Intégration des outils Persée
(Proposition d'architecture)
Déliverable 3.1 du projet PERSÉE - Partie 1

S. Bardin and F. Herbreteau and M. Sighireanu and G. Sutre and A. Vincent

2 juin 2005

Participants au groupe "Intégration" : A. Griffault, F. Herbreteau, G. Point, G. Sutre, A. Vincent (Labri), M. Sighireanu (Liafa) et S. Bardin, A. Finkel, D. Nowak (LSV)

1 Introduction

L'embarquement d'applications informatiques dans des systèmes critiques a conduit au développement de modèles et de techniques de vérification automatique depuis une vingtaine d'années [22, 48, 21, 49]. Lorsque le modèle est fini, les propriétés à vérifier sont généralement décidables, mais les algorithmes énumératifs usuels se heurtent au problème de l'explosion combinatoire de l'espace d'états. De nombreuses techniques [25, 46, 18, 47, 34, 40, 43, 33] ont été développées pour contourner ce problème, conduisant à des outils efficaces [17, 39]. L'une d'entre elles [18] a introduit l'idée de la *vérification symbolique* qui consiste à passer d'une représentation énumérative à une représentation symbolique pour les ensembles d'états.

Cette technique a non seulement permis un bond significatif dans la taille des modèles vérifiables, mais elle a aussi donné naissance aux techniques de vérification de systèmes infinis. La vérification de systèmes infinis pose non seulement le problème de la représentation d'ensembles infinis d'états, mais aussi leur calcul. Or, justement, les systèmes infinis font apparaître un nouveau problème : très souvent, les problèmes de vérification sont indécidables. Deux approches sont alors utilisées : soit on se restreint à des sous-classes de systèmes infinis pour lesquelles le model-checking est décidable [42, 41, 19, 32], soit on utilise des semi-algorithmes [9, 3, 16, 8, 14, 5, 30, 31, 13, 2].

La première approche se base soit sur la construction d'un modèle fini permettant de conclure [4], soit sur des techniques de troncature [29] qui permettent l'examen d'une partie finie du modèle. La première approche se base sur le développement de méthodes capturant, partiellement et éventuellement de façon approchée, les causes de l'indécidabilité, comme le *widening* [27, 3, 5] ou les méthodes d'*accélération* [9, 30, 31, 13, 2]. Ces techniques ont permis l'élaboration d'outils [12, 38, 6, 11] qui ont montré leur pertinence sur de nombreux exemples [26, 31, 15].

Or, ces méthodes ont été développées pour la vérification de modèles homogènes, bien souvent des automates finis étendus avec des variables d'un type fixé (par exemple, des files ou des compteurs). Mais la plupart des systèmes réels sont naturellement plus riches : ainsi, un protocole de communication, comme le BRP de Philips, utilise des files pour l'envoi et la réception de messages, des horloges pour détecter les erreurs (timeout), et des compteurs pour respecter les nombres maximaux de réḿissions spécifiés. Dans le cadre du projet Persée, nous souhaitons développer des techniques pour ces *systèmes hétérogènes*. De telles techniques ont été mises en œuvre dans le cadre de l'outil TReX [6], mais nous souhaitons aller plus loin :

- Il apparaît tout d'abord que ces techniques sont spécialisées : elles ne sont réellement efficaces que dans un cadre précis. Pour l'analyse d'un modèle présentant potentiellement des situations diverses, il est donc utile de disposer de plusieurs techniques que l'on applique là où elles sont les plus adaptées.

- La difficulté à représenter des espaces d'états dépend des opérations que l'on réalise sur ceux-ci. Or, dans un même modèle certaines données sont complexes à représenter, et d'autres simples. Il est donc bénéfique de disposer d'une hétérogénéité au niveau des représentations symboliques afin d'augmenter l'efficacité des algorithmes de vérification.
- La comparaison et la validation de nouvelles techniques sont rendues difficiles par les spécificités des différents outils, sans que cela ne soit intrinsèque au problème considéré. Nous souhaitons donc disposer d'un cadre permettant de les tester et de les mettre en œuvre.

Dans le cadre du projet PERSÉE, nous développons par ailleurs une approche complémentaire à cette attaque frontale des modèles hétérogènes : l'abstraction automatique. En effet, la complexité des modèles issus d'applications réelles est bien souvent hors de portée de nos techniques d'analyse. Mais cette complexité est en même temps excessive : l'information qu'elle apporte n'est pas utile pour tel ou tel but de vérification. Par exemple, dans le cas de la validation d'un protocole, on peut imaginer que les variables horloges n'influent pas sur la bornitude des files de communication, contrairement aux variables files et compteurs, et qu'elles peuvent être «retirées» du modèle tout en permettant de conclure sur ce problème. Le but est donc de vérifier des propriétés du modèle concret sur une abstraction de celui-ci, conservative vis-à-vis des propriétés considérées. Dans le cadre de l'interprétation abstraite [27], des techniques de construction automatique de modèles abstraits comme l'abstraction de données ou l'abstraction par prédicats [45, 23] ont été inventées, puis mises en œuvre avec succès [37, 10].

Ces méthodes d'abstraction se basent sur la production d'un premier modèle abstrait sur lequel on tente de vérifier la propriété. Soit la satisfaction ou l'insatisfaction de la propriété sur le modèle abstrait se transmet au modèle concret, soit les approximations introduites ne permettent pas le transfert du résultat de l'abstrait vers le concret. Il est alors nécessaire de raffiner le modèle : en affinant les critères d'abstraction (par exemple, les prédicats) utilisés lors de la construction du permier modèle abstrait. On obtient ainsi un second modèle abstrait, mais plus précis que le premier, et on continue la boucle abstraction-vérification-raffinement jusqu'à pouvoir prouver la satisfaction ou l'insatisfaction de la propriété. Notre interêt est donc d'utiliser conjointement les deux méthodes [24, 1].

Du point de vue de l'outillage, la vérification symbolique et l'abstraction automatique présentent des besoins similaires : la représentation et le calcul d'ensembles d'états infinis, et l'évaluation des opérations du modèle sur ceux-ci. Nous nous sommes donc fixés le but de développer une architecture pour la vérification et l'abstraction de modèles hétérogènes, qui prenne en compte ces besoins. Elle doit donc être :

- *Générique* : l'hétérogénéité des systèmes fait que nous ne pouvons pas nous restreindre à un modèle particulier, mais que nous devons choisir un formalisme ouvert permettant la modélisation d'applications diverses.
- *Modulaire* : afin de pouvoir comparer les techniques et bénéficier de leurs avantages conjointement, il est nécessaire de trouver un cadre dans lequel ces techniques peuvent collaborer. Ainsi, suivant la propriété à vérifier, on prévilégiera telle ou telle technique d'abstraction, et pendant la phase de vérification, on utilisera une première technique sur une partie du modèle et une seconde technique sur le reste du modèle afin de pouvoir conclure.

L'absence de généricité et de modularité dans les outils de model-checking usuels provient souvent de la restriction de leur utilisation (prévue) à un domaine d'application étroit. En effet, le langage de spécification limite les modèles que l'on peut soumettre à la vérification. Par exemple, PROMELA/SPIN ne permettent pas de modéliser le temps, ni des canaux de communication non bornés. Le manque de modularité est du à une trop grande intégration, allant jusqu'à la fusion, entre les composants logiciels. Elle provient souvent du fait que le concepteur a cherché à obtenir un outil entièrement automatique, donc sans expertise permettant de choisir la méthode la mieux adaptée. Ces aspects révèlent un seul et même point : la *séparation entre syntaxe et sémantique* dans les outils existants est très faible et conduit à un outil figé.

Traditionellement, un model-checker se compose de 3 parties : le formalisme de modélisation, le module de représentation symbolique (représentation d'ensembles d'états et opérations sur ceux-ci), et l'algorithme de model-checking (parcours du graphe sémantique, heuristiques, etc.). Notre proposition consiste à isoler ces 3 parties et à introduire uniquement les interactions nécessaires

entre elles. La figure 1 détaille notre architecture.

– La *généricité* est obtenue d'une part au niveau du modèle en permettant l'ajout de nouveaux types et opérateurs via des *signatures* qui sont décrites en section 2. Elle est obtenue d'autre part au niveau des représentations symboliques : elles ne sont pas fixées et il est donc possible d'en ajouter autant que nécessaire tant que les interfaces sont respéctées, comme indiqué en section 3. Le lien entre la syntaxe (le modèle) et la sémantique (la représentation symbolique) se fait au niveau des variables : quelle représentation pour quelle variable ? Ce choix fixé lors de la vérification, est décrit en section 4.
– La *modularité* se retrouve au niveau des représentations symboliques qui son éclatées en trois aspects : la *région* qui correspond à la structure de données munie des opérations ensemblistes usuelles, la *représentation symbolique* qui est une région munie d'une sémantique, et enfin, l'*accélération* (au sens large : accélération exacte, interpolation, widening, etc) est elle aussi construite au-dessus des régions. Il est donc aisé de remplacer une représentation symbolique par une autre tout en conservant le même algorithme de model-checking pour peu que les interfaces soient respectées. De même, remplacer une accélération par une autre ne nécessite pas de changer la région ou la représentation symbolique. La section 3 décrit ces trois entités, ainsi que la façon dont sont évaluées les opérations sur les régions, et comment elles sont composées pour obtenir d'autres régions.

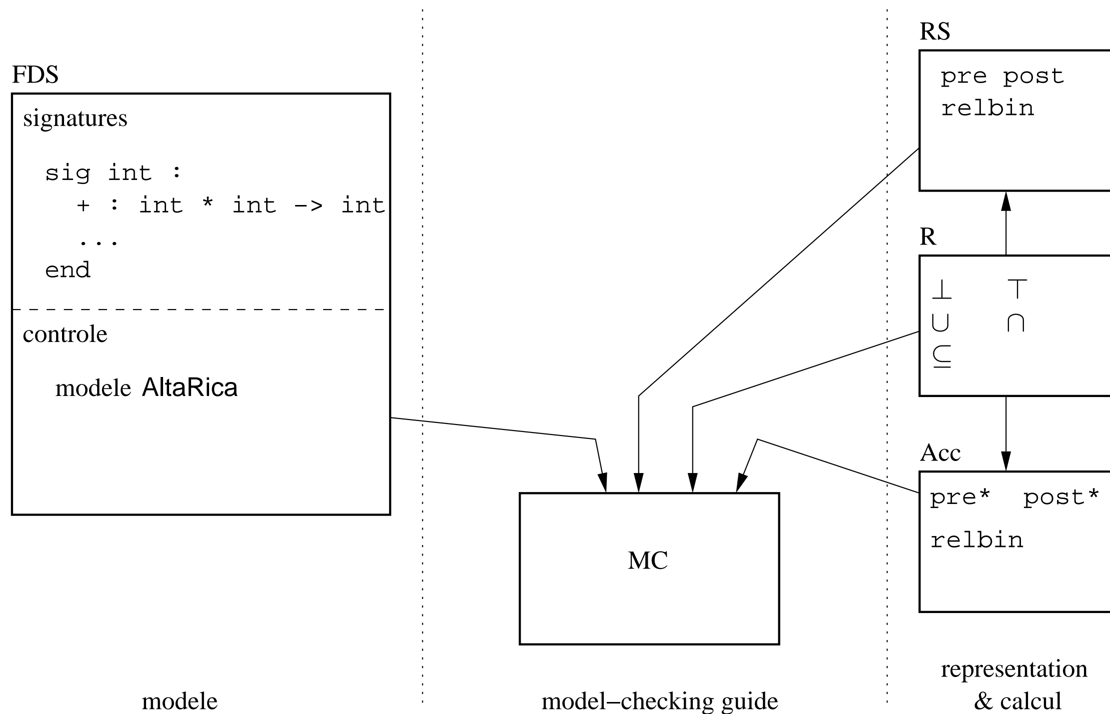


Fig. 1 – Architecture de l'outil Persée.

2 Formalisme de description

Le formalisme AltaRica [7, 44] a été conçu au LaBRI pour permettre d'utiliser une même description d'un système avec plusieurs outils d'analyse. Il est par exemple possible à partir de la description AltaRica d'un modèle de le simuler, de générer des arbres de défaillances, ou de vérifier qu'il satisfait des propriétés logiques.

Plusieurs organismes ont participé à l'élaboration du formalisme et du langage associé, no-

tamment le LaBRI et aussi des entreprises privées comme IXI (maintenant GFI Consulting) ou Dassault Systèmes. Cette collaboration a été fructueuse et nous a convaincus de l'importance des deux propriétés suivantes d'AltaRica :

- une sémantique clairement définie : on peut associer à toute description AltaRica le système de transitions qui lui correspond,
- un langage dont la syntaxe est proprement spécifiée.

Le but de ces deux points étant que les outils autour d'AltaRica, développés par les divers partenaires, puissent accepter les mêmes descriptions et que leurs résultats puissent être mis en corrélation.

