View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade do Minho: RepositoriUM

Bigraphical Modelling of Architectural Patterns

Alejandro Sanchez[1], Luís S. Barbosa[2], and Daniel Riesco[1]

[1] Departamento de Informática, Universidad Nacional de San Luis,
Ejército de los Andes 950, D5700HHW San Luis, Argentina
{asanchez,driesco}@unsl.edu.ar
[2] DI - HASLab, Universidade do Minho,
Campus de Gualtar, 4710-057 Braga, Portugal
lsb@di.uminho.pt

Abstract. Archery is a language for behavioural modelling of architectural patterns, supporting hierarchical composition and a type discipline. This paper extends Archery to cope with the patterns' structural dimension through a set of (re-)configuration combinators and constraints that all instances of a pattern must obey. Both types and instances of architectural patterns are semantically represented as bigraphical reactive systems and operations upon them as reaction rules. Such a bigraphical semantics provides a rigorous model for Archery patterns and reduces constraint verification in architectures to a type-checking problem.

1 Introduction

In a number of contexts the term architectural pattern is used as an architectural abstraction. The expression is taken in the usual sense in classical software architecture – a known solution to a recurring design problem. In [4] it is characterised as a description of element and configuration types, and a set of constraints on how to use them. Available catalogs such as [8] provide a vocabulary for their use at a high abstraction level. However, the lack of formality in their pattern documentation prevents its usage for developing precise architectural specifications on top of them, and in consequence, any tool-supported analysis and verification.

Such is the motivation behind Archery, a language to describe the behaviour of pattern elements, a subset of which was recently presented at [13]. Its semantics is given by translation to mCRL2 [10]. A pattern specification in Archery comprises a set of architectural elements (connectors and components) and their associated behaviours. An architecture describes a particular configuration that instances of a pattern's elements assume. This configuration has an emergent behaviour and constitutes an instance of the pattern. Then, both patterns and elements define the types of behaviour expected from instances. The language supports hierarchical composition of architectures.

This paper, extends Archery to the so-called *structural* dimension of architectural patterns. This comprises the usage of typed variables to contain and reference instances, a set of scripting operations to build architectural configurations, and a set of primitives to specify constrains over such configurations.

Constraints restrict the class of valid configurations that architectures, instances of a particular pattern, may adopt. Then, reconfigurations are only enabled if respecting the pattern constraints. For instance, a reconfiguration script that connects two clients in a Client-Server architecture violates the intended use of the pattern and should be prevented.

A second contribution of this paper is a semantics for the structural dimension of Archery on top of Bigraphical Reactive Systems (BRS) [11]. The theory of BRSs was developed to study systems in which locality and linking of computational agents varies independently, and to provide a general unifying theory in which existing calculi for concurrency and mobility can be represented. The two main constituents of a BRS are a bigraph and a set of parametric reaction rules. The former specifies the BRS structure as two orthogonal graphs upon the same set of nodes, one modelling locality, and another linking. Rules model its dynamics, *i.e.*, how the structure is reconfigured through reaction.

The theory of BRSs has a precise definition. A bigraph, expressed as a tuple of functions, is an arrow in a category. Its domain and codomain are objects. A more restrictive category can be defined for bigraphs by including in their definition a mechanism, called sorting, that constrains the configurations they can adopt. This setting allows the formal treatment of the encoded system. In particular, if conditions are met [11], it allows to automatically derive a labelled transition system (LTS) from a BRS, in which behavioural equivalence is a congruence.

The choice of BRS as a semantical framework for Archery arose naturally as the language was expected to allow for independently modifying both placing and linking of pattern instances. At a more fundamental level, the structural dimension of patterns and architectures become encoded as arrows in a suitable category[3]. Finally, the use of bigraphs reduces the problem of verifying whether an architectural constraint holds for a pattern to a certain kind of type-checking. Actually, once a structural constraint is encoded as a sorting, to check if it is verified by an architecture amounts to translating the latter to a bigraph and prove that such a bigraph belongs to the category defined by the sorting.

The bigraphical encoding presented here is also the basis, along the work in [5], to explore in [12] the automatic derivation of LTS whose states stand for the different configurations the corresponding architecture can adopt. This makes possible to resort to behavioural equivalence to compare the application of different patterns in reconfiguring systems.

The following sections illustrate how Archery can be endowed with a bigraphical semantics. For such purposes we limit ourselves to a subset of the scripting operations and an example constraint. The full version of the language can be found in [12]. The rest of the paper is organised as follows: section 2 introduces Archery. Section 3 briefly recalls the basic theory of BRS and section 4 develops a formal semantics for the structural dimension of the presented language. Finally, section 5 concludes and discusses future work.

[3] In fact, the name Archery comes from a comment in Steve Awodey's book [3] emphasising the importance of arrows in category theory: " *...the subject might better have been called abstract function theory, or perhaps even better: archery.*"

2 The Archery Language

We structure Archery as a core and two extensions, respectively named Archery-Core, Archery-Script, and Archery-Structural-Constraint. The first is a slightly modified version of the language presented in [13], the second adds the operations for building configurations, and the third incorporates the primitives for defining structural constraints. The structure follows the differences in how their semantics are defined. While both behavioural and structural semantics are defined for Archery-Core, only structural semantics are given to Archery-Script and Archery-Structural-Constraint. The three language subsets are endowed structural semantics by translations to bigraphs. However, the codomain of each translation differs, and the third subset requires a more involved approach.

2.1 Archery-Core

A specification in Archery-Core comprises one or more patterns and a main architecture. The first rule of the grammar, shown in Figure 1, indicates this by equating the *Spec* non-terminal to one or more *Pat* and a *Var* non-terminals. Note that several non-terminal are undefined; the grammar leaves out the definition of the ones that are not relevant to the structural dimension.

Spec	::=	*Pat*+ *Var*
Pat	::=	**pattern** TYPEID **(** *PatPars*? **)** **elements** Elem+ **end**
Elem	::=	**element** TYPEID **(** *ElemPars*? **)** *Behaviour ElemInterface*
ElemInterface	::=	**interface** *Port*+
Port	::=	(**in**|**out**) ID **;**
Var	::=	ID **:** TYPEID = *Inst* **;**
Inst	::=	(*ElemInst*|*PatInst*)
ElemInst	::=	TYPEID **(** *ElemInstPars*? **)**
PatInst	::=	**architecture** TYPEID **(** *PatInstPars*? **)** *ArchBody* **end**
ArchBody	::=	*Instances Attachments*? *ArchInterface*?
Instances	::=	**instances** *Var*+
Attachments	::=	**attachments** *Att*+
Att	::=	**from** *PortRef* **to** *PortRef* **;**
ArchInterface	::=	**interface** *Ren*+
Ren	::=	*PortRef* **as** ID **;**
PortRef	::=	ID**.**ID

Fig. 1: Grammar Fragment for Archery-Core

A pattern is specified according to the rule expanding the *Pat* non-terminal. Its definition contains, a TYPEID token that represents the identifier for it, an optional list of formal parameters, and one or more architectural elements *Elem*, *i.e.*, specified according to the *Elem* non-terminal. For instance, the specification in Listing 1 includes two patterns: `ClientServer` and `PipeFilter`.

