Implémentation des Bigraphes dans Coq

Cécile Marcon, Cyril Allignol, Xavier Thirioux, Celia Picard

Implémentation des Bigraphes dans Coq

Cécile Marcon, Cyril Allignol, Xavier Thirioux, Celia Picard

Résumé

Les bigraphes sont un modèle mathématique introduit par Robin Milner. Ils peuvent être utilisés pour représenter des systèmes concurrents et distribués. Nous avons implémenté une sémantique formelle des bigraphes dans l'assitant de preuve Coq. Cet article présente l'implémentation choisie et comment instancier un bigraphe dans notre bibliothèque.

1 Introduction

La vérification formelle permet de raisonner rigoureusement sur des systèmes complexes et d'en vérifier la correction. L'une des façons de vérifier des systèmes est d'utiliser des assistants de preuve. Ces assistants de preuve peuvent être utilisés pour modéliser des systèmes et raisonner sur eux.

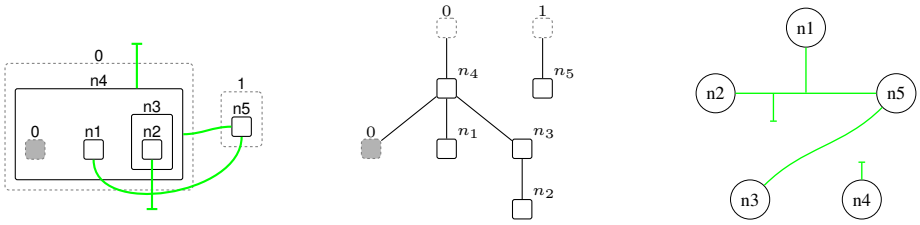
Un modèle mathématique qui nous intéresse particulièrement est celui des bigraphes : une théorie introduite par Robin Milner [1]. Les bigraphes permettent de représenter et d'analyser les systèmes distribués, ils ont par ailleurs déjà été outillés pour le model checking [2].

Notre objectif est double : tout d'abord formaliser les concepts fondamentaux de la théorie des bigraphes dans Coq [3] –soit la représentation des bigraphes et la sémantique opérationnelle– et ensuite utiliser cette formalisation pour permettre la vérification formelle de systèmes interactifs modélisés à l'aide de bigraphes [4].

Cet article présente un travail réalisé dans le cadre du premier objectif : nous avons implémenté une bibliothèque contenant le type bigraphe ainsi que quelques opérateurs usuels dans Coq. Nous décrivons ci-après comment est défini ce type et comment instancier un bigraphe dans notre bibliothèque.

2 Bigraphes : définitions

Cette section présente les bigraphes tels qu'ils ont été introduits par Milner. Un bigraphe est essentiellement constitué de deux structures : une forêt et un hypergraphe, basés sur le même ensemble fini de sommets appelé *nœuds*. La forêt est appelée *graphe de places* et représente la hiérarchie et le placement des nœuds. L'hypergraphe est appelé *graphe de liens* et représente les connexions entre les nœuds, ou plus précisément entre leurs *ports*. Nous pouvons représenter ces deux structures dans un même graphe appelé bigraphe, comme le montre Figure 1.



(a) Bigraphe (b) Graphe de places (c) Graphe de liens

FIGURE 1 – Un bigraphe et sa décomposition en graphes de places et de liens

Les bigraphes ont été conçus pour modéliser des systèmes concurrents et distants. Pour cela ils peuvent être assemblés côte à côte ou les uns dans les autres : ils possèdent une *interface* qui permet ou non l'interaction. L'interface d'un bigraphe est constituée de ses *sites*, ses *racines*, ses *ports ouverts* et de ses *arêtes ouvertes*.

Les rectangles pointillés vides sont les racines : elles se trouvent au sommet de la hiérarchie et hébergent des bigraphes. Les rectangles pointillés gris sont les sites : ce sont des espaces vides dans lesquels un bigraphe peut être inséré. Les sites et les racines sont appelés *places* et sont représentés par des entiers naturels.

Le graphe de places peut être traduit en une fonction acyclique *parent* qui associe chaque site et nœud à son nœud ou à sa racine parent :

$$parent : n\text{œ}ud \uplus site \longrightarrow n\text{œ}ud \uplus racine$$

Les arêtes ouvertes sont les arêtes brisées vers le haut, et les ports ouverts sont celles brisées vers le bas. Nous les appelons *liens ouverts*, ils portent des noms (identifiants) tirés d'un ensemble infini de *Noms*.

Le graphe des liens peut être traduit en une fonction *lien* qui associe à chaque port et port ouvert son arête ou arête ouverte connectée :

$$lien : port\ ouvert \uplus port \longrightarrow ar\hat{e}te \uplus ar\hat{e}te\ ouverte$$

Les bigraphes obéissent également à une *signature* de la forme $(\kappa,\ ar\acute{i}t\acute{e})$ où κ est un ensemble de catégories de nœuds et $ar\acute{i}t\acute{e} : \kappa \to \mathbb{N}$ est une fonction qui associe un nombre de ports pour chaque catégorie de nœud.

Une fonction *contrôle* associe chaque nœud à une catégorie (et donc implicitement un nombre de ports).

Ainsi, pour l'interface $\langle sites, ports\ ouverts\rangle \to \langle racines, ar\hat{e}tes\ ouvertes\rangle$, un bigraphe est donc défini par un 5-tuple $\langle n\text{œ}uds, ar\hat{e}tes, contrôle, parent, lien\rangle$. À l'aide de ces définitions, nous implémentons un type *bigraphe* dans l'assistant de preuve Coq.

