BiCoq : Bigraphs Formalisation with Coq

Cécile Marcon
Fédération ENAC ISAE-SUPAERO
ONERA, Université de Toulouse
Toulouse, France

Cyril Allignol
Fédération ENAC ISAE-SUPAERO
ONERA, Université de Toulouse
Toulouse, France

Celia Picard
Fédération ENAC ISAE-SUPAERO
ONERA, Université de Toulouse
Toulouse, France

Blair Archibald
University of Glasgow
Glasgow, Scotland

Michele Sevegnani
University of Glasgow
Glasgow, Scotland

Xavier Thirioux
Fédération ENAC ISAE-SUPAERO
ONERA, Université de Toulouse
Toulouse, France

Abstract

Bigraphs are a formal model for representing (ubiquitous) systems with strong notations of both *space*, e.g. a person in a room, and *non-spatial relations*, e.g. mobile phone communication regardless of location. They have been used in a wide range of scenarios including sensor systems, IoT configuration languages, and communications protocol design. While implementations of the bigraph theory exist, e.g. BigraphER, until now, there has been no attempt to formalise the theory in a theorem prover. We show an implementation of the bigraph theory in the Coq theorem prover, including the main bigraph type specification and common manipulation operators, e.g. composition and tensor product. This is a key step to fully formalising the theory and paves the way for a certified implementation for use in safety critical scenarios.

CCS Concepts

• Software and its engineering → Formal software verification.

Keywords

Formal methods, theorem proving, bigraphs

ACM Reference Format:
Cécile Marcon, Cyril Allignol, Celia Picard, Blair Archibald, Michele Sevegnani, and Xavier Thirioux. 2025. BiCoq : Bigraphs Formalisation with Coq. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC'25), March 31-April 4, 2025, Catania, Italy.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3672608.3707824

1 Introduction: Reasoning upon Bigraphs

Milner's bigraphs [20] are an expressive modelling formalism particularly for systems that feature both spatial and non-spatial interactions [6, 9, 25], and those with strong notions of concurrency and interaction. A system's state is modeled by a bigraph, and system interactions are be modeled as a set of (reaction) *rules* that change a pattern inside of bigraph with another. A bigraph completed with a set of rules is called a Bigraphical Reactive System (BRS).

Multiple tools [23, 26] allow working with, and reasoning over, bigraphs, via e.g. model checking. They allow efficient property verification such as determining whether a pattern exists within each state of a BRS, and reachability properties such as deadlocks. However, these existing tools focus mainly on *system/model* verification: there are no guarantees the underlying theory is implemented correctly, and no ability to generate correct-by-construction code. We go beyond model checking by providing verified bigraph semantics.

Theorem proving is a powerful tool to rigorously reason about (computational) models and provide a strong mathematical grounding for theory. We choose to use the Coq [5] proof assistant for its ability to extract computer (OCaml) code, ensuring correctness at the implementation level and facilitating integration into real-world systems. Moreover, bigraphs have demonstrated their relevance to represent language semantics [8, 21], which paves the way to (verified) compiling using bigraphs as an intermediate representation. Coq has been widly used for verified compilation whose goal is ensuring that the implementation of a system complies to its formal specification (an approach similar to Compcert [18] or Velus [7]).

We present BiCoq [19], our formalization of the bigraph theory with the Coq proof assistant. We describe the main bigraph type, major bigraph operators, e.g. composition and tensor products, and derived operators, e.g. parallel product, nesting and merge product, and prove that our encoding satisfies category axioms, therefore showing that our formalisation is trustworthy and consistent with the theory as originally presented by Milner. This provides a foundation to encode more of the bigraph semantics, including the rewriting theory that allows systems to evolve over time.

Paper Outline. Section 2 presents the bigraph theory, main definitions and notations. In Section 3, we justify our choices for implementing bigraphs in Coq. Section 4 presents our equivalence definition, and Section 5 presents two different operators and proofs of the correctness of their behaviour. Section 6 presents derived operators. Related work is in Section 7 and we conclude in Section 8.

2 Bigraphs: Definitions

While bigraphs have been extended in several ways, we only model *pure* bigraphs as introduced by Milner [20].

We give key bigraph definitions and notations used throughout the paper. We then describe elementary bigraphs used for later definitions, and briefly present the category theory behind bigraphs.

SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03
https://doi.org/10.1145/3672608.3707824

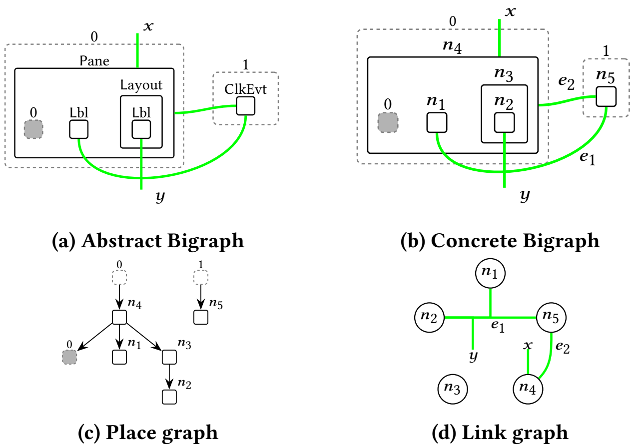

Figure 1: A bigraph modelling a user interface and corresponding concrete bigraph, place graph, and link graph.

2.1 Elements of Bigraphs

We start from the definitions of a forest—a disjoint union of trees—and of a hypergraph—a graph where edges can connect more than two vertices. A bigraph is these two structures (called a *place graph* and *link graph* respectively) over the same finite set of vertices called *nodes* (V). We represent these two structures in one diagram (e.g. Fig. 1 represents a basic user interface).

2.1.1 Place Graph. The *place graph* represents the hierarchy and placing of nodes (e.g. in Fig. 1c, the edge $n_4 \longrightarrow n_1$ means n_1 is nested in n_4). The unfilled dashed rectangles are called *roots* or *regions* and are at the top of the hierarchy. The grey dotted rectangles are called *sites*. Sites are blank spaces where any bigraph with one root can fit (they are used to nest bigraphs into one another). *Sites* and *roots* are called *places*. Places are represented by natural numbers which denote ordinal sets of that order, e.g. root = 2 means there are two roots labelled $\{0, 1\}$ (see Fig. 1b).

The place graph can be represented by an acyclic function *prnt* of type: $node \uplus site \longrightarrow node \uplus root$, where $\uplus$ denotes disjoint union. It associates each *site* and *node* to its parent *node* or *root*. The *prnt* function is total so it is not possible to have a disconnected node or site that is not in a root.