Each architectural element in a pattern is specified as described by *Elem*. Its definition comprises: a TYPEID token as its identifier, an optional list of formal parameters, a description *Behaviour* of its behaviour, and a description *ElemInterface* of its interface. The behaviour is specified with a slightly modified subset of mCRL2 limiting its expressivity to sequential processes. Its description must contain one ore more process expressions, as the one shown in line 5, and a list of action definitions, like in line 4. The first process is the initial behaviour of the instance and may call other processes defined within the element. The interface contains one or more ports *Port*. A port is defined by a direction indicator, either `in` or `out`, and an ID token that must match an action name in the list of action definitions. For instance, the interface of `Server` defines two ports in line 6. We adopt the underlying metaphor of water flow in [2] for ports: an `in` port receives input from any port connected to it, and an `out` port sends output to all ports connected to it. Ports are synchronous: actually a suitable process algebra expression can be used to emulate any other port behaviour.

Listing 1: Example Patterns and Architectures

```
1  pattern ClientServer()
2  elements
3    element Server()
4      act rreq, sres, cres;
5      proc Server() = rreq.cres.sres.Server();
6      interface in rreq; out sres;
7    element Client()
8      act prcs, sreq, rres;
9      proc Client() = prcs.sreq.rres.Client();
10     interface in rres; out sreq;
11 end
12 pattern PipeFilter()
13 elements
14 element Pipe()
15   act accept, forward;
16   proc Pipe() = accept.forward.Pipe();
17   interface in accept; out forward;
18 element Filter()
19   act rec, trans, send;
20   proc Filter() = rec.trans.send.Filter();
21   interface in rec; out send;
22 end
23 cs : ClientServer = architecture ClientServer()
24 instances
25     s1 : Server = architecture PipeFilter()
26     instances
27       f1:Filter=Filter(); f2:Filter=Filter();
28       p1:Pipe=Pipe();
29     attachments
```

```
30       from f1.send to p1.accept;
31       from p1.forward to f2.rec;
32     interface
33       f1.rec as rreq;
34       f2.send as sres;
35     end
36   c1 : Client = Client();  c2 : Client = Client();
37 attachments
38   from c1.sreq to s1.rreq;  from c2.sreq to s1.rreq;
39   from s1.sres to c1.rres;  from s1.sres to c2.rres;
40 end
```

A variable and its value is defined according to *Var*. The variable has an ID token as its identifier, followed by a TYPEID token that must match an element or pattern name. The value can be either a pattern *PatInst* or an element *ElemInst* instance. Note that the variable that follows the pattern definitions, as indicated in the first grammar rule, and as shown in line 23 of the example, must contain an architecture (the main one).

An architecture defines a set of variables and describes the configuration adopted by the instances in them. It contains: a TYPEID token that must match a pattern name, an optional list of actual arguments, a set of variables *Var*, an optional set of attachments *Att*, and an optional interface *ArchInterface*. Each variable in the set must have as type an element defined in the pattern the architecture is instance of. If the variable has as assigned value an element instance *ElemInst*, it is defined by a TYPEID and a list of actual parameters. If it has a pattern instance, like between lines 25 and 35 of the example, a nested architecture is defined. Each attachment *Att* includes a port reference *PortRef* to an out port, and another to an in port. A port reference is an ordered pair of ID tokens, with the first matching a variable identifier, and the second a port of the variable's instance. Then, an attachment indicates that the out port communicates with the in port, such as in the case of `f1.send` with `p1.accept` in line 30. The architecture interface is a set of one or more port renames *Ren*. Each port rename contains a port reference and an ID token with the external name for the port. Ports not included in the set are not visible from the outside. Including the same port in an attachment and in the interface is incorrect. An example interface with two renames is shown in lines 33 and 34.

2.2 Archery-Script

Archery-Script is used to specify scripts for creating architectures or for reconfiguring existing ones. It assumes the existence of a process that triggers a scripts under some conditions. Its operations (informally described in Table 1), are defined independently of any pattern. The design principles of patterns are enforced through constraints, as it is shown in Section 2.3. This independence, and the fact that a variable may contain an instance whose type may not necessarily match the variable's type, allows the reuse of a script in an open family of patterns (related by some refinement relation). We illustrate the operations through the example in Listing 2.

Table 1: Set of Operations in Archery–Script

Name	**Format**	**Description**
Import	`import(s)`	Receives as a parameter a reference `s` to an Archery specification and imports it to the environment of the executing script (*e.g.*, line 2 in Listing 2)
Create Variable	`v:type`	Creates a variable with name `v` and type `type` (line 3)
Create Instance	`v=type()`	Creates a new instance of type `type` and assigns it to a variable `v` (line 4)
Add Instance	`addInst(a,v)`	Adds a variable `v` and the instance in it, to the architecture in variable `a` (line 5)
Attach	`attach(f.o, t.i)`	Attaches the port `o` of the instance in variable `f` to the port `i` of the instance in variable `t` (line 8)
Deattach	`deattach(f.o, t.i)`	Removes the attachment between the port `o` of the instance in variable `f` and the port `i` of the instance in variable `t` (line 6)
Add Rename	`addRen(v.p,q)`	Renames port `p` in variable `v` with name `q` (line 15)
Remove Rename	`remRen(v.q)`	Removes rename `q` in the architecture in variable `v` (line 14)
Move	`move(s,t)`	Moves the instance in variable `s` to the variable `t` (line 16); the reference to the contents of `t` are lost, but its attachments and renames remain

The example is divided in three parts and assumes the existence of an initial configuration we call $cs_{initial}$. The configuration is similar to the one in Listing 1, but differs in that the nested architecture (between lines 25 and 35) is replaced by a `Server` instance (in a single line `s1:Server=Server();`).

The first part of the example reconfigures $cs_{initial}$ by adding and connecting a second server. It starts with an import operation that leaves the configuration in variable `cs`. The operations in lines 3 and 4, create a new variable `s2` and assign a fresh instance of `Server` to it. Upon that, `s2` is included in the architecture in `cs`. Then the operations in the next two lines remove the attachments among the instances in variables `cs.c2` and `cs.s1`. Subsequently, new attachments are created between the instance in variable `cs.c2` with the instance in variable `cs.s2`. We will refer to the obtained configuration as cs_{first}.

Listing 2: Example Script

```
1 script
2   import("initial");  // first part
3   s2 : Server;
4   s2 = Server();
```

```
5    addInst(cs, s2);
6    deattach(cs.c2.sreq, cs.s1.rreq);
7    deattach(cs.c2.rres, cs.s1.sres);
8    attach(cs.c2.sreq, cs.s2.rreq);
9    attach(cs.c2.rres, cs.s2.sres);
10   import("pf");  // second part
11   f3 : Filter = new Filter();
12   addInst(pf, f3);
13   attach(pf.p1.forward, pf.f3.rec);
14   remRen(pf.sres);
15   addRen(pf.f3.send, sres);
16   move(pf, cs.s2);
17   c3 : Client =  Client();  // third part
18   addInst(cs, c3);
19   deattach(cs.c2.sreq, cs.s2.rreq);
20   deattach(cs.c2.rres, cs.s2.sres);
21   attach(cs.c2.sreq, cs.c3.rres);
22   attach(cs.c2.rres, cs.c3.sreq);
23 end
```

The second part of the example starts in the line 10 and shows how the interface of an architecture is modified and then a server is replaced. It assumes the existence of a configuration pf, similar to the one described between the lines 25 and 35 in Listing 1, but contained in a variable `pf` of type `PipeFilter`. The script imports such configuration, creates a new instance of `Filter` in variable `f3` and includes it in `pf`. Line 14 removes rename `sres` from `pf`. This removal has the same effect as deleting line 34 from Listing 1. Then, a new rename is included in the interface, but now for port `send` in variable `pf.f3`. Subsequently, the instance in `pf` is moved to the variable `cs.s2`. The instance in the variable `cs.s2` is now the architecture of type `PipeFilter` but connected as it was the previous instance in such variable.

