Représentation de programmes SMALA grâce à la théorie des bigraphes

Cécile Marcon, Nicolas Nalpon, Cyril Allignol, Célia Picard
ENAC, Toulouse, France

Résumé

Dans le but de certifier le compilateur du langage réactif-interactif SMALA, nous cherchons à représenter sa sémantique en instanciant la théorie des bigraphes. En nous limitant dans un premier temps à un sous-ensemble de ce langage, nous avons pu représenter ses différents acteurs comme des entités distribuées dans l'espace. Des règles de réaction sur les bigraphes permettent de mettre ces entités en mouvement et en relation les unes avec les autres afin de représenter l'exécution du programme en plus de sa structure.

Mots-clés : bigraphes, sémantique, compilation certifiée, langage réactif

1 Introduction

Dans le secteur aéronautique comme dans de nombreux autres, les humains interagissent avec des systèmes toujours plus nombreux et critiques, y compris pour leur sécurité. Un système *interactif* est manipulé par des humains et doit en réponse réagir de façon correcte, fiable voire certifiée. Un système interactif-reactif réagit aux entrées utilisateurs mais aussi à d'autres (issues de capteurs par exemple).

La plupart des langages de programmation ne facilitent pas le passage de la phase de design à celles de conception, architecture et développement, compliquant la création de logiciels interactifs. Ce constat motive le développement de SMALA, un langage réactif [1] mettant l'accent sur l'aspect interactif des programmes. Il permet de décrire un programme à la manière des systèmes basés composant [2]. On y retrouve des entités qu'on peut composer et connecter entre elles afin de les coordonner. SMALA repose sur de larges bibliothèques de composants, notamment graphiques, développés en C++. Il est surtout utilisé pour concevoir des IHM pour l'aviation, comme l'interface du tableau de bord de l'hélicoptère électrique Volta [3].

Pour pouvoir utiliser un langage dans la conception de systèmes critiques il faut fournir des garanties sur l'exécution de ses programmes. Ainsi, nous envisageons de vérifier le compilateur de SMALA en commençant par s'assurer de la préservation des liens de causalité entre les entités. Pour cela, nous nous inspirons de l'approche des compilateurs Vélus [4] et CompCert [5] : nous implémentons et vérifions un

compilateur SMALA grâce à l'assistant de preuve Coq. Pour l'instant, nous ignorons la partie graphique du langage. De plus, pour faciliter la vérification, nous travaillons avec SMALIGHT, un sous-ensemble de SMALA, couvrant ses principes de base et permettant de redéfinir les bibliothèques actuelles en C++.

Notre première définition formelle de ce sous-ensemble [6] a soulevé plusieurs problèmes. En effet, l'ensemble choisi était trop petit, ne permettant pas de couvrir tous les concepts de SMALA. De plus, l'expression de l'ordre d'exécution des entités du programme nécessitait l'introduction de trop nombreux paramètres, rendant les règles peu intuitives et difficiles à faire évoluer. De même que le lambda calcul est approprié pour définir les sémantiques opérationnelles de langages fonctionnels, nous voulons exprimer notre sémantique avec une théorie mathématique adaptée. La sémantique de SMALA est basée sur une double structure d'arbre et de graphe définissant les règles d'activation des entités du programme. Mais dans notre première approche, nous avons exprimé la priorité d'exécution avec une liste d'ensembles, compliquant la compréhension. Étant donnée la structure de SMALA, nous avons décidé d'utiliser la théorie des bigraphes pour formaliser la sémantique d'une version étendue de SMALIGHT. La capacité des bigraphes à modéliser et implémenter des langages de programmation a déjà été démontrée [7]. Ils ont ainsi été utilisés pour exprimer la sémantique du langage Kappa basé sur des règles d'interaction entre protéines [8]. Ici, nous utilisons la théorie des bigraphes pour modéliser la structure d'un programme SMALIGHT et les règles d'activation de ses éléments.

La section 2 présente SMALA et SMALIGHT. La section 3 définit les éléments de la théorie des bigraphes utiles pour la suite. La section 4 détaille la formalisation de la sémantique avec les bigraphes. La section 5 présente la suite des travaux.

2 Réduction de SMALA en SMALIGHT

SMALA repose sur deux concepts : les processus et une notion de dynamicité.