2.1.2 Link Graph. The *link graph* represents the (hyper-)*edges* (E) between nodes (Fig. 1d). Bigraphs have a *basic signature* of the form (κ, *arity*) where κ is the set of entity types e.g. {Lbl, Layout, ...}, and $arity : \kappa \rightarrow \mathbb{N}$ is the function that maps an entity to the number of *ports* it has. A function *ctrl* associates each *node* to its entity types/control (and so its number of *ports*), e.g. in Fig. 1b, $ctrl(n_1) =$ Lbl and $arity($Lbl$) = 1$.

Some edges of the link graph connect upwards to *outernames* (x in Fig. 1b). Others connect downwards to *innernames* (y in Fig. 1b). *names* are drawn from an infinite set $\mathcal{X}$. Links with no names are *closed*. The link graph can be represented by a function *link* of type $innername \uplus port \longrightarrow edge \uplus outername$. This function is also total, which means all *ports* must connect to an *edge*/name. When a link is open (i.e. it has an *outername*) it does not need an *edge* label since the name uniquely identifies this link. This definition of *link* permits multiple *innernames* to connect to the same link, but you cannot have multiple *outernames* on a link.

We define the *support* of a bigraph (notation $|b|$) as the union of nodes and edges: $|b| = V \uplus E$. The notion of support will be used later (see Section 4) to build an equivalence between bigraphs.

Lastly, we distinguish between two types of bigraphs. *Abstract* bigraphs have unnamed support. They assign *controls*, e.g. Lbl, to nodes but do not give nodes specific identifiers (see Fig. 1a). *Concrete* bigraphs use controls as well, but also assign identifiers to nodes, e.g. n_1 (see Fig. 1b). It is possible to move between these representations, e.g. by forgetting identifiers, or assigning arbitrary identifiers to nodes and edges.

Bigraphs are compositional (algebraic) objects, i.e. larger bigraphs can be built from smaller ones. Every bigraph has an inner and outer *interface* that may or may not allow such operations. The interface of a bigraph is spcified by its *innernames*, *sites* (*inner face*), *outernames* and *roots* (*outer face*). An interface can then be described as an arrow in the form $\langle site, innername \rangle \rightarrow \langle root, outername \rangle$.

In summary, we can give the following definition for a bigraph:

Definition 2.1 (Bigraph). For each interface $\langle site, innername \rangle \rightarrow \langle root, outername \rangle$, a bigraph is a 5-tuple $\langle node, edge, ctrl, prnt, link \rangle$

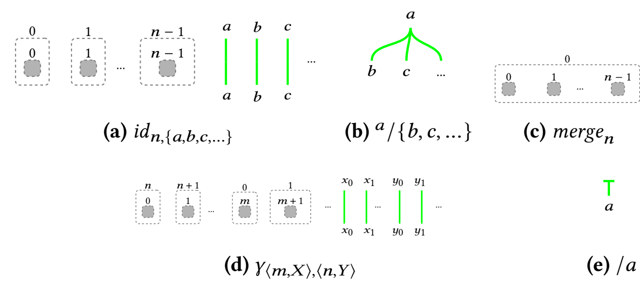

Figure 2: Elementary bigraphs.

2.2 Elementary Bigraphs

We present here some common bigraphs used throughout the paper.

2.2.1 Identity Bigraphs (Fig. 2a). A family of node-free bigraphs that map sites to regions, and names to names such that the interface is maintained:

$\forall s \in \mathbb{N}, \forall i \subset \mathcal{X}, id_{s,i} : \langle s, i \rangle \rightarrow \langle s, i \rangle = \langle \emptyset, \emptyset, \emptyset, id, id \rangle$

We call ϵ the empty interface $\langle 0, \emptyset \rangle$, and id_ϵ the empty bigraph.

2.2.2 Merge Bigraphs (Fig. 2c) . A family of node-free and name-free bigraphs collapsing n sites into 1 root:

$\forall n \in \mathbb{N}, merge_n : \langle n, \emptyset \rangle \rightarrow \langle 1, \emptyset \rangle = \langle \emptyset, \emptyset, \emptyset, s \mapsto 0, \emptyset \rangle$

2.2.3 Symmetries (Fig. 2d). $\gamma_{\langle m,X \rangle,\langle n,Y \rangle}$ are node-free bigraphs that allow the movement of roots. Bigraph roots do not commute, instead they use explicit symmetries to move.

$$\forall m, n \in \mathbb{N}, \forall X, Y \subset \mathcal{X},$$
$$\gamma_{\langle m,X \rangle,\langle n,Y \rangle} : \langle m + n, X \uplus Y \rangle \rightarrow \langle m + n, X \uplus Y \rangle$$
$$= \langle \emptyset, \emptyset, \emptyset, s \mapsto (n + s) \bmod (m + n), id \rangle$$

2.2.4 Substitutions (Fig. 2b). $^y/X$ are place-free bigraphs that rename (sets of) innernames X to a specific outername y.

$\forall y \in \mathcal{X}, \forall X \subset \mathcal{X}, ^y/X : \langle 0, X\rangle \to \langle 0, \{y\}\rangle = \langle \emptyset, \emptyset, \emptyset, \emptyset, i \mapsto y\rangle$

2.2.5 Closures (Fig. 2e). $/x$ are place-free bigraphs that link an innername x to an *idle edge* (it does not export an outer name). Idle edges are "invisible" and can be ignored for equivalence (see Section 4.2): $/x : \langle 0, x\rangle \to \langle 0, \emptyset\rangle = \langle \emptyset, \{e\}, \emptyset, \emptyset, i \mapsto e\rangle$

2.3 Categorical Axioms

While there are many bigraph categories, we focus on the category of abstract bigraphs (i.e. bigraphs with unnamed support).

As stated by Milner [20] (Theorem 2.20) and further explained in Section 4, abstract bigraphs form an *spm-category* (a symmetric partial monoidal category) quotiented over $\eqsim$ (see Section 4.2). Objects are interfaces and morphisms are bigraphs. The category allows composition $\circ$, tensor product $\otimes$, symmetries $\gamma_{I,J}$ and *id* of Section 2.2.1 are the left and/or right neutral elements for these operators. This category is *partial* because $\otimes$ is only defined on disjoint interfaces as $\langle m, X\rangle \otimes \langle n, Y\rangle = \langle m+n, X \uplus Y\rangle$. For interfaces X and Y, $X \otimes Y$ and $Y \otimes X$ are either both defined or both undefined.