The third part begins upon line 17. It creates a new client and connects it in a wrong way. A new variable `c3` is created and a new instance of the type `Client` is assigned to it. Next, the fresh variable is included in the architecture in `cs`. Subsequently, the attachments between the instances in variables `cs.c2` and `cs.s2` are removed. Then, the script creates two attachments between instances in variables `cs.c3` and `cs.c2`. The resulting configuration violates the design principle behind a Client-Server architecture by connecting two clients. We refer to the configuration obtained upon the script execution as cs_{wrong}.

2.3 Archery-Structural-Constraint

To rule out configurations such as cs_{wrong}, entails the need for mechanisms to constrain what may count as valid instances of a pattern. Since the variable cs in the script of Listing 2 is of type $ClientServer$, we could add to the pattern specification a constraint φ to express that clients can only connect to servers and vice versa. We define φ for all attachments att in an architecture of type

$ClientServer$ as follows

$$client(from(att)) \Leftrightarrow server(to(att)) \wedge client(to(att)) \Leftrightarrow server(from(att))$$

with $from$ (respectively, to) a function that returns the variable with the out (respectively, in) port in att, and with $client$ (respectively, $server$) a predicate yielding true when its argument is of type $Client$ (respectively, $Server$).

By constraining patterns in this way, we can prevent an operation in a script that generates an invalid configuration. Clearly, cs_{wrong} does not satisfy it. In contrast, the configuration cs_{first} does. Given a configuration c and a constraint φ, the satisfaction problem can be formulated as $c \models \varphi$, which can be rendered as a type checking assertion in the bigraphical semantics for Archery. Such is the the topic of the following sections.

3 Bigraphical Reactive Systems

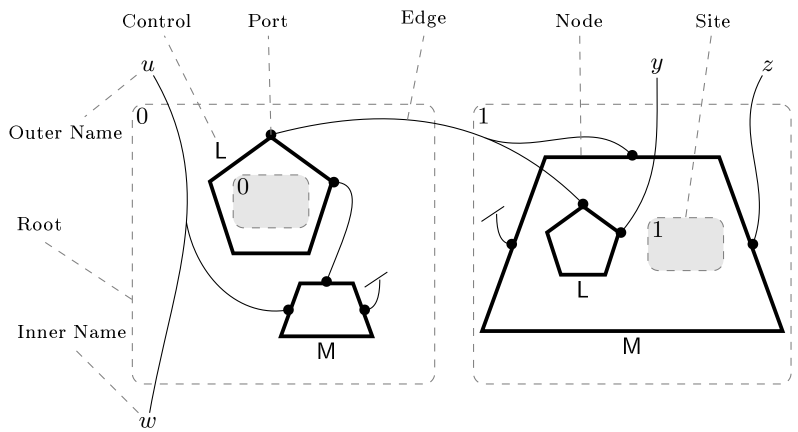
A Bigraphical Reactive System (BRS) is an inhabitant of a category. The operations and the elementary bigraphs in such category enable an algebraic treatment of BRSs. In the next sections we briefly describe the notions of bigraphs, their algebra, and the parametric reaction rules that make them dynamic. We refer the reader to [11] for more detail on these notions and their precise definitions.

3.1 Bigraphs

A bigraph contains a set of nodes related through a parent-child relationship and through edges. The former gives rise to a forest structure called *place graph*, in which the roots of the trees are the nodes without parent. The latter defines a hypergraph called *link graph*: a node is related to others by an edge, if each one has a *port* linked to an end of such edge. A bigraph is said to be *concrete* if its nodes and edges have identity, and *abstract* if they not. Figure 2 shows the structure of bigraphs following the anatomy style used in [11]. The abstract bigraph in it has a forest with two trees and a hypergraph with two edges.

The encoding of a system is enabled by the *basic signature* of a bigraph. Every node has an associated *control* from a set $\mathcal{K}$ that distinguishes its kind of contribution to the encoding. The control also establishes the number of ports the node has with an arity function $ar : \mathcal{K} \to \mathbb{N}$. The tuple $(\mathcal{K}, ar)$ is the basic signature of a bigraph and in the case of our example $\mathcal{K} = \{\mathsf{L} : 2, \mathsf{M} : 3\}$.

New bigraphs can be built from existing ones by plugging one into another. The interface of a bigraph defines the form of bigraphs it can contain – *inner face*, and the form that containers must accept – *outer face*. Suppose we divide a bigraph into two parts. A division in a tree leaves a *site* in one part, and a new *root* on the other. A division in an edge generates two open links: one called *inner name* and another called *outer name* The roots and outer names are the outer face, and the sites and inner names the inner face of a bigraph. Figure 2 shows the graphic conventions to depict them.



$${}^{u}/_{\{w\}}\ /x\ (/y\ (\mathsf{L}_{\{xy\}}.(\mathsf{id}_0)\ |\ {}^{y}/_{\{x\}}\ /z\ \mathsf{M}_{\{wxz\}})\ \parallel\ \mathsf{id}_w\ \parallel\ /w\ \mathsf{M}_{\{wxz\}}.(\mathsf{L}_{\{xy\}}\ |\ \mathsf{id}_1))$$

Fig. 2: Anatomy of Bigraphs

The category in which a bigraph lives depends on whether it is abstract or not and the signature $\mathcal{K}$ over which it is defined. An abstract bigraph becomes an arrow $F : I \rightarrow J$ in a category $\textsc{Bg}(\mathcal{K})$. Its domain I and codomain J are objects in such category. The domain is a tuple $I = \langle n, X \rangle$, in which n is a set of ordinals $\{0, 1, ..., n-1\}$ that index its sites, and X is its set of inner names. Similarly, the codomain is a tuple $J = \langle m, Y \rangle$ with m indexing its roots, and Y its set of outer names. If the bigraph is concrete, the space is a precategory $`\textsc{Bg}(\mathcal{K})$ instead. The reason for using a precategory is that composition is not always defined when nodes and edges have identity.

Undesired arrangements of controls can be ruled out by defining a *sorting* $\Sigma = (\Theta, \mathcal{K}, \Phi)$. The controls in $\mathcal{K}$ are classified in a set of sorts $\Theta = \{\theta_0, ..., \theta_n\}$, and valid arrangements of sorts are restricted with a formulation rule Φ. The sorts can be assigned to the controls – *place sorting*, or to the links according to the ports in controls – *link sorting*. Abstract (respectively, concrete) bigraphs over a sorting Σ inhabit a category $\textsc{Bg}(\Sigma)$ (respectively, precategory $`\textsc{Bg}(\Sigma)$).