De nombreuses études comme par exemple [35, 20, 36] ont été réalisées en utilisant le formalisme AltaRica, et ce formalisme est largement utilisé par nos partenaires industriels.

Nous allons dans cette section présenter les concepts d'AltaRica, et nous utiliserons des exemples écrits dans le langage AltaRica. La syntaxe du langage est définie en annexe et sa sémantique formelle sont définies dans la thèse de Gérald Point [44].

2.1 Décrire les comportements par contraintes

Nous donnons tout de suite un exemple simple qui permet de montrer comment on décrit un automate à contraintes en AltaRica.

```
node exemple
    state x : [ 0, 8 ];
    event e;

    trans
        x <= 2 |- e -> x := x + 1;
        x >= 3 |- e -> x := x + 2;

    init
        x := 0;
edon
```

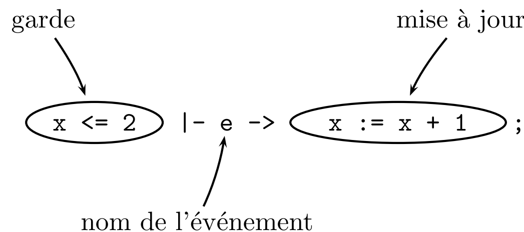
L'espace d'états est donné par les valuations possibles des variables d'état (ici, `x`, qui peut prendre 9 valeurs différentes). A l'exception de l'événement ε que nous décrirons plus loin, tous les événements doivent être nommés, et nous nous donnons donc ici un événement `e`. Dans la section `trans` sont énumérées les dites *macro-transitions* (2 dans l'exemple).

Afin d'expliquer le comportement de modèles AltaRica, nous donnerons l'équivalent de ces modèles sous forme de systèmes de transitions.

2.1.1 Sémantique d'une macro-transition

Chaque macro-transition peut donner plusieurs transitions dans la sémantique du nœud.

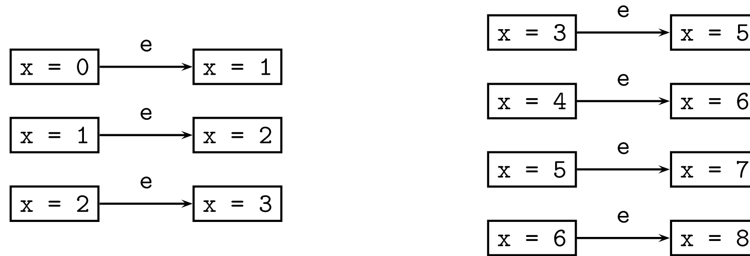
Voici comment s'effectue le passage de la macro-transition à sa sémantique sur l'exemple ci-dessus.



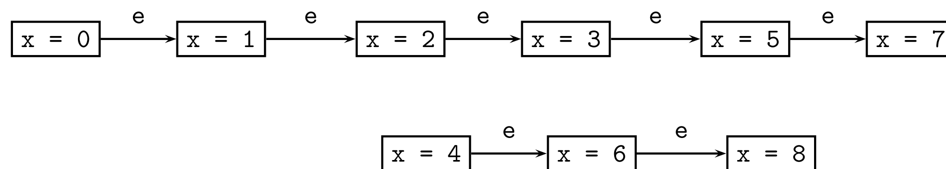
La garde spécifie l'ensemble des configurations qui peuvent potentiellement être source d'une transition engendrée par la macro-transition. Dans ce cas-ci, il s'agit des valuations pour lesquelles `x` vaut 0, 1, ou 2. Le nom de l'événement est celui qui sera utilisé pour chaque transition engendrée.

Pour chaque configuration de départ, les variables d'état sont affectées par la mise à jour, et cela donne l'ensemble des configurations qu'il est possible d'atteindre à partir de cette configuration.

Voici donc les transitions que l'on obtient pour la première et la deuxième macro-transitions :



Et nous obtenons graphiquement le système de transitions suivant après identification des configurations égales.



De plus, des ε-transitions n'apparaissant pas ici font partie de la sémantique du modèle, sous la forme de boucles sur chaque configuration.

Notez que depuis l'état initial `x = 0`, la composante où `x` vaut 4, 6, ou 8 n'est pas accessible.

2.1.2 Non-déterminisme

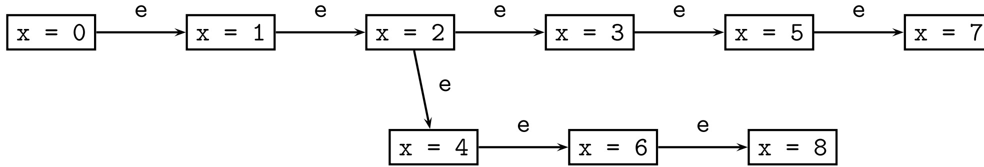
Si les gardes de deux macro-transitions ayant le même nom d'événement sont d'intersection non vide, plusieurs transitions différentes peuvent être tirées lorsque cet événement survient dans une configuration appartenant à l'intersection des deux gardes. Le système peut engendrer ou accepter les divers comportements indifféremment : c'est du *non-déterminisme*.

Le non-déterminisme est utile pour modéliser des comportements dont on ne sait pas *a priori* lequel aura lieu. En autorisant les deux comportements de manière non-déterministe, on est sûr que l'énumération de tous les comportements du système les contiendra tous.

Voici une légère modification de la garde de la deuxième macro-transition de l'exemple précédent.

```
node exemple
    state x : [ 0, 8 ];
    event e;

    trans
        x <= 2 |- e -> x := x + 1;
        x >= 2 |- e -> x := x + 2;

    init
        x := 0;
edon
```

On obtient alors le système de transitions suivant :



On voit ici que dans la configuration `x = 2`, le système peut passer indifféremment à la configuration `x = 3` ou à la configuration `x = 4` par un événement `e`.

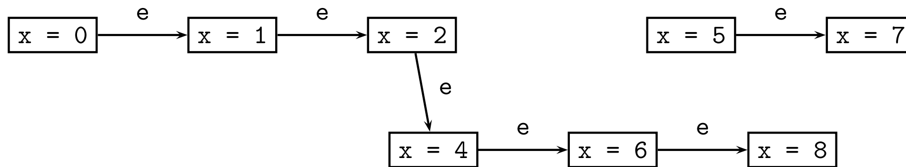
2.1.3 Assertion et contraintes de domaines

Il est possible de spécifier une contrainte appelée *assertion* qui *doit* être satisfaite par toute configuration du système.

Nous pouvons par exemple interdire la configuration dans laquelle `x` vaut 3. Il suffit pour cela d'ajouter à notre nœud `exemple` précédent le champ :

```
assert
    x != 3;
```

Ceci donnera le système de transitions suivant :



L'assertion contient aussi implicitement les contraintes de domaines des variables du nœud. Ainsi, dans les exemples précédents, nous n'avons pas eu besoin de préciser dans la garde de la deuxième macro-transition qu'elle n'était possible que si `x <= 6`. Du fait que les deux configurations `x = 7` et `x = 8` mèneraient à des valeurs de `x` hors de son domaine, ces transitions n'existent pas dans la sémantique du modèle. Le fait que la valeur des variables *après* tirage d'une transition puisse affecter la tirabilité de la transition est souvent appelé test de *post-condition*.

L'assertion dans sa totalité (et pas seulement les contraintes de domaines) sert à évaluer la post-condition.

2.2 Eléments introduits pour la hiérarchie

AltaRica permet de réutiliser des nœuds à l'intérieur d'autres nœuds. Nous allons ici présenter deux concepts qui, bien qu'ayant déjà un sens dans un nœud simple, prennent tout leur sens lorsqu'on souhaite modéliser un système qui contient des sous-nœuds.

> En accord avec [44], nous utiliserons parfois le terme *composant* pour désigner un nœud qui ne contient pas de sous-nœud.

2.2.1 Variables de flux/variables d'état

Il existe deux sortes de variables dans un nœud AltaRica : les variables d'état et les variables de flux.

Les variables d'état représentent l'état du système, elles peuvent être utilisées à tout endroit où une variable peut être utilisée : dans une garde, dans l'affectation qui suit une transition, ou dans un prédicat global appelé *assertion* qui contraint les valeurs que peuvent prendre les variables.

Les variables de flux sont comme les variables d'état, à ceci près qu'elles ne font pas partie de l'"état" du système : elles représentent des valeurs auxiliaires qui sont uniquement en relation avec l'état courant grâce à l'assertion. Les variables de flux ne peuvent pas être affectées explicitement lors du franchissement d'une transition.

Les variables de flux ont deux rôles possibles :

– elles permettent d'extraire de l'information du système, de plus haut niveau que simplement l'état courant du système, par exemple pour abstraire de l'information pertinente
– elles permettent aussi de contraindre l'état du système puisqu'elles apparaissent dans l'assertion, qui est un prédicat que toutes les configurations du système doivent satisfaire.

L'intérêt principal de ces variables de flux est de permettre la communication d'information entre nœuds. (cf. sous-section 2.3.2, page 10)

Il est temps d'introduire le vocabulaire adéquat : on appelle *état* d'un modèle AltaRica une valuation de ses variables d'état, et *configuration* une valuation de toutes ses variables, y compris les variables de flux.

2.2.2 Les ε-transitions

Comme nous l'avons laissé entendre un peu plus haut dans l'exemple, des transitions ε (c'est-à-dire sans nom) sont ajoutées à tous les états sous la forme de boucles.

On peut déjà noter que ces ε-transitions ne peuvent pas être spécifiées explicitement dans un modèle AltaRica. Elles font uniquement partie de la sémantique du modèle.

Leur rôle est d'assurer qu'un nœud ne "bloque" jamais : pour une configuration donnée, il est toujours possible de tirer une ε-transition sortante. Cette particularité est très utile lors de la composition hiérarchique de nœuds AltaRica, comme nous le verrons dans la section suivante.

Toutes les configurations correspondant au même état du système sont reliées par des ε-transitions. En particulier, à toute configuration est associée une boucle ε. Les variables de flux peuvent toujours changer de valeur lors du tirage d'une transition, et les ε-transitions ne font pas exception.

2.2.3 Un exemple

Afin d'éclaircir l'interaction entre les variables de flux et les ε-transitions, nous allons prendre l'exemple d'une salle qui peut contenir jusqu'à cinq personnes, et dont le battant de porte permet à une personne de se dissimuler (voir la figure 2).

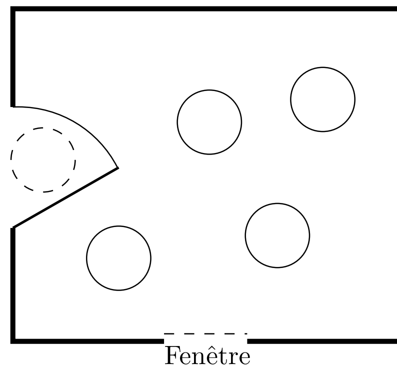


Fig. 2 – Salle avec vue partielle sur l'intérieur

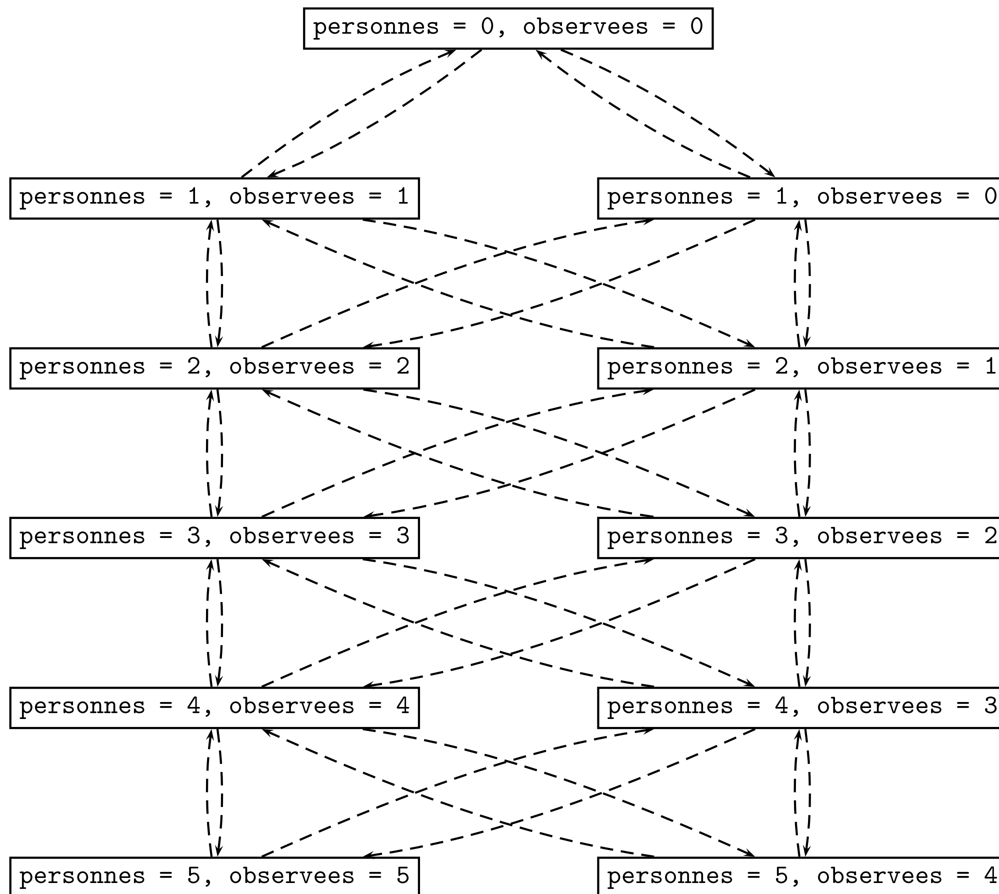
Une personne observant la salle depuis l'extérieur par la fenêtre ne peut jamais savoir s'il y a une personne derrière le battant de la porte. A tout instant, le nombre de personnes qu'elle observera dans la salle sera donc faux d'au plus une personne par défaut.