An *spm-category* follows the following equations:

$$\forall g, f, \exists g \circ f \Leftrightarrow cod(f) = dom(g) \tag{C1}$$

$$\forall f : K \to L, g : J \to K, h : I \to J, h \circ (g \circ f) = (h \circ g) \circ f \tag{C2}$$

$$\forall f, id \circ f = f \circ id = f \tag{C3}$$

Eqs. (M1), (M2) and (M3) define a partial monoidal category:

$$\forall f, g, h, h \otimes (g \otimes f) = (h \otimes g) \otimes f \tag{M1}$$

$$\forall f, id \otimes f = f \otimes id = f \tag{M2}$$

$$\begin{aligned}&\forall f_1 : I \to J, f_0 : M \to I, g_1 : K \to L, g_0 : N \to K,\\ &\qquad (f_1 \otimes g_1) \circ (f_0 \otimes g_0) = (f_1 \circ f_0) \otimes (g_1 \circ g_0)\end{aligned} \tag{M3}$$

These equations allow symmetry for the partial monoidal category:

$$\forall I, \gamma_{I,\epsilon} = id_I \tag{S1}$$

$$\forall I, J, \gamma_{J,I} \circ \gamma_{I,J} = id_{I \otimes J} \tag{S2}$$

$$\forall f : I_0 \to I_1, g : J_0 \to J_1, \gamma_{I_1,J_1} \circ (f \otimes g) = (g \otimes f) \circ \gamma_{I_0,J_0} \tag{S3}$$

$$\forall I, J, K, \gamma_{I \otimes J,K} = (\gamma_{I,K} \otimes id_J) \circ (id_I \otimes \gamma_{J,K}) \tag{S4}$$

We now implement abstract bigraphs as a type `bigraph` in the proof assistant Coq and prove these axioms.

3 Coq Formalisation

The first challenge to implementing bigraph theory in Coq is getting the right type to formally describe a bigraph. As bigraphs are presented algebraically, within a categorical framework, Coq with its higher-order logic and dependent types is a good fit for implementation. We draw experience from the Gross *et al.* [16] category implementation in Coq that revealed that unwise implementation choices may lead to poor efficiency limits. Specifically, parameterized types should be used carefully.

3.1 Representing Sets and Acyclic Functions

To model bigraphs we need some mathematical components. We use some existing tools from the MathComp library [12] (e.g. finite types), but implement some new tools. These are available in the `MyBasics`, `Names` and, `MathCompAddings` files.

Bigraphs are defined over different types of sets. The nodes (V) and edges (E) are finite sets (MathComp's `finType`). As regards interfaces, places are ordinals (MathComp's `Ordinal`), while inner/outer names are finite sets drawn from an infinite set of names.

To this purpose, we implement a **`Module Type`** `NamesParameter` (a module type is an abstract specification for a module), which includes a type $\mathcal{X}$, a proof it has a decidable equality (`EqDec` $\mathcal{X}$), and a characterization of its infinite nature: $\forall l : list\ \mathcal{X}, \exists n : \mathcal{X}, n \notin l$.

We draw the subsets of $\mathcal{X}$ using `NoDupList`, that gives lists of $\mathcal{X}$ with no duplicates. We decide to represent sets of names as lists for the vast list library that will aid implementing bigraph algorithms.

Innernames and outernames types are built from the `NoDupList` `innername` and `outername` using the constructor `NameSub` where `NameSub ndl` $= \{name \in \mathcal{X} \mid name \in$ `ndl`$\}$.

To ensure the directed place graph is a forest, we need to enforce it is acyclic. Our definition is based on `Acc` from the standard library that forbids building an infinite ascending chain:

```
Definition Acyclic {N I O} (p : N+I -> N+O) :=
    forall n, Acc (fun n' n => p (inl n) = inl n') n.
```

Here `Acyclic` is defined generically for `p` where its domain and range are respectively the disjoint sums `N+I` and `N+O`. Acyclicity is only meaningful when iterating `p` through `N` (nodes), so generic injection functions `inl:N -> N+X` are used in the definition. It will be applied to the *prnt* function of a place graph.

3.2 The `bigraph` Type

We represent the set of bigraphs with interface $\langle s, i\rangle \to \langle r, o\rangle$ as the dependent type `bigraph s i r o`, implemented as follows in the `AbstractBigraphs` file:

```
Record bigraph  (site : nat) (innername : NoDupList)
               (root : nat) (outername : NoDupList) : Type :=
  Big {
    node : finType; edge : finType; control : node -> Kappa;
    (* Place graph *)
    parent : node + (ordinal site) -> node + (ordinal root);
    ap : Acyclic parent;
    (* Link graph *)
    link : (NameSub innername) + Port control ->
            (NameSub outername) + edge;
  }.
```

Much like the interpretation of bigraphs as functions between interfaces, this definition hides the internal details, i.e. the specific nodes/edges are hidden in the output type and kept internal to the record. This is a natural choice when performing categorical operations on bigraphs, e.g. composition or tensor product (see Sections 5.2 and 5.3) that need to check the inner/outer names and sites/regions are adequate without looking at the internal structure.

Two module parameters are used here. `NamesParameter` handles the global set of names (see Section 3.1). `SignatureParameter` contains the bigraph signature (κ, *arity*), that determines the specific entities in the system and their (fixed) number of ports, and is encoded as two parameters: **`Variable`** `Kappa : ` **`Type`** and **`Variable`** `Arity : Kappa -> nat`.

Following Section 2.1 the bigraph record contains `node` and `edge` and a function `control` providing the specific control for each node.

The place graph is encoded by `parent`, determining the parent (*node*/*root*) for each *node*/*site*. We use `ap` to ensure *prnt* is acyclic.

For link graphs we first construct a set of `Ports`. These can be determined from the `control` function (which allows us to infer the node set, and to compute how many ports each node has). Then, we create a set of dependent pairs (notation `&`) of nodes and their port labels (up to the specified arity) as follows:

```
Definition Port {node : Type} (control : node -> Kappa):
   Type := { n : node & fin (Arity (control n)) }.
```

Using this, the function `link` determines the assignment of *outernames* or edges to ports (or *innernames*).

Remark: bigraph notations. To lighten the presentation, in the following, b_1 and b_2 will implicitly denote bigraphs with respective interfaces $\langle s_1, i_1\rangle \to \langle r_1, o_1\rangle$ and $\langle s_2, i_2\rangle \to \langle r_2, o_2\rangle$. Similarly, any internal element (e.g. nodes) of b_1 and b_2 will also be referred to through the index notation (e.g. n_1 and n_2).

4 Equivalence Between Bigraphs

Defining equivalences between bigraphs is desirable as it allows bigraph comparisons and thus is necessary for pattern matching and in order to prove that our implementation is correct.

The native Coq support for comparison is Leibniz equality. However, it is too restrictive for our purpose as Leibniz equality compares objects and compels their inner types to be equal. In a typed setting, Leibniz equality is homogeneous, meaning we can only compare objects with the same type: that is, bigraphs with the exact same interface. The issue can be made explicit with a simple example: the set of names $\{a, b\}$ can be represented as the `NoDupList` $[a, b]$ as well as $[b, a]$, which the Leibniz equality would deem as unequal, even though we do not care about ordering in a set. Representing set of names as characteristic functions of type $\mathcal{X}$ `-> bool` has the same issue. Instead, we implement our own (reflexive, symmetric, transitive) equivalences on bigraphs, that are also congruences with respect to bigraph operations.