3.2 Algebra

All bigraphs can be built from elementary ones by applying three basic operations: composition, product and identities. The *composition* $G \circ F : I \rightarrow K$, also denoted $G\ F$, of two bigraphs $F : I \rightarrow J$ and $G : J \rightarrow K$, represents a new bigraph obtained by plugging F into G. This operation is only defined when the inner face of G matches the outer face of F. The set $|F|$ of node and edge identifiers of F needs to be disjoint with $|G|$ if bigraphs are concrete. When $G \circ F$ is defined, we say that G is a context for F. The *product* of two bigraphs $F_i : \langle m_i, X_i \rangle \rightarrow \langle n_i, Y_i \rangle$ $(i = 0, 1)$, is a new bigraph $F_0 \otimes F_1 : \langle m_0 + m_1, X_0 \uplus X_1 \rangle \rightarrow \langle n_0 + n_1, Y_0 \uplus Y_1 \rangle$, (with $\uplus$ the union of disjoint sets) that represents placing F_0 besides F_1. $|F_0| \cap |F_1| = \emptyset$ also needs to hold for concrete bigraphs. The *identity* bigraph (arrow) of an interface (object) $I = \langle m, X\rangle$ is a tuple $\langle id_m, id_X\rangle$. In practice, a set of derived operations defined on top of the basic ones and elementary bigraphs is actually used.

The elementary bigraphs that do not have nodes are divided in the ones that only have roots and sites – *placings* (ϕ), and the ones that only have (outer and inner) names – *linkings* (λ). Placings can be generated from three elementary forms: a root with no sites $1 :\ 0 \rightarrow 1$; a symmetry $\gamma_{1,1} :\ 2 \rightarrow 2$ that exchanges the indexes of roots with the ones of sites; and a join $join :\ 2\ \rightarrow 1$ of two sites into one root. A merge bigraph can be derived as $merge_{n+1} = join \circ (id_1 \otimes merge_n)$. Similarly, linkings can be generated from two elementary forms: the substitution y/X of a set of names X with one name y; and the closure $/x$ of a link x. The only elementary bigraph that introduces nodes is $\mathsf{K}_{\vec{x}}$: $1 \rightarrow \langle 1, \{\vec{x}\}\rangle$, defined for each control $\mathsf{K} : n$ (with n ports), gives rise to a bigraph with a single node whose n ports are bijectively linked to n names in $\vec{x}$.

Some abbreviations for operations we may use are as follows: we may write $F \circ G$ instead of $(F \otimes id_I) \circ G$ when there is no ambiguity; given a linking $\lambda : Y \rightarrow Z$ and a bigraph $G : I \rightarrow \langle m, X\rangle$ with $Y = X \uplus X'$, we may write $\lambda \circ G$ instead of $(id_m \otimes \lambda) \circ (G \otimes X')$ when m and X are clear from the context.

The derived operations are: parallel product, nesting and merge product. The *parallel product* of two bigraphs $F_i : \langle m_i, X_i\rangle \rightarrow \langle n_i, Y_i\rangle$ $(i = 0, 1)$ is defined as $F_0 \parallel F_1 : \langle m_0 + m_1, X_0 \cup X_1\rangle \rightarrow \langle n_0 + n_1, Y_0 \cup Y_1\rangle$, a tensor product of the two bigraphs, with the exception that the link map allows name sharing. The result of the *nesting* of two bigraphs $F : I \rightarrow \langle m, X\rangle$ and $G : m \rightarrow \langle n, Y\rangle$ that may share names is a bigraph $G.F : I \rightarrow \langle n, X \cup Y\rangle$ defined by the expression $(id_X \parallel G) \circ F$. The *merge product* of two bigraphs G_i $(i = 0, 1)$ is $merge\ \circ\ (G_0 \parallel G_1)$, i.e., the merge of the parallel product of them. Abbreviations that we may use are as follows: $y/X \circ G$ instead of $(y/X \parallel id_I) \circ G$ with $I = \langle n, Z\rangle$, when G has outer face $\langle n, X \uplus Z\rangle$; A for the bigraph $\mathsf{A}.1$ when the control A has no children.

The algebraic expression in Figure 2 represents the bigraph shown above it, and is defined in terms of these elementary bigraphs and operations.

3.3 Reaction Rules

A parametric reaction rule is a tuple $\langle R : m \rightarrow J, R' : m' \rightarrow J', \eta\rangle$, with R and R' bigraphs respectively called *redex* and *reactum*, and η an instantiation map. Map η assigns to each ordinal in $m' = \{0, 1, .., i, .., m' - 1\}$ an ordinal $m = \{0, 1, .., j, .., m - 1\}$. When a bigraph F matches the redex, it is replaced with the reactum. The sites in F are placed in the sites of the reactum according to η. If we name the bigraphs contained by F according to the sites m in the redex in which they are placed, we obtain a sequence $d_0, d_1, .., d_j, .., d_m$. Then, the expression $\eta(i) = j$ tells that d_j will be placed in the i^{th} site of the reactum.

Bigraphs that have an associated set of reaction rules are defined over a *dynamic signature*. It differs from the basic in that each control is assigned one of the three values as follows: *atomic* – for controls of nodes without children (barren), *active* – for non-atomic controls that allow reactions to occur among the nodes inside, *passive* – for non-atomic and non-active controls. A reaction only takes place if the bigraph matching the redex is in an active context, *i.e.*, in a root, or in an active node with all ancestors active as well.

The abstract (respectively concrete) BRS with sorting Σ and parametric reaction rules $\mathcal{R}$ (${}^{\backprime}\mathcal{R}$) live in a category $\textsc{Bg}(\Sigma, \mathcal{R})$ (${}^{\backprime}\textsc{Bg}(\Sigma, {}^{\backprime}\mathcal{R})$).

4 Bigraphical Modelling of Archery Specifications

In this section we provide a bigraphical semantics for Archery. We respectively translate Archery-Core and Archery-Script specifications into bigraphs in categories $\textsc{Bg}(\Sigma_{Arch-Core}, \mathcal{R}_{Arch-Core})$ and $\textsc{Bg}(\Sigma_{Arch-Script}, \mathcal{R}_{Arch-Script})$. Since each Archery-Structural-Constraint constraint generates a different category, we limit to define $\textsc{Bg}(\Sigma_{\varphi}, \mathcal{R}_{Arch-Core})$ for the example constraint φ described in Section 2.3 and leave a generic method to [12].