2.1 Processus

Un processus est une entité qui a une sémantique et un état (Activé ou Désactivé). Si un processus est Activé, il s'exécute selon sa sémantique. S'il est Désactivé, il est inactif. De plus, un processus peut être *persistant* (son activation perdure jusqu'à la fin de l'exécution ou sa désactivation par un autre processus) ou *transitoire* (il s'exécute selon sa sémantique à l'activation puis se désactive immédiatement).

Les processus de SMALA couvrent tous les aspects du langage : graphiques, interactifs, structures de contrôle, création de composants, gestion de l'activation et de la mémoire. Néanmoins, la plupart peuvent être vus comme la composition de processus plus élémentaires. Dans SMALIGHT, nous visons à identifier l'ensemble minimal de ces processus permettant la redéfinition de tous les autres. Ainsi, actuellement, les processus de SMALIGHT (en omettant la partie graphique) sont :

— binding : processus persistant permettant à un processus p_1 (le notifiant) de

communiquer son état d'activation à un processus p_2 (l'abonné). Ainsi, le binding $p_1 \rightarrow p_2$ active p_2 quand p_1 s'active. Il en existe trois autres types :
  - Le binding $p_1 \rightarrow! p_2$ désactive p_2 quand p_1 s'active
  - Le binding $p_1 !\rightarrow p_2$ active p_2 quand p_1 se désactive
  - Le binding $p_1 !\rightarrow! p_2$ désactive p_2 quand p_1 se désactive
- component : processus persistant correspondant à un contenant nommé. On appelle enfants les processus qu'il contient. Ils sont activés dans l'ordre de leur définition dans le code (figure 1a) lorsque le component est activé.
- spike : processus transitoire dont la sémantique est de ne rien faire. Il est généralement utilisé pour notifier ses abonnés d'un changement d'état.
- property : processus persistant encapsulant une valeur stockée en mémoire et notifiant ses abonnés lorsque cette valeur change. Il en existe de différents types mais SMALIGHT contient seulement les IntProperty pour l'instant, sur lesquelles on peut effectuer des opérations arithmétiques.
- assignment : processus transitoire noté $=:$. Ainsi, $e_1 =: p_2$ copie la valeur de l'expression e_1 dans la property p_2 lorsqu'il est activé.

La figure 1a présente un exemple d'un programme SMALIGHT très simple.

2.2 La dynamicité

La notion de dynamicité dans SMALA couvre trois aspects : la propagation (seule traité pour l'instant), la mémoire et la création dynamique de processus.

La propagation concerne toutes les modifications dues à des changements d'état de processus dans un programme SMALA. Ces propagations sont de deux types différents : celles issues des components et celles issues des bindings.

La définition d'un component induit la création d'une relation hiérarchique avec ses enfants, représentée par un arbre (figure 1c). Si la racine change d'état, on propage ce changement aux enfants en parcourant les dans l'ordre de leur déclaration et récursivement. De plus, un enfant est Activé seulement si son parent l'est aussi.

Un binding ou un assignment crée une relation entre les processus qui peut être représentée par un graphe orienté acyclique (figure 1b). Ainsi, si un processus source (i. e., n'ayant pas de prédécesseur) change d'état, ce changement est propagé aux autres processus du graphe en suivant un ordre total issu d'un tri topologique.

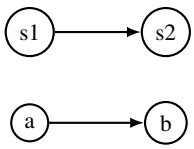
3 La théorie des bigraphes

La théorie des bigraphes, formellement définie par Milner [9], représente les relations et les interactions entre des entités grâce à un système réactif bigraphique consistant en une paire de graphes (graphe de places et graphe de liens, représentant respectivement l'imbrication des entités les unes dans les autres et leurs relations) appelée bigraphe et un ensemble de règles de réaction sur ce bigraphe.

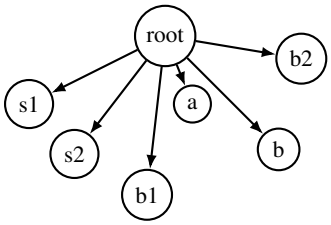
Le graphe de places (figure 2b) est une forêt d'arbres qui permet de décrire la localisation des nœuds et en particulier leur imbrication les uns dans les autres. La

```
_main_
Component root {
    Spike s1
    Spike s2
    Int a (42)
    Int b (0)
    b1: s1 -> s2
    b2: a =: b
}
```

(a) Programme Smala



(b) Graphe binding et assignment



(c) Arbre du component root

FIGURE 1 – Structure d'un programme SMALA

racine de ses arbres est une région (une entité qui peut en contenir d'autres mais ne peut être contenue ; dans la figure 2a ce sont les rectangles blancs, dans la figure 2b, les racines) et leurs feuilles sont des nœuds ou des sites (une entité qui peut être contenue par une région ou des nœuds et pouvant être substituée par un graphe de places ; dans la figure 2a, c'est le rectangle grisé et dans la figure 2b, la feuille '0'). Les régions et les sites permettent de composer des graphes de places entre eux.

