éditeur correspondant.

[3] ou même d'aucune !

[4] Le fait que le composant de contrôle est unique n'apparait pas sur le modèle UML ; c'est une contrainte OCL qui permet d'imposer ce choix au niveau de la syntaxe abstraite.

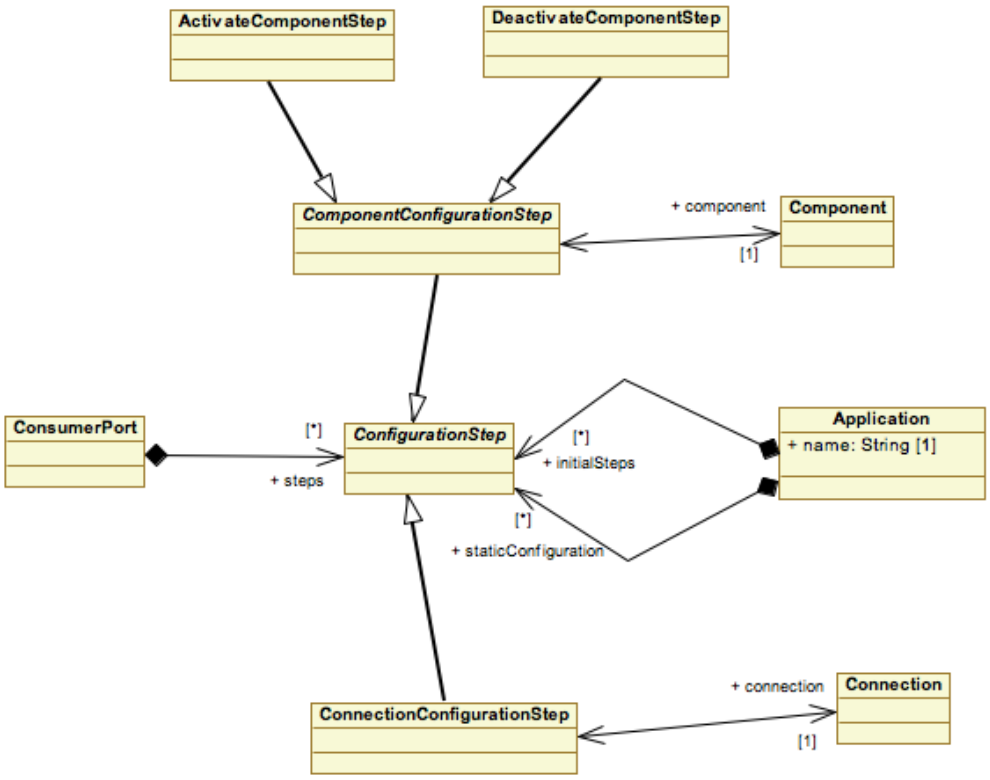

FIG. 13 – Syntaxe abstraite : configuration

Ces différentes actions sont nommées $ConfigurationStep$, et se divisent donc en $ActivateComponent-Step$, $DeactivateComponentStep$ et $ConnectionConfigurationStep$. On notera l'absence de déconnexion : nous avons en effet décidé qu'un port ne peut être au plus qu'une fois un $toPort$ dans une connexion. Ainsi, établir une nouvelle connexion vers un port qui est déjà destinataire dans une autre connexion supprime cette dernière. Par contre, un port peut participer comme $fromPort$ dans plusieurs connexions simultanées (c'est le cas par exemple du port d'application $current\_speed$ dans l'exemple présenté dans le chapitre 3), la sémantique est alors une duplication de la donnée vers les différents ports destinataires.

La configuration au lancement le l'application est divisée en 2 parties : la configuration statique ($staticConfiguration$) donne les composants qui seront toujours actifs et les connexions qui seront toujours établies. La configuration initiale ($initialSteps$) indique les composants qui sont actifs au lancement de l'application, mais qui pourront être désactivés par la suite, ainsi que les connexions établies au lancement mais qui sont susceptibles d'être supprimées par la suite. Cette distinction provient du fait que la plate-forme cible de notre outil est Inflexion, et que la version actuelle de cette plate-forme ne propose pas la reconfiguration dynamique. Comme indiqué ci-dessus, l'évolution de la configuration est provoquée par l'émission d'un événement particulier, a priori produit par le contrôleur. Pour rester cohérent avec notre approche de modélisation des applications, un tel événement arrivera donc sur un $ProducerPort$, et sera transmis via une $Connection$ vers un $ConsumerPort$. Un tel port sera qualifié de port interne à l'application, et se verra offert la possibilité de déclencher des $step$ à la réception d'un événement.

5.4 Composants internes et clauses de vérification

Se reporter à l'extrait de la syntaxe abstraite sur la figure 14.

Nous avons déjà évoqué les composants internes, et donné des exemples de ceux-ci dans le chapitre 3. Le comportement de ces composants internes est décrit dans notre langage et utilise de manière générale des expressions ($Expression$).

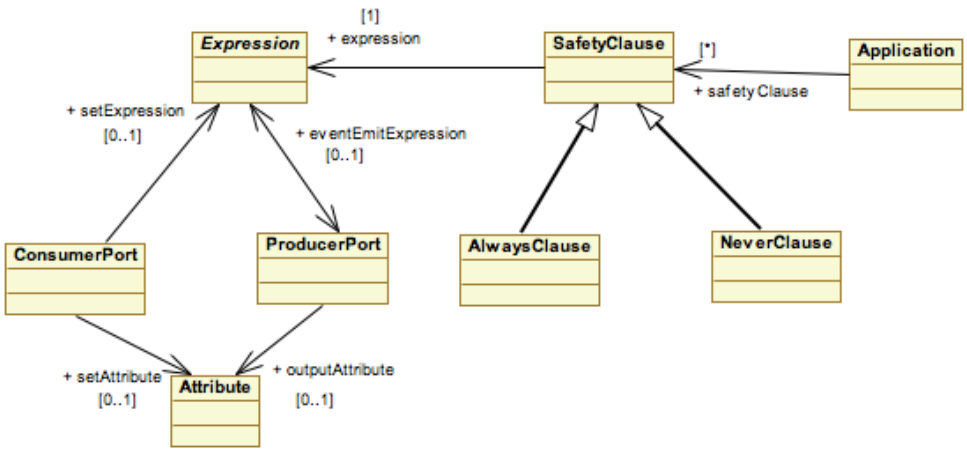

FIG. 14 – Syntaxe abstraite : composants internes et clauses de vérification

– Un composant émettant un événement quand une expression booléenne (pouvant faire référence à des flots de données entrantes) passe de faux à vrai est donc un composant offrant un $ProducerPort$ et lui associe une $eventEmitExpression$.
– Un composant émettant la valeur d'un attribut comme flot de données, valeur calculée par une expression sur réception d'un événement est donc un composant ayant un $ProducerPort$ associé à un $Attribute$ et un $ConsumerPort$ associé à une $setExpression$ pour indiquer la nouvelle valeur de l'attribut lors de la réception de l'événement.

Notre méthodologie a aussi pour vocation de pouvoir faciliter la vérification en proposant à l'utilisateur de pouvoir définir des propriétés portant sur l'application, propriétés qui seront ensuite traduites en observateurs destinés au contrôleur et écrits dans le langage de celui-ci. Ces propriétés sont de deux sortes : celles qui doivent être toujours vraies ($AlwaysClause$) et celles qui ne doivent jamais être vraies ($NeverClause$). Ces propriétés sont donc définies au niveau de l'application ($safetyClause$) et sont exprimées à l'aide d'une $expression$.

5.5 Expression

Se reporter à l'extrait de la syntaxe abstraite sur la figure 15.

Les différentes sortes d'expressions sont :

– les expressions binaires ($BinaryExpression$) qui combinent deux expressions par un opérateur binaire ($BinaryOperator$) ; les opérateurs possibles sont les opérateurs de comparaison, les opérateurs logiques, les opérateurs temporels et l'opérateur prédicat de connexion ($CONNECTED\_TO\_FROM$) ;
– les expressions unaires ($UnaryExpression$) qui préfixent un opérateur unaire ($UnaryOperator$) à une expression ; les opérateurs possibles sont les opérateurs logiques et temporels et l'opérateur absent ($NONE$) qui permet de représenter une expression quelconque parenthésée ;

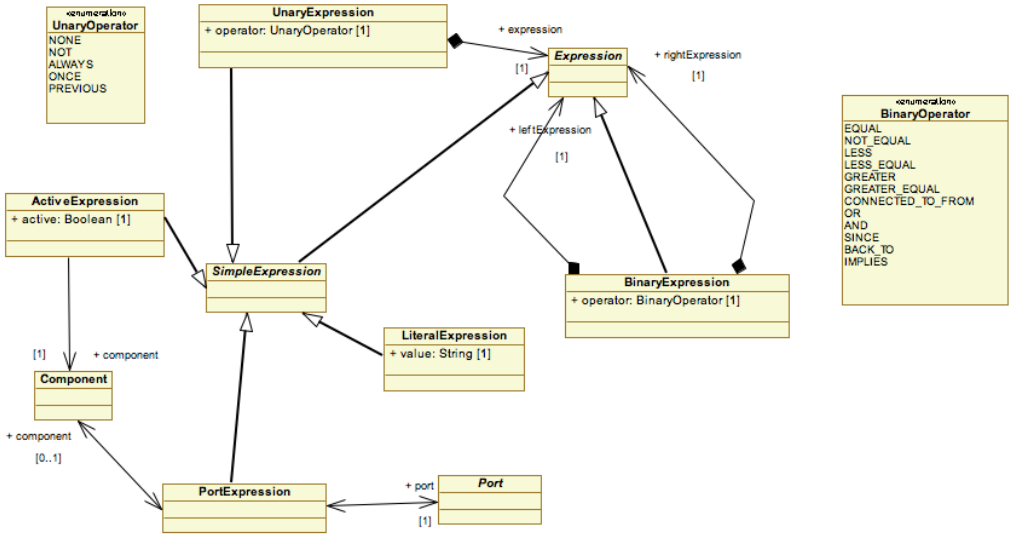

FIG. 15 – Syntaxe abstraite : expressions

- les expressions littérales ($LiteralExpression$) pour représenter des valeurs ;
- les expressions faisant référence à un port ($PortExpression$) et donc à la valeur arrivant sur ce port ;
- les prédicats sur les composant actifs ou inactifs ($ActiveExpression$).

6 Vérification de propriétés

Dans une description ADLV, outre la description de l'application et de ses composants, il est possible de faire figurer des *propriétés* qui doivent être vérifiées par le système. Ces propriétés doivent appartenir à un sous-ensemble de la logique temporelle linéaire (LTL), qui est habituellement appelé ensemble des *formules de sûreté canoniques*. La section 6.1 définit formellement la structure des formules utilisables.