4.1 Archery-Core

Function $\mathcal{T}$ (1) translates an Archery-Core specification into a bigraph in category $\textsc{Bg}(\Sigma_{Arch-Core}, \mathcal{R}_{Arch-Core})$. It takes a *Spec* and returns the parallel product of bigraphs that result of translating each *Pat* in *Pat*+, and a variable *Var* containing the main architecture. Table 2 lists the controls in $\Sigma_{Arch-Core}$ and the sort assignment to their ports, and Table 3 the rules in $\mathcal{R}_{Arch-Core}$. We describe the signature and rules as we describe the encoding of an example pattern and architecture, and leave the sorting for the end of the section.

$$\mathcal{T}(Spec) = \prod_{Pat+} \mathcal{T}(Pat) \;\|\; \mathcal{T}(Var) \tag{1}$$

$$\mathcal{T}(Pat) = \mathsf{Pat}_{TYPEID}.(\Big|_{Elem+} \mathcal{T}(Elem)) \tag{2}$$

$$\mathcal{T}(Elem) = \mathsf{Elem}_{TYPEID}.(\Big|_{Port+} \mathcal{T}(Port)) \tag{3}$$

$$\mathcal{T}(\texttt{in}\ ID) = \mathsf{NewIn}_{ID},\ \mathcal{T}(\texttt{out}\ ID) = \mathsf{NewOut}_{ID} \tag{4}$$

$$\mathcal{T}(Var) = \mathcal{T}(Var, 1) \tag{5}$$

$$\mathcal{T}(Var, B) = \mathsf{NewVar}_{ID,TYPEID}.(\mathcal{T}(Inst, ID, B))$$

$$\mathcal{T}(ElemInst, idVar, B) = \mathsf{NewInst}_{TYPEID,idVar}.(B) \tag{6}$$

$$\mathcal{T}(PatInst, idVar, B) = \mathsf{NewInst}_{TYPEID,idVar}.(\mathcal{T}(idVar, Var+, Att*, Ren*, B))$$

$$\mathcal{T}(idVar,\ Var\ Var*, Att*, Ren*, B) = \mathcal{T}(Var, \mathsf{AddVar}_{idVar,ID}.(\mathcal{T}(idVar, Var*, Att*, Ren*, B))) \tag{7}$$

$$\mathcal{T}(idVar, [\,], Att*, Ren*, B) = \mathcal{T}(Att*, Ren*, B)$$

$$\mathcal{T}(idIF\ idPF\ idIT\ idPT\ Att*, Ren*, B) = \tag{8}$$

$$
\begin{aligned}
&\quad \mathsf{NewAtt}\,_{idIF,\ idPF,\ idIT,\ idPT,\ uniqueId()}.(\mathcal{T}(Att*, Ren*, B))\\
&\mathcal{T}([\,], Ren*, B) = \mathcal{T}(Ren*, B)\\
&\mathcal{T}(idInst\ idPrt\ idNew\ Ren*, B) = \\
&\quad \mathsf{NewRen}_{idInst,\ idPrt,\ idNew,\ uniqueid()}.(\mathcal{T}(Ren*, B))\\
&\mathcal{T}([\,], B) = B
\end{aligned} \tag{9}
$$

Table 2: Sorting for Archery-Core

Ctrl	**Arity**	**Activeness**	**Sorts**	**Represented Item**
Pat	1	passive	u	A pattern
Elem	1	passive	u	An element
NewIn	1	passive	u	An in port within an element definition
In	1	atomic	i	An in port within an instance
NewOut	1	passive	u	An out port within an element definition
Out	1	atomic	o	An out port within an instance
NewInst	2	passive	uu	Instance creation and assignment
Inst	1	active	u	An Instance
NewVar	2	passive	uu	Variable creation
Var	2	active	uu	A variable
AddVar	2	passive	uu	Movement of one variable into another
NewAtt	5	passive	uuuuu	Attachment creation
From	2	atomic	fu	Attachment end for out port
To	2	atomic	tu	Attachment end for in port
NewRen	4	passive	uuuu	Rename creation
Int	2	passive	rr	Rename end for internal variable
Ext	2	passive	rr	Rename end for external instance

The result of applying Function $\mathcal{T}$ (2) to pattern `ClientServer` in Listing 1 is the bigraph shown in Figure 3a and in (10): a Pat node with *ClientServer* as outer name and the nesting of the merge product of applying (3) to each element. In the case of element `Client`, (3) creates an Elem node with the element identifier as outer name and the nesting of the merge product of respectively calling first and second functions in (4) with each in and out port of the element. The former function creates a NewIn node with *rres* as outer name, and the latter a node NewOut with *sreq* as outer name.

$$
\begin{aligned}
\mathsf{Pat}_{ClientServer}.(\ &\mathsf{Elem}_{Client}.(\ \mathsf{NewIn}_{rres} \mid \mathsf{NewOut}_{sreq}\ ) \mid \\
&\mathsf{Elem}_{Server}.(\ \mathsf{NewIn}_{rreq} \mid \mathsf{NewOut}_{sres}\ ))
\end{aligned} \tag{10}
$$

The result of applying Function $\mathcal{T}$ (5) to the architecture between lines 25 and 35 is shown in Figure 3b and in (11). The translation involves Rules in Table 3 triggered by intermediate bigraphs generated by Function $\mathcal{T}$ (5) to (9). It begins when (5) receives the example architecture and in combination with (6)

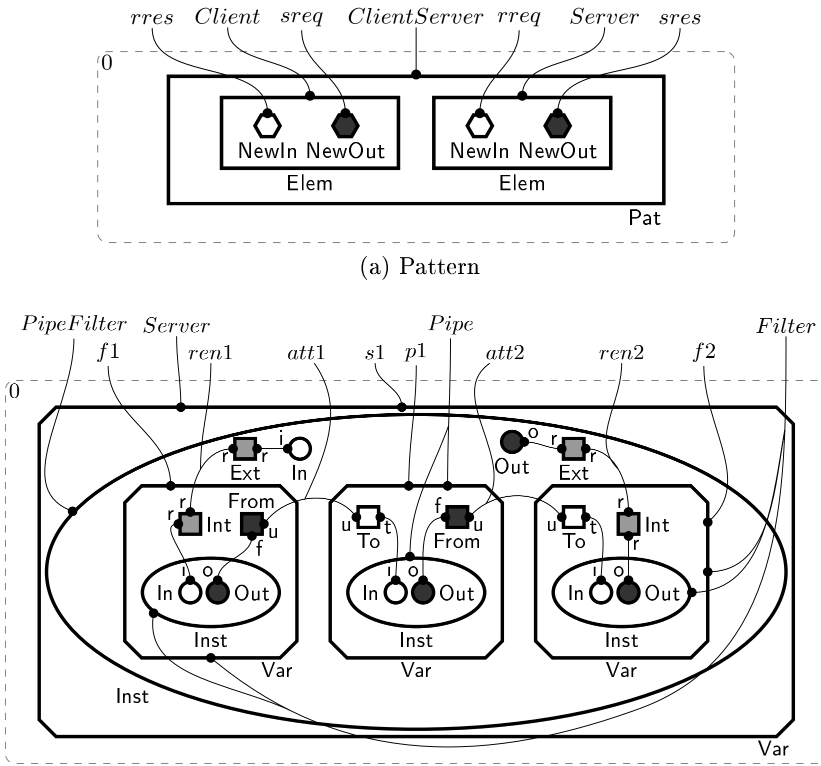
Table 3: Parametric Reaction Rules for Archery-Core