L'état de la salle est clairement constitué du nombre exact de personnes présentes et nous nous donnons une variable de flux pour représenter la quantité de personnes observables de l'extérieur de la salle. Ceci se traduit aisément en AltaRica, cf. figure 3.

```
node SalleAvecRecoin
    state personnes : [ 0, 5 ];
    flow observees : [ 0, 5 ];

    event entre, sort;

    trans
        personnes < 5 |- entre -> personnes := personnes + 1;
        personnes > 0 |- sort -> personnes := personnes - 1;

    assert
        observees <= personnes & observees >= (personnes - 1);
edon
```

Fig. 3 – Modèle AltaRica de la salle de la figure 2

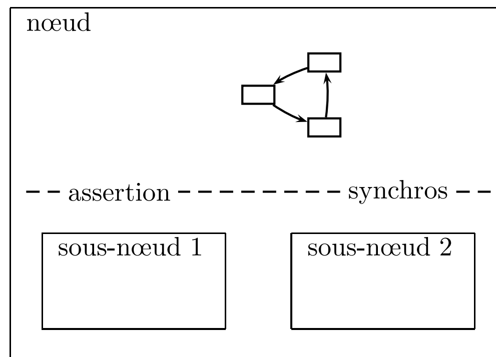
Nous représentons ci-dessous le graphe des configurations de ce nœud. Les événements `entre` et `sort` ne sont pas écrits pour que le graphe reste lisible mais leurs transitions sont représentées en lignes pointillées : les transitions descendantes correspondent à l'événement `entre`, les transitions montantes à l'événement `sort`. On peut observer que les ε-transitions conservent bien les états, mais laissent les flux libres à l'intérieur de leurs contraintes.



2.3 Composition hiérarchique

Le formalisme AltaRica permet d'utiliser un nœud AltaRica à l'intérieur d'un autre, ou de faire communiquer plusieurs nœuds entre eux. Dans ce dernier cas, il est toujours nécessaire qu'un nœud englobe les nœuds communiquants : ce nœud joue le rôle de contrôleur.

Un nœud se compose donc d'une partie locale et de ses sous-nœuds potentiels.



Les comportements de la partie locale du nœud et les sous-nœuds sont liés par l'assertion et les *synchronisations d'événements*, afin de décrire le comportement du nœud dans sa globalité.

Notez que les deux sous-nœuds n'ont pas connaissance de l'existence l'un de l'autre. La communication qui s'établit entre eux est *contrôlée* par leurs ancêtres communs.

2.3.1 Sémantique d'un nœud sans synchronisations

Nous allons d'abord décrire comment un nœud se comporte lorsqu'aucune interaction n'est imposée entre lui et ses sous-nœuds.

Chaque configuration du nœud est prise dans les valuations de la réunion des variables du nœud et de ses sous-nœuds. Cette réunion est disjointe et on utilise les noms des sous-nœuds comme préfixes pour nommer les variables afin de les distinguer. La variable `x` du sous-nœud `N` est nommée `N.x` lorsqu'on veut y faire référence.

Vecteurs d'événements Une transition est désormais étiquetée par un *vecteur d'événements*. Chaque vecteur d'événements est constitué d'un des événements du nœud local et d'un événement par sous-nœud : un vecteur d'événements code donc précisément les événements de chacun des composants apparaissant dans le nœud, qui seront activés si ce vecteur l'est.

En l'absence d'autre contrainte, chacun des événements (local ou apparaissant dans un sous-nœud direct) constitue un vecteur d'événements où tous les autres éléments du vecteur sont l'événement ε du nœud correspodant. Dans ce contexte, deux événements nommés ne peuvent pas avoir lieu en même temps. On peut parler d'*asynchronisme fort*.

Exemple Nous définissons dans la figure 4 un nœud `Simple` qui contient une variable booléenne qui passe de l'état faux à l'état vrai sur un événement `e`.

Le nœud `Simple` a donc deux événements : l'événement `e` qui change son état, et l'événement ε qui boucle sur chaque configuration (qui ne sont ici que des états).

Le nœud `PasDeSynchro` contient deux sous-nœuds `N1` et `N2` qui sont des nœuds `Simple`.

Comme il ne définit pas d'événement propre, son seul événement local est ε. Ses vecteurs d'événements sont donc <ε, N1.ε, N2.ε>, <ε, N1.e, N2.ε>, et <ε, N1.ε, N2.e>.

```
node Simple
    state v : bool;
    event e;
    trans
        ~v |- e -> v := true;
edon

node PasDeSynchro
    sub N1, N2 : Simple;
edon
```

FIG. 4 – Nœuds `Simple` et `PasDeSynchro`

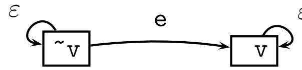


FIG. 5 – Sémantique du nœud `Simple`

2.3.2 Synchronisation par assertion

Il est possible dans l'assertion d'un nœud de contraindre à la fois les variables locales au nœud, et les variables de ses sous-nœuds. Grâce à cela, il est par exemple possible de partager des variables entre nœuds : il suffit de contraindre deux variables à être égales (figure 7). L'assertion ne peut qu'*interdire* certaines configurations, ainsi que par conséquent les transitions qui en partent ou qui y arrivent.

Toute autre contrainte est acceptée, et parmi les utilisations possibles, on peut par exemple contraindre un nœud prévu de manière générique à limiter ses variables dans un certain domaine, ce qui est illustré par la figure 8.

2.3.3 Synchronisation d'événements

L'autre mécanisme de communication disponible en AltaRica est la synchronisation d'événements. Elle permet de préciser les vecteurs d'événements que l'on souhaite garder dans le nœud. Ces vecteurs sont appelés *vecteurs de synchronisation*.

Chaque vecteur fait apparaître au plus un événement par sous-nœud ainsi qu'au plus un événement du nœud local, et est complété par des événements ε. Dès qu'un événement nommé d'un sous-nœud apparaît dans un vecteur de synchronisation, cet événement ne sera plus synchronisé avec l'événement ε du nœud local, comme c'est le cas en l'absence de synchronisation.

Voyons sur notre exemple simple ce qui se passe lorsqu'on synchronise les deux événements nommés des deux sous-nœuds (figure 9).

Comme on peut le voir en comparant les figures 6 (page 11) et 10, les vecteurs `<ε, N1.e, N2.ε>` et `<ε, N1.ε, N2.e>` n'apparaissent plus, puisque respectivement `N1.e` et `N2.e` sont explicitement cités dans le vecteur de synchronisation.

Il est d'ailleurs intéressant de constater sur cet exemple que l'on aurait obtenu exactement la même sémantique si au lieu de synchroniser les événements `N1.e` et `N2.e` on avait interdit les deux configurations où `N1.v` et `N2.v` sont différentes en imposant l'égalité dans l'assertion (`assert N1.v = N2.v`).

Bien qu'anecdotique sur cet exemple, cette dualité entre contrainte sur les configurations et contrainte sur les vecteurs d'événements est une des caractéristiques importantes d'AltaRica. Elle est le témoin du fait que le langage AltaRica est un langage de haut niveau, qui permet de faire des choix de modélisation différents pour un "même" système.

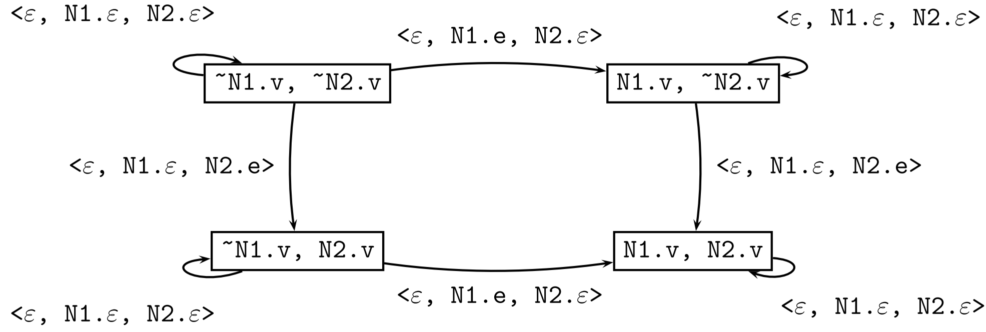


Fig. 6 – Sémantique du nœud PasDeSynchro

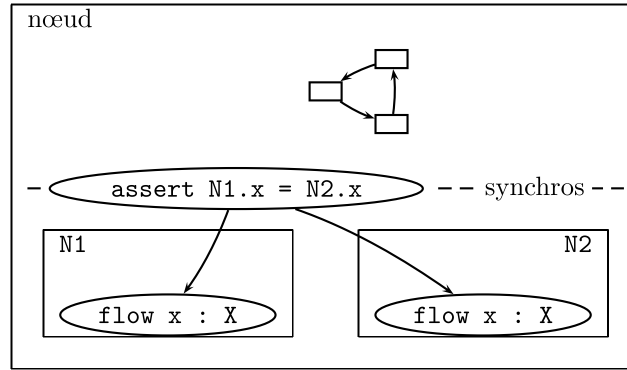


Fig. 7 – Partage de variables

2.4 Notion de visibilité

Jusqu'ici, nous avons montré plusieurs exemples d'assertions et de contraintes d'événements qui lient des nœuds différents entre eux. Ceci n'est possible qu'à la condidition que les éléments (variables ou événements) utilisés dans ces liaisons soient *visibles* depuis un autre nœud.

Ces propriétés de visibilité interviennent en amont de la sémantique : elles sont un aspect "langage" d'AltaRica. Les contraintes de visibilité interdisent l'utilisation de certaines variables dans certains nœuds, de manière purement "syntaxique".

Les trois propriétés de visibilité disponibles en AltaRica sont les suivantes : privé, parent, public.

Un élément privé n'est accessible que dans le nœud où il est déclaré, un élément ayant la visibilité "parent" sera en plus accessible dans tout nœud qui l'utilise comme sous-nœud. Enfin, un élément public est visible par tous les nœuds qui sont au-dessus de lui dans la hiérarchie AltaRica induite par la relation d'utilisation.

Il y a une analogie évidente avec les langages de programmation orientés objet puisqu'il s'agit de répondre au même problème d'autorisation d'accès. Le langage C++ propose les trois qualificateurs de visibilité privé, protégé, et public. Nous n'avons pas conservé le terme *protégé* car il correspond en C++ à une notion d'héritage alors qu'il correspond en AltaRica à une notion d'utilisation. Du fait qu'il n'existe pas de mécanisme de passage de paramètres en AltaRica, et donc à plus forte raison de mécanisme permettant de passer en paramètre un nœud AltaRica à un autre en dehors de la ré-utilisation statique d'un nœud, un élément "public" sera visible le long du chemin qui remonte de parent en parent, et ne sera jamais visible à l'extérieur de ce chemin.

Ces règles sont résumées graphiquement dans la figure 11. Les nœuds AltaRica sont symbolisés par des rectangles, et la relation d'utilisation d'un nœud par les arêtes. Les nœuds dans lesquels l'élément x est visible sont encadrés par un double cadre. L'élement x peut être indifféremment

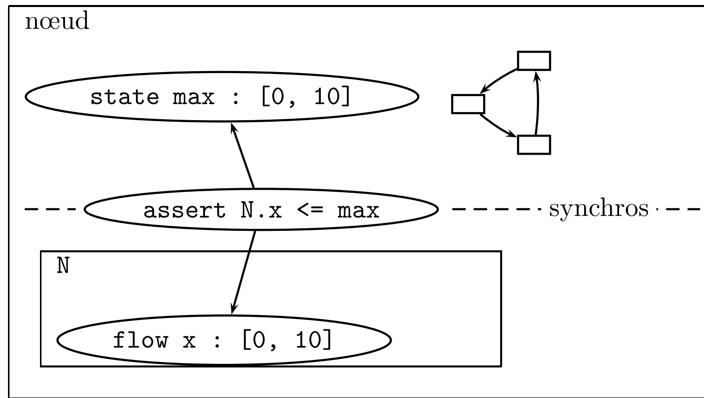


Fig. 8 – Contrainte sur une sous-variable

```
node SynchroSimple
    sub N1, N2 : Simple
    sync <N1.e, N2.e>
edon
```

Fig. 9 – Le nœud `SynchroSimple`

une variable d'état ou de flux, ou un événement à l'exception des événements publics qui sont contraints par une règle supplémentaire.