3 Implémentation du type bigraphe avec Coq

L'assistant de preuve Coq permet aux développeurs de définir formellement des types et de vérifier les preuves de théorèmes. Nous utilisons Coq pour formaliser

la théorie des bigraphes et vérifier notre implémentation.

Cela débute par la définition d'un type `bigraph` dans un module `Bigraphs`. Ce module est paramétré par deux Module : `Signature` et `Names`. `Signature` instancie $(\kappa, arite)$. Ce choix implique que lorsqu'on instancie `Bigraphs`, ses bigraphes partagent tous cette même signature et modélisent des systèmes similaires. De plus, les liens ouverts sont du même type `Names` que nous axiomatisons infini et muni d'une égalité décidable.

Notre objectif est de n'implémenter que des bigraphes concrets dont tous les champs sont correctement identifiés. Dans cette optique, nous pouvons définir un bigraphe comme un **`Record`** (un tuple étiqueté qui peut avoir des paramètres) :

```
Record bigraph (site: nat) (*site*)
               (innername: NoDupList) (*port ouvert*)
               (root: nat) (*racine*)
               (outername: NoDupList) (*arête ouverte*)
:= Big { node : FinDecType;
         edge : FinDecType;
         control : type node -> Kappa;
         parent : type node + fin site ->
                    type node + fin root;
         link : NameSub innername + Port control ->
                     NameSub outername + type edge;
         ap : Acyclic parent }.
```

Cela signifie que `bigraph s i r o` est le type de bigraphes avec l'interface $\langle s, i\rangle \to \langle r, o\rangle$, où $s \in \mathbb{N}$, $r \in \mathbb{N}$ et `i` et `o` les liens ouverts, sous-ensembles de `Names` représentés par des `NoDupList` (listes sans doublons de `Names`). Nous accédons à toutes les places avec `fin p` $= \{n \in \mathbb{N} \mid n < p\}$ et à tous les liens ouverts avec `NameSub l` $= \{n \in$ `Name` $\mid n \in l\}$.

Nous représentons les nœuds et les arêtes avec le type `FinDecType` qui décrit un type fini et muni d'une égalité décidable.

Les autres champs suivent les définitions de Section 2 :

— `control` attribue un type à chaque nœud.

— `parent` donne le parent (nœud ou racine) de chaque nœud et site. La notation $+$ doit être lue comme une somme disjointe ($\uplus$).

— `ap` est la preuve que *parent* est acyclique. Nous définissons Acyclic avec le constructeur `Acc` de la bibliothèque standard qui assure que nous n'avons pas une chaîne ascendante infinie de nœuds parents.

— `link` donne l'arête ou l'arête ouverte associée à chaque port ou port ouvert. L'ensemble des ports est atteignable depuis la fonction contrôle avec le constructeur `Port control` $= \{(n, i) \in$ `node` $\times \mathbb{N} \mid i <$ `control` $n\}$.

Enfin, nous souhaitons aborder le choix de placer l'interface en tant que paramètre du bigraphe. Notre motivation est de pouvoir composer et juxtaposer les bigraphes. Cependant, pour ce faire, nous devons avoir accès à leurs interfaces sans nécessairement avoir besoin d'accéder aux autres éléments du bigraphe.

4 Instanciation d'un bigraphe

Figure 2 représente le bigraphe relativement basique que nous voulons créer.

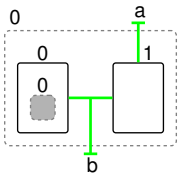


FIGURE 2 – Un simple bigraphe

Tout d'abord, nous instancions les Module Type Signature ($\kappa = \mathbb{N}$, $arite = id$) et Names ($Name = char$). Trivialement nous avons $s = 1, r = 1, i = [b], o = [a]$. `node = bool`, `edge = unit` permettent d'éviter les calculs arithmétiques.

Il reste à écrire nos fonctions descriptives (`z` est 0 dans `fin 1`, donc la racine).

```
Definition mycontrol (node:node) : Kappa := match node with
  | false => 1 | true => 2 end.
Definition myparent (ns : node + site) : node + root :=
  match ns with | inl n => inr z | inr s => inl false end.
```

Nous écrivons la fonction `lien` directement en vernaculaire (dans le le langage de preuve) : il s'agit de faire une disjontion de cas d'abord entre les ports ouverts et ports, puis sur les nœuds, et enfin sur les deux ports du nœud *true*.

La preuve de l'acyclicité de parent se fait en dépliant la définition de Acc qui amène à la contradiction de l'hypothèse d'une égalité entre un nœud et une racine.

5 Conclusion et travaux à venir

Nous avons présenté la formalisation en Coq de la théorie des bigraphes que nous avons réalisée et discuté des choix d'implémentation. Cependant, l'exemple que nous avons présenté soulève des objections : définir un bigraphe à partir de zéro est fastidieux, peu intuitif, et ne permet pas de se le représenter facilement. Afin de pallier ceci, nous avons ajouté à notre bibliothèque des opérateurs sur les bigraphes (composition, juxtaposition etc.) que nous détaillerons dans de futurs travaux.

Références

[1] R. MILNER, *The space and motion of communicating agents*. Cambridge University Press, 2009.

[2] M. SEVEGNANI et M. CALDER, « BigraphER: rewriting and analysis engine for bigraphs », in *Computer Aided Verification: 28th International Conference, Proceedings, Part II 28*, Springer, 2016.

[3] Y. BERTOT, « A Short Presentation of Coq », in *Theorem Proving in Higher Order Logics*, Berlin, Heidelberg : Springer Berlin Heidelberg, 2008.

[4] N. NALPON, C. ALLIGNOL et C. PICARD, « Towards a User Interface Description Language Based on Bigraphs », in *ICTAC 2022*, 2022.