Ces formules peuvent porter sur toutes les entrées et sorties de l'application. En se basant sur la description de cette dernière en ADLV, une méthode de notre conception permet de passer à une formule de logique temporelle dépendant uniquement des *événements* en entrée et en sortie du *contrôleur* de l'application. Cette méthode fait l'objet de la section 6.2.

Nous sommes alors capables de traduire cette dernière formule (dite *forme intermédiaire*) en un *observateur*, c'est-à-dire un module écrit dans le *même* langage que le contrôleur (par exemple, Esterel ou Lustre). On peut alors utiliser les outils propres à ce langage pour *prouver* que le contrôleur, et donc l'application, vérifient les propriétés de sûreté. La section 6.3 traite de la génération des programmes en langage cible, et de la vérification des propriétés.

La figure 16 résume l'enchaînement de ces différentes étapes.

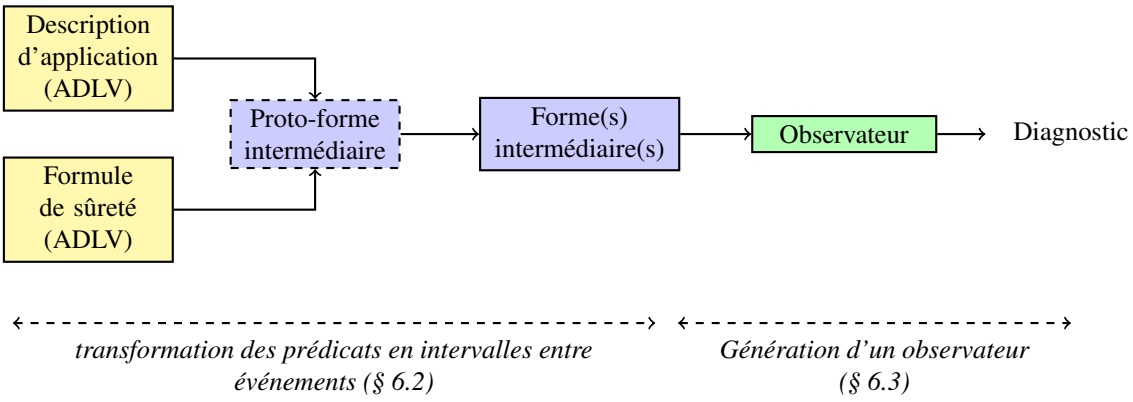

FIG. 16 – Vue synoptique du système de vérification de propriétés.

Notons que les propriétés sont vérifiées sur l'implémentation synchrone du contrôleur, destinée à piloter l'application au moment de l'exécution, grâce à des observateurs générés directement à partir de la description de l'application. Nous mettons donc en œuvre *doublement* le principe WYPIWYE : *what you prove is what you execute* [Berry 1992].

6.1 Formules de logique temporelle manipulées

6.1.1 Formules de sûreté canoniques

Notre but est de pouvoir exprimer le fait que certaines propriétés sont *toujours* vérifiées par l'application, c'est-à-dire qu'elles sont vraies à chaque instant de son fonctionnement. La logique temporelle linéaire fournit un opérateur permettant d'exprimer ce type de propriétés : $\Box$ (« toujours »). Ainsi, pour une formule f, dire que le système vérifie $\Box f$ signifie que la formule f est vraie à chaque instant de chacune de ses exécutions possibles. Notons que cet opérateur permet d'exprimer le « jamais » aussi bien que le « toujours » : « jamais f » se note $\Box\neg f$.

Dans le cadre de ce projet, nous nous sommes restreints aux formules de la forme $\Box f$, où f est une formule du *fragment passé* de la logique temporelle linéaire. Cela signifie que les opérateurs qui apparaissent dans f sont soit issus de la logique propositionnelle, soit des opérateurs *passés* de la logique temporelle linéaire. Ces formules f seront appelées en abrégé *formules passées* ; leur constitution est précisée ci-après.

De telles formules sont appelées *formules de sûreté canoniques*[5] dans la littérature [Manna 1992b, Chang 1992]. Ces formules ont l'avantage de proposer une bonne expressivité [Laroussinie 1995] et une bonne simplicité d'emploi [Halbwachs 1992, Jagadeesan 1995] tout en étant facilement traduisibles dans des langages synchrones classiques (par exemple, Esterel ou Lustre). À l'inverse, le fragment futur de la logique temporelle linéaire semble assez difficile à traduire dans ces langages.

Les formules passées que nous traitons sont composées de *prédicats* décrits dans la section 6.1.2, mis en relation par des *opérateurs* décrits dans la section 6.1.3.

Le composant ADLV produit un observateur par formule de façon à ce que :

– dès le moment où la formule d'une clause **`always`** devient fausse, l'observateur émet un signal d'erreur ;
– dès le moment où la formule d'une clause **`never`** devient vraie, l'observateur émet un signal d'erreur.

6.1.2 Prédicats

Les formules sont construites à partir des prédicats donnés dans le tableau 1.

Type	Forme	Signification
signal	s	vrai lorsque le signal s de type événement est présent
comparaison	$s\texttt{op}k$	vrai lorsque le signal s de type flot vérifie la comparaison par rapport à la constante k. `op` est un opérateur de comparaison parmi $\{<, \leq, =, \geq, >\}$
activation	$(\texttt{in})\texttt{active}(c)$	vrai lorsque le composant c est (dés)activé
connexion	$c_i.p_j \ll c_k.p_\ell$	vrai lorsque le port p_ℓ du composant c_k est connecté vers le port p_j du composant c_i
littéral	`true`, `false`	constantes booléennes

TAB. 1 – Prédicats utilisables dans les formules de logique.

Tous ces prédicats peuvent être *inversés* (*niés*) de façon naturelle. Par exemple :

– $\text{negation}(s < k) = s \geq k$
– $\text{negation}(\texttt{active}(c)) = \texttt{inactive}(c)$
– $\text{negation}(c_i.p_j \ll c_k.p_\ell) = c_i.p_j \not\ll c_k.p_\ell$

Il est donc possible de transformer simplement toute formule en une formule dans laquelle aucun prédicat n'est directement précédé d'une négation. Cette propriété sera utilisée dans la suite.

Notons qu'à aucun moment le système ne donne de sémantique particulière aux opérateurs de comparaison. Tous les traitements de ces opérateurs sont réalisés de façon purement *syntaxique*. Ainsi, dans une formule du type $s < k$, k est considéré comme une simple chaîne de caractères. Seule est connue du système la règle de réécriture déjà évoquée : $\text{negation}(s < k) = s \geq k$.

[5] En anglais : *canonical safety formulae*.

6.1.3 Opérateurs du fragment passé de la logique temporelle linéaire

Dans nos formules dites *passées*, nous prenons en compte les opérateurs de la logique des propositions et les opérateurs temporels passés classiques :

– logique propositionnelle :
  $\vee$ ou (binaire)
  $\wedge$ et "
  $\rightarrow$ implique " $a \rightarrow b = \neg a \vee b$
  $\neg$ non (unaire)
– opérateurs temporels (passés)
  $\mathcal{S}$ depuis (since) (binaire)
  $\mathcal{B}$ depuis faible (back-to) " $a \mathcal{B} b = (\boxdot a) \vee (a \mathcal{S} b)$
  $\boxdot$ toujours (has-always-been) (unaire)
  $\diamondsuit$ déjà (once) "
  $\odot$ précédent (previous) "

6.1.4 Exemple, relation avec les formules futures

Par exemple, imaginons que nous voulions spécifier que la propriété suivante doit toujours être vérifiée : *si on appuie sur les freins, alors le régulateur n'est pas connecté jusqu'à ce qu'on presse sur le bouton de régulation et que l'on relâche les freins*.

Cette spécification pose problème : elle fait apparaître un opérateur de logique temporelle *future* : *jusqu'à*. Il faut donc la reformuler pour ne faire apparaître que des opérateurs passés : *si depuis un appui sur les freins on n'a pas à la fois appuyé sur le bouton de régulation et relâché les freins, alors le régulateur n'est pas connecté.*

Notons que cela peut se formaliser en un schéma de réécriture plus général : en notant $\mathcal{U}$ l'opérateur *until* (*jusqu'à*), $\Box(a \rightarrow (b \mathcal{U} c))$ équivaut à $\Box((\neg c \mathcal{S} a) \rightarrow b)$. Cependant, en l'absence de tels schémas dans tous les cas, il est préférable de prendre l'habitude d'écrire systématiquement les propriétés de sûreté sous la forme de formules de sûreté canoniques (c'est-à-dire composées uniquement d'opérateurs passés après le *toujours* initial).

Au final, cette propriété s'écrit de la manière suivante dans la syntaxe concrète textuelle d'ADLV, introduite au paragraphe 4.5.3 (la syntaxe abstraite a été présentée au paragraphe 5.4) :

```
always {
  (!(regulation_on && !brakes_on)) since brakes_on =>
  !(injection_command << regulator.regulated_injection_command);
}
```

6.1.5 Mise sous forme canonique des formules non temporelles

Un sous-ensemble des formules de logique temporelles passées présentées plus haut est utilisé dans la description des composants internes en ADLV : les formules non temporelles, c'est-à-dire de logique propositionnelle *pure*. Ces dernières sont utilisées pour indiquer les conditions selon lesquelles des signaux sont émis en sortie de composants internes (clauses **when**). Il sera donc nécessaire de pouvoir reconnaître deux formules *égales*, même si elles sont écrites sous des formes différentes. Pour ce faire, nous introduisons les *formes canoniques*.

6.1.5.1 Forme canonique

Une forme canonique est une écriture *unique* d'une formule. Toute autre écriture de la formule peut être transformée de façon systématique vers la forme canonique.

On considère selon les cas deux formes canoniques possibles : *forme conjonctive* (conjonction de disjonctions) et *forme disjonctive* (disjonction de conjonctions). Ces formes canoniques sont basées respectivement sur les formes normales conjonctive et disjonctive, auxquelles sont appliquées des contraintes supplémentaires afin d'assurer l'unicité de l'écriture.

6.1.5.2 Algorithme général

Pour mettre une expression e sous forme canonique, on applique les étapes suivantes :

1. remplacer e par $\mathrm{developpe}(e)$, définie ci-après au paragraphe 6.1.5.3, et répéter jusqu'à invariance. $\mathrm{developpe}$ est une fonction chargée de mettre en œuvre certaines règles de distributivité, selon que l'on cherche à obtenir une forme canonique disjonctive ou conjonctive ;
2. si e est un *ou* ou un *et*, l'aplatir (i.e. transformer une cascade de *ou* et de *et* binaires en niveaux de *et* et *ou* n-aires) ;
3. traiter les *et* et *ou* n-aires de façon à éliminer d'éventuelles scories.