Le graphe de liens (figure 2c) est un hypergraphe décrivant des relations entre entités par des hyperarêtes (arêtes connectant plusieurs nœuds). Ses nœuds ont des ports permettant aux arêtes de se fixer. Les arêtes brisées ont une extrémité non connectée et permettent la composition de graphes de liens ('e4' en figure 2).

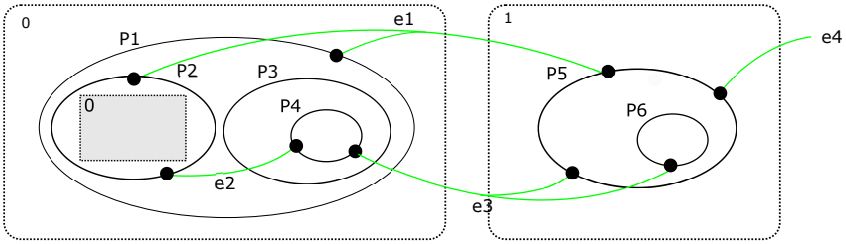
Une règle de réaction R est une paire de bigraphes (G, D) ayant autant de sites, de régions et d'arêtes brisées. On la note $R : G \longrightarrow D$. Appliquer R à un bigraphe B consiste à remplacer une occurrence de G se trouvant dans B par D.

4 De SMALIGHT vers les bigraphes

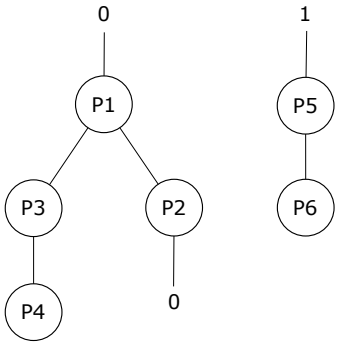
Nous avons choisi de représenter les règles d'activation d'un programme SMALIGHT en l'assimilant à un système réactif bigraphique. L'assignment et la property ne sont pas présentés ici. En effet, bien qu'ils possèdent un mécanisme de propagation d'activation, une part important de leur fonctionnement est liée à la mémoire qui n'est pas traitée dans cet article.

4.1 Représentation générique des processus

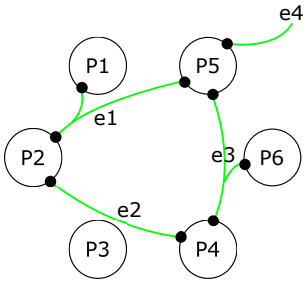
Nous proposons tout d'abord une représentation générique des processus de SMALIGHT qui nous permet d'appliquer les règles de réaction de façon efficace. Ainsi, un processus est représenté par un nœud Process. Il contient un ensemble de nœuds, possédant chacun un site, décrivant ses attributs (voir figure 3a). A ce stade, nous avons identifié 4 attributs qui suffisent pour décrire totalement n'importe quel processus de SMALIGHT. Ils sont décrits ici avec les conventions graphiques utilisées le cas échéant : un identifiant, (représenté si nécessaire par une police



(a) Bigraphe

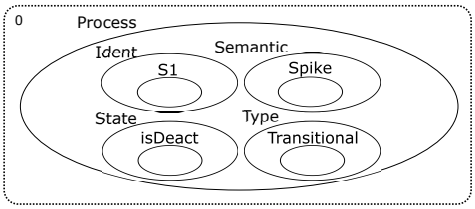


(b) Le graphe de places

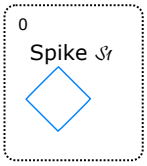


(c) Le graphe de liens

FIGURE 2 – Un bigraphe et les graphes de places et de liens correspondants



(a) En bigraphe



(b) Avec les conventions graphiques choisie

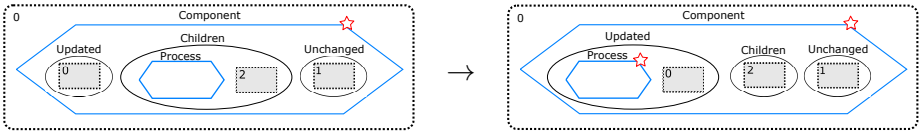
FIGURE 3 – Le modèle générique présenté avec le spike de SMALIGHT

cursive), une sémantique (component, spike, etc. représentant un processus), un état (un nœud Activé est violet ; Désactivé, il est bleu) et un type (transitoire est représenté par un losange, persistant par un hexagone). Nous ne représentons les ports que si la règle de réaction concerne le graphe de liens.