2.4.1 Le cas particulier des événements publics

Les événements publics nécessitent de généraliser un peu la notion de synchronisation d'événements que nous avons présentée, et du point de vue de la visibilité ils sont restreints de la manière suivante : dès qu'un événement public est synchronisé dans un nœud, il cesse d'être public. Un schéma explicatif est donné par la figure 12.

2.4.2 Visibilités comme réécriture syntaxique

Le fait qu'une variable d'un sous-nœud `A` soit publique ou visible du nœud parent peut se voir comme une permission accordée d'accéder à cette variable dans le sous-nœud, mais peut aussi s'expliquer en imaginant que l'on remonte une image de cette variable dans le nœud parent. C'est cette vision que nous allons adopter ici car elle permet d'expliquer sans ambiguïté les règles de visibilité et de voir clairement l'impact qu'ont les événements publics sur la sémantique du nœud parent.

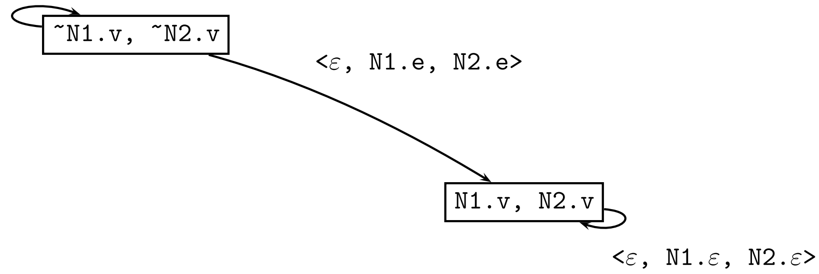


FIG. 10 – Sémantique du nœud SynchroSimple

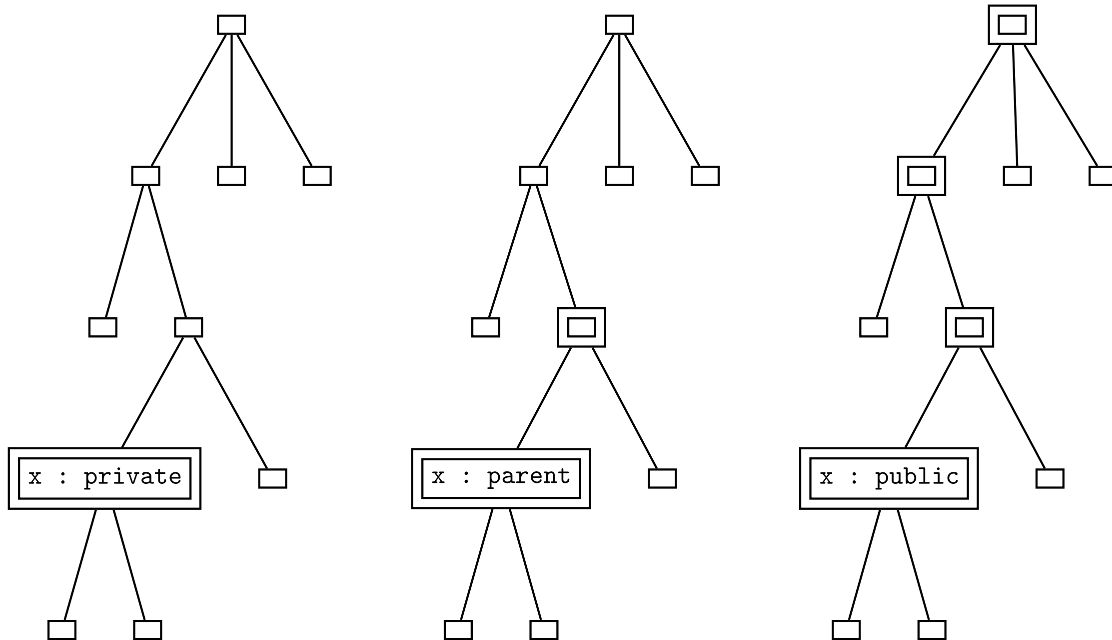


FIG. 11 – Les trois qualificatifs de visibilité

Objet dans le sous-nœud A	*Incarnation de l'objet dans le nœud parent*
state s : bool : private ;	*Rien de visible*
flow f : bool : private ;	
event e : private ;	
state s : bool : parent ;	flow s_A : bool : private ; assert s_A = A.s ;
flow f : bool : parent ;	flow f_A : bool : private ; assert f_A = A.f ;
event e : parent ;	event e_A : private ;
state s : bool : public ;	flow s_A : bool : public ; assert s_A = A.s ;
flow f : bool : public ;	flow f_A : bool : public ; assert f_A = A.f ;
event e : public ;	*si* A.e *n'est pas synchronisé dans ce nœud* event e_A : public ; sync <e_A, A.e> ; trans true |- e_A -> ;
event e : public ;	*si* A.e *est synchronisé dans ce nœud* event e_A : private ;

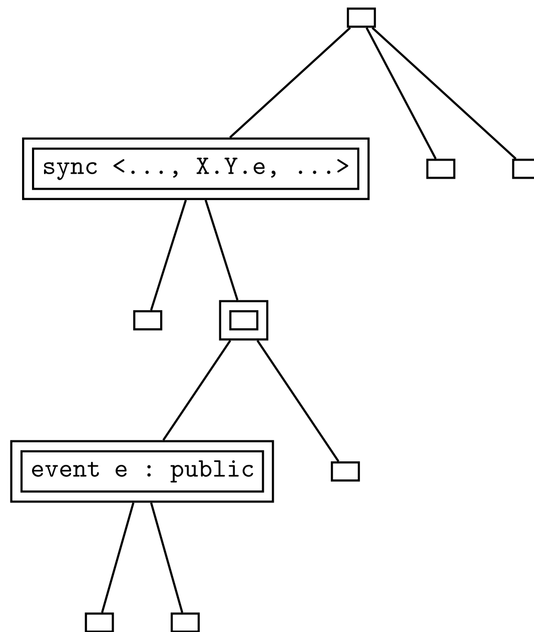


FIG. 12 – Propagation des événements publics

2.4.3 Visibilité par défaut et compatibilité ascendante

Le langage AltaRica permet de spécifier des qualificatifs de visibilité. Les visibilités par défaut correspondent à l'ancienne version du langage afin de préserver la compatibilité ascendante pour les modèles déjà écrits.

Dans la précédente version du langage, aucun élément n'était public, les variables d'état étaient privées, et les variables de flux et les événements étaient visibles des parents.

Le point de départ de cette extension a été le besoin exprimé par les industriels de rendre les pannes des feuilles visibles jusqu'à la racine de la hiérarchie. Comme les concepts privé/parent existaient déjà sans être nommés explicitement dans les modèles, c'est tout naturellement que ces trois qualificatifs ont été introduits et que leur sens a été défini.

Voici un tableau récapitulatif des règles de visibilité par défaut :

Type d'objet	*Visibilité par défaut*
Etat	privé
Flux	parent
Evénement	parent

2.5 La spécification de nouveaux types

2.5.1 Les signatures

Pour nos besoins de généricité, nous souhaitons étendre le formalisme AltaRica de façon à ce qu'il permette la définition de nouveaux types et des opérations associées. En effet, AltaRica offre principalement un seul type de données : les énumérations finies. Les opérations permises dépendent de la déclaration de la variable : si celle-ci est booléenne alors les opérateurs classiques (conjonction, disjonction, négation, etc) sont permis, si elle est entière, alors la comparaison, la somme, la négation, etc, sont autorisées.

Nous proposons donc un mécanisme permettant de définir de nouveaux types à base de signatures. Nous pouvons par exemple ajouter un type `file` qui permet de modéliser les communications entre processus :

```
domain msg = { SND, ACK, NACK, END };

sig fifo
  empty :            -> fifo
  is_empty : fifo    -> bool
  top :      fifo    -> msg
  push : fifo * msg -> fifo
  pop :        fifo -> fifo
gis
```

Cette signature définit les opérations classiques sur les files FIFO. Avec ce mécanisme, le formalisme AltaRica permet de modéliser la plupart de systèmes dont le nombre de processus est borné. Notons au passage qu'il est aussi désormais possible de spécifier des types infinis. D'autres exemples de signatures sont présentées en annexe A (page 25). Nous considérons donc qu'un modèle PERSÉE est composé d'un ensemble de signatures et d'un modèle AltaRica.

2.5.2 Syntaxique et sémantique des modèles Persée

L'ajout des signatures au formalisme AltaRica pose des difficultés lors de l'analyse syntaxique. En effet, l'ensemble des types et des opérateurs du langage n'est pas connu *a priori*. Par exemple, pour la transition suivante :

```
( ! is_empty(f)) && (top(f) != m) |- e -> f := push(f, m)
```

`is_empty`, `push` et `!=` ne sont pas des opérateurs du langage : ils sont introduits par la signature `fifo` de la section précédente. La solution consiste donc à les traiter comme des identificateurs, sans leur associer de sémantique particulière (de la même manière que les noms de fonctions sont gérés par un compilateur `C` par exemple). La figure 13 présente l'arbre d'analyse syntaxique obtenu pour la transition précédente.

La sémantique est donnée par les représentations symboliques comme il est décrit en section 3.1. Nous obtenons ainsi une séparation forte entre syntaxe : un modèle AltaRica paramétré par des signatures, et sémantique : le système de transitions symbolique construit en section 3.

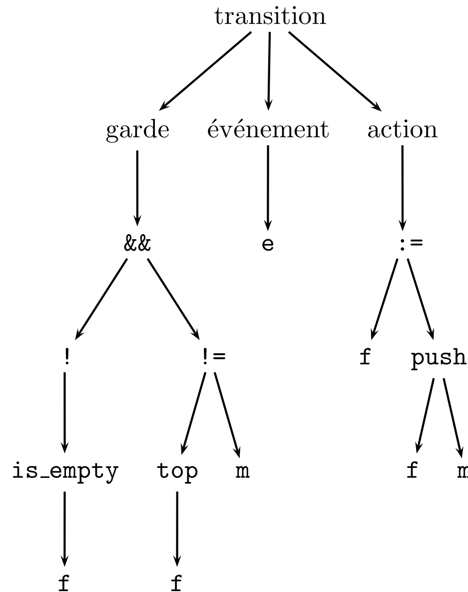


FIG. 13 – Arbre d'analyse syntaxique pour une transition.

L'affectation d'une sémantique donnée à un opérateur se fait par l'attribution d'une représentation symbolique à chaque variable du modèle. Ainsi, pour la transition précédente, la sémantique de `top` est fixée par le choix de la représentation symbolique pour `f` dans le terme `top(f)`. Le fait d'associer une représentation à chaque variable et non pas à chaque type offre une plus grande souplesse dans la stratégie de vérification du modèle. Notons que cela permet par exemple qu'un même modèle soit interprété avec des files de communication parfaites, ou non fiables, ou encore, que la sémantique du temps soit discrète ou continue, simplement en changeant l'affectation d'une représentation symbolique aux variables. Ce choix est réalisé par l'utilisateur via les mécanismes d'interaction décrits en section 4. La représentation symbolique reçoit, lors d'un calcul de `pre` ou `post`, une région d'une part, et un arbre syntaxique comme celui décrit en figure 13. L'évaluation de l'opération est décrite en section 3.1.

Un outil de vérification de typage se basant sur les signatures sera inclus à notre architecture Persée. Il sera capable, à partir d'un arbre syntaxique comme celui de la figure 13 de vérifier que les opérateurs définis dans les signatures sont appelés avec le bon nombre de paramètres, qui satisfont les contraintes de typages. De même, pour les affectations comme `f := push(f,m)`, le module de vérification de type s'assure que la valeur renvoyée par `push` et le type de `f` coïncident.

3 Calcul symbolique

Nous précisons dans cette section les notions de *région*, de *représentation symbolique*, et d'*accélération*. Nous prenons dans la suite de cette section un point de vue abstrait, théorique, afin de ne pas surcharger la présentation par des détails de syntaxe ou d'implantation.