6.1.5.3 Développement des opérateurs

Les opérateurs *et* et *ou* sont développés de façon spécifique au type de forme canonique souhaitée (disjonctive ou conjonctive). Ainsi, pour obtenir une forme canonique disjonctive, la méthode est la suivante :

– *et* : distribué sur *ou* de façon à obtenir une forme disjonctive (disjonction de conjonctions).
$\mathrm{developpe}(a \wedge (g \vee h)) = (a \wedge \mathrm{developpe}(g)) \vee (a \wedge \mathrm{developpe}(h))$
$\mathrm{developpe}((g \vee h) \wedge b) = (\mathrm{developpe}(g) \wedge b) \vee (\mathrm{developpe}(h) \wedge b)$
$\mathrm{developpe}(a \wedge b) = \mathrm{developpe}(a) \wedge \mathrm{developpe}(b)$
– *ou* : lui n'est pas distribué, car nous construisons une forme disjonctive.
$\mathrm{developpe}(a \vee b) = \mathrm{developpe}(a) \vee \mathrm{developpe}(b)$
– *implication*.
$\mathrm{developpe}(a \rightarrow b) = \neg a \vee b$

Cette méthode est directement dérivée d'une méthode de construction de formes normales. Les autres opérateurs sont traités de façon identique, quel que soit le type de forme canonique souhaité :

– *non* : éliminé par distribution.
$\mathrm{developpe}(\neg(a \wedge b)) = \neg\mathrm{developpe}(a) \vee \neg\mathrm{developpe}(b)$
$\mathrm{developpe}(\neg(a \vee b)) = \neg\mathrm{developpe}(a) \wedge \neg\mathrm{developpe}(b)$
$\mathrm{developpe}(\neg p) = \mathrm{negation}(p)$, où p est un prédicat
$\mathrm{developpe}(\neg e) = \neg\mathrm{developpe}(e)$
– *prédicats*.
$\mathrm{developpe}(e)$ vaut e si e est un prédicat.
– *autres*, par exemple un opérateur temporel ($\boxdot$, $\mathcal{B}$, $\diamondsuit$, $\odot$, $\mathcal{S}$)
Le résultat n'est pas défini (levée d'une erreur), car on ne devrait pas rencontrer d'opérateur temporel dans une expression d'une clause **`when`**.

6.1.5.4 *Traitement des opérateurs n-aires*

Les opérateurs n-aires sont construits de façon particulière de façon à tendre vers l'unicité des écritures :

- les opérandes sont classés par ordre lexicographique ;
- un même opérande ne peut pas apparaître plusieurs fois, que ce soit dans un « et » ou un « ou ». En effet, $a \wedge a = a$, et $a \vee a = a$.

Cependant, ces règles n'assurent pas à elles seules l'unicité des écritures. Après la construction structurelle de la forme canonique, il reste donc à appliquer une étape de *simplification* sur les opérateurs restants (« et » et « ou » n-aires). En effet, des *scories* peuvent subsister dans les formules. Par exemple, dans un « et » n-aire :

- il peut figurer des constantes : $a_1 \wedge a_2 \cdots \wedge a_n \wedge$ `true` se simplifie en $a_1 \wedge a_2 \cdots \wedge a_n$; $a_1 \wedge a_2 \cdots \wedge a_n \wedge$ `false` se simplifie en `false` ;
- il peut figurer un terme et son inverse : $a \wedge \neg a \wedge \ldots$ se simplifie en `false` ;
- une conjonction n-aire à zéro terme se simplifie en `true`.

Des règles équivalentes existent pour les disjonctions n-aires.

L'application récursive de toutes ces règles assure effectivement une unicité des formes canoniques. Mais cela a une autre conséquence très intéressante : *pour une formule qui est une tautologie, la* forme canonique conjonctive *ainsi définie se réduit à* `true`.

6.1.5.5 *Utilisation de la forme canonique pour l'identification des tautologies*

Les observations précédentes conduisent au résultat suivant : une formule f est une tautologie si et seulement si sa forme canonique conjonctive est `true`.

Démonstration. Soit une formule f.

Supposons que f soit une tautologie, et mettons-la sous forme normale conjonctive : $f = f_1 \wedge f_2 \wedge \cdots \wedge f_n$. Alors pour tout i, $1 \leq i \leq n$, f_i est une tautologie.

Écrivons l'un des ces f_i : $f_i = a_{i,1} \vee a_{i,2} \vee \cdots \vee a_{i,p_i}$. Comme f_i est une tautologie, alors :

- soit il existe deux indices k et ℓ tels que $a_{i,k} = \neg a_{i,\ell}$, et donc l'expression f_i se simplifie en `true` ;
- soit l'un des termes est égal à la constante `true`, et donc f_i se simplifie également en `true`.

Au final, pour tout i, $1 \leq i \leq n$, f_i se simplifie en `true`. f est donc une conjonction de zéro ou plus opérandes se simplifiant en `true`. La forme canonique de f est donc `true`.

La réciproque est immédiate : si f a pour forme canonique `true`, f est une tautologie.

□

6.2 Des formules de logique temporelle aux formes intermédiaires

6.2.1 Vue d'ensemble

Le point de départ est le modèle ADLV, construit à partir du fichier d'entrée.

On considère alors chacune des formules de logique temporelle contenues dans les clauses **`never`**. L'idée générale est de construire pour chaque formule et sous-formule un *signal* émis lorsque la formule est vérifiée (et maintenu tant qu'elle l'est). Il suffit alors d'émettre un signal `failure` (ou autre) lorsque le signal associé à la formule de la clause **`when`** est présent. Pour une formule f d'une clause **`always`**, on procède comme ci-dessus, mais avec $\neg f$ (car `always`$\{\Box f\}$ équivaut à `never`$\{\neg f\}$).

Le problème à résoudre est donc le suivant :

> Soit une formule de logique temporelle f portant sur des signaux d'entrée et de sortie *de l'application*. Il s'agit de construire un signal s lui correspondant, construit à partir d'événements d'entrée et de sortie *du composant de contrôle*, réalisé par des modules écrits dans le langage de ce dernier (Esterel, Lustre, etc.)

Ce problème se décompose en deux sous-problèmes :

1. transformer les conditions impliquant les signaux d'entrée et de sortie de l'application en conditions portant sur les événements d'entrée et de sortie du composant de contrôle. La majeure partie de notre travail porte sur ce point, et fait l'objet de la présente section ;
2. générer un observateur dans le langage cible à partir d'une formule de logique temporelle. Or il a été démontré que la traduction d'opérateurs de logique temporelle en langages synchrones peut se faire assez simplement [Halbwachs 1992, Jagadeesan 1995]. Les principes de fonctionnement dans notre cas seront donnés dans la section 6.3.

De cette étude du problème nous déduisons ceci : nous pouvons temporairement conserver une notation fondée sur des formules de logique (qui sera facilement traduisible en langage cible plus tard), mais il est nécessaire de transformer les références à des signaux de l'application en signaux du composant de contrôle. Voyons donc où apparaissent les signaux de l'application dans les formules de logique. On constate facilement que l'on peut les trouver dans les *prédicats*, précisément ceux de type signal, comparaison, activation et connexion. Examinons la façon dont on peut traiter ces prédicats :

– les signaux sont des événements ; ils peuvent se ramener aux événements du contrôleur en suivant les connexions internes à l'application ;
– les comparaisons portent sur des signaux de type flot. Ces derniers sont traités par des composants internes (placés en amont du contrôleur) qui produisent des événements lorsque les comparaisons deviennent vraies ou fausses. Un prédicat de type comparaison est donc vrai *entre* les occurrences d'un ensemble d'événements « de début » et un ensemble d'événements « de fin » ;
– les activations et les connexions sont réalisées soit à l'initialisation de l'application, soit par des composants internes placés en aval du contrôleur, et qui effectuent les (dés)activations et (dé)connexions sur réception d'événements produits en sortie du contrôleur. Ici aussi, ces prédicats sont vrais entre les occurrences d'un ensemble d'événements « de début » et un ensemble d'événements « de fin ».

Nous devons donc définir des valeurs de vérité vraies entre l'occurrence de deux ensembles d'événements. Nous introduisons pour cela un nouveau type de prédicat, l'*intervalle*. Un prédicat intervalle $[u, v[$ sera vrai si l'événement u est survenu, mais que l'événement v n'est pas encore survenu. Pour spécifier complètement un intervalle, il est également nécessaire d'indiquer si l'on se trouve dans l'intervalle au moment de l'initialisation du système. Si c'est le cas, on utilisera une notation légèrement différente : $[^*u, v[$. La figure 17 illustre les différences entre l'intervalle $[u, v[$ et l'intervalle $[^*u, v[$.

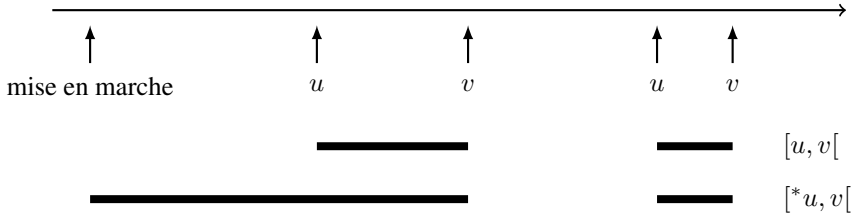

FIG. 17 – Différences entre les intervalles $[u, v[$ et $[^*u, v[$.

Notons que la valeur de vérité d'un simple événement s peut même être assimilée à celle de l'intervalle $[s, \bar{s}[$, ce qui ajoute à la cohérence de l'ensemble. Ainsi, résoudre le sous-problème numéro 1 *revient globalement*

à remplacer dans la formule de départ tous les prédicats non constants par des prédicats de type intervalle portant sur des signaux en entrée ou en sortie du contrôleur.

Une formule de logique temporelle dans laquelle tous les prédicats non constants auront été remplacés par des intervalles sera appelée *forme intermédiaire*. Par opposition, les formules de logique temporelle spécifiées par l'utilisateur dans des clauses **`never`** ou **`always`** sont appelées *formules de départ*. Les formules des deux types sont constituées d'opérateurs propositionnels et temporels ; elles diffèrent par les types de prédicats possibles (voir table 3 plus bas).

Nous verrons plus loin qu'il n'est pas toujours possible de trouver une forme intermédiaire équivalente à la formule de départ. Cependant, il est parfois possible d'*approcher* cette dernière, de l'*encadrer* par *deux* formes intermédiaires, l'une trop stricte, l'autre trop lâche. Ces encadrements seront formellement définis au paragraphe 6.2.3.

Notre algorithme général ne produit donc pas directement des formes intermédiaires, mais des *proto*-formes intermédiaires construites à partir de *proto*-intervalles. Dans le cas où l'on obtient une formule *équivalente* à la formule de départ, la proto-forme intermédiaire est simplement égale à la forme intermédiaire. Dans les cas où l'on ne peut proposer qu'une solution approchée, la proto-forme intermédiaire correspond au meilleur encadrement possible de la formule de départ. Les proto-formes intermédiaires sont extensivement traitées au paragraphe 6.2.4. Nous y donnons un algorithme de passage d'une proto-forme intermédiaire à une ou deux forme intermédiaires. Le tableau 2 présente les nouveaux prédicats intervalle et proto-intervalle. Le tableau 3 donne une comparaison entre formules de départ, proto-formes intermédiaires et formes intermédiaires.