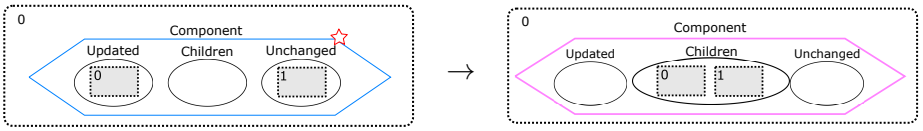
Nous décrivons l'activation et la désactivation des processus par un ensemble de règles de réaction, afin de représenter l'exécution d'un programme SMALIGHT.

4.2 Logique de base des règles de réaction

La section 2.2 présente l'aspect dynamique de SMALA notamment les deux règles de propagation d'activation. L'une concerne le component et dit que son activation provoque celle de tous ses enfants. On le traduit par des règles de réaction sur le graphe de places. L'autre concerne l'activation des bindings et se traduit par



(a) Règle récursive appliquée tant que *Children* n'est pas vide



(b) Règle d'arrêt quand *Children* est vide

FIGURE 4 – Activation du *Component*

des règles de réaction sur le graphe de liens, détaillées en section 4.3.2.

Pour activer un component on applique plusieurs règles de réaction sur le bigraphe. On doit savoir que le component est en train de s'activer et que ses enfants doivent l'être aussi. Pour cela, on place le component dans un nœud *Activate* servant de *flag*, symbolisé par une étoile rouge en haut à droite du processus. Finalement, par souci de généralisation et pour ne pas avoir à écrire trop de règles de réaction, nous appliquons le flag *Activate* à tous les processus dès leur activation avant de passer leur état à l'état Activé. La désactivation suit un principe similaire.

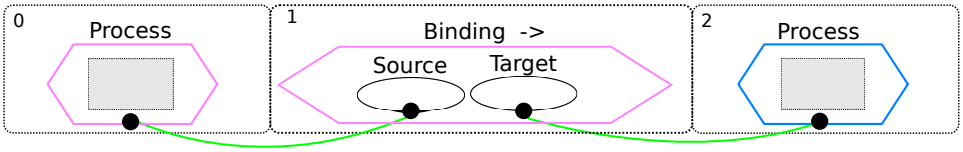
4.3 Instanciation sur les processus de SMALIGHT

Ces modèles généraux sont instanciés ici pour le component et le binding. Le spike est représenté comme un nœud *Spike* unique (voir figure 3). La property et l'assignment, utilisant la mémoire pas encore traitée, ne sont pas présentés.

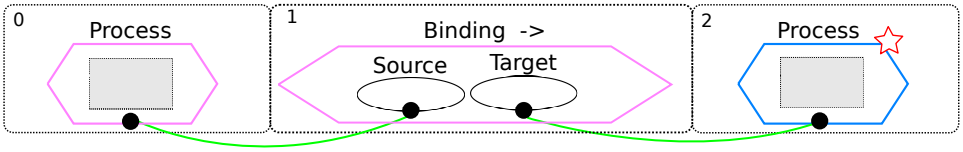
4.3.1 Le component

Le component est un processus qui en contient d'autres. On le représente par un nœud *Component* possédant un enfant *Children* contenant tous ses enfants.

Pour activer un *Component*, et donc tous ses enfants, on doit connaître les processus restant à activer à chaque application d'une règle de réaction. De plus l'activation d'un enfant dépend de son type (transitoire ou persistant). Nous avons donc ajouté deux autres sites comme enfants de *Component* pour gérer cela. Ainsi, à l'activation d'un Component, ses enfants persistants sont déplacés dans un nœud *Updated* (voir figure 4) et ses enfants transitoires dans un nœud *Unchanged*. Lorsque tous les nœuds ont été traités, le nœud *Children* est vide. On peut alors passer l'état du component à Activé et replacer l'ensemble des enfants dans le nœud *Children*.