3.1 Opérations et sémantique associée

Comme nous l'avons dit en section 2.5.2, l'évaluation des opérations par les fonctions `pre` ou `post` consiste à recevoir un arbre d'analyse syntaxique comme celui de la figure 13 (page 15), à associer une sémantique à chaque opérateur en fonction de la représentation associée aux variables, et enfin, à évaluer le résultat de l'opération. Nous décrivons maintenant ce processus.

Nous considérons les opérations sous une forme abstraite, non structurée, et sans aucun "sucre syntaxique". Les opérations seront simplement les éléments d'un alphabet noté `Op`. Cela se traduit par la signature de module (OCaml) suivante :

```
module Operation :
sig
  type t                              (* type for all operations *)
end
```

On s'attend naturellement à ce que l'interprétation d'une opération tienne compte de la valeur des variables, afin :
- d'autoriser ou non l'exécution de l'opération (*garde*), et
- de modifier éventuellement la valeur des variables (*affectation*).

Aussi utiliserons-nous la définition suivante pour donner une signification aux opérations. Afin de simplifier la présentation, nous supposons dans la suite de cette section que toutes les variables prennent leur valeurs dans le même domaine d'interprétation.

Une *sémantique* d'un alphabet d'opérations `Op` est un triplet $\Delta = (V, \mathbb{D}, \delta)$, où :
- V est un ensemble fini de *variables*, et
- $\mathbb{D}$ est un *domaine* d'interprétation pour les variables, et
- δ est une fonction totale de `Op` dans $\mathcal{P}(\mathbb{D}^V \times \mathbb{D}^V)$.

La fonction δ associe à chaque opération une relation binaire sur $\mathbb{D}^V$, permettant de décrire l'effet d'une opération sur les variables. Nous étendons naturellement la fonction δ : `Op` $\rightarrow \mathcal{P}(\mathbb{D}^V \times$

$\mathbb{D}^V$) aux séquences d'opérations $\sigma \in \mathtt{Op}^*$, par la définition récursive suivante :

$$\begin{cases} \delta(\varepsilon) &= \{(\overrightarrow{d}, \overrightarrow{d}) \mid \overrightarrow{d} \in \mathbb{D}^V\} \\ \delta(\mathtt{op} \cdot \sigma) &= \{(\overrightarrow{d}, \overrightarrow{d'}) \mid \exists \overrightarrow{d''} \in \mathbb{D}^V, (\overrightarrow{d}, \overrightarrow{d''}) \in \delta(\mathtt{op}) \text{ et } (\overrightarrow{d''}, \overrightarrow{d'}) \in \delta(\sigma)\} \end{cases}$$

Nous notons *post* et *pre* les fonctions totales $post, pre : \mathcal{P}(\mathbb{D}^V) \times \mathtt{Op}^* \to \mathcal{P}(\mathbb{D}^V)$ de *successeurs* et de *prédécesseur* définies par :

$$\begin{aligned} post(X, \sigma) &= \{\overrightarrow{d'} \in \mathbb{D}^V \mid \exists \overrightarrow{d} \in X, (\overrightarrow{d}, \overrightarrow{d'}) \in \delta(\sigma)\} \\ pre(X, \sigma) &= \{\overrightarrow{d} \in \mathbb{D}^V \mid \exists \overrightarrow{d'} \in X, (\overrightarrow{d}, \overrightarrow{d'}) \in \delta(\sigma)\} \end{aligned}$$

Exemple. L'ensemble d'opérations $\mathtt{Op} = \{\mathtt{x} \stackrel{?}{=} \mathtt{0}, \mathtt{y} \leftarrow \mathtt{y} + 2\}$, où x et y sont deux compteurs, a pour sémantique naturelle $(\mathbb{N}, \{x, y\}, \delta)$, où la fonction $\delta : \mathtt{Op} \to \mathcal{P}(\mathbb{N}^{\{x,y\}} \times \mathbb{N}^{\{x,y\}})$ est définie par :

$$\begin{aligned} (\overrightarrow{d}, \overrightarrow{d'}) &\in \delta(\mathtt{x} \stackrel{?}{=} \mathtt{0}) &&\text{si} \quad \overrightarrow{d}[x] = 0 \text{ et } \overrightarrow{d'} = \overrightarrow{d} \\ (\overrightarrow{d}, \overrightarrow{d'}) &\in \delta(\mathtt{y} \leftarrow \mathtt{y} + 2) &&\text{si} \quad \overrightarrow{d'}[x] = \overrightarrow{d}[x] \text{ et } \overrightarrow{d'}[y] = \overrightarrow{d}[y] + 2 \end{aligned}$$

■

Nous considérons maintenant, pour le reste de cette section, un alphabet d'opérations fixé Op muni d'une sémantique fixée $\Delta = (V, \mathbb{D}, \delta)$. Un élément de $\mathbb{D}^V$ est appelé une *valuation* (des variables).

3.2 Structure de régions

Nous utilisons des régions afin de représenter symboliquement des ensembles potentiellement infinis de valuations. Intuitivement, chaque région est un sous-ensemble de $\mathbb{D}^V$, et l'ensemble des régions doit (1) contenir $\emptyset$ et $\mathbb{D}^V$, et (2) être clos par union et intersection. Plus précisément, nous appelons région l'encodage (par exemple sous la forme d'une formule) d'un tel sous-ensemble de $\mathbb{D}^V$. La définition suivante tient compte du fait que deux régions peuvent représenter le même sous-ensemble de $\mathbb{D}^V$.

Une *structure de régions* pour $(\mathbb{D}, V)$ est un 7-uplet $(R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, [\![\cdot]\!])$ où :

- R est un ensemble de *régions*, dont l'*interprétation* est donnée par $[\![\cdot]\!] : R \to \mathcal{P}(\mathbb{D}^V)$,
- la relation binaire $\sqsubseteq \subseteq R \times R$ d'*inclusion symbolique* entre régions satisfait :

$$r \sqsubseteq r' \quad \text{ssi} \quad [\![r]\!] \subseteq [\![r']\!] \tag{1}$$

- les régions $\bot, \top \in R$ de *vide symbolique* et d'*univers symbolique* satisfont :

$$[\![\bot]\!] = \emptyset \quad \text{et} \quad [\![\top]\!] = \mathbb{D}^V \tag{2}$$

- les fonctions totales $\sqcup, \sqcap : R \times R \to R$ sont les opérateurs d'*union symbolique* et d'*intersection symbolique* satisfaisant :

$$[\![r \sqcup r']\!] = [\![r]\!] \cup [\![r']\!] \quad \text{et} \quad [\![r \sqcap r']\!] = [\![r]\!] \cap [\![r']\!] \tag{3}$$

Cette présentation formelle se traduit par la signature de module (OCaml) suivante :

```
module type REGION =
sig
  type t                                  (* type for all regions *)
  val bot : t                                 (* the empty region *)
  val top : t                             (* the universal region *)
  val leq : t -> t -> bool                    (* region inclusion *)
  val cup : t -> t -> t                       (* union of regions *)
  val cap : t -> t -> t                (* intersection of regions *)
end
```

Remarque. Afin d'obtenir des approximations supérieures (resp. inférieures) des calculs symboliques, la contrainte (3) peut être relâchée en remplaçant les égalités par des inclusions. D'autre part, la contrainte (1) peut également être relâchée, pour des raisons d'effectivité, par l'implication $r \sqsubseteq r' \implies [\![r]\!] \subseteq [\![r']\!]$, mais les calculs de point fixe (par itérations successives et test d'inclusion) termineront alors moins souvent (en théorie).

3.3 Représentation symbolique

Une structure de régions permet de manipuler de manière symbolique des ensembles de valuations. Cependant, nous avons également besoin d'interpréter de manière symbolique les opérations de Op sur ces régions. Nous complétons donc les structures de régions avec des opérateurs symboliques post et pre.

Une *représentation symbolique* pour la sémantique $\Delta = (V, \mathbb{D}, \delta)$ est un 9-uplet $(R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \mathsf{post}, \mathsf{pre}, [\![\cdot]\!])$ où :

- $(R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, [\![\cdot]\!])$ est une structure de régions pour $(\mathbb{D}, V)$,
- les fonctions totales $\mathsf{post}, \mathsf{pre} : R \times \mathtt{Op} \to R$ sont les opérateurs de *successeur symbolique* et de *prédécesseur symbolique* satisfaisant :

$$[\![\mathsf{post}(r, \mathsf{op})]\!] = \mathit{post}([\![r]\!], \mathsf{op}) \tag{4}$$
$$[\![\mathsf{pre}(r, \mathsf{op})]\!] = \mathit{pre}([\![r]\!], \mathsf{op}) \tag{5}$$

Cette présentation formelle se traduit par la signature de module (OCaml) suivante :

```
module type SYMBOLIC_REPRESENTATION =
sig
  module R : REGION                              (* region module *)
  val post : R.t -> Operation.t -> R.t           (* symbolic post *)
  val pre  : R.t -> Operation.t -> R.t            (* symbolic pre *)
end
```

Remarque. Afin d'obtenir des approximations supérieures (resp. inférieures) des calculs symboliques, les contraintes (4) et (5) peuvent être relâchées en remplaçant les égalités par des inclusions.

Exemple. Considérons l'ensemble d'opérations $\mathtt{Op} = \{\mathtt{y} \leftarrow \mathtt{y} + 2\}$ de sémantique $(\mathbb{N}, \{y\}, \delta)$, où la fonction $\delta : \mathtt{Op} \to \mathcal{P}(\mathbb{N} \times \mathbb{N})$ est définie par :

$$(n, n') \in \delta(\mathtt{y} \leftarrow \mathtt{y} + 2) \quad \text{si} \quad n' = n + 2$$

Nous pouvons définir une représentation symbolique, par sous-ensemble clos supérieurement de $\mathbb{N}$, pour cette sémantique. Elle est formellement définie par $\mathcal{R} = (R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \mathsf{post}, \mathsf{pre}, [\![\cdot]\!])$ avec $R = \mathbb{N} \cup \{\bot\}$, $\top = 0$, et pour tout $n, n' \in \mathbb{N}$:

$$\begin{aligned}
[\![\bot]\!] &= \emptyset \\
[\![n]\!] &= \uparrow n \\
n \sqcup n' &= \min(n, n') \\
n \sqcup \bot = \bot \sqcup n &= n \\
n \sqcap n' &= \max(n, n') \\
n \sqcap \bot = \bot \sqcap n &= \bot \\
\mathsf{post}(\bot, \mathtt{y} \leftarrow \mathtt{y} + 2) = \mathsf{pre}(\bot, \mathtt{y} \leftarrow \mathtt{y} + 2) &= \bot \\
\mathsf{post}(n, \mathtt{y} \leftarrow \mathtt{y} + 2) &= n + 2 \\
\mathsf{pre}(n, \mathtt{y} \leftarrow \mathtt{y} + 2) &= \max(0, n - 2)
\end{aligned}$$

où $\min(n, n')$ (resp. $\max(n, n')$) désigne l'élément minimal (resp. maximal), pour l'ordre total habituel sur les entiers, de $\{n, n'\}$. ■

3.4 Accélération

Les représentations symboliques permettent de réaliser des calculs symboliques d'atteignabilité, par exemple le calcul par itération de point fixe de l'ensemble d'atteignabilité d'un système. Cependant, la région obtenue à chaque étape d'un tel calcul peut *ne représenter qu'une portion finie* de l'ensemble (souvent infini) d'atteignabilité. Dans un tel cas, on bénéficie peu de l'avantage d'utiliser une représentation symbolique (sauf, bien sûr, si le but poursuivi est la représentation compacte d'ensembles finis d'états, comme par exemple avec les BDD).

Les techniques d'accélération sont basées sur l'observation suivante : si l'itération d'une séquence d'opérations $\sigma \in \mathtt{Op}^*$ (étiquetant un circuit de contrôle du système) à partir d'une valuation $\overrightarrow{d} \in \mathbb{D}^V$ produit un ensemble infini qui peut être calculé symboliquement, alors une portion infinie de l'ensemble d'atteignabilité peut être capturée en une seule étape.

Une *accélération* pour la sémantique $\Delta = (V, \mathbb{D}, \delta)$ est un 8-uplet $(R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \nabla, [\![\cdot]\!])$ où :
- $(R, \sqsubseteq, \bot, \top, \sqcup, \sqcap, [\![\cdot]\!])$ est une structure de régions pour $(\mathbb{D}, V)$,
- la fonction totale $\nabla : R^+ \times \mathtt{Op}^* \to R$ est l'opérateur d'*itération symbolique* satisfaisant[1] :

$$[\![\nabla(r_0 \cdots r_n, \sigma)]\!] = \bigcup_{i \in \mathbb{N}} post([\![r_0]\!], \sigma^i) \tag{6}$$

pour tous $\sigma \in \mathtt{Op}^*$ et $r_0 \cdots r_n \in R^+$ tels que : $[\![r_k]\!] = post([\![r_{k-1}]\!], \sigma)$ pour tout $1 \leq k \leq n$.