Type	Forme	Signification
intervalle	$[s_i, s_j[$	vrai lorsqu'on a observé le signal s_i (de type événement), mais pas encore le signal s_j (de type événement)
proto-intervalle	(A_I, A_O)	approximation de la valeur de vérité d'une formule par un intervalle trop petit A_I (intérieur) et un intervalle trop grand A_O (extérieur)

TAB. 2 – Prédicats à usage interne, s'ajoutant aux prédicats déjà définis sur le tableau 1.

Type de formule	Signal	Comparaison	Activation	Connexion	Littéral	Proto-intervalle	Intervalle
Formule de départ	×	×	×	×	×		
Proto-forme intermédiaire					×	×	
Forme intermédiaire					×		×

TAB. 3 – Types de prédicats autorisés dans les trois types de formules

6.2.2 Algorithme de passage à une proto-forme intermédiaire

Cet algorithme, qui associe à une formule de départ une forme intermédiaire équivalente, occupe une place centrale dans notre système de vérification. En effet, il est chargé d'analyser la description en ADLV afin d'en extraire une *méthode de vérification* que constitue la forme intermédiaire. Ensuite, cette forme intermédiaire ne subit plus qu'une transposition *mécanique* vers l'un ou l'autre des langages cibles.

Le cœur de cet algorithme se trouve la fonction `pfi` (« proto-forme intermédiaire ») suivante.

Fonction `pfi`(f) :

1. si f est un prédicat d'activation, du type `(in)active`(c), on cherche les conditions d'activation et de désactivation du composant c. Ces conditions correspondent à des événements reçus par un composant

interne, lui-même connecté à des ports du contrôleur. On peut donc renvoyer un intervalle, dont les bornes sont les ports du contrôleur qui provoquent l'activation et la désactivation du composant c. Ce composant peut être activé ou non lors de l'initialisation du système, ce qui détermine la présence initiale dans l'intervalle.

2. si f est un prédicat de connexion, on procède à peu près comme ci-dessus. On cherche les conditions de connexion et de déconnexion correspondantes, en termes d'événements issus de ports du contrôleur. On renvoie un intervalle dont les bornes sont les ports du contrôleur qui provoquent la connexion et la déconnexion des ports spécifiés. La connexion peut exister ou non lors de l'initialisation du système, ce qui détermine la présence initiale dans l'intervalle.
3. si f est un prédicat de type *signal* (s), soit s est un port du contrôleur, auquel cas on note $p = s$, soit on cherche le port p du contrôleur qui est connecté à s. Si on trouve un tel p, le résultat est l'intervalle $[p, \bar{p}[$.
4. on essaie alors de trouver la formule f et sa négation $\neg f$ en tant que condition d'une ou plusieurs clauses **`when`** d'un composant interne de la description ADLV. Si les ports du contrôleur connectés à ces ports internes sont notés p_f et $p_{\neg f}$, alors on renvoie comme résultat un intervalle du type $[p_f, p_{\neg f}[$. Pour l'identification de f et $\neg f$ dans des clauses **`when`**, on utilise la mise des formules sous forme canonique, décrite au paragraphe 6.1.5.1.

   Il peut également arriver que la recherche dans les clauses **`when`** donne un résultat *approché*, c'est-à-dire un *encadrement* de f. Dans ce cas, on stocke temporairement ce résultat approché dans une variable *approx*, et on passe à la suite.
5. si f est de la forme $a \star b$ où $\star$ est un opérateur binaire, alors on essaie de déterminer $a' = \texttt{pfi}(a)$ et $b' = \texttt{pfi}(b)$. Si a' et b' existent :
   - si *approx* n'est pas définie, alors on renvoie $a' \star b'$;
   - si *approx* est définie, et a' et b' sont des équivalences exactes de respectivement a et b, alors on renvoie $a' \star b'$;
   - sinon, on renvoie *approx*.
6. de même, si f est de la forme $\star\ a$ où $\star$ est un opérateur unaire, alors on essaie de déterminer $a' = \texttt{pfi}(a)$. Si a existe :
   - si *approx* n'est pas définie, alors on renvoie $\star\ a'$;
   - si *approx* est déinie, et que a' est une équivalence exacte de a, alors on renvoie $\star\ a'$;
   - sinon, en renvoie *approx*.
7. si l'on arrive à ce point, la fonction `pfi` échoue.

Remarque 1. Notons bien que les proto-formes intermédiaires approchées n'apparaissent que lors de la recherche de f dans des clauses **`when`** (étape 4). Dans ce cas, on ne renvoie pas la proto-forme intermédiaire approchée tout de suite : on essaie d'abord de voir s'il est possible de renvoyer une proto-forme intermédiaire exacte en décomposant la formule f (étapes 5 et 6). De cette façon, nous privilégions toujours les proto-formes intermédiaires exactes (qui donnent lieu directement à une forme intermédiaire équivalente à la formule de départ), et n'avons recours à des formes approchées qu'en dernière possibilité.

Remarque 2. La décomposition des formules dans les étapes 5 et 6 se fait directement selon leur structure sous forme d'arbre résultant à la fois de l'écriture par l'utilisateur et de la phase de *parsing*. Nous ne réorganisons pas les formules selon les règles d'associativité, commutativité ou distributivité.

Ainsi, si nous considérons une formule f de la forme $a \vee (b \wedge c)$, nous allons essayer de traiter successivement les sous-formules $a \vee (b \wedge c)$ (formule f en entier), a, $b \wedge c$, b et c. Nous ne nous intéresserons jamais, par exemple, à la formule $a \vee b$, qui est pourtant une sous-formule de f si l'on écrit cette dernière $f = (a \vee b) \wedge (a \vee c)$.

Nous proposons deux justifications principales pour ce choix de comportement :

1. cela permet à l'algorithme de rester d'une complexité raisonnable ;

2. les sous-formules qui sont recherchées dans la description ADLV sont celles qui ont été écrites par l'utilisateur dans les clauses **never** et **always**. Or, on peut raisonnablement imaginer que cette personne tendra à toujours utiliser les mêmes notations, que ce soit dans les clauses de vérification ou dans les clauses **when**. La recherche des sous-formules a donc toutes les chances d'aboutir si nous restons proches des notations de l'utilisateur.

Cette recherche de sous-expressions rejoint des problématiques traitées en compilation. En effet, on cherche à éliminer les calculs redondants de sous-expressions dans la forme intermédiaire des programmes. Ces optimisations sont appelées en anglais *Global Common Subexpression Elimination* (GCSE) [Cocke 1970, Aho 2006]. Cependant, cette méthode se base elle aussi sur l'écriture a priroi des sous-expressions, sans chercher à les réorganiser par associativité ou commutativité, ce qui justifie encore notre choix.

6.2.3 Traitement des clauses **when**

Nous traitons dans cette section de la recherche d'une formule de départ f dans un certain nombre de clauses **when**. En particulier, nous nous intéressons à la recherche d'*encadrements* de f qui ont été évoqués plus haut.

On suppose que l'on dispose de n clauses **when**, parmi lesquelles on recherche f :

$$\begin{cases} s_1 & \texttt{when} & a_1 \\ s_2 & \texttt{when} & a_2 \\ & \vdots & \\ s_n & \texttt{when} & a_n \end{cases}$$

Nous cherchons à associer un *intervalle* à f ; il s'agit donc de trouver deux bornes (une borne « gauche », de début, et une borne « droite », de fin), correspondant aux périodes auxquelles f est vraie. La borne gauche correspond au moment où f devient vraie après avoir été fausse, et la borne droite au moment où f devient fausse après avoir été vraie (ou encore, ce qui est équivalent, au moment où $\neg f$ devient vraie après avoir été fausse). On essaie donc de trouver la formule f, ou sa négation $\neg f$ en tant que condition d'une clause **when** de la description ADLV.

Afin de déterminer la borne gauche, on commence par établir des listes des signaux s_i pour lesquels $a_i = f$, pour lesquels $a_i \Rightarrow f$, et pour lesquels $f \Rightarrow a_i$. Pour la borne droite, on procédera de la même façon avec $\neg f$.

Déterminer ces égalités et implications est simple à l'aide des formes canoniques : $u = v$ ssi u et v ont la même forme canonique (conjonctive ou disjonctive) ; $u \Rightarrow v$ ssi la formule $u \rightarrow v$ (qui est égale à $\neg u \vee v$) est une tautologie, i.e. ssi la forme canonique conjonctive de $\neg u \vee v$ est `true` (voir paragraphe 6.1.5.5).

On dispose donc d'une relation d'ordre partiel entre formules ($\cdot \Leftarrow \cdot$), et d'une relation d'équivalence associée ($\cdot = \cdot$).

À chaque formule a_i correspond un signal s_i. À la formule f on associe le « signal » spécial I^+ (borne gauche de l'intervalle), et à la formule $\neg f$ le signal I^- (borne droite de l'intervalle). L'intervalle envisagé est donc de la forme $I = [I^+, I^-[$. À toute formule correspond donc un signal. On note φ la fonction qui à une formule associe le signal correspondant.

Ceci nous permet de définir des relations d'ordre partiel ($\cdot \preceq \cdot$) et d'équivalence ($\cdot \leftrightarrow \cdot$) sur les signaux, par morphisme de structure à partir de l'ensemble des formules :

– $a \Leftarrow b$ ssi $\varphi(a) \preceq \varphi(b)$;
– $a = b$ ssi $\varphi(a) \leftrightarrow \varphi(b)$.

Si $b \Rightarrow a$, cela veut dire qu'au moins les conditions a doivent être réunies pour que b soit vérifiée. En termes d'événements, cela veut dire que l'événement qui déclenche b ne peut pas se produire avant celui qui déclenche a. Donc le signal $\varphi(b)$ doit intervenir *après* le signal $\varphi(a)$. Ceci donne donc une signification temporelle simple à la relation $\preceq$: il s'agit d'un ordre sur l'occurrence les signaux. De même, la relation d'équivalence $\leftrightarrow$ s'interprète en termes de concomitance de signaux.

On est donc capable d'identifier les signaux s_i qui se trouvent *avant* I^+ ($s_i \preceq I^+$), ceux qui se trouvent *après* ($s_i \succeq I^+$), et ceux qui ont lieu *en même temps* ($s_i \leftrightarrow I^+$). Mieux, entre deux signaux s_i et s_j qui, par exemple, se trouvent avant I^+, on peut, s'il y a lieu, repérer des relations d'ordre (par exemple, il est possible que $s_i \preceq s_j$). *Bien entendu, ceci s'applique également à I^- ou tout autre signal de « référence ».*

Globalement, on peut donc construire un schéma de type *diagramme de Hasse*, dont un exemple est donné sur la figure 18.