1	New Variable	$\mathsf{NewVar_{yx}}.d_0 \multimap \mathsf{Var_{yx}}.1 \parallel d_0$
2	Create Element Instance	$\mathsf{Elem_x}.d_0 \parallel \mathsf{Var_{y\_}}.1 \parallel \mathsf{NewInst_{yx}}.d_1 \multimap$ $\mathsf{Elem_x}.d_0 \parallel \mathsf{Var_{y\_}}.(\mathsf{Inst_x}.d_0) \parallel d_1$
3	Create Pattern Instance	$\mathsf{Pat_x}.d_0 \parallel \mathsf{Var_{y\_}}.1 \parallel \mathsf{NewInst_{yx}}.d_1 \multimap$ $\mathsf{Pat_x}.d_0 \parallel \mathsf{Var_{y\_}}.\mathsf{Inst_x}.1 \parallel d_1$
4	Create In Port	$\mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(\mathsf{NewIn_y} \mid d_0) \mid d_1) \multimap$ $/y\ \mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(\mathsf{In_y} \mid d_0) \mid d_1)$
5	Create Out Port	$\mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(\mathsf{NewOut_y} \mid d_0) \mid d_1) \multimap$ $/y\ \mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(\mathsf{Out_y} \mid d_0) \mid d_1)$
6	Add Instance	$\mathsf{Var_{x\_}}.(\mathsf{Inst_\_}.d_0 \mid d_1) \parallel \mathsf{Var_{y\_}}.d_2 \parallel \mathsf{AddVar_{xy}}.d_3 \multimap$ $\mathsf{Var_{x\_}}.(\mathsf{Inst_\_}.(\mathsf{Var_{y\_}}.d_2 \mid d_0) \mid d_1) \parallel d_3$
7	Add Attachment	$\mathsf{Var_{f\_}}.(\mathsf{Inst_\_}.(\mathsf{Out_o} \mid d_0) \mid d_1) \parallel$ $\mathsf{Var_{t\_}}.(\mathsf{Inst_\_}.(\mathsf{In_i} \mid d_2) \mid d_3) \parallel \mathsf{NewAtt_{fotia}}.d_4 \multimap$ $\mathsf{Var_{f\_}}.(\mathsf{Inst_\_}.(\mathsf{Out_o} \mid d_0) \mid \mathsf{From_{oa}} \mid d_1) \parallel$ $\mathsf{Var_t}.(\mathsf{Inst_\_}.(\mathsf{In_i} \mid d_2) \mid \mathsf{To_{ia}} \mid d_3) \parallel d_4$
8	Add Rename Out	$\mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(/p\ \mathsf{Var_{v\_}}.(\mathsf{Inst_\_}.(\mathsf{Out_p} \mid d_0) \mid d_1) \mid d_2) \mid d_3) \parallel$ $\mathsf{NewRen_{vpqr}}.d_4 \multimap$ $/q\ \mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(/p\ \mathsf{Var_{v\_}}.(\mathsf{Inst_\_}.(\mathsf{Out_p} \mid d_0) \mid \mathsf{Int_{pr}} \mid d_1) \mid$ $\mathsf{Ext_{qr}} \mid \mathsf{Out_q} \mid d_2) \mid d_3 \parallel d_4$
9	Add Rename In	$\mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(/p\ \mathsf{Var_{v\_}}.(\mathsf{Inst_\_}.(\mathsf{In_p} \mid d_0) \mid d_1) \mid d_2) \mid d_3) \parallel$ $\mathsf{NewRen_{vpqr}}.d_4 \multimap$ $/q\ \mathsf{Var_{\_\_}}.(\mathsf{Inst_\_}.(/p\ \mathsf{Var_{v\_}}.(\mathsf{Inst_\_}.(\mathsf{In_p} \mid d_0) \mid \mathsf{Int_{pr}} \mid d_1) \mid$ $\mathsf{Ext_{qr}} \mid \mathsf{Out_q} \mid d_2) \mid d_3 \parallel d_4$

and Rules 1 and 3 creates a Var node with a nested Inst. The former has *s1* and *Server*, and the later *PipeFilter*, as outer names. This node nesting is used to represent variable-instance pairs in general, and in particular corresponds to the variable `s1` of type `Server` containing a pattern instance of type `PipeFilter`. In turn, the latter nests the merge product of the encoding of each of the three variable-instance pairs of the architecture, obtained after successive applications of (5), (6) and (7) and the effects of Rules 1, 2, and 6.

$$
\begin{aligned}
&/rreq\ /sres\ \mathsf{Var}_{s1,Server}.(\ \mathsf{Inst}_{PipeFilter}.(\\
&\quad /rec\ /send\ \mathsf{Var}_{f1,Filter}.(\ \mathsf{Inst}_{Filter}.(\ \mathsf{In}_{rec} \mid \mathsf{Out}_{send}\ ) \mid \\
&\quad\quad \mathsf{From}_{send,att1} \mid \mathsf{Int}_{rec,ren1}) \mid \\
&\quad /accept\ /forward\ \mathsf{Var}_{p1,Pipe}.(\ \mathsf{Inst}_{Pipe}.(\ \mathsf{In}_{accept} \mid \mathsf{Out}_{forward}\ ) \mid \\
&\quad\quad \mathsf{To}_{accept,att1} \mid \mathsf{From}_{forward,att2}) \mid \\
&\quad /rec\ /send\ \mathsf{Var}_{f2,Filter}.(\ \mathsf{Inst}_{Filter}.(\ \mathsf{In}_{rec} \mid \mathsf{Out}_{send}\ ) \mid \\
&\quad\quad \mathsf{To}_{rec,att2} \mid \mathsf{Int}_{send,ren2}) \mid \\
&\quad \mathsf{In}_{rreq} \mid \mathsf{Ext}_{rreq,ren1} \mid \mathsf{Out}_{sres} \mid \mathsf{Ext}_{sres,ren2}\ )\ )
\end{aligned}
\tag{11}
$$

For instance, the encoding of `f1` has closures for outer names *rec* and *send*, and a Var with a nested Inst, that in turn nests one In and one Out node, with



(a) Pattern

(b) Architecture

Fig. 3: Bigraphs for the Client-Server Example

respective *rec* and *send* names. The encoding of attachments is generated by (8) and Rule 7. In the case of the one between `f1.send` and `p1.accept`, it respectively includes in each Var, a From and a To node. The created nodes share as outer name a unique identifier *att1* that establishes a link between them. The renames are translated by (9) and Rules 8 and 9. The encoding for the renaming of `f1.rec` as `rreq` of `s1` respectively includes an Int and an Ext nodes inside their Var nodes representing `f1` and `s1`. An In node is also created inside the latter. These three nodes have shared outer names: Int and Ext have the unique identifier *ren1*, Int and (internal) In have *rec*, and Ext and (external) In have *rreq*.

The link sorts $\Theta = \{\mathsf{o}, \mathsf{f}, \mathsf{t}, \mathsf{i}, \mathsf{r}, \mathsf{u}\}$ and the formulation rule Φ ensure valid configurations representing attachments: they can only connect ports with opposite direction. Rule Φ restricts the structure as follows: a link with a point o (port or inner name with sort o) can only have other points f or r; a link with a point i can only have other points t or r; a link with a point u has sort u and no constraints. The sorting assignment in Table 2 and Φ prevent a bigraph representing attachments between two ports with the same direction. Figure 3b shows two edges (with respective sort assignments) satisfying Φ.