Cette présentation formelle se traduit par la signature de module (OCaml) suivante :

```
module type ACCELERATION =
sig
  module R : REGION                              (* region module *)
  val acc : R.t list -> Operation.t list -> R.t    (* acceleration *)
end
```

Remarque. Afin d'obtenir des approximations supérieures (resp. inférieures) des calculs symboliques, la contrainte (6) peut être relâchée en remplaçant l'égalité par une inclusion.

Les algorithmes de vérification peuvent utiliser l'accélération de deux façons :
- *statique* : l'utilisateur indique les séquences d'opérations à accélérer avant de lancer les calculs symboliques. Cette option correspond essentiellement aux méta-transitions de [9].
- *dynamique* : les séquences d'opérations à accélérer sont choisies automatiquement au cours des calculs symboliques [6, 11]. Des heuristiques peuvent alors être utilisées afin de limiter le choix [11].

4 Module de vérification

4.1 Interface avec la description

A la suite de la compilation de la description, le module de vérification doit obtenir une spécification du modèle qui doit inclure au minimum :
- la liste des variables représentées explicitement et leur type
- la liste des variables représentées symboliquement et leur signature
- la liste des transtions du système
- la liste des invariants (assertions) du système
- l'état initial des variables (s'il existe)

[1] On note R^+ l'ensemble des séquences finies (non vides) $(r_k)_{0 \leq k \leq n}$ de régions, notées plus simplement $r_0 \cdots r_n$.

4.2 Interface avec les représentations symboliques

En général, le module de vérification doit pouvoir appeler toute opération fournie par une représentation symbolique choisie.

Diagnostique utilisateur Comme montré dans l'architecture générale 1, la correspondance entre les signatures de la description et les représentations symboliques est faite à l'aide d'un fichier de correspondances. Le module de vérification doit être capable de vérifier que ce fichier est correct.

Diagnostique automatique De plus, depuis ce module, on doit pouvoir appeler une commande qui permet d'indiquer, en partant de la description, les différentes représentations symboliques qui peuvent être utilisées pour les signatures de la description. Ceci correspond à un diagnostique "automatique" de la description.

Choix des accélerations Si pour une représentation symbolique, plusieurs accelerations sont possible, l'utilisateur peut choisir l'une en spécifiant le module accéleration à utiliser par la suite.

4.3 Exploration et vérification : place à l'expertise

Il est évident qu'un outil "presse-bouton" n'est pas une bonne solution pour une plate-forme d'expérimentation. Nous devons donc proposer des moyens pour que l'expert utilisateur puisse exploiter, aussi librement que possible, la puissance des bibliothèques (pour les représentations symboliques ou pour les accélérations) interfacées à la plate-forme. Ceci n'empêche pas de proposer également des solutions "génériques" (classiques) d'analyse, vu qu'elles sont déjà développées dans les outils existants.

C'est pourquoi nous proposons pour le module de vérification un langage de commandes pour guider et superviser le model-checker. Ce langage peut être compilé (comme en HyTech) ou interprété.

Vu la diversité des options qui sont à présent utilisées dans les outils Fast et TReX, il nous a semble essentiel de proposer un langage qui permet une extension facile avec des options spécifiques. C'est pourquoi le langage de commandes que nous proposons est divisé en deux parties :

- *partie de base* (générale) qui inclut les commandes qui sont apparues comme communes aux outils existants.
- *partie spécifique* (dépendante d'outil ou bibliothèque) qui inclut tous ce qui n'est pas générique. Cette partie sera à définir à l'interface avec la plate-forme Persée et sera accessible par une commande de type `load`, par exemple :

  ```
  mck> load Fast;;
  mck> load Trex;;
  ```

  À la fin de cette section, nous indiquons quelques unes des options spécifiques des outils existants.

Dans la suite, nous présentons la partie de base du langage de commandes. La syntaxe que nous employons est celle d'OCAML. Ceci n'est en aucun cas l'expression d'un choix, mais plutôt une solution pour éviter (pour le moment) le choix d'une syntaxe spécifique.

Déclaration et affectation de variables de type région ou configuration (produit cartésien de représentations).

```
mck> let R = str2region(" ... ") ;;             # region
mck> let Rinit = init ;;                        # config initiale description
mck> let Rbad = (str2auto(...), str2sre(...)) ;; # config quelconque
mck> let Rnew = cap Rinit Rbad ;;               # expression
```

Les expression utilisées pour l'affectation des variables région (représentation) peuvent faire appel à toute fonction fournie par l'interface des implémentations choisies pour ces régions (représentations).

Supervision des calculs effectués par le model-checker. Ici, plusieurs commandes sont nécessaires :

– affichage d'une variable à l'écran ou dans un fichier :

```
mck> print R;;                          # sur stdout
...
mck> print Rnew to "fileRnew.aut";; # dans un fichier
```

– suivi des calculs pas à pas automatique et complet :

```
mck> trace on;;                         # debut
...
mck> trace off;;                        # fin
```

ou partiel :

```
mck> trace only transitions;;       # indique les trans appliquées
...
mck> trace only accelerations;;     # indique les acc calculées
```

Il est également utile de pouvoir interrompre ou suspendre les calculs lancés et afficher les calculs partiels. Une solution "système" est de rediriger des signaux (CTRL-S ou CTRL-Q) vers des fonction d'affichage qui seraient éventuellement réentrant.

Instanciation d'un model-checker générique fourni par défaut. L'idée est de fournir un "squelette"de model-checker, avec des fonctions génériques (choix des cycles, heuristique de recherche, quand déclencher l'accélération) instanciées soit à partir de code nouveau, soit avec du code déjà écrit. On aurait ainsi une petite bibliothèque de codes classiques (ex : les différents types de parcours, différentes recherche de cycles, . . .), ce qui permettrait d'expérimenter très vite des idées. Pour des besoins spécifiques, d'autres modules dédiés de "model-checker"peuvent être implantés.

Le model-checker générique aurait le code suivant :

```
let reach =
        let loops = precompute_loops()   # pre-calcul ds cycles à la Fast
        in
            reach_star reach_star.strategy loops
            ...
            ;;
```

Ce code générique (les fonctions `precompute_loops` ou `reach_star.strategy` sont génériques) pourra être ensuite instancié avec différentes fonctions concrètes.

Par exemple, le pre-calcul des cycles à accélérer pourra être fait en utilisant une des heuristiques de Fast :

```
mck> let precompute_loops = Fast.post_star_heur1;;
```

ou en lisant les meta-transitions données par l'utilisateur :

```
mck> let precompute_loops = read_meta_transitions "meta.tr" ;;
```

La stratégie de parcours (en largeur ou en profondeur) serait également à instancier :

```
mck> let reach_star.strategy = depth;;
```

Paramétrisation du `reach_star` avec des options classiques comme, par exemple :

- direction du parcours :

```
let reach_star.dir = forward ;;      # autre : backward
```

- définition d'un calcul sûr (*safe*) :

```
mck> let safety = ... ;;
```

  (`safety` étant un mot clé) pour l'analyse d'accéssibilité. Cette option vise à offrir une équivalence à l'option d'accessibilité avec observateur de TReX. Elle nous a semble plus élégante que le calcul explicite demandé par HyTech.
- masquage des transitions ou définition d'un observateur comportamental :

```
mck> let reach_star.set = [ t1, t2, [t2 , t3] ] lang;;
```

  (`set` étant un mot clé) pour le calcul des accélérations ou l'analyse d'accessibilité, définit l'ensemble des transitions à considérer.
  Ceci permet donc de compléter l'option d'accessibilité avec observateur de TReX : le calcul d'accessibilité est interrompu s'il est possible de tirer des transitions en dehors du langage spécifié par `reach_set`.
  Pour Fast, elle permet d'indiquer explicitement les combinaison de transitions à prendre en compte pour l'accélération.

Options spécifiques Fast à définir dans un module à charger avec la commande `load` :

- `int LOOP_LENGTH` : la longueur initiale des cycles à calculer.
- `int MAX_SIZE` : la taille maximale de l'automate courant.
- `int MAX_ACC` : le nombre maximale d'accélérations enchaînées.
- `string ALGO` : l'algorithme d'accélération employé.

Options spécifiques TReX à définir dans un module à charger avec la commande `load` :

- `int DFS_LOOP_DEEP` : la profondeur maximale des cycles à tester dans le parcours en profondeur.
- `state list EXTRAPOL_STATES` : la liste des états de contrôle pour lesquels on doit essayer d'extrapoler les cycles trouvés.
- `enum DECISION_PROCEDURE_REAL` : sélectionne la procédure de décision à utiliser pour résoudre les contraintes arithmétiques dans les réels.
- `enum DECISION_PROCEDURE_INT` : pareil pour les entiers.
- `bool SIMPLIFY_CONSTRAINTS` : simplification systématique des contraintes résultantes des calculs.
- `int MAX_CALL_REDUCE` : nombre maximal d'appels pour Reduce/redlog.
- `bool SUBSTITUTE_VARS` : appel d'heuristiques pour minimiser le nombre de paramètres d'extrapolation.
- `string OUTPUT_FILES` : préfixe pour les fichiers de sortie (graphe symbolique et configurations accessibles).

Références

[1] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In *Proc. 11th Int. Conf. on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 146–159. Springer-Verlag, 1999.

[2] P.A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using Forward Reachability Analysis for Verification of Lossy Channel Systems. *Formal Methods in System Design*, 25(1) :39–65, 2004.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1) :3–34, 1995.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[5] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic Techniques for Parametric Reasoning about Counter and Clock Systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434, Chicago (IL), USA, July 2000. Springer.

[6] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX : A Tool for Reachability Analysis of Complex Systems. In *Proc. 13th Int. Conf. Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372, Paris, France, July 2001. Springer.

[7] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2–3) :109–124, 1999.

[8] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. *Formal Methods in System Design*, 14(3) :237–255, 1999.

[9] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In David L. Dill, editor, *Proc. of the 6th Int. Conf. on Computer-Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67, Standford, California, USA, 1994. Springer-Verlag.

[10] T. Ball and S. K. Rajamani. The SLAM project : Debugging system software via static analysis. In *Conf. Record of the 29th Annual ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, January 2002.

[11] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST : Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121, Boulder, CO, USA, July 2003. Springer.

[12] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[13] B. Boigelot, F. Herbreteau, and S. Jodogne. Hybrid acceleration using real vector automata. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 193–205, Boulder, CO, USA, July 2003. Springer.

[14] A. Bouajjani and P. Habermehl. Symbolic Reachability Analysis of Fifo-Channel Systems with Nonregular Sets of Configurations. *Theoretical Computer Science*, 221(1-2) :211–250, 1999.

[15] M. Boyer and M. Sighireanu. Synthesis and Verification of Constraints in the PGM protocol. In *Proc. 12th Int. Symposium of Formal Methods Europe (FME '03)*, volume 2805 of *Lecture Notes in Computer Science*, Pisa, Italy, September 2003. Springer.

[16] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Proc. of the 9th Int. Confȯn Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, 1997.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th ACM/IEEE Design Automation Conference (DAC'90)*, pages 46–51, Orlando, Florida, 1990.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, 98(2) :142–170, June 1992.

[19] O. Burkart and J. Esparza. More infinite results. In *Current Trends in Theoretical Computer Science, Entering the 21th Century*, pages 480–503. World Scientific, 2001.

[20] C. Castel and Ch. Seguin. Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée. In *AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, June 2001.

[21] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J. W. De Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, pages 124–175, Noordwijkerhout, The Netherlands, 1993. REX School/Symposium, Springer-Verlag. Lecture Notes in Computer Science, vol. 803.

[22] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2) :244–263, 1986.

[23] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic modelchecking. *Journal of the ACM*, 50(5) :752–794, 2003.

[24] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5) :1512–1542, 1994.

[25] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. 4th Annual Symp. on Logic in Computer Science (LICS'99)*, pages 353–362. IEEE Computer Society, 1989.

[26] A. Collomb-Annichini and M. Sighireanu. Parameterized Reachability Analysis of the IEEE 1394 Root Contention Protocol using TReX. In P. Pettersson and S. Yovine, editors, *Proc. Workshop on Real-Time Tools (RTTOOLS'01), Aalborg, August 2001.* Uppsala University, Department of Information Technology, 2001. Appears as technical report 2001-014.

[27] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 238–252, Toronto, Canada, 1977.

[28] P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time ! In *Proceedings of 3rd Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–432. Springer, 1997.

[29] J. Esparza, S. Römer, and W. Vogler. An improvement of mcmillan's unfolding algorithm. In *Proc. of the 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer-Verlag, 1996.

[30] A. Finkel, S. P. Iyer, and G. Sutre. Well-abstracted transition systems. In *Proc. 11th Int. Conf. Concurrency Theory (CONCUR'00)*, volume 1877 of *Lecture Notes in Computer Science*, pages 566–580, University Park, PA, USA, August 2000. Springer.