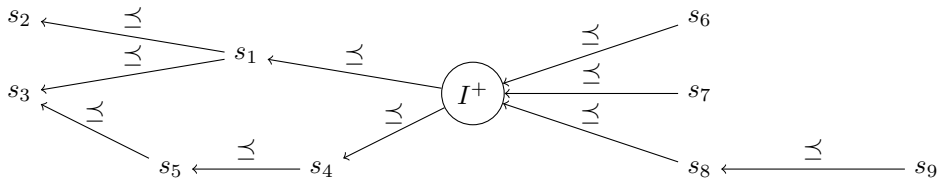

FIG. 18 – Exemple de « diagramme de Hasse » pour le signal I^+. On pourrait construire un diagramme du même type pour le signal I^-.

Il reste à déterminer ce qu'il est possible de faire des signaux que nous avons identifiés. Tout d'abord, parmi les signaux situés « avant » I^+, il ne sert à rien de tous les prendre en considération. On a tout intérêt à ne tenir compte que des signaux les « plus proches » de I^+, donc en l'occurrence s_1 et s_4. Ces derniers sont les signaux qui réalisent une meilleure approximation de I^+. Formellement, on note cet ensemble de signaux S_O^+ (notation expliquée par la suite), et on le définit ainsi ($\mathbb{S}$ est l'ensemble de tous les signaux issus de l'étude des clauses **`when`**) :

$$S_O^+ = \left\{ s \in \mathbb{S} | s \preceq I^+ \wedge \neg \left(\exists s' \in \mathbb{S}, s \preceq s' \preceq I^+ \right) \right\}$$

De même, on définit les ensembles suivants :

Définition formelle	Signification
$S_I^+ = \{s \in \mathbb{S} \| I^+ \preceq s \wedge \neg (\exists s' \in \mathbb{S}, I^+ \preceq s' \preceq s)\}$	signaux les plus proches (minimaux) après I^+
$S_I^- = \{s \in \mathbb{S} \| s \preceq I^- \wedge \neg (\exists s' \in \mathbb{S}, s \preceq s' \preceq I^-)\}$	signaux les plus proches (maximaux) avant I^-
$S_O^- = \{s \in \mathbb{S} \| I^- \preceq s \wedge \neg (\exists s' \in \mathbb{S}, I^- \preceq s' \preceq s)\}$	signaux les plus proches (minimaux) après I^-

La figure 19 reprend la figure 18, sur laquelle on fait figurer les ensembles S_I^+ et S_O^+.

Soit maintenant quatre signaux $s_O^+ \in S_O^+$, $s_I^+ \in S_I^+$, $s_I^- \in S_I^-$ et $s_O^- \in S_O^-$. La position relative de ces signaux est représentée sur la figure 20.

On constate que l'on dispose de deux approximations de I :

– une approximation « extérieure » $[s_O^+, s_O^-[$ (l'indice « O » signifie *outer*) ;
– une approximation « intérieure » $[s_I^+, s_I^-[$ (l'incide « I » signifie *inner*).

Notons que cet encadrement de I est valable quels que soient les signaux pris dans leurs ensembles respectifs. En particulier, on a :

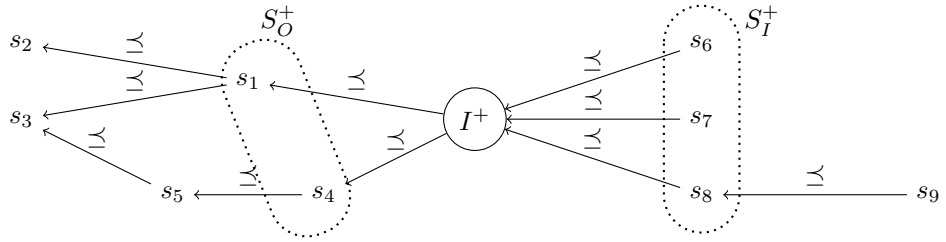

FIG. 19 – Ensembles S_I^+ et S_O^+ sur le diagramme de Hasse de la figure 18.

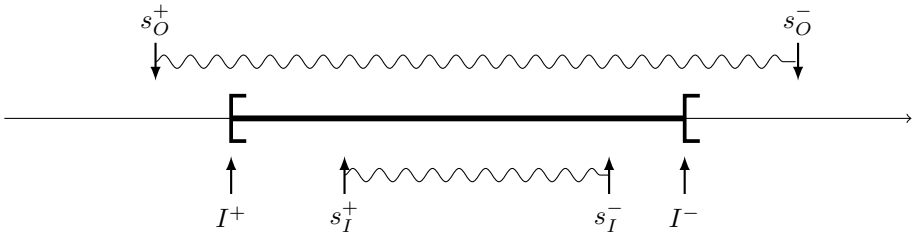

FIG. 20 – Intervalle étudié et approximations de ses bornes.

$$\underbrace{\bigcup_{\substack{s_I^+ \in S_I^+,\\ s_I^- \in S_I^-}} [s_I^+, s_I^-[}_{A_I} \quad \subset \quad I \quad \subset \quad \underbrace{\bigcap_{\substack{s_O^+ \in S_O^+,\\ s_O^- \in S_O^-}} [s_O^+, s_O^-[}_{A_O}$$

Soit E un ensemble de signaux. On définit les signaux *first* et *last* de la façon suivante :
– $\mathrm{first}(E)$ est émis lorsque le premier (au sens temporel) des signaux de E est émis ;
– $\mathrm{last}(E)$ est émis lorsque le dernier (au sens temporel) des signaux de E est émis.
Notons bien que ces notions de *premier* et *dernier* s'entendent pour chaque occurrence temporelle individuelle, car par construction, les signaux des ensembles considérés *ne sont pas comparables* dans l'absolu.

On considère qu'il est possible de « réarmer » ce « mécanisme » à tout moment pour une prochaine émission. La figure 21 illustre les modalités d'émission de ces signaux sur un exemple.

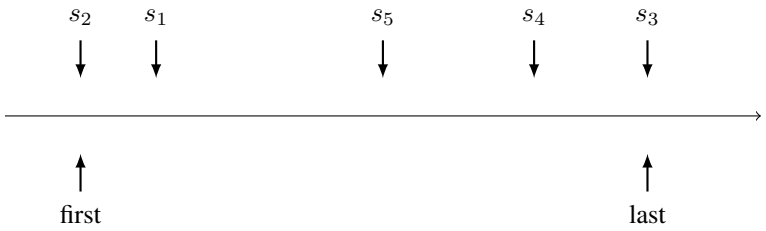

FIG. 21 – Signaux *first* et *last* pour une occurrence des signaux de l'ensemble de signaux $E = \{s_1, s_2, s_3, s_4, s_5\}$.

Avec ces définitions, on peut facilement réécrire la définition ci-dessus intervalles A_I et A_O :

$$\underbrace{[\text{first}(S_I^+), \text{last}(S_I^-)[}_{A_I} \subset I \subset \underbrace{[\text{last}(S_O^+), \text{first}(S_O^-)[}_{A_O}$$

Il est possible de percevoir ce résultat d'une autre manière. Pour tous les $s_I^+ \in S_I^+, s_I^- \in S_I^-, s_O^+ \in S_O^+, s_O^- \in S_O^-$, on a (voir figure 20) :

$$\begin{cases} s_O^+ \preceq I^+ \preceq s_I^+ \\ s_I^- \preceq I^- \preceq s_O^- \end{cases}$$

On cherche une *meilleure approximation possible* de I^+ et I^-. Donc dans les membres de gauche on prend les *derniers* signaux (*last*), et dans les membres de droite les *premiers* (*first*). □

Précision sur la notation employée. A_I s'entend de la façon suivante : « l'intervalle qui débute lorsqu'on détecte pour la première fois l'un des signaux de S_I^+ et qui se termine lorsqu'on détecte le dernier des signaux de s_I^-, sachant que chaque détection est réarmée lorsque l'autre a lieu. » Ainsi, les ensembles S_I^+ et S_I^- font partie intégrante de la définition de l'intervalle A_I. Les mêmes considérations s'entendent pour A_O.

On est donc capable de traduire la formule de départ f dans le cas général. Ceci précise l'étape n°4 de l'algorithme du paragraphe 6.2.2 :

- s'il existe $s^+ \leftrightarrow I^+$ et $s^- \leftrightarrow I^-$, alors l'intervalle $[s^+, s^-[$ convient ;
- sinon, on essaie de définir les intervalles A_I et/ou A_O. Le résultat est alors le couple (A_I, A_O), que l'on qualifie de *proto-intervalle* ;
- sinon, on passe à l'étape suivante (n°5) dans l'algorithme du paragraphe 6.2.2.

6.2.4 Proto-intervalles et proto-formes intermédiaires

6.2.4.1 Définitions

Un proto-intervalle (A_I, A_O) constitue un encadrement de l'intervalle associé à une formule f. f étant destinée à apparaître dans une clause `never`(f), examinons la signification des intervalles A_I et A_O.

Tout d'abord, si $A_I = A_O$, nous nous trouvons dans un cas un peu particulier, où le proto-intervalle peut être considéré comme un intervalle. Dans ce cas, le proto-intervalle correspond à une équivalence exacte ; il ne réalise pas d'approximation. Plus précisément, l'ensemble des intervalles peut être *plongé* trivialement dans l'ensemble des proto-intervalles via la fonction $I \mapsto (I, I)$.

Nous considérons à partir de maintenant que $A_I \neq A_O$.

Si on génère un code de vérification fondé sur A_I, on court le risque de ne pas détecter certains cas d'erreur. Par contre, si le test échoue, on a à coup sûr détecté une *vraie erreur*. On parle de *sous-vérification*.

Si on génère un code de vérification fondé sur A_O, on peut être amené à détecter des erreurs qui n'en sont pas (*faux positifs*). Par contre, si le test réussit, on est sûr que le code vérifié est correct. On parle de *sur-vérification*.

On le voit, le recours à des proto-intervalles ne fournit que des résultats *approchés*, soumis à interprétation. Ce n'est donc qu'un pis-aller, qui ne doit être utilisé qu'en dernière nécessité.

Ces considérations sont valables pour des formules f *directement* associées à des proto-intervalles. Il est nécessaire d'étudier le cas plus général de formules construites à partir d'intervalles, de proto-intervalles, et d'opérateurs de logique temporelle et classique. On appelle de telles formules des *proto-formes intermédiaires*.

6.2.4.2 *Réécriture d'une proto-forme intermédiaire en une ou deux formes intermédiaires*

Le but est de définir une réécriture des proto-formes intermédiaires en formes intermédiaires qui ne contiennent que des intervalles et des opérateurs logiques. Une proto-forme intermédiaire donnera lieu à *deux* formes intermédiaires (au plus), l'une correspondant à des conditions trop faibles, appelée « extérieure » et notée f'_O, et l'autre correspondant à des conditions trop fortes, appelée « intérieure » et notée f'_I. On notera : $f' \rightsquigarrow (f'_I, f'_O)$ (f', proto-forme intermédiaire associée à la formule f, se réécrit en f'_I et f'_O).