In the theory, there are two equivalences of interest: support-equivalence, denoted $\bumpeq$, and lean-support equivalence, denoted $\Bumpeq$. Recall the support of a bigraph $|b|$ is the set of nodes and edges. Support-equivalence between b_1 and b_2 implies the existence of a bijection between $|b_1|$ and $|b_2|$ that respects the structures of both bigraphs. This also implies equality of the interfaces, i.e. if a node connects to a name, it connects to the same name in the mapping. Lean-support equivalence has the same requirements but disregards *idle edges*, which are unconnected edges (this can occur during composition with a closure Fig. 2e). Both definitions are similar, lean-equivalence only requiring a filter of idle edges.

Our implementation requires defining bigraph isomorphisms using bijections (see the `Bijections` file) for each element of the support, and checking the functions describing the structure (*prnt*/*link*) behave the same through the bijections. Doczkal *et al.* [10] implement a similar equivalence modulo isomorphism on graphs.We write $A \cong B$ for the set of bijections between sets A and B and $bij_{A,B}$ for an element of this set.

Bijections already enjoy a group structure, with function composition, inverse and identities as elements. We add operators to compose bijections through typical set constructions such as disjoint sum $A \uplus B$, function space $A \longrightarrow B$, subset $\{a : A \mid P(a)\}$, dependent sum $\{a : A \,\&\, B(a)\}$ and ordinal $[0, a[$ and prove that they respect the group structure. The main operators we need are:

$$
\begin{aligned}
\_ \twoheadrightarrow \_ &: A \cong B \to C \cong D \to (A \to C \cong B \to D) \\
\_ \mathbin{\langle + \rangle} \_ &: A \cong B \to C \cong D \to (A \uplus C \cong B \uplus D) \\
\langle\{\_ \mid \_\}\rangle &: \forall bij_{A,B} : A \cong B, (\forall a : A, P(a) \Leftrightarrow Q(bij_{A,B}(a))) \to \\
&\quad \{a : A \mid P(a)\} \cong \{b : B \mid Q(b)\} \\
\langle\{\_ \,\&\, \_\}\rangle &: \forall bij_{A,B} : A \cong B, (\forall a : A, C(a) \cong D(bij_{A,B}(a))) \to \\
&\quad \{a : A \,\&\, C(a)\} \cong \{b : B \,\&\, D(b)\} \\
\underline{=} &: \forall a, b \in \mathbb{N}, a = b \to [0, a[\cong [0, b[
\end{aligned}
$$

4.1 Support-Equivalence

We define (in the `SupportEquivalence` file) support equivalence over bigraphs $b_1 \bumpeq b_2$, by the conjunction of these 10 named properties:

$$
\begin{aligned}
& s_1 = s_2 && (equ_s) \\
& \forall\ name \in \mathcal{X},\ name \in i_1 \Leftrightarrow name \in i_2 && (equ_i) \\
& r_1 = r_2 && (equ_r) \\
& \forall\ name \in \mathcal{X},\ name \in o_1 \Leftrightarrow name \in o_2 && (equ_o) \\
& \exists bij_n \in n_1 \ \cong\ n_2 && (bij_n) \\
& \exists bij_e \in e_1 \ \cong\ e_2 && (bij_e) \\
& \forall n_1 \in node_1, \\
& \exists bij_{p,n} \in [0, Ar(ctrl_1(n_1))] \cong [0, Ar(ctrl_2(bij_n(n_1)))] && (bij_p) \\
& (bij_n \twoheadrightarrow id_\kappa)(ctrl_1) = ctrl_2 && (equ_c) \\
& ((bij_n \mathbin{\langle + \rangle} \overline{equ_s}) \twoheadrightarrow (bij_n \mathbin{\langle + \rangle} \overline{equ_r}))(prnt_1) = prnt_2 && (equ_p) \\
& (((\langle\{id \mid equ_i\}\rangle \mathbin{\langle + \rangle} \langle\{bij_n \,\&\, bij_{p,n}\}\rangle) \twoheadrightarrow (\langle\{id \mid equ_o\}\rangle \mathbin{\langle + \rangle} bij_e)) \\
& \quad (link_1) = link_2 && (equ_l)
\end{aligned}
$$

The first four requirements enforce equality on the bigraph interfaces, i.e. they have the same sites/roots and inner/outer names.

For nodes and edges, we require a bijection between the sets (i.e. between the support, but it is clearer to treat the support set in parts). As ports operate a little like additional nodes we also require a bijection between the sets of ports (for each node) through bij_n, i.e. we can only map ports if we can map nodes. Moreover, for nodes, their controls must be maintained (Eq. (equ_c)) through $bij_n \twoheadrightarrow id_\kappa$.

Finally, we need to confirm the *prnt* (place graph) and *link* (link graph) remain valid under the bijections constructed from the node, edge and port bijections, as is handled by Eqs. (equ_p) and (equ_l). These are essential as it proves the bijections are structure preserving so that we have a real isomorphism between two bigraphs.

To prove $\bumpeq$ is an equivalence, we provide the bijections between elements, proofs of equality, and prove the functions equations.

For the reflexivity, symmetry and transitivity proofs, we respectively use identities, inverse and composition of bijections, and the morphism properties described in the introduction of this section.

Remark: homogeneous vs. heterogeneous equivalence. Our equivalence being between two heterogeneous types implies that we cannot directly use rewriting strategies. Coq users will know that we use a well-known method of packing the bigraph and its interface into `bigraph_packed`. This allows us to create a second equivalence based on the packed bigraphs: two packed bigraphs are equal iff their bigraph are $\simeq$ equivalent. Our previous proofs trivially give us that this new relation is an equivalence.

4.2 Lean-Support Equivalence

As mentioned before, lean-support equivalence is support equivalence that ignores idle edges.

To lean a bigraph, we filter out the idle edges using a predicate `not_is_idle` which states that there exists an preimage for an edge through *link*. This predicate allows us to create a new `finType` for edges. We rebuild the new *link* function from the old one trivially since there is no change to it (by definition, the idle edges that we removed were never an image of any point).

We now implement $b_1 \risingdotseq b_2 := (lean(b_1)) \simeq (lean(b_2))$. This relation is clearly an equivalence. We also prove the useful lemma: $b_1 \simeq b_2 \implies b_1 \risingdotseq b_2$.

We now move to defining operators on bigraphs.

5 Bigraphs as an *spm-category*

We prove that our abstract bigraphs implementation complies with the axioms of an *spm-category* (Section 2.3), by implementing *composition* (in `Composition`), which places regions into sites and joins like-names, and *tensor product* (in `TensorProduct`), which juxtaposes bigraphs side-by-side, as well as symmetries (in `Symmetries`).