[31] A. Finkel and J. Leroux. How to compose presburger-accelerations : Applications to broadcast protocols. In M. Agrawal and A. Seth, editors, *Proc. of the 22nd Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'02)*, volume 2556 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[32] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere ! *Theoretical Computer Science*, 256(1-2) :63–92, 2001.

[33] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[34] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2) :305–326, May 1994.

[35] A. Griffault. Conception et validation d'un protocole avec le modèle AltaRica. In *AFADL : Approches Formelles dans l'Assistance au Développement de Logiciels*, pages 293–307, January 2003.

[36] A. Griffault and A. Vincent. Vérification de modèles AltaRica, October 2003. MAJECSTIC : Manifestation des jeunes chercheurs STIC.

[37] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. 10th Int. SPIN Workshop (SPIN'03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, USA, May 2003. Springer. Tool paper.

[38] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH : A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2) :110–122, 1997.

[39] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering (TSE)*, 23(5) :279–295, 1997.

[40] K. L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1) :45–65, 1995.

[41] F. Moller. Infinite results. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer-Verlag, 1996.

[42] P. A. Abdulla, K. Čerâns, B. Jonsson, and Y-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. 11th Annual Symp. on Logic in Computer Science (LICS'96)*, pages 313–321. IEEE Computer Society, 1996.

[43] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1) :39–64, 1996.

[44] G. Point. *Altarica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement*. PhD thesis, LaBRI, Université Bordeaux 1, January 2000.

[45] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[46] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Proc. 10th Int. Conf. on Applications and Theory of Petri Nets (ATPN)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[47] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4) :297–322, 1992.

[48] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences (JCSS)*, 32(2) :183–221, 1986.

[49] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1) :1–37, November 1994.

A Exemples de signatures

Les signatures sont le moyen de spécifier les types infinis et les opérations sur ces types qui peuvent être utilisés dans la partie contrôle.

Dans la suite, nous présentons des exemples de signatures pour les types infinis actuellement traités par les outils Fast et TReX.

Type compteur Un premier type de compteur est le compteur "DBM". Il s'agit de variables entières qui peuvent être comparées ou affectées deux à deux. Plus précisement, uniquement les opérations suivantes sont permises :

```
sig DBMcnt
  0 : DBMcnt
  set : DBMcnt * int -> DBMcnt          /* x := set(y, n) */
  <= : DBMcnt * DBMcnt * int -> bool     /* x - y <= n */
   = : DBMcnt * DBMcnt * int -> bool     /* x - y = n */
gis
```

Des types de compteurs plus généraux, comme ceux représentables par les NDD ou UBA peuvent être définis comme suit :

```
sig counter
  0 : counter
  + : counter * int -> counter
  - : counter * counter -> counter
  <= : counter * int -> bool
gis
```

Type horloges Il s'agit d'un type réel dont les valeurs changent dans les états avec une pente de 1 par rapport à l'avacement du temps.

Un exemple de signature pour laquelle la représentation symbolique peut être un RVA est la suivante :

```
sig clock
  0 : clock
  -   : clock * real  -> clock
  -   : clock * clock -> clock
  +   : clock * real  -> clock
  +   : clock * clock -> clock
  <   : clock * clock -> bool
  <=  : clock * clock -> bool
gis
```

La signature du type réel nécessaire au type horloge ci-dessus étant :

```
sig real
  str2real : string -> real
  -   : real        -> real
  -   : real * real -> real
  <   : real * real -> bool
  <=  : real * real -> bool
gis
```

Si la représentation symbolique est les DBMs, la signature pour le type horloge est similaire à celle proposée pour le type compteur DBM.

Type canal avec pertes Une signature possible est :

```
sig lossy_fifo
  empty :                 -> lossy_fifo
  is_empty : lossy_fifo   -> bool
  top :      lossy_fifo   -> msg
  push : lossy_fifo * msg -> lossy_fifo
  pop :        lossy_fifo -> lossy_fifo
gis
```

où le type `msg` définit le contenu des messages dans le canal.

B Exemple de description : Bounded Retransmission Protocol

Dans cette section on montre comment le protocole BRP de Philips [28] peut être décrit en AltaRica étendu avec les signatures.

```
/**
 *************************************************
 *  BRP
 *************************************************
 *  Complete version with only 2 messages in the
 *  input file.
 *  Contains channels, clocks and counters.
 *************************************************
 */

/* finite type for program counters */
domain brp_state = { r_init, r_wait_next, r_ok,
                     s_init, s_fst, s_success, s_error };

/* finite type for messages exchanged */
domain brp_msg   = { m_fst0, m_fst1, m_last0, m_last1, m_ack0, m_ack1 };


/* infinite type for reals */
sig real
    str2real : string -> real
    -   : real        -> real
    -   : real * real -> real
    <   : real * real -> bool
    <=  : real * real -> bool
gis

/* infinite type for clocks */
sig clock
  0 : clock
  set : clock * real -> clock           /* x := set(y, n) */
  lt : clock * clock * real -> bool     /* x - y < r */
  le : clock * clock * real -> bool     /* x - y <= r */
  eq : clock * clock * real -> bool     /* x - y = r */
gis

/* infinite type for lossy chanels */
sig lossy_fifo
    empty :                 -> lossy_fifo
    is_empty : lossy_fifo   -> bool
    top :      lossy_fifo   -> brp_msg
    push : lossy_fifo * brp_msg -> lossy_fifo
    pop :        lossy_fifo -> lossy_fifo
gis

/* infinite type for counters */
sig counter
  0 : counter
  set : counter * int -> counter           /* x := y + n */
  le : counter * counter * int -> bool     /* x - y <= n */
  eq : counter * counter * int -> bool     /* x - y = n */
gis

/* global parameters */
```

```
const t1 : real = /* symbolic */;
      synch : real = /* symbolic */;
      max : int = /* symbolic */;
      t2 : real = /* symblic */;

/* Global system */
node BRP
  /* components */
  sub R : Receiver;
      S : Sender;

  /* events */
  event REQ, SOK, SNOK, SDNK,
        ROK, RNOK, RFST,
i;

  /* components connection */
  assert (S.M = R.M) && (S.A = R.A);

edon


/* Receiver */
node receiver
  /* local vars */
  flow  z : clock;
        M : lossy_fifo;
        A : lossy_fifo;

  state expab : bool;
        r_pc : brp_state; /* state counter */

  /* transitions */
  trans
    (r_pc = r_init) && (top(M) = m_fst0) |- RFST ->
        z := 0, M := pop(M), A := push(A, m_ack0), expab := true,
        r_pc := r_wait_next;

    (r_pc = r_init) && (top(M) = m_fst1) |- RFST ->
        z := 0, M := pop(M), A := push(A, m_ack1), expab := false,
        r_pc := r_wait_next;


    (r_pc = r_wait_next) && lt(z,0,t2)
                         && (expab) && (top(M) = m_last1) |- ROK ->
       z := 0, M := pop(M), A := push(A, m_ack1),
       r_pc := r_ok;

    (r_pc = r_wait_next) && lt(z,0,t2)
                         && (not(expab)) && (top(M) = m_last0) |- ROK ->
       z := 0, M := pop(M), M := push(A, m_ack0),
       r_pc := r_ok;

    (r_pc = r_wait_next) && lt(z,0,t2)
```

```
                        && (expab) && (top(M) = m_fst0) |- i ->
       M := pop(M), A := push(A, m_ack0);

    (r_pc = r_wait_next) && lt(z,0,t2)
                        && (expab) && (top(M) = m_last0) |- i ->
       M := pop(M), A := push(A, m_ack0);

    (r_pc = r_wait_next) && lt(z,0,t2)
                        && (not(expab)) && (top(M) = m_fst1) |- i ->
       M := pop(M), A := push(A, m_ack1);

    (r_pc = r_wait_next) && lt(z,0,t2)
                        && (not(expab)) && (top(M) = m_last1) |- i ->
       M := pop(M), A := push(A, m_ack1);

    (r_pc = r_wait_next) && eq(z,0,t2)
                        && is_empty(M) |- RNOK ->
       M := empty, A := empty,
       r_pc := r_init;


    (r_pc = r_ok) && lt(z,0,t2) && (top(M) = m_last1) |- i ->
       M := pop(M), A := push(A, m_ack1);

    (r_pc = r_ok) && lt(z,0,t2)
                  && (top(M) = m_last0) |- i ->
       M := pop(M), A := push(A, m_ack0);

    (r_pc = r_ok) && eq(z,0,t2)
                  && (is_empty(M)) |- i ->
       M := empty, A := empty,
       r_pc := r_init;

    (r_pc = r_ok) && lt(z,0,t2)
                  && (top(M) = m_fst0) |- RFST ->
       z := 0, M := pop(M), A := push(A, m_ack0), expab := true,
       r_pc := r_wait_next;

    (r_pc = r_ok) && lt(z,0,t2)
                  && (top(M) = m_fst1) |- RFST ->
       z := 0, M := pop(M), A := push(A, m_ack0), expab := false,
       r_pc := r_wait_next;

  /* time invariants in states */
  assert (r_pc = r_wait_next) && le(z,0,t2)
         (r_pc = r_ok) && le(z,0,t2);

edon



/* Sender */
node sender
  /* local vars */
```

```
flow  x : clock;
      M : lossy_fifo;
      A : lossy_fifo;

state ab : bool;        /* alternating bit */
      rc : counter;     /* retransmission counter */
      s_pc : brp_state; /* state counter */

trans
  (s_pc = s_init) && (ab) |- REQ ->
     x := 0, rc := 0, M := push(M, m_fst1),
     s_pc := s_fst;

  (s_pc = s_init) && not(ab) |- REQ ->
     x := 0, rc := 0, M := push(M, m_fst0),
     s_pc := s_fst;


  (s_pc = s_fst) && eq(rc,0,max)
                 && eq(x,0,t1)
                 && is_empty(M) && is_empty(A) |- SNOK ->
     x := 0, M := empty, A := empty,
     s_pc := s_error;

  (s_pc = s_fst) && lt(x,0,t1)
                 && (ab)
                 && (top(A) = m_ack1) |- i ->
     ab := not(ab), x := 0, rc := 0,
     A := pop(A), M := push(M, m_last0),
     s_pc := s_success;

  (s_pc = s_fst) && lt(x,0,t1)
                 && not(ab)
                 && (top(A) = m_ack0) |- i ->
     ab := not(ab), x := 0, rc := 0,
     A := pop(A), M := push(M, m_last0),
     s_pc := s_success;

  (s_pc = s_fst) && eq(x,0,t1)
                 && le(rc,0,max - 1)
                 && (ab) |- i ->
     x := 0, rc := set(rc,1), M := push(M, m_fst1);

  (s_pc = s_fst) && eq(x,0,t1)
                 && le(rc,0,max - 1)
 && not(ab) |- i ->
     x := 0, rc := set(rc,1), M := push(M, m_fst0);


  (s_pc = s_success) && eq(x,0,t1)
                     && eq(rc,0,max)
                     && is_empty(M) |- SDNK ->
     x := 0, M := empty,
     s_pc := s_error;
```

```
    (s_pc = s_success) && lt(x,0,t1)
                       && (ab)
                       && (top(A) = m_ack1) |- SOK ->
       x := 0, ab := not(ab),
       M := empty, A := empty,
       s_pc := s_init;

    (s_pc = s_success) && lt(x,0,t1)
                       && not(ab)
       && (top(A) = m_ack0) |- SOK ->
       x := 0, ab := not(ab),
       M := empty, A := empty,
       s_pc := s_init;

    (s_pc = s_success) && eq(x,0,t1)
                       && le(rc,0,max - 1)
       && (ab) |- i ->
       rc := set(rc,1), x := 0, M := push(M, m_last1);

    (s_pc = s_success) && eq(x,0,t1)
                       && le(rc,0,max - 1)
       && not(ab) |- i ->
       rc := set(rc,1), x := 0, M := push(M, m_last0);


    (s_pc = s_error) && eq(x,0,synch) |- i ->
       ab := false,
       s_pc := s_init;

  assert (s_pc = s_fst) && le(x,0,t1);
         (s_pc = s_success) && le(x,0,t1);
         (s_pc = s_error) && le(x,0,synch);

edon
```

C Exemple de diagnostique utilisateur

Pour la description du BRP, un exemple de diagnostique utilisateur est le suivant (la syntaxe n'est pas encore fixée) :

```
  domain
        dclk   (var t1, synch, t2, R.z, S.x) :dense; iby dbm
```

On indique que le domaine `dclk` (celui des horloges) est dense et implémenté par des DBMs. Les variables sur ce domaine sont `t1`, `synch`, `t2`, `R.z` et `S.x`.

```
  domain
        dcnt   (var S.rc, max) iby ndd;
```

Le domaine `dcnt` est celui des compteurs, qui est implémenté par des NDDs. Les variables `S.rc` et `max` sont représentées de cette façon.

```
  domain
        dfifo  (var R.M, R.A) iby sre;
```

notation	*domaine*	*signification*
O	Ops	fonction/opérateur
S	Sig	signature
V	Var	variable

TAB. 1 – Identificateurs pour les signatures

Enfin, le domaine `dfifo` est celui des canaux de communication. Cela concerne les variables `R.M` et `R.A` qui sont représentées par des SREs.