Pour un proto-intervalle $P = (A_I, A_O)$, on a simplement :

$$(A_I, A_O) \rightsquigarrow (A_I, A_O)$$

Méthode générale de démonstration

Une proto-forme intermédiaire f', associée à une formule de départ f, se réécrit en (f'_I, f'_O).

D'après ce qui précède, nous avons les deux jeux de relations suivants, équivalentes via le morphisme φ :

$$(a)\begin{cases} \varphi(f'_O) & \preceq & \varphi(f) & \preceq & \varphi(f'_I) \\ \varphi(\neg f'_I) & \preceq & \neg\varphi(f) & \preceq & \varphi(\neg f'_O) \end{cases}, \qquad (b)\begin{cases} f'_O & \Leftarrow & f & \Leftarrow & f'_I \\ \neg f'_I & \Leftarrow & \neg f & \Leftarrow & \neg f'_O \end{cases}$$

Le jeu (a) traite de l'ordre des signaux, et le jeu (b) permet d'aboutir aux relations d'implication entre formules.

La première ligne de chaque jeu caractérise l'ordre du *début* des intervalles ; la deuxième ligne l'ordre de leur *fin*. Or, on constate sur le jeu (b) que les deux lignes sont équivalentes : la deuxième n'est autre que la négation de la première. Dans la suite, on ne tiendra donc compte que de la première ligne.

Ces relations ont en réalité été écrites par *équivalence*, donc on a de façon *réciproque* le résultat suivant.

Soit une formule f. S'il existe des formes intermédiaires g et h qui vérifient :

$$g \Leftarrow f \Leftarrow h$$

alors f admet des formes intermédiaires intérieure et extérieure, qui sont respectivement g et h ; autrement dit, $f' \rightsquigarrow (g, h)$. □

Cas de la négation

Soit une formule $g' = \neg f'$, où f' est une proto-forme intermédiaire associée à une formule de départ f, $f' \rightsquigarrow (f'_I, f'_O)$. Une représentation graphique de la situation est donnée sur la figure 22.

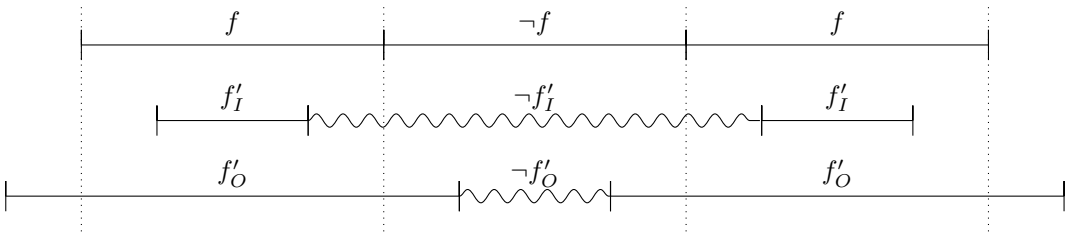

FIG. 22 – Construction des formes intermédiaires pour une proto-forme intermédiaire $g' = \neg f'$.

On voit sur la figure que l'on assiste dans ce cas à un *inversion* des formes intermédiaires intérieure et extérieure. Formellement, on peut écrire :

$$f'_I \Leftarrow f \Leftarrow f'_O$$

En passant à la contraposée il vient :

$$\neg f'_O \Leftarrow \neg f \Leftarrow \neg f'_I$$

On conclut :

$$\text{si } \ f' \rightsquigarrow (f'_I, f'_O) \ \text{ alors } \ \neg f' \rightsquigarrow (\neg f'_O, \neg f'_I)$$

Conjonction, disjonction, implication

Soit f' et g' deux proto-formes intermédiaires associées respectivement aux formules de départ f et g. On suppose que $f' \rightsquigarrow (f'_I, f'_O)$ et $g' \rightsquigarrow (g'_I, g'_O)$.

D'une part, $f'_I \Leftarrow f \Leftarrow f'_O$, et d'autre part $g'_I \Leftarrow g \Leftarrow g'_O$. Par compatibilité de la relation $\Leftarrow$ avec le « et »[6], il vient :

$$f'_I \wedge g'_I \Leftarrow f \wedge g \Leftarrow f'_O \wedge g'_O$$

Au final : $f' \wedge g' \rightsquigarrow (f'_I \wedge g'_I, f'_O \wedge g'_O)$.

Et de même (compatibilité de la relation $\Leftarrow$ avec le « ou ») : $f' \vee g' \rightsquigarrow (f'_I \vee g'_I, f'_O \vee g'_O)$.

Implication. La formule $f' \rightarrow g'$ est égale à $\neg f' \vee g'$. Or $\neg f' \rightsquigarrow (\neg f'_O, \neg f'_I)$. Donc $f' \rightarrow g' \rightsquigarrow (\neg f'_O \vee g'_I, \neg f'_I \vee g'_O)$. Au final :

$$f' \rightarrow g' \rightsquigarrow (f'_O \rightarrow g'_I, f'_I \rightarrow g'_O)$$

Opérateurs temporels unaires

Soit f' une proto-forme intermédiaire associée à une formule de départ f, $f' \rightsquigarrow (f'_I, f'_O)$. On a $f'_I \Leftarrow f \Leftarrow f'_O$.

Soit Ψ un opérateur temporel unaire. D'après [Manna 1992a], tous les opérateurs temporels unaires sont monotones, donc :

$$\Psi(f'_I) \Leftarrow \Psi(f) \Leftarrow \Psi(f'_O)$$

Et donc :

$$\Psi(f') \rightsquigarrow (\Psi(f'_I), \Psi(f'_O))$$

[6] Pour démontrer cette propriété, il suffit d'écrire la table de vérité de la formule $[(a \rightarrow b) \wedge (c \rightarrow d)] \rightarrow [(a \wedge c) \rightarrow (b \wedge d)]$, ce qui permet de constater qu'il s'agit d'une tautologie.

Opérateurs temporels binaires

De même, [Manna 1992a] indique que tous les opérateurs temporels binaires sont monotones.

Soient donc f' et g' deux proto-formes intermédiaires associées respectivement aux formules de départ f et g. On suppose que $f' \rightsquigarrow (f'_I, f'_O)$ et $g' \rightsquigarrow (g'_I, g'_O)$. Soit Ψ un opérateur temporel binaire. Alors :

$$\Psi(f', g') \rightsquigarrow (\Psi(f'_I, g'_I), \Psi(f'_O, g'_O))$$

Conclusion

À l'exception des négations et des implications[7], tous les opérateurs combinent de façon *naturelle* les proto-intervalles. Il n'existe que deux cas particuliers :

– $\neg f' \rightsquigarrow (\neg f'_O, \neg f'_I)$
– $f' \rightarrow g' \rightsquigarrow (f'_O \rightarrow g'_I, f'_I \rightarrow g'_O)$

La méthode générale de transformation des formules de départ fournit des proto-formes intermédiaires construites à base de proto-intervales. Dans cette section, nous avons élaboré un moyen simple pour passer d'une proto-forme intermédiaire à une ou deux formes intermédiaires. Deux cas peuvent se présenter :

1. la proto-forme intermédiaire ne contient que des proto-intervalles *exacts*. Dans ce cas, on obtient au final une seule forme intermédiaire, exacte elle aussi[8] ;
2. la proto-forme intermédiaire contient au moins un proto-intervalle *approché*. Dans ce cas, on obtient au final une forme intermédiaire extérieure et/ou une forme intermédiaire intérieure.

Exemple

Un exemple de transformation d'une proto-forme intermédiaire en deux formes intermédiaires est donnée sur la figure 23. Les règles de réécriture sont appelées progressivement, à chaque fois qu'un opérateur combine un couple de formes intermédiaires (sur la figure, couple de deux formules en gris), jusqu'à obtenir au plus haut niveau un tel couple :

1. la règle de réécriture naturelle de l'opérateur $\vee$ est d'abord appliquée ;
2. ensuite, on peut appliquer celle de l'opérateur $\neg$, qui « croise » les formes intérieures et extérieures ;
3. il ne reste plus ensuite qu'à appliquer celle de l'opérateur $\mathcal{S}$ pour obtenir le résultat.

6.3 Des formes intermédiaires aux observateurs

Une forme intermédiaire est une formule de logique temporelle qui ne doit jamais être vraie. Elle doit donc être traduite en un observateur dans le langage cible, chargé d'émettre un signal d'erreur dans les cas où elle devient vraie. Un outil de vérification du langage cible (`checkblif` pour Esterel, `lesar` pour Lustre, etc.) peut alors soit *prouver* que le signal n'est émis en aucun cas, et donc que la formule ne peut pas être vraie, soit au contraire exhiber un contre-exemple.

Une forme intermédiaire est composée d'opérateurs logiques (combinatoires et temporels passés), de prédicats de type intervalle, et éventuellement de constantes booléennes. Notre système repose sur la définition

[7] Pour ce qui est de l'implication, ceci est dû à l'existence d'une « négation cachée » dans l'implication.

[8] Plus exactement, en suivant la méthode générale, on obtient une forme intermédiaire extérieure et une forme intermédiaire intérieure, identiques.

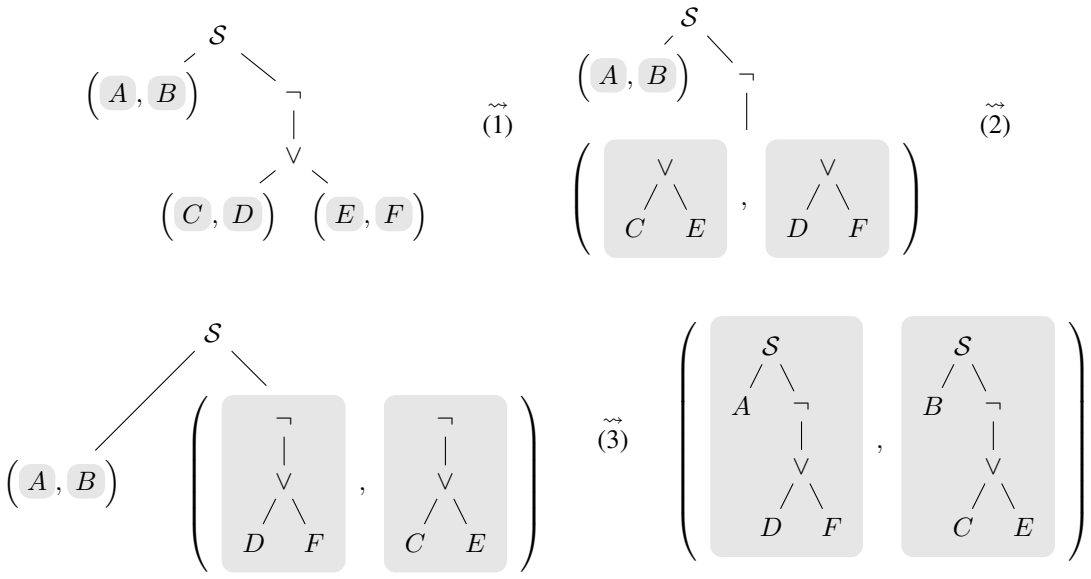

FIG. 23 – Exemple de réécriture d'une proto-forme intermédiaire en un couple de formes intermédiaires. *Les proto-formes intermédiaires sont écrites sur fond blanc, les formes intermédiaires sur fond gris.*

d'un *visiteur*[9] de formes intermédiaires pour chaque langage cible. Ce visiteur se charge de construire pour chaque formule f un signal $\mathcal{S}_f$ dans le langage cible, dont la présence ou l'absence à chaque instant correspond à la valeur de vérité à cet instant de la formule f.