5.1 Interface Requirements

Bigraphs operators usually come with constraints on their arguments. For example, composition of bigraphs b_1 and b_2 requires innerface of b_1 and outerface of b_2 to be equal. On the contrary, for tensor product of b_1 and b_2, their interfaces must be name disjoint.

All these requirements are encased in classes so they are discharged by Coq's automated class instance search without the need to provide explicit proofs. For the search to succeed, we must provide base cases and reasoning rules as class instances. For example, with sites and roots we may generate instances such as: $n = n$; $n + 0 = n$; $n + m = m + n$; $n = m \rightarrow m = n$. Similar rules apply for equality or disjointness of inner and outer names.

The class for natural numbers equality is in MyEqNat and the classes for equality between finite subtypes and disjointness of finite types are in Names.

Remark: requirement notations. In the following, interface requirements will appear as: $Requirement \Rrightarrow Definition$.

5.2 Composition

When working with abstract bigraphs, composing bigraphs b_1 and b_2 requires the outerface of b_2 to be equal to the innerface of b_1 (cf Fig. 3). This means the same number of sites in b_1 as roots in b_2 and a set isomorphism (that we implemented through a list permutation) between the innernames of b_1 and outernames of b_2. For place graphs, $\circ$ joins the roots to sites, connecting any nodes as required, e.g. redefining *prnt*. The sites and roots merge and disappear. In the link graph, innernames and outernames of the same name connect. They merge and disappear into a new link. The *link* function is updated to reflect new links between ports/edges/names. Nodes and edges are assumed disjoint in the theory, and enforced in our implementation as node and edge types are unique to each bigraph.

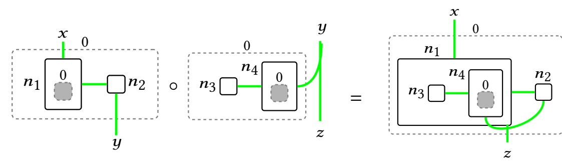

Figure 3: Bigraph Composition.

We can build a sequence operator, $\gg$, between *prnt* and *link* functions of b_1 and of b_2, that bypasses the merged interface. That is, when the function from b_2 gives an image in its outerface (and thus in the innerface of b_1), the new function bypasses it by returning the image of its b_1 twin. Using this we define composition:

$$s_1 = r_2 \wedge i_1 \cong o_2 \Rrightarrow b_1 \circ b_2 : \langle s_2, i_2 \rangle \rightarrow \langle r_1, o_1 \rangle :=$$
$$\langle n_1 \uplus n_2, e_1 \uplus e_2, ctrl_1 \uplus ctrl_2, prnt_2 \gg prnt_1, link_2 \gg link_1 \rangle$$

As expected, the resulting bigraph has the innerface of b_2 and the outerface of b_1, and the merged interface disappears. We use disjoint sum $\uplus$ when combining nodes and edges, using the proof that `finType` is closed for the disjoint sum. The *ctrl* function similarly applies under this disjoint sum (of nodes).

The final element of a bigraph is the proof that *prnt* is acyclic, and so we need to prove $prnt_2 \gg prnt_1$ is also acyclic. This is straightforward as both place graphs are already proved acyclic and we only ever change a parent relation upwards from the outerface.

In our implementation, as stated in Section 5.1, the requirements for equality of the innerface of b_1 and the outerface of b_2 are automatically discharged, which allows us to simply write $b_1 \circ b_2$.

5.2.1 Properties of Composition. We prove that this operator behaves as it should in an spm-category as described in Section 2.3.

Eq. (C1) is automatic from our definition. We prove that composition is neutral to the left and right (Eq. (C3)) with the family of identity bigraph with which it is allowed to commute. The hypothesis we have to check before we can proceed with the composition between a bigraph b and the id is that $s_{id} = r_b$ and $i_{id} \cong o_b$. That means in that case that the correct family of id is id_{r_b, o_b}. To prove Eqs. (equ_s), (equ_i), (equ_r) and (equ_o) for places and names, we expose the reflexivity of equality on natural numbers and the reflexivity of $\in$ for the sum of a list and an empty one. Then the bijections for nodes bij_n and edges bij_e are pretty much identities, i.e. bijections between A and $A \uplus \emptyset$ (and symmetrically). For ports, we have a small lemma that checks that $\forall n, Arity(ctrl(id \circ b)n) = Arity(ctrl\, b(bij_n n))$, we build bij_p from this equation. Proving that the function equations equ_c, equ_p and equ_l hold with these bijections is simply a matter of simplification and case analysis to go back to the origin bigraph of each element.

We also prove that composition is associative (Eq. (C2)). We proceed similarly to what we just described, i.e. use reflexivity and bijections that reorder elements to their assigned category from $A \uplus (B \uplus C)$ to $(A \uplus B) \uplus C$.

This proves that $\circ$ respects category equations used by Milner.

We also prove that equality is a congruence with respect to this composition. This allows us to declare composition as a morphism and directly use rewrite mechanisms of Coq.

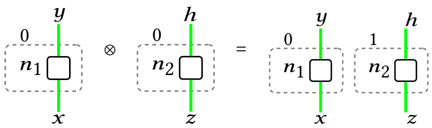

Figure 4: Tensor product

5.3 Tensor Product

Tensor product ($\otimes$) is the juxtaposition of two bigraphs with disjoint interfaces (see Fig. 4).

Like the $\gg$ operator we defined for composition, we define an operator $\bar{\otimes}$ between *prnt* and *link*. $\bar{\otimes}$ acts like $\uplus$, while also increasing the places of the second bigraph by the number of places of the first bigraph, i.e. to account for the increased ordinal. We define tensor product as:

$$i_1 \cap i_2 = \emptyset \wedge o_1 \cap o_2 = \emptyset \Rightarrow$$
$$b_1 \ \otimes \ b_2 : \langle s_1 + s_2, i_1 \uplus i_2 \rangle \longrightarrow \langle r_1 \uplus r_2, o_1 + o_2 \rangle :=$$
$$\langle n_1 \uplus n_2, e_1 \uplus e_2, ctrl_1 \uplus ctrl_2, prnt_1 \ \bar{\otimes} \ prnt_2, link_1 \ \bar{\otimes} \ link_2 \rangle$$

Like with composition, disjointness requirements have been nested into classes and useful lemmas added to a pool of class instances (see Section 5.1). Tensor product adds the sites/roots numbers (ordinals) and joins name sets. The union of name sets is still a `NoDupList` as we have $i_1 \cap i_2 = o_1 \cap o_2 = \emptyset$.