D Définition formelle des signatures

D.1 Syntaxe

Pour donner la syntaxe (abstraite) des signatures, nous utilisons le méta-langage de description syntaxique BNF (*Backus-Naur Form, BNF*). Dans ce formalisme, la description d'un langage est donnée sous la forme d'un ensemble de *productions* (ou règles de réécriture). Les éléments constitutifs de ces productions sont donnés dans la table ci-dessous.

élément	*signification*
xyz	symbole terminal xyz
ABC	symbole non-terminal *ABC*
|	choix entre deux éléments
[...]	zéro ou une occurrence d'un élément
{...}	zéro ou plusieurs occurrences d'un élément
(...)	groupement de plusieurs éléments
	concaténation (notée implicitement)

Une production a la forme suivante :

non-terminal ::= méta-expression

où la méta-expression est une expression construite en utilisant les éléments du tableau ci-dessus. Une production est interprétée comme suit : une définition du symbole non-terminal en partie gauche de "::=" peut être remplacée par toute sous-méta-expression alternative de la méta-expression donnée. Tout élément (y compris la concaténation) a priorité par rapport au choix sauf quand cette priorité est forcée par "(...)".

Les terminaux de cette syntaxe sont les suivants :

- les mots clés **sig** et **gis**
- les symboles réservés sont `:`, `->`, `*`
- les classes d'identificateurs utilisés, leur notation et leur domaine sont présentés dans le tableau 1. Les identificateurs d'opération peuvent être également des identificateurs spéciaux comme les symboles mathématiques.

Les classes de symboles non-terminaux utilisés et leur notation sont présentées dans le tableau 2.

La syntaxe (abstraite) proposée pour les signatures est donnée par la grammaire suivante :

SL	::=	[SD $\{SD\}$]	*liste de définitions de signatures*	(S1)
SD	::=	**sig** S OL **gis**	*définition de signature*	(S2)
OL	::=	OD { OD }	*liste de déclarations d'opérations*	(S3)
OD	::=	O `:` $\vec{S}$ `->` S	*déclaration d'opération*	(S4)
$\vec{S}$	::=	[S {`*` S}]	*liste de signatures paramètres*	(S5)

Les signatures seront utilisées :

notation	*domaine*	*signification*
SL	SigList	liste de définition de signatures
SD	SigDef	définitions de signature
OL	OpsList	liste de déclaration d'opérations
OD	OpsDef	déclaration d'opération
$\vec{S}$	SigIdList	liste d'identificateurs d'opérations

Tab. 2 – Non-terminaux pour la partie données

prédicat	*section*
$\mathcal{C} \vdash SL \Rightarrow \mathcal{C}'$	
$\mathcal{C} \vdash SD \Rightarrow \mathcal{C}'$	
$\mathcal{C} \vdash OL \Rightarrow \mathcal{C}'$	
$\mathcal{C} \vdash OD \Rightarrow \mathcal{C}'$	
$\mathcal{C} \vdash \vec{S} \Rightarrow \vec{S}$	
$\mathcal{C} \vdash VD \Rightarrow \mathcal{C}'$	
$\mathcal{C} \vdash E \Rightarrow S$	

Tab. 3 – Prédicats de typage pour les signatures et les expressions.

– dans la déclaration de variables pour préciser leur type et
– dans les expressions employées dans les gardes et les affectations des transitions.

Nous précisons ci-dessous la syntaxe de ces constructions afin de pouvoir leur préciser, dans la suite, leur sémantique.

$$VD ::= V \ : S \ ; \qquad \textit{déclaration de variable} \quad \text{(S1)}$$
$$E ::= O \ (\ EL \ ) \qquad \textit{expression appel d'opérateur d'une signature} \quad \text{(S2)}$$
$$EL ::= [\ E \ \{ \ , \ E\} \ ] \qquad \textit{liste d'arguments} \quad \text{(S3)}$$
$$\text{(S4)}$$

D.2 Sémantique statique

La sémantique statique pour la définition de signatures a deux rôles : (1) vérifier que la définition de la signature est bien formée et (2) construire l'environnement de typage contenant les définitions de signatures et ses opérations.

Dans cette section, nous ne donnons pas la sémantique statique des expressions construites en utilisant les opérations des signatures, car elle fait naturellement partie de la définition formelle du langage AltaRica.

Nous présentons la sémantique statique sous la forme d'un système de types Milner-Hindley, afin de faciliter la lecture. Le tableau 3 présente les prédicats de typage définis par les règles d'inférence.

Dans les règles d'inférence est utilisé l'objet sémantique "environnement de typage", noté $\mathcal{C}$,

qui prends des valeurs dans l'algèbre de termes engendrée par la grammaire suivante :

$$\begin{array}{rcll}
\mathcal{C} & ::= & S \mapsto \ \textbf{type} & \textit{type/signature}\,(\mathcal{C}1) \\
& | & O \mapsto \{\textit{profile}\} & \textit{opération}\,(\mathcal{C}2) \\
& | & V \mapsto S & \textit{variable}\,(\mathcal{C}3) \\
& | & () & \textit{vide}\,(\mathcal{C}4) \\
& | & \mathcal{C}, \mathcal{C} & \textit{composition disjointe}\,(\mathcal{C}5) \\
\textit{profile} & ::= & [\vec{S} \to S] & \textit{profile d'opération}\,(\mathcal{C}6)
\end{array}$$

Nous utiliserons la notation $\mathcal{C} \vdash S \Rightarrow \textbf{type}$ pour dire que $(S \mapsto \textbf{type}) \in \mathcal{C}$.

La surcharge des opérations nous impose de relier un identificateur d'opération à une liste de profils. La résolution des surcharges doit lier chaque identificateur d'opération au profil unique correspondant à son utilisation. Informellement, deux opérations O_i et O_j appartenant respectivement aux signatures S_i et S_j (S_i peut être égal à S_j) peuvent avoir le même nom (surchargé), $S_i = S_j$, si au moins une des conditions ci-dessous est satisfaite :

- les signatures de leur résultat sont différentes,
- le nombre de paramètres de O_i et de O_j est différent,
- les produits cartésiens des types des paramètres de O_i et de O_j sont différents

Ces contraintes pour la surcharge des noms sont formalisées dans la définition suivante de la composition disjointe "," entre les environnements de typage correspondant aux opérations.

$$(\mathcal{C}_1, \mathcal{C}_2)(O) \stackrel{\text{def}}{=} \begin{cases} \mathcal{C}_1(O) & \textbf{si } O \notin \textit{Dom}(\mathcal{C}_2) \\ \mathcal{C}_2(O) & \textbf{si } O \notin \textit{Dom}(\mathcal{C}_1) \\ \mathcal{C}_1(O) \cup \mathcal{C}_2(O) & \textbf{si } \forall \vec{S}_1 \to S_1 \in \mathcal{C}_1(C)\ \forall \vec{S}_2 \to S_2 \in \mathcal{C}_2(C) \\ & \quad ((S_1 \neq S_2) \vee (\vec{S}_1 \neq \vec{S}_2)) \\ \textit{erreur} & \textbf{sinon} \end{cases}$$

Comme pour les types, nous utilisons la notation $\mathcal{C} \vdash O \Rightarrow \textit{profile}$ pour dire que $\textit{profile} \in \mathcal{C}(O)$, afin d'obtenir le profile (unique) correspondant à l'utilisation de l'opérateur O.

Règles de typage : Nous donnons maintenant les règles de typage pour les définitions de signatures ainsi que pour le typage des expressions.

Pour les définitions de signatures, nous définissons un prédicat de typage de la forme $\mathcal{C} \vdash SD \Rightarrow \mathcal{C}'$, qu'il faut lire : "dans l'environnement $\mathcal{C}$, la définition SD est bien formée et élabore l'environnement $\mathcal{C}'$".

Cette règle utilise le prédicat de typage pour la liste d'identificateur de signatures (utilisée pour déclarer le profile des opérateurs) $\vec{S}$ ayant la forme $\mathcal{C} \vdash \vec{S} \Rightarrow \vec{S}$, qu'il faut lire : "dans l'environnement $\mathcal{C}$, la liste $\vec{S}$ est bien formée et élabore la liste de types $\vec{S}$".

Les règles ci-dessous donnent le typage d'une liste de définitions de signature. Afin de ne pas compliquer l'écriture de ces règles, nous n'avons pas géré ici les définitions récursives de signatures. On suppose donc qu'il existe une phase initiale de construction de l'environnement de typage pour les signatures définies. Cette phase doit également signaler les définitions multiples d'un même identificateur de signature.

Les premières n pré-conditions de la première règle disent que chaque définition doit être correcte dans l'environnement de typage où les autres signatures sont présentes (afin de permettre des définitions récursives de signatures). La $n+1$-ème pré-condition dit que les environnements de typage générés doivent être disjoint pour les signatures (noms uniques) et les noms des opérateurs doivent être correctement surchargé. Le résultat du typage est l'environnement composé de toutes les définitions de signatures.

La deuxième règle dit que la liste d'opérateurs doit être correctement déclarée en présence de l'identificateur de la signature définie.

$$\frac{\mathcal{C} \vdash SD_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C} \vdash SD_n \Rightarrow \mathcal{C}_n \quad \mathcal{C}_1, ..., \mathcal{C}_n \neq \mathit{erreur}}{\mathcal{C} \vdash (SD_1 ... SD_n) \Rightarrow \mathcal{C}_1, ..., \mathcal{C}_n}$$

$$\frac{\mathcal{C}, S \mapsto \mathbf{type} \vdash OL \Rightarrow \mathcal{C}_O}{\mathcal{C} \vdash (\mathbf{sig}\ S\ OL\ \ \mathbf{gis}) \Rightarrow (S \mapsto \mathbf{type}), \mathcal{C}_O}$$

Les règles suivantes typent les déclarations d'opérateurs. La première dit que chaque déclaration doit être correcte et que la composition des déclarations doit respecter les règles de la surcharge. La deuxième dit qu'une déclaration est correcte quant tous les signatures des arguments et du résultat sont bien définies dans l'environnement de typage.

$$\frac{\mathcal{C} \vdash OD_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C} \vdash OD_n \Rightarrow \mathcal{C}_n \quad \mathcal{C}_1, ..., \mathcal{C}_n \neq \mathit{erreur}}{\mathcal{C} \vdash (OD_1 ... OD_n) \Rightarrow \mathcal{C}_1, ..., \mathcal{C}_n}$$

$$\frac{\mathcal{C} \vdash \vec{S} \Rightarrow \vec{S} \quad \mathcal{C} \vdash S \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (O \ \ \mathbf{:}\ \vec{S}\ \texttt{->} S) \Rightarrow (O \mapsto \vec{S} \rightarrow S)} \qquad \frac{\mathcal{C} \vdash S_1 \Rightarrow \mathbf{type} \quad \cdots \quad \mathcal{C} \vdash S_n \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (S_1 \texttt{*} ... \texttt{*} S_n) \Rightarrow (S_1, ..., S_n)}$$

Ces dernières règles montrent comment il faut intégrer le typage des signatures dans le typage AltaRica pour les variables et les expressions. La première règle dit qu'une déclaration de variable est correcte si la signature a été définie ; le résultat du typage est un environnement de typage où V est liée à la signature S. La deuxième règle dit qu'une expression appel d'opérateur est bien typée si les expressions des arguments sont bien typées et qu'il existe un seul opérateur qui a le nom O et les types des arguments $S_1, ..., S_n$ dans l'environnement de typage. Le type de l'expression est la signature résultat de l'expression.

$$\frac{\mathcal{C} \vdash S \Rightarrow \mathbf{type}}{\mathcal{C} \vdash (V \ \ \mathbf{:}\ \vec{S}) \Rightarrow (V \mapsto S)} \qquad \frac{\mathcal{C} \vdash E_1 \Rightarrow S_1 \quad \cdots \quad \mathcal{C} \vdash E_n \Rightarrow S_n \quad \mathcal{C} \vdash O \Rightarrow (S_1, ..., S_n) \mapsto S}{\mathcal{C} \vdash O(E_1, ..., E_n) \Rightarrow S}$$

D.3 Sémantique dynamique

La sémantique dynamique des signatures n'est pas définie. En effet, l'évaluation des opérateurs définis par les signatures dépend de la représentation symbolique associée à cette signature.