(a) Motif recherché pour appliquer la règle de réaction



(b) Réécriture du motif : activation du processus Target

FIGURE 5 – Règle de réaction sur le *Binding* $\rightarrow$ avec un processus source persistant

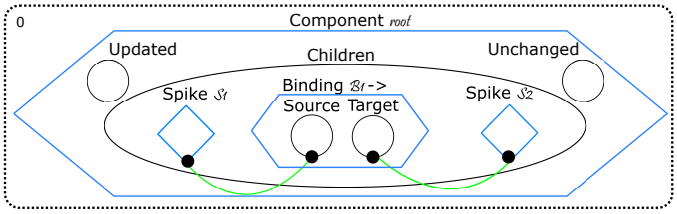


FIGURE 6 – Le bigraphe représentant le programme SMALIGHT de la figure 1a

4.3.2 Le binding

Le binding est représenté comme un nœud ayant deux enfants *Source* et *Target*. Chacun comprend un port qui le lie à un nœud Process correspondant au processus source, respectivement cible, du binding. Ainsi, les règles de réaction du binding $\rightarrow$ correspondent à la recherche de bindings dont le processus source est activé et le processus cible pas encore (figure 5). Les autres bindings suivent le même principe.

4.3.3 Exemple

On peut donc décrire un programme SMALIGHT en bigraphe grâce aux éléments précédents. Le bigraphe de la figure 6 correspond ainsi au code de la figure 1a.

5 Travaux en cours et à venir

Cet article présente une formalisation du langage SMALIGHT et de ses règles d'activation sous forme d'un système réactif bigraphique. Cette formalisation permet de représenter des interactions simples entre les différents processus de SMALIGHT. Toute la sémantique de SMALIGHT n'est pas encore couverte. Parmi les aspects importants à traiter, il y a la synchronisation de la propagation d'activation et les modifications en mémoire. La synchronisation vise à activer un processus si et seulement si tous les processus qui le précèdent ont fini leur exécution. Pour

cela, nous travaillons sur des règles de réaction qui propagent l'activation selon un tri topologique. Concernant la mémoire, nous cherchons aussi à exploiter la théorie des bigraphes pour la représenter et raisonner dessus. Nous avons testé cette formalisation en l'implémentant à l'aide de BigraphER [10], un outil développé en OCaml qui permet de simuler des systèmes réactifs bigraphiques. Par ailleurs, BigraphER a une place importante dans notre chaîne de compilation certifiée. En effet, nous travaillons à la traduction de cet outil dans Coq afin de pouvoir l'utiliser pour implémenter et vérifier notre sémantique. Enfin, voulant avoir une chaîne de compilation certifiée de bout en bout, nous cherchons à générer du code séquentiel C à partir de l'outil BigraphER afin de pouvoir s'appuyer sur CompCert.

Références

[1] E. BAINOMUGISHA, A. L. CARRETON, T. van CUTSEM et al., « A Survey on Reactive Programming », *ACM Computing Surveys*, 2013.

[2] F. ARBAB, « Abstract Behavior Types: a foundation model for components and their composition », *Sci. Comput. Program.*, t. 55, n° 1-3, p. 3-52, 2005.

[3] P. ANTOINE et S. CONVERSY, « Volta: the first all-electric conventional helicopter », *Proceedings of the More Electrical Aircraft Conference*, 2017.

[4] T. BOURKE, L. BRUN, P.-E. DAGAND et al., « A Formally Verified Compiler for Lustre », *PLDI*, 2017.

[5] X. LEROY, « Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assitant », *POPL*, 2006.

[6] N. NALPON, C. PICARD, C. ALLIGNOL et al., « Vers la vérification de SMALA, un langage réactif interactif », *AFADL 2020*, 2020.

[7] E. HØJSGAARD, « Bigraphical Languages and their Simulation », PhD Thesis, The IT University of Copenhagen, 2011.

[8] V. DANOS, J. FERET, W. FONTANA et al., « Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models », *FSTTCS 2012*, Hyderabad, India, 2012, p. 276-288.

[9] R. MILNER, *The Space and Motion of Communicating Agents*. New-York : Cambridge University Press, 2009.

[10] M. SEVEGNANI et M. CALDER, « BigraphER: Rewriting and Analysis Engine for Bigraphs », *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada*, 2016, p. 494-501.