The extended functions for *ctrl*, *prnt*, and *link* naturally apply the original function based on where the node originated from. Some additional manipulation is required to handle cases involving sites/roots and names, e.g. to handle the increased ordinals In practice we use bijections from our library that manipulate finite sets. The same argument also applies to ports.

Finally, we need to ensure the new *prnt* function remains acyclic. The proof simply uses both components' acyclicity.

5.3.1 Properties of Tensor Product. We prove that this operator behaves as it should in an spm-category as described in Section 2.3.

We prove that our tensor product has the empty identity bigraph id_ϵ for a neutral element from the left and the right (cf. Eq. (M2)). Before proceeding with the tensor product, we need to check that any list is disjoint to the empty list, which is trivial. Then, proving Eqs. (equ_s) and (equ_r) boils down to proving that 0 is the neutral element for addition and proving Eqs. (equ_i) and (equ_o) requires to check that $\forall n \in \mathcal{X}, n \in io \Leftrightarrow name \in io \cup \emptyset$. Since the resulting tensor product support is the same as for composition, our methods are very similar. For bij_n and bij_e, we use the same bijection between A and $A + \emptyset$ (and symmetric). For bij_p, it is done as for composition. The equ_c, equ_p, equ_l are also a simple matter of case analysis.

Associativity (cf. Eq. (M1)) is proven in a similar way, with the same bijections from $A \uplus (B \uplus C)$ to $(A \uplus B) \uplus C$ we used in composition. The only care taken here is with the reordering of places.

We prove that tensor product commutes with composition (cf. Eq. (M3)) by reordering the elements. This proof is the longest, with a lot of variables and cases but still remains a mere case analysis.

We also prove that tensor product is a congruence to our equivalence. Intermediate lemmas and proving congruence of $\in$ are useful here. This allows us to declare the tensor product as a morphism.

Some remarks about the proofs: despite their straightforwardness proofs like the ones of transitivity, distributivity or congruence become pretty long because of the multitude of cases to expose.

These last two subsections proofs show that our implementation of $\otimes$ and $\circ$ respect spm-category's rules. In order to have a fully implemented spm-category, we need to have symmetry arrows.

5.4 Symmetries and Axioms

To prove that abstract bigraphs are indeed an *spm-category*, we need to prove the remaining axioms of Section 2.3 and the bigraphical structure axioms defined by Milner [20](page 31). We implement them in the `Symmetries` file, but we do not expand on these proofs.

To prove Eq. (S1), we use the *id* bijection and usual neutral element's rules. For Eq. (S2), both bigraphs have no nodes, so we create bijections from void to void $\uplus$ void. Interfaces simply need to commute to get equivalence. For Eq. (S4) as well both bigraphs have no nodes and bijections go from void to sums of void. Interfaces equality stems from commutativity and associativity of + and $\uplus$ operators. Eq. (S3) requires similar tools of commutativity.

With these final theorems, we have proven that we have implemented an spm-category.

6 Derived Operators

When specifying bigraphs it is useful to work at a higher level than composition/tensor by defining a set of derived operators: parallel product, nesting and merge product (respectively implemented in ParallelProduct, Nesting and MergeProduct). These derived operators use some of the classical node-free bigraphs introduced in Section 2.2 to rename or rearrange the interface.

6.1 Parallel Product

Parallel product (denoted $\|$, and shown in Fig. 5) is similar to tensor product, in that it places bigraphs side-by-side, but additionally it joins any like-names shared between the bigraphs (recall that tensor product requires disjoint names). For example, two bigraphs both with an outername y will bear a (new) outername y. To be able to parallel product two bigraphs there is a requirement on innernames: if bigraphs b_1 and b_2 share an innername i, then both in b_1 and b_2, i must be linked to a common outername (else it is impossible to know which outername to link to). We call this requirement *iToO* and implement it in the `UnionPossible` file. Formally, $iToO(b_1, b_2) := \forall i \in i_1 \cap i_2, link_1(i) = link_2(i) \in o_1 \cap o_2$.

As with the other operators, we write this requirement in a **`Class`** called `UnionPossible`. To allow Coq to automatically infer some proofs, we export some clever class instances of `UnionPossible`. For instance, Eq. (1) is used several times in the rest of this section:

$$\forall b_1, \forall b_2, \quad i_1 \cap i_2 = \emptyset \implies iToO(b_1, b_2) \tag{1}$$

Although parallel product is a derived operator, we specify it with a new definition (rather than directly in terms of tensor). Our implementation introduces $\bar{\|}$. $\bar{\|}$ that acts like $\bar{\otimes}$ but allows name-sharing:

$$iToO(b_1, b_2) \Rightarrow b_1 \ \ \| \ \ b_2 : \langle s_1 + s_2, i_1 \cup i_2 \rangle \longrightarrow \langle r_1 + r_2, o_1 \cup o_2 \rangle :=$$
$$\langle n_1 \uplus n_2, e_1 \uplus e_2, ctrl_1 \uplus ctrl_2, prnt_1 \ \bar{\otimes} \ prnt_2, link_1 \ \bar{\|} \ link_2 \rangle$$

We use the same *prnt* function acyclicity proof as in Section 5.3.

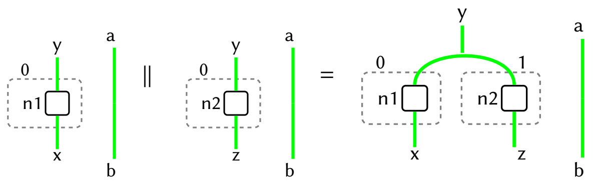

Figure 5: Parallel product

Properties of parallel product. To prove correctness of this operation, we first prove that $\|$ is a derivative of $\otimes$: when names are disjoint, $b_1 \parallel b_2 = b_1 \otimes b_2$. To do so, we require Eq. (1) to have $iToO(b_1, b_2)$. Then we use the definition of $\overline{\|}$ and break down whether the innernames were from i_1 or i_2.

Then we prove that id_ϵ is a neutral element for parallel product. To generate the *iToO* proof, we use the disjointness of i and $\emptyset$ and Eq. (1). The proof is then the one of tensor product Section 5.3.

We also prove that parallel product is associative. This first requires proving that : $iToO(b_1, b_2) \wedge iToO(b_2, b_3) \wedge iToO(b_1, b_3) \implies iToO((b_1 \parallel b_2), b_3)$. This amounts to proving that $\forall i \in (i_1 \cup i_2) \cap i_3, link_{b_1 \| b_2}(i) = link_3(i)$. This is directly deduced from $\forall i \in i_1 \cap i_3, link_1(i) = link_3(i)$ and $\forall i \in i_2 \cap i_3, link_2(i) = link_3(i)$.