4.2 Archery-Script

We translate a script into a bigraph in $\text{BG}(\Sigma_{Arch-Script}, \mathcal{R}_{Arch-Script})$. Both the sorting and the parametric reaction rules extend the ones defined for Archery-Core. $\Sigma_{Arch-Script}$ includes three more controls and $\mathcal{R}_{Arch-Script}$ adds the parametric reaction rules in Table 4.

Table 4: Parametric Reaction Rules for Archery-Script

10	Remove Attachment	$\mathsf{Var_{f-}}.(\mathsf{Inst_{-}}.(\mathsf{Out_{o}} \mid d_0) \mid \mathsf{From_{oa}} \mid d_1) \parallel$ $\mathsf{Var_{t,-}}.(\mathsf{Inst_{-}}.(\mathsf{In_{i}} \mid d_2) \mid \mathsf{To_{ia}} \mid d_3) \parallel \mathsf{RemAtt_{a}}.d_4 \rightarrowtriangle$ $\mathsf{Var_{f-}}.(\mathsf{Inst_{-}}.(\mathsf{Out_{o}} \mid d_0) \mid d_1) \parallel$ $\mathsf{Var_{t-}}.(\mathsf{Inst_{-}}.(\mathsf{In_{i}} \mid d_2) \mid d_3) \parallel d_4$
11	Remove Rename Out	$/q\ \mathsf{Var_{--}}.(\mathsf{Inst_{-}}.(/p\ \mathsf{Var_{v-}}.(\mathsf{Inst_{-}}.(\mathsf{Out_{p}} \mid d_0) \mid \mathsf{Int_{pr}} \mid d_1) \mid$ $\mathsf{Ext_{qr}} \mid \mathsf{Out_{q}} \mid d_2) \mid d_3) \parallel \mathsf{RemRen_{r}}.d_4 \rightarrowtriangle$ $\mathsf{Var_{--}}.(\mathsf{Inst_{-}}.(/p\ \mathsf{Var_{v-}}.(\mathsf{Inst_{-}}.(\mathsf{Out_{p}} \mid d_0) \mid d_1) \mid d_2) \mid d_3) \parallel d_4$
12	Remove Rename In	$/q\ \mathsf{Var_{--}}.(\mathsf{Inst_{-}}.(/p\ \mathsf{Var_{v-}}.(\mathsf{Inst_{-}}.(\mathsf{In_{p}} \mid d_0) \mid \mathsf{Int_{pr}} \mid d_1) \mid$ $\mathsf{Ext_{qr}} \mid \mathsf{In_{q}} \mid d_2) \mid d_3) \parallel \mathsf{RemRen_{r}}.d_4 \rightarrowtriangle$ $\mathsf{Var_{--}}.(\mathsf{Inst_{-}}.(/p\ \mathsf{Var_{v-}}.(\mathsf{Inst_{-}}.(\mathsf{In_{p}} \mid d_0) \mid d_1) \mid d_2) \mid d_3) \parallel d_4$
13	Move Instance	$\mathsf{Var_{d-}}.d_0 \parallel \mathsf{Var_{o}}.(\mathsf{Inst_{-}}.(d_1) \mid d_2) \parallel \mathsf{MoveInst_{od}}.d_3 \rightarrowtriangle$ $\mathsf{Var_{d-}}.\mathsf{Inst_{-}}.(d_1) \parallel \mathsf{Var_{o-}}.(d_2) \parallel d_3$

Function $\mathcal{TS}$ carries out the translation of a script $t = [t_1\ t_2\ \ldots\ t_n]$ by processing the first operation and returning a combination of the result and the recursive call with the tail of the sequence. Each operations t_i has as type one of the listed in Table 1. Expression (12) translates an import operation into the parallel product of the application of $\mathcal{T}$ to the specification *Spec*, and the recursive call with the rest of the script. Expressions (13) to (19) translate t by nesting the translation of the tail of t in a node that results from translating t_1. The created node partially triggers one of the reaction rules in $\mathcal{R}_{Arch-Script}$.

We introduce the (passive) controls and rules related to expressions (17), (18) and (19) since they are not present in $\Sigma_{Arch-Core}$ and $\mathcal{R}_{Arch-Core}$. The first expression creates a RemAtt node that represents a remove attachment operation and has one port of sort u. The outer name of the port is a unique id that matches the nodes involved in the encoding of the attachment. RemAtt partially triggers Rule 10, that removes such nodes, making the edge representing the attachment disappear. It also places the contents of RemAtt, matching parameter d_4, in a parallel root. The second (18) creates a RemRen node that represents a remove renaming operation and has one port of sort u. In a similar way, the outer name is a unique id that matches the nodes involved in the representation of the renaming. RemRen triggers either Rule 11 or 12, depending on whether the renaming is respectively over an out or an in port. Both rules have the same effect: the removal of all nodes encoding the renaming and placing the contents of RemRen in a parallel root. The third (19) creates a node MoveInst that represents an instance movement operation. The control has two ports with sort u: one identifier vo representing the original container for the instance, and another vd for the container to where it is moved. The node partially matches the redex of Rule 11. The reaction nests the contents of $\mathsf{Var}_{vo,\,-}$, matching $\mathsf{Inst}_{-}.(d_1)$, into $\mathsf{Var}_{vd,\,-}$. The former contents of the destination are lost. The original variable keeps the contents matching d_2 (outside the instance), and the contents matching d_3 are place in a parallel root.

$$\mathcal{TS}([\mathtt{import}(Spec);\ t]) = \mathcal{T}(Spec) \parallel \mathcal{TS}(t) \tag{12}$$

$$\mathcal{TS}([\mathtt{v : type};\ t]) = \mathsf{NewVar}_{v,\,type}.\mathcal{TS}(t) \tag{13}$$

$$\mathcal{TS}([\mathtt{v = type}();\ t]) = \mathsf{NewInst}_{v,\,type}.\mathcal{TS}(t) \tag{14}$$

$$\mathcal{TS}([\mathtt{addInst(a, v)};\ t]) = \mathsf{AddVar}_{a,\,v}.\mathcal{TS}(t) \tag{15}$$

$$\mathcal{TS}([\mathtt{attach(vf.pf, vt.pt)};\ t]) = \mathsf{NewAtt}_{vf,\,pf,\,vt,\,pt,\,uniqueId()}.\mathcal{TS}(t) \tag{16}$$

$$\mathcal{TS}([\mathtt{deattach(vf.pf, vt.pt)};\ t]) = \mathsf{RemAtt}_{id(vf,\,pf,\,vt,\,pt)}.\mathcal{TS}(t) \tag{17}$$

$$\mathcal{TS}([\mathtt{remRen(v.q)};\ t]) = \mathsf{RemRen}_{id(v,\,q)}.\mathcal{TS}(t) \tag{18}$$

$$\mathcal{TS}([\mathtt{move(vo, vd)};\ t]) = \mathsf{MoveInst}_{vo,vd}.\mathcal{TS}(t) \tag{19}$$