Nous avons réalisé des visiteurs pour les langages Esterel (paragraphe 6.3.2) et Lustre (paragraphe 6.3.3). Cependant, des visiteurs pour d'autres langages peuvent être très facilement ajoutés au système.

6.3.1 Simplification préalable des formes intermédiaires

Il existe deux cas particuliers dans lesquels un bloc `never` $\{f\}$ peut être traité à part. Soit m la forme intermédiaire associée à la formule de départ f.

6.3.1.1 Cas où m est équivalente à `false`

Dans ce cas, m ne sera bien entendu jamais vérifiée, donc le bloc **`never`** ne sert à rien. On peut donc facilement optimiser le programme de vérification obtenu en ne générant rien pour cette clause.

6.3.1.2 Cas où m est équivalente à `true`

Ici, m est *toujours vérifiée*, ce qui fait que le signal d'erreur sera émis quel que soit l'état de l'automate. Dans ce cas, le programme de vérification est quand-même généré, mais un message d'avertissement est fourni à l'utilisateur : on connaît d'ores et déjà l'issue de la phase de vérification, sans même qu'il soit nécessaire de l'effectuer...

[9] Au sens du patron de conception *visiteur* [Gamma 1999].

6.3.1.3 Exemple

On peut aboutir à une forme intermédiaire équivalente à `false` dans le cas d'un bloc **`always`** de la forme suivante :

```
always { active(controller); }
```

Ceci est traité en tant que bloc **`never`** :

```
never { !active(controller); }
```

Or, le contrôleur d'une application est toujours actif dans notre exemple, ce que le moteur de génération de formules intermédiaire est capable de détecter :

– le composant `controller` est vu actif à l'état initial ;
– il n'existe ni transition qui le désactive, ni qui le (ré)active.

La forme intermédiaire de **`active`**`(controller)` est donc la constante `true`, d'où au final un bloc **`never`** sur une condition tout le temps fausse, qui ne donne lieu à aucun code généré.

Notre outil est donc capable d'aboutir à certaines conclusions (certes un peu simplistes !), sans même qu'il soit nécessaire d'utiliser un vérificateur externe.

6.3.2 Générateur de code Esterel

6.3.2.1 Introduction

Esterel [Berry 1992] est un langage réactif synchrone qui dispose des outils nécessaires pour effectuer des *preuves* sur les programmes. En effet, après compilation d'un programme, il est possible avec la commande `checkblif` de vérifier qu'un signal donné n'est jamais émis.

Ceci est bien adapté au traitement des clauses **`never`** (auxquelles sont ramenées par négation les clauses **`always`**). Le module principal de l'observateur est chargé d'émettre un signal *failure* si l'une des expressions des blocs *never* devient vraie.

Un module Esterel est associé à chaque formule ou sous-formule f. Ce module est chargé de calculer un signal $\mathcal{S}_f$ correspondant à f. $\mathcal{S}_f$ est émis à tous les instants où la formule est vérifiée. Le paragraphe suivant donne quelques exemples de génération de ces signaux $\mathcal{S}_f$.

6.3.2.2 Génération des signaux associés aux formules

Intervalles

Un intervalle $I = [s_1, s_2[$ donne lieu trivialement au code suivant :

```
every immediate s1 do
    abort
        sustain S_I
    when s2
end every
```

Ce processus Esterel maintient un signal de sortie $\mathcal{S}_I$ *entre* la réception de deux signaux *événementiels* (début et fin).

Expressions « et »

Une expression $e = a \wedge b$ donne lieu au code suivant :

```
run C_a
||
run C_b
||
[
    every immediate [S_a and S_b] do
        emit S_e
    end every
]
```

`C_a` et `C_b` sont les modules correspondant à respectivement a et b. Leurs signaux de sortie associés sont respectivement `S_a` ($\mathcal{S}_a$) et `S_b` ($\mathcal{S}_b$).

Le calcul des signaux pour des expressions « ou » et « non » est similaire.

Expression temporelle : « since »

Soit l'expression $e = a \; \mathcal{S} \; b$. Le code suivant est directement tiré de [Jagadeesan 1995] :

```
run C_a
||
run C_b
||
[
    every immediate S_b do
        do
            sustain S_e
        watching immediate [not S_a]
    end every
]
```

À chaque fois que l'on rencontre $\mathcal{S}_b$, le signal correspondant à b, on maintient $\mathcal{S}_e$ jusqu'à ce que $\mathcal{S}_a$, le signal correspondant à a, soit absent.

Cela correspond à l'idée intuitive suivante : la formule $a \; \mathcal{S} \; b$ est vraie entre un instant où $a \wedge b$ devient actif et un l'instant suivant où a devient inactif.

6.3.2.3 Relations

Dans le code Esterel du contrôleur, il est généralement nécessaire de faire figurer des *relations* entre signaux, par exemple pour indiquer que deux signaux sont mutuellement exclusifs. Cela donne lieu à des indications du type :

```
relation brakes_pushed # brakes_released;
```

Sous peine de détecter des cas d'erreur qui n'en sont pas, il est nécessaire de faire figurer ces relations dans l'observateur. Le générateur se contente de *recopier* toutes les relations trouvées dans le code du contrôleur.

6.3.2.4 *Fonctionnement de la vérification*

À partir de la description ADLV de l'application, notre outil génère un fichier `check.strl` contenant l'observateur Esterel correspondant aux propriétés de sûreté à vérifier. La procédure de génération suppose que le contrôleur est implémenté dans un module Esterel portant le nom du composant de contrôle de l'application.

Il reste deux étapes à accomplir pour effectuer la vérification :

1. compilation des deux fichiers, le contrôleur et les modules de vérification :
   `esterel -Lblif -B proof controller.strl check.strl;`
2. vérification : `checkblif -output failure -rel proof.rel.blif proof.blif.`

6.3.3 Génération de code Lustre

Pour la réalisation d'un visiteur pour le langage Lustre [Halbwachs 1991], nous avons adopté une stratégie légèrement différente. En effet, nous commençons dans ce cas par définir dans chaque fichier généré une bibliothèque de *nœuds* Lustre, qui est ensuite utilisée pour l'écriture directe des expressions sous forme intermédiaire.

Par exemple, on définit ainsi un nœud chargé de calculer $b \mathcal{S} a$:

```
node since(B, A: bool) returns (B_since_A: bool);
let
B_since_A =
    if A then B
        else if B then true
        else (true -> pre(B_since_A));
tel
```

La définition de ces nœuds est directement inspirée des exemples donnés dans [Halbwachs 1993].

Il est alors possible de combiner directement des instances des nœuds, à la manière d'appels de fonctions, de façon à calculer les signaux associés aux formules des clauses **never**. Par exemple, la traduction de la formule

$$\neg\{(\neg([\text{regulation\_on}, \overline{\text{regulation\_on}}[\wedge [\text{brakes\_released}, \text{brakes\_pushed}[)) \\ \mathcal{S}\ [\text{brakes\_pushed}, \text{brakes\_released}[\} \\ \Rightarrow \neg[\text{start\_speed\_regulation}, \text{stop\_speed\_regulation}[$$

donne :

```
never_3 = not(
  implies(
    since(
      not(
        interval(false, regulation_on, not(regulation_on))
        and
        interval(false, brakes_released, brakes_pushed)
      ), interval(false, brakes_pushed, brakes_released)),
  not(interval(false, start_speed_regulation, stop_speed_regulation))));
```

Enfin, un nœud spécifique nommé `check` est chargé de mettre un signal `ok` à *faux* dans tous les cas où l'une des clauses **never** devient vraie. Le nœud `check` prend donc comme entrées les signaux d'entrée du contrôleur, et possède une sortie `ok`.

Le fichier Lustre contenant le contrôleur est inclus dans le programme de vérification, et le nœud `check` invoque le contrôleur.

Deux versions différentes du nœud `check` peuvent être générées :

– une *version standard*, qui ne possède que la sortie `ok`. Les signaux de sortie du contrôleur sont des signaux locaux ; le code généré est concis car il fait peu usage des signaux locaux ;
– une *version de débogage*, qui possède en plus de la sortie `ok` toutes les sorties du contrôleur, ainsi que des signaux intermédiaires correspondant au calcul des intervalles, et à chacune des clauses *never* individuelles. Cette version est destinée à la simulation : elle permet de se rendre compte facilement des problèmes dans le contrôleur ou les clauses vérifiées.

La vérification d'un code Lustre met en œuvre les étapes suivantes :

1. compilation du vérificateur (qui inclut le contrôleur) : `lustre check.lus check` ;
2. vérification : `lesar check.lus check -forward -diag`. Le paramètre `-forward` correspond à l'algorithme de vérification choisi. Il peut être remplacé par `-enum` ou `-backward` ;
3. en cas de problème, pour effectuer la simulation afin d'identifier la cause de l'erreur : `lux check.oc`, puis `./check`.

6.4 Conclusion sur la vérification

Dans cette section, nous avons décrit une solution complète de vérification des applications ADLV, fondée sur le principe WYPIWYE, de façon à apporter un maximum de sécurité. À partir de propriétés exprimées librement sur les entrées d'une application, notre méthode permet de passer à une vérification sur des événements de modules synchrone, bien prise en charge par les outils desdits langages.

L'architecture de vérification est indépendante du langage : il serait très facile d'ajouter le support d'un nouveau langage synchrone, en plus d'Esterel et Lustre.

7 Outil

7.1 Architecture

L'ensemble des outils ont été réalisés sous Eclipse et écrits en Java. Toutes les classes Java sont dans un sous-paquetage de `fr.supelec.adlv`.

Les différents outils sont :

– L'outil de représentation des modèles ADLV (EMF_ADLV)
– L'outil de traintement de la syntaxe concrète ADLV (Parser_ADLV)
– L'outil de génération pour Inflexion (Generators_ADLV)
– L'outil de génération des observateurs à partir des propriétés (Checker_ADLV)

Les 3 derniers outils utilisent le premier.