6.2 Merge Product

Merge product (denoted |) produces a bigraph with a single root, i.e. it creates siblings. It is itself derived from parallel product (it is a parallel product composed with a *merge* bigraph, cf Section 2.2). As it is based on parallel product, we require *iToO* to hold. Merge product is defined as:

$$iToO(b_1, b_2) \Rrightarrow b_1 \mid b_2 : \langle s_1 + s_2, i_1 \cup i_2 \rangle \to \langle 1, o_1 \cup o_2 \rangle := (merge_{r_1+r_2} \otimes id_{0,o_1 \cup o_2}) \circ (b_1 \parallel b_2)$$

Implementation-wise, we can't explicitely state that the type of $b_1 \mid b_2$ is $\langle s_1 + s_2, i_1 \cup i_2 \rangle \to \langle 1, o_1 \cup o_2 \rangle$ because Coq computes the type of $(merge_{r_1+r_2} \otimes id_{0,o_1 \cup o_2}) \circ (b_1 \parallel b_2)$ as $\langle s_1 + s_2, i_1 \cup i_2 \rangle \to \langle 1 + 0, \emptyset \cup (o_1 \cup o_2) \rangle$. This may be fixed through an explicit casting, but for now this does not pose any issues.

Properties of merge product. To prove correctness, we first prove that $merge_0$ is a unit for merge product. To do so, we reuse proof of *iToO* of Section 6.1, the proof from then is straightforward.

Then we prove | is associative, this requires the same proof as for $\|$ that $iToO(b_1, b_2) \wedge iToO(b_2, b_3) \wedge iToO(b_1, b_3) \implies iToO((b_1 \mid b_2), b_3)$, the main proof also follows the same structure.

6.3 Nesting

Nesting (denoted $\cdot$) is like a composition allowing nested bigraphs to pass their outernames to the top-level, i.e. $(\langle 1, \{\} \rangle \to \langle 1, \{y\} \rangle) \cdot (\langle 0, \{\} \rangle \to \langle 1, \{x\} \rangle) = \langle 0, \{\} \rangle \to \langle 1, \{x, y\} \rangle$. In a composition this would not be allowed, as x is not in the innerface of the context. Nesting is defined as:

$$i_1 = \emptyset \wedge s_1 = r_2 \Rrightarrow b_1 \cdot b_2 : \langle s_2, i_2 \rangle \to \langle r_1, o_1 \cup o_2 \rangle := (id_{0,o_2} \parallel b_1) \circ b_2$$

Implementation-wise, we have the same remark as for |, which is that the type of $(id_{0,o_2} \parallel b_1) \circ b_2$ is $\langle s_2, i_2 \rangle \to \langle 0 + r_1, o_2 \cup o_1 \rangle$.

Properties of nesting. To prove correctness of the nesting operator, we first prove that nesting has $id_{s,\emptyset}$ as a right neutral element and $id_{r,\emptyset}$ as a left. The proof follows the same flow as composition.

Then we prove $\cdot$ is associative, again similarly to composition.

These derived operators should allow to uniquely express a bigraph as an algebraic formula of elementary bigraphs.

7 Related Work

Coq has been used to implement many theories similar to the bigraph theory, although many are abstract and do not provide direct access to the results we required to encode bigraphs. For instance, Wiegley has implemented large parts of category theory in Coq [27], which we hope could be used to instanciate a category of our bigraphs. However, such a deep embedding would likely prove itself too stiff when dealing with changes, especially in early exploratory stages of our formalization. The categories of bigraphs are also not always well defined due to the need for disjoint supports, i.e. they are special categories (called paracategories or precategories) where composition may not always be defined. Another example of abstract theories is Geuvers *et al.* implementation of monoids [14] or Gaspar *et al.* component-based approach [13].

Doczkal and Pous implement graph theory in Coq [11]. They represent graphs as a record of vertices, represented as a finite type, and edges, represented as a boolean relation between two vertices.We cannot directly use this representation as our links are hyperedges so cannot be represented with binary relations, and our parent relation needs to be acyclic. Additional aspects of bigraphs such as open edges, innernames etc. are also missing. Graph theory has been implemented in other theorem provers, for example in Lean 4 [17]. Here graphs are represented as an array of adjacency lists. Graphs have also been formalized in Isabelle [22] as a record of vertices, edges and two functions that map the edges to their heads and tails (this is a common approach in graph transformation literature), and a different Isabelle implementation [24] uses sets of nodes and sets of sets of nodes to represent the edges.

Although not computer certified, bigraphs have already been implemented in BigraphER [26]. BigraphER is a powerful toolkit that allows to compute or simulate the transition system of a bigraphical reactive system. It enables the use of (external) model checkers for verification. BigraphER supports many theory extension including bigraphs with sharing, stochastic and probabilistic reaction rules, rule priorities and predicate checking, and it is an open question how we extend our formalism to also model these features. In future, we will use the code extraction mechanism of Coq to provide a correct-by-construction implementation of bigraphs that can be used to *execute* bigraphical reactive systems, e.g. creating a verified version of (parts of) BigraphER.

8 Conclusion and Future Work

We have taken the first step into formally encoding bigraphs theory in the Coq theorem prover, approximately amounting to 1,500 lines of specification. We have implemented the main bigraph type that, like in the theory, hides the specific concrete details (e.g. nodes and edges) behind a well defined interface (based on places and names).

We have implemented and shown correctness of the main operators on bigraphs: composition, tensor product and main derived operators. We have shown an approach to recovering support and lean-support equivalence that determines when two bigraphs are structurally equivalent. We used these equivalences to prove relevant *spm-category* axioms. We are now confident our implementation is sound to reason upon.

With more complex operators come more complex interface requirements. Therefore, our automated proof search based upon Coq classes and instances could be extended to cope with more general situations.

We think it is also possible to encode sorting [3]. It extends the interface to add domain specific constraints determining when composition is legal, e.g. buildings are not nested within rooms.

Now that we can prove properties of bigraph structures, the next step is to encode bigraph dynamics, which are essential to model interactive systems. Dynamics are specified using a set of rewriting rules (reaction rules) that replace sub-bigraphs with sub-bigraphs, and example reaction rules are shown in [2]. Implementing rewriting requires identifying patterns within a larger bigraph. This may be achieved through decompositions [15], although for efficiency we may require a custom pattern-matching algorithm which may be inspired by [4]. Pattern-matching within graphs and bigraphs is a complex combinatorial task where enumeration of candidate sub-graphs is central. For example, BigraphER uses a sophisticated custom subgraph isomorphism algorithm [1]. As a first step we will explore an approach path using lazy enumeration strategies and lazy lists (lists that do not get fully computed until we need to access a specific element).

The wider outcome is the access to a well defined bigraph theory that can be used to underpin analysis and verification of critical systems. We are particularly interested in the verification of user interfaces for aviation scenarios and aim to use the new formalism within this domain. This will require verified code extraction from the Coq specification.