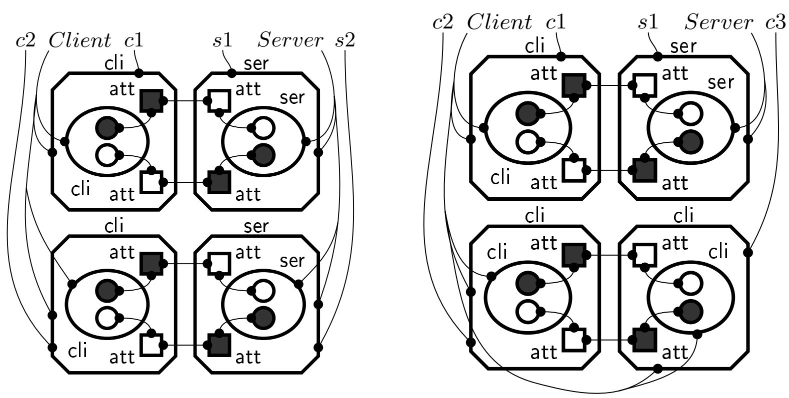
$$\mathcal{TS}([\,]) = 1$$

4.3 Archery-Structural-Constraint

The way constraints are dealt within the bigraphical framework discussed in this paper is now illustrated through an example. Let us consider the constraint φ formulated in Section 2.3. We derive from it a place sorting Σ_φ. Note that, in general, this derivation can be automated [12]. Then, a specification that fulfils φ is translated to a bigraph in $\mathrm{B}\textsc{g}(\Sigma_\varphi,\, \mathcal{R}_{Arch-Core})$.

For this example, we define Θ as $\{\mathsf{cli}, \mathsf{ser}, \mathsf{att}, \mathsf{oth}\}$ and Φ. The sort of a $\mathsf{Var}_{-,\,type}$ node depends on *type*: cli if it is `Client`, and ser if it is `Server`. From and To nodes have sort att, and other nodes have sort oth. Φ is as follows: a node att immediately *in* a node cli can only have an edge to an att immediately in a node ser. Given two nodes w and w', w is in w' if the former has w' as ancestor in the parent-child relationship.

It can now be verified whether a specification Var of a `ClientServer` instance preserves constraint φ, by checking if the type of bigraph $\mathcal{T}(Var)$ is $\mathrm{B}\textsc{g}(\Sigma_\varphi,\, \mathcal{R}_{Arch-Core})$. In Section 2.2 we described cs_{first} and cs_{wrong} as two configurations. Figure 4 partially shows the bigraphs that encode them. Only the sorts att, cli and ser, and nodes that participate in attachments are shown. Figure 4a contains a bigraph that partially encodes cs_{first}. It can be observed that all four nodes att in cli (respectively, ser) only have edges to nodes att in nodes ser (respectively, cli). Then, the bigraph is $\mathrm{B}\textsc{g}(\Sigma_\varphi,\, \mathcal{R}_{Arch-Core})$ and configuration cs_{first} satisfies φ. In contrast, the encoding of cs_{wrong} shown in Figure 4b, does not fulfil formation rule Φ: the nodes att in the node cli with outer name $c1$, have edges with nodes att in another node cli. Therefore, the bigraph is not an inhabitant of $\mathrm{B}\textsc{g}(\Sigma_\varphi,\, \mathcal{R}_{Arch-Core})$.



(a) cs_{first} (b) cs_{wrong}

Fig. 4: Bigraphs for Example Configurations

5 Conclusions

In this paper we introduced Archery, a modelling language for software architectural patterns rooted in the process algebra trend [10]. The language allows the specification of both structural and behavioural dimensions of architectures (Archery-Core), scripts to (re)configure such architectures (Archery-Script), and constraints to ensure that they obey the design principles of the pattern they are instance of (Archery-Structural-Constraint).

A second contribution of the paper was the development of a bigraphical semantics for Archery. To respect space limits, this was fully presented for Archery-Core, partially for the scripting component and illustrated through an example for constraints. By doing so, we were able to reduce the constraint satisfaction verification to a type checking problem.

We can distinguish two approaches in the design of languages that provide support for both the behavioural and structural dimensions, in architectural design. One is to extend a structure-based language with a behavioural model [6], and the other is to build the architectural language on top of the behavioural model [1], by upgrading it with architectural constructs. Our work is along the lines of the latter approach but with the difference that we used bigraphs as a foundation for the structural dimension. Benefits of using the bigraphical theory include its solid categorical framework, its independent treatment of locality and linking of computational agents, and its role of unifying theory for concurrency and mobile calculi. The work in [9] also provides a bigraphical semantics to an architectural description language. While our encoding uses a single signature to encode any pattern, theirs requires different signatures for different patterns. There are two main approaches to the reconfiguration of pattern instances: one is to define a generic set of operations and reflect a pattern's design principles with constraints that prevent illegal configurations; and another is to design a pattern-specific set of operations that allow to correctly (re)configure instances [7]. Our work is aligned with the former.

As part of future work we mention the derivation process for sortings that encode constraints. The process must ensure that the resulting sorting does not prevent the automatic derivation of an LTS for a BRS, and consider the decidability and complexity of type-checking.

Acknowledgements

This research was partially supported by the project EVOLVE (Evolutionary Verification, Validation and Certification) under contract QREN `1621`.

References

1. Aldini, A., Bernardo, M., Corradini, F.: A Process Algebraic Approach to Software Architecture Design, vol. 54. Springer London (2010)
2. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (Jun 2004)
3. Awodey, S.: Category Theory (Oxford Logic Guides). Oxford University Press, USA, second edn. (Aug 2010)
4. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley Longman Publishing Co., Inc., second edn. (2003)
5. Birkedal, L., Debois, S., Hildebrandt, T.: On the construction of sorted reactive systems. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008 - Concurrency Theory, Lecture Notes in Computer Science, vol. 5201, pp. 218–232. Springer Berlin/Heidelberg (2008)
6. Bodeveix, J.P., Filali, M., Gaufillet, P., Vernadat, F.: The AADL real-time model A behavioural annex for the AADL. In: Proceedings of the DASIA 2006 – DAta Systems In Aerospace – Conference (2006)
7. Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., Lluch Lafuente, A.: Concurrency, graphs and models. chap. Graph-Based Design and Analysis of Dynamic Software Architectures, pp. 37–56. Springer-Verlag, Berlin, Heidelberg (2008)
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley (1996)
9. Chang, Z., Mao, X., Qi, Z.: An Approach based on Bigraphical Reactive Systems to Check Architectural Instance Conforming to its Style. In: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, 2007. TASE '07. pp. 57 – 66. IEEE Computer Society (2007)
10. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language mCRL2. In: Methods for Modelling Software Systems: Dagstuhl Seminar 06351 (2007)
11. Milner, R.: The space and motion of communicating agents, vol. 54. Cambridge University Press (2009)
12. Sanchez, A.: A Calculus of Architectural Patterns (to appear). Ph.D. thesis, Universidad Nacional de San Luis (2012)
13. Sanchez, A., Barbosa, L.S., Riesco, D.: A Language for Behavioural Modelling of Architectural Patterns. In: Proceedings of the 3rd Workshop on Behavioural Modelling - Foundations and Applications (BM-FA 2011). ACM DL (2011)