7.2 Outil de représentation des modèles ADLV

Cet outil est en fait le modèle EMF de notre syntaxe abstraite augmenté de l'API automatiquement générée.

Dans la version actuelle de l'outil, et en raison de la forte instabilité du modèle dans cette étape de définition et de mise au point du langage ADLV, cette API n'a pas été augmentée de comportements utilitaires facilitant le parcours du modèle, ni a fortiori des comportements nécessaires à la génération vers Inflexion. Ces ajouts sont pour l'instant localisés dans le composant Parser_ADLV.

7.3 Outil de traintement de la syntaxe concrète ADLV

Cet outil est basé sur ANTLR et n'appelle pas de remarques particulières sur l'aspect analyse lexicale et syntaxique.

Cet outil fournit un moyen de charger un fichier ADLV textuel dans l'outil de représentation des modèles ADLV ; d'autres moyens sont possibles, par exemple par transformation de modèles, mais n'ont pas été mis en œuvre dans le cadre de MoDrival.

Outre les classes générées par ANTLR, les classes suivantes sont présentes :
– une classe $EmfAdlvFeeder$ chargée d'encapsuler l'utilisation d'ANTLR,
– une classe $EmfAdlvHelper$ pouvant être considérée comme le conteneur racine des objets ADLV et regroupant toutes les méthodes qui ont été jugées utiles pour extraire de l'information des modèles,
– une classe $TestAdlvParser$ permettant de tester le chargement de modèles.

7.4 Outil de génération pour Inflexion

Cet outil regroupe l'ensemble des classes en charge de générer des fichiers pour Inflexion. On trouve ainsi :
– $AdlvGenerator$: c'est la classe racine de tous les générateurs. Elle regroupe quelques méthodes de portée générale et des constantes configurant certains aspects de la génération (nom du composant « miroir » de l'application, type de déploiement...).
– $MpcGenerator$: c'est la classe chargée de la génération des fichiers MakeProject (fichier workspace `.mwc` et fichiers projets `.mpc`).
– $EventTypeGenerator$: tous les événements ADLV sont convertis en événements IDL et regroupés dans un seul fichier référencé par les autres ; c'est la classe $EventTypeGenerator$ qui se charge de cette action.

– $Idl3Generator$: cette classe se charge de générer les fichiers IDL pour les composants internes et externes, ainsi que pour le composant réalisant le routage dynamique et le composant représentant l'extérieur de l'application (composant « miroir »).
– $CppGenerator$: cette classe génère les fichiers d'entête et la glue des composants externes.
– $InternalComponentImplGenerator$: cette classe génère l'implémentation des composants internes.
– $ApplicationComponentImplGenerator$: cette classe génère l'implémentation du composant de routage dynamique et du composant « miroir ».
– $CcdGenerator$: cette classe génère les fichiers de déploiment (`.ccd` : CORBA Component Descriptors et `.cdp` : Component Deployment Plan).
– $TestGenerator$: classe pour le test de l'ensemble de la chaîne de génération vers Inflexion à partir d'une description textuelle en ADLV d'une application.

Il faut d'autre part préciser que les interfaces qui se substituent aux ports flots de données, ainsi que les duplicateurs de flots, ont été définis une fois pour toutes (fichier de projet `.mpc` pour MakeProject, fichier IDL, fichiers d'implémentation et fichiers de description `.ccd`), et sont supposés se trouver dans un répertoire Supelec pour que la compilation avec Inflexion se passe bien.

7.4.1 Interfaçage avec les outils externes

Le comportement des composants externes n'est pas défini avec ADLV, mais par d'autres outils. Cette section explique les conventions à respecter avec ces outils pour que la glue générée par notre outil s'adapte correctement au code C ou C++ généré par les outils externes. Actuellement, seuls les outils Esterel (composant de contrôle) et Simulink (composants de traitement) sont supportés.

7.4.2 Interfaçage avec Esterel

Les conventions suivantes doivent être respectées :

– le nom du composant ADLV est le nom du module Esterel,
– les signaux d'entrée du module Esterel doivent être déclarés comme ports (avec les mêmes noms) consommateurs d'événements dans le composant ADLV,
– les signaux de sortie du module Esterel doivent être déclarés comme ports producteurs d'événements dans le composant ADLV,
– le fichier C généré à partir du module Esterel doit être placé dans un sous-répertoire nommé esterel après la génération des fichiers Inflexion et avant le lancement de la compilation.

À titre d'exemple, voici la partie interface du module Esterel pour notre régulateur de vitesse (voir la définition ADLV textuelle dans la section 4.4.1) :

```
module controller:
    input regulation_on, regulation_off;
    input brakes_released, brakes_pushed;
    input speed_correct, speed_incorrect;

    output start_speed_regulation, stop_speed_regulation;
    output set_target_speed;

    %
    %  Corps du module de contrôle
    %

end module
```

7.4.3 Interfaçage avec Simulink

Les conventions à respecter pour Simulink sont identiques, en adaptant la terminologie, à celles d'Esterel. Les fichiers C++ générés par l'outil "RealTime-Workshop" (l'interfaçage avec le code C généré n'a pas encore été réalisé) doivent être copiés dans un sous-répertoire simulink du répertoire correspondant au composant Inflexion généré.

La figure 24 montre la conception Simulink de notre régulateur simplifié.

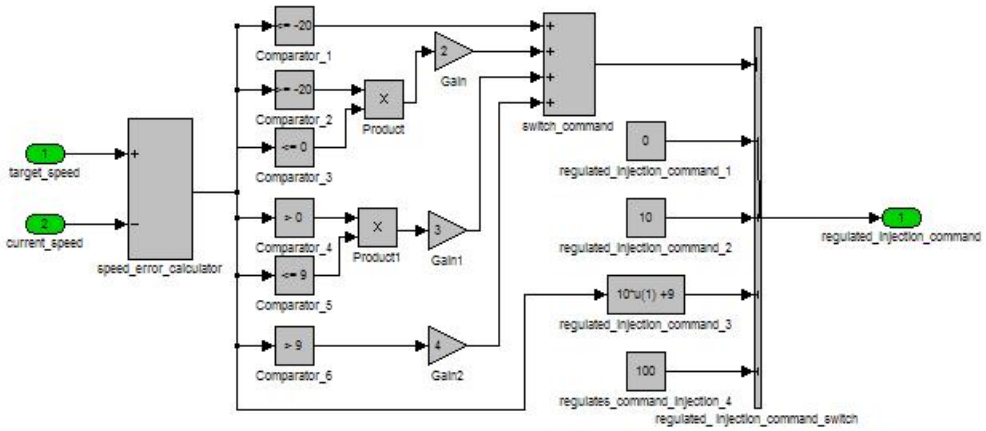

FIG. 24 – Schéma Simulink de la conception du composant régulateur

7.5 Outil de génération des observateurs à partir des propriétés

Cet outil est situé dans un paquetage `fr.supelec.adlv.checker.logic`.

7.5.1 Fonctionnement global

L'outil fondé autour d'une classe nommée `Checker`, qui possède une référence sur un modèle EMF d'une application ADLV. Dans ce modèle, elle recherche toutes les clauses **`never`** et **`always`**. Les expressions logiques que ces dernières contiennent sont converties dans une représentation particulière à l'outil de vérification (voir paragraphe 7.5.2).

Sur ces formules, l'outil applique alors les algorithmes énoncées dans la section 6 afin de construire une proto-forme intermédiaires, puis une ou deux formes intermédiaires.

Un *visiteur* spécifique au langage cible parcourt enfin les formes intermédiaires afin de générer le code de l'observateur associé. Le fonctionnement des visiteurs est décrit au paragraphe 7.5.3.

Un programme principal de démonstration, donné dans la classe `Main`, utilise Parser_ADLV pour charger un fichier ADLV à partir du disque, et de générer les observateurs Esterel et Lustre correspondants.

7.5.2 Représentation des expressions logiques

La représentation des expressions logiques est prise en charge par le sous-paquetage `logic`. Tous les types d'expressions héritent d'une classe `Expr` ; elles se répartissent en trois grandes familles :

- les *prédicats*, aussi bien ceux présents en entrée (`ConnectedPredicate`, `ComparisonPredicate`, etc.), que ceux des proto-formes intermédiaires (`ProtoInterval`) et des formes intermédiaires (`Interval`) ;
- les *opérateurs unaires*, comme `Not`, `Once`, etc. ;
- les *opérateurs binaires* (ex : `And`, `Since`, etc.), à partir desquels sont fabriquées les formules de logique temporelle manipulées ;
- les *opérateurs n-aires*, au nombre de deux, `NaryAnd` et `NaryOr`. Ils sont utilisés uniquement dans les formes canoniques ; toutes les autres formules utilisent des cascades d'opérateurs binaires, pas des opérateurs n-aires.

Les expressions sont munies de méthodes qui permettent de les comparer, de les simplifier, et de savoir si l'on a affaire à une forme intermédiaire (exacte) ou bien à une proto-forme intermédiaire (avec formes intérieure et extérieure).

7.5.3 Visiteurs d'expressions logiques

L'interface `CheckingCodeGenerator` décrit les méthodes que doit offrir un générateur de code pour un langage cible donné. Elle comprend notamment toutes les méthodes d'un *visiteur* de formes intermédiaires.

Nous fournissons trois classes qui implémentent `CheckingCodeGenerator` :

- `EsterelGenerator`, qui génère des observateurs écrits en Esterel. Le fonctionnement de ce visiteur a été décrit au paragraphe 6.3.2 ;
- `LustreGenerator`, qui génère des observateurs écrits en Lustre. Le fonctionnement de ce visiteur a été décrit au paragraphe 6.3.3. Ce générateur peut soit produire un observateur très compact (mode « normal »), soit produire un observateur plus détaillé (mode « debug »), utile par exemple pour obtenir un contre-exemple détaillé lorsqu'une propriété n'est pas vérifiée ;
- `DummyStringGenerator`, un visiteur très simple qui écrit sur la console une représentation textuelle des formules de logique. Ce visiteur est un *exemple* qui peut être utilisé comme base afin de créer un visiteur pour un nouveau langage cible.

Afin de faciliter le développement de visiteurs pour d'autres langages, nous fournissons en outre une classe abstraite `DefaultAbstractCheckingCodeGenerator`, qui implémente `CheckingCodeGenerator`, et fournit une implémentation de base de certains opérateurs ($a \rightarrow b$ est par exemple défini à partir de $\wedge$ et $\neg$, sous la forme : $\neg a \wedge b$). Ceci permet de ne gérer que les opérateurs de base dans un premier temps, en utilisant les règles classiques de logique pour gérer les autres. Dans un deuxième temps, et après mise au point de la première partie, il est alors possible de gérer de façon *optimisée* les opérateurs restants.