The formal approach is also relevant for those wanting to reason soundly on π-calculcus or other process algebras such as BigrTiMo (a process calculus based on bigraphs) [28].

Acknowledgments

This work is partially supported by the British Council through the Alliance Hubert Curien Programme and an Amazon Research Award on Automated Reasoning. We thank the reviewers for their valuable comments.

References

[1] Blair Archibald, Kyle Burns, Ciaran McCreesh, and Michele Sevegnani. 2021. Practical Bigraphs via Subgraph Isomorphism. In *27th International Conference on Principles and Practice of Constraint Programming, CP (Virtual Conference), October 25-29, 2021 (LIPIcs, Vol. 210)*, Laurent D. Michel (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:17. https://doi.org/10.4230/LIPIcs.CP.2021.15

[2] Blair Archibald, Muffy Calder, and Michele Sevegnani. 2024. Practical Modelling with Bigraphs. arXiv:2405.20745 [cs.LO]

[3] Blair Archibald and Michele Sevegnani. 2024. A Bigraphs Paper of Sorts. In *Graph Transformation - 17th International Conference, ICGT 2024, Held as Part of STAF 2024, Enschede, The Netherlands, July 10-11, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14774)*, Russ Harmer and Jens Kosiol (Eds.). Springer, 21–38. https://doi.org/10.1007/978-3-031-64285-2_2

[4] Samuel Arsac. 2022. *Coq formalization of graph transformation.* Master's thesis. Équipe PLUME, LIP, ENS Lyon. supervised by R. Harmer and D. Pous.

[5] Yves Bertot. 2008. A Short Presentation of Coq. In *Theorem Proving in Higher Order Logics.* Springer Berlin Heidelberg, Berlin, Heidelberg, 12–16. https://doi.org/10.1007/978-3-540-71067-7_3

[6] Lars Birkedal, Søren Debois, Ebbe Elsborg, Thomas Hildebrandt, and Henning Niss. 2006. Bigraphical models of context-aware systems. In *Foundations of Software Science and Computation Structures: 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006. Proceedings 9.* Springer, 187–201.

[7] Timothy Bourke, Basile Pesin, and Marc Pouzet. 2023. Verified Compilation of Synchronous Dataflow with State Machines. *ACM Transactions on Embedded Computing Systems* 22, 5s (Sept. 2023), 137:1–137:26. https://doi.org/10.1145/3608102 ESWEEK special issue including presentations at the 23rd Int. Conf. on Embedded Software (EMSOFT 2023).

[8] Pierre Boutillier, Mutaamba Maasha, Xing Li, Héctor F Medina-Abarca, Jean Krivine, Jérôme Feret, Ioana Cristescu, Angus G Forbes, and Walter Fontana. 2018. The Kappa platform for rule-based modeling. *Bioinformatics* 34, 13 (2018), i583–i592.

[9] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, Jonathan Hayman, Jean Krivine, Chris Thompson-Walsh, and Glynn Winskel. 2012. Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models. In *FSTTCS 2012 - IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (LIPIcs, Vol. 18)*, Schloss Dagstuhl Leibniz-Zentrum fuer Informatik (Ed.). Hyderabad, India, 276–288. https://doi.org/10.4230/LIPIcs.FSTTCS.2012.276

[10] Christian Doczkal and Damien Pous. 2020. Completeness of an axiomatization of graph isomorphism via graph rewriting in Coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs.* 325–337. https://doi.org/10.1145/3372885.3373831

[11] Christian Doczkal and Damien Pous. 2020. Graph theory in Coq: Minors, treewidth, and isomorphisms. *Journal of Automated Reasoning* 64 (2020), 795–825. https://doi.org/10.1007/s10817-020-09543-2

[12] A. Mahboubi et E. Tassi. 2022. *The Mathematical Components team: Mathematical components.* Zenodo. https://doi.org/10.5281/zenodo.4457887

[13] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. 2014. Bringing Coq into the World of GCM Distributed Applications. *Int. J. Parallel Program.* 42, 4 (Aug. 2014), 643–662.

[14] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. 2002. A Constructive Algebraic Hierarchy in Coq. *Journal of Symbolic Computation* 34, 4 (2002), 271–286. https://doi.org/10.1006/jsco.2002.0552

[15] Arne John Glenstrup, Troels Christoffer Damgaard, Lars Birkedal, and Espen Højsgaard. 2010. *An implementation of bigraph matching.* working paper TR-2010-135. IT-Universitetet i København, Copenhagen, Denmark.

[16] Jason Gross, Adam Chlipala, and David I Spivak. 2014. Experience implementing a performant category-theory library in Coq. In *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5.* Springer, 275–291.

[17] Peter Kementzey. 2021. *A Graph Library for Lean 4.* Ph. D. Dissertation. Vrije Universiteit Amsterdam.

[18] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.*

[19] Cécile Marcon and Xavier Thirioux. 2024. *BiCoq: Modeling bigraphs with Coq.* https://doi.org/10.5281/zenodo.12522237

[20] Robin Milner. 2009. *The space and motion of communicating agents.* Cambridge University Press.

[21] Nicolas Nalpon, Cyril Allignol, and Célia Picard. 2022. Towards a User Interface Description Language Based on Bigraphs. In *International Colloquium on Theoretical Aspects of Computing.* Springer, 360–368.

[22] Lars Noschinski. 2014. A Graph Library for Isabelle. *Mathematics in Computer Science* 9 (03 2014), 23–39. https://doi.org/10.1007/s11786-014-0183-z

[23] Gian Perrone, Søren Debois, and Thomas Hildebrandt. 2012. A model checker for Bigraphs. *Proceedings of the ACM Symposium on Applied Computing* (03 2012). https://doi.org/10.1145/2245276.2231985

[24] Tom Ridge. 2005. Graphs and Trees in Isabelle/HOL.

[25] Hamza Sahli, Thomas Ledoux, and Éric Rutten. 2020. Modeling self-adaptive fog systems using bigraphs. In *Software Engineering and Formal Methods: SEFM 2019 Collocated Workshops: CoSim-CPS, ASYDE, CIFMA, and FOCLASA, Oslo, Norway, September 16–20, 2019, Revised Selected Papers 17.* Springer, 252–268.

[26] Michele Sevegnani and Muffy Calder. 2016. BigraphER: Rewriting and analysis engine for bigraphs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28.* Springer, 494–501. https://doi.org/10.1007/978-3-319-41540-6_27

[27] John Wiegley. 2022. Category Theory in Coq. GitHub. https://github.com/jwiegley/category-theory

[28] Wanling Xie, Huibiao Zhu, and Qiwen Xu. 2021. A process calculus BigrTiMo of mobile systems and its formal semantics. *Formal Aspects of Computing* 33 (2021), 207–249. https://doi.org/10.1007/s00165-021-00530-x