BIGACTORS - A MODEL FOR STRUCTURE-AWARE COMPUTATION

Eloi Pereira, Christoph Kirsch, Raja Sengupta, and João Sousa

Working Papers

CPCC-WP-2012-11-19

CPCC Berkeley

November 2012

BigActors - A Model for Structure-Aware Computation

Eloi Pereira[1,2], Chirstoph Kirsch[3], Raja Sengupta[1] and João Sousa[4]

Abstract

This paper describes a model of computation for structure aware computing called the BigActor model. The model is hybrid. It combines the Actor model [1] and the Bigraph model [12]. The contributions of this paper are an operational semantics, an example illustrating how the model supports the concise programming of a mobile agent working in a ubiquitous computing world, a query language enabling a bigActor to observe the world around it, and a three sufficient conditions for correct execution of the model under the presence of concurrency and dynamic structure.

I. Introduction

This paper describes a model of computation for structure-aware computing. The model is a hybrid. It combines the Actor model [1] and the Bigraph model [12]. Hence the name *BigActor model*. In this model, computation is modelled using the Actor model, enriched by three new semantics rules (Figure 8) to make it structure-aware. Structure is a bigraph. The dynamics of the structure is modelled as a Bigraph Reactive System [11]. The three extra semantic rules provide means for an actor to observe and control a bigraph, and migrate from one host to another. Hence it is called a bigActor. When the structure models a physical world as in Example 3 of Section IV, bigActors become cyber entities in a physical world.

The contributions of this paper are an operational semantics for the BigActor model stated in Section IV-A. BigActors are linked to a model of the structure (bigprah) by a hosting relation. Without a structure (or context as in context-aware computing), a bigActor is just an actor. But when a bigActor is hosted in a bigraph it can leverage the three new semantic rules enabling it to observe the bigraph, control it, and migrate in it. The set of observations a bigActor can make in the bigraph hosting it, is formalized as a query language. Section V formalizes one possible query language. The operational semantics is well defined for a class of query languages as expressed by Equation 1. Section IV-A includes an example program (Example 3) to illustrate how this model contributes to the concise programming of a mobile agent (`app@sp` in Figure 7) computing in a ubiquitous computing environment (Figure 10).

The BigActor model is defined over a bigraph transition system. This is formally a sequence of bigraphs generated by applying Bigraph Reaction Rules [12] to an initial bigraph. Thus, structure can be dynamic and the three new semantic rules are designed to program actors that adapt to changing structure. Context-Aware programming needs a semantics for the interaction of program and context. In this paper we borrow the one underpinning control theory (see Figure 1 for an example of bigActors depicted as a feedback loop), because it is widely used. In control theory the system is separated into plant and controller with the controller being composed in feedback. It observes, then controls, and the control generates new observations, repeating the cycle. In our case, bigActors observe the bigraph, then control the bigraph or migrate to other hosts in the bigraph.

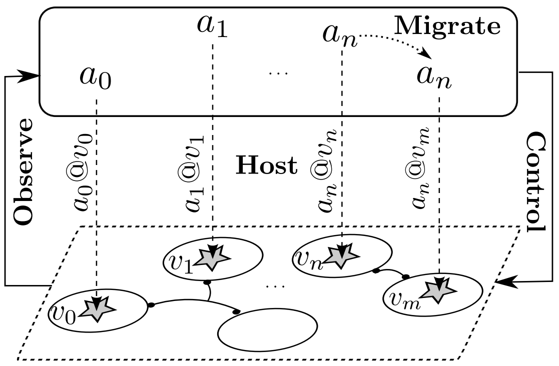


Fig. 1. BigActors: actors composed as a feedback loop with a bigraph.

[1]Systems Engineering, Department of Civil and Environmental Engineering, UC Berkeley, USA. Email: eloi@berkeley.edu, sengupta@ce.berkeley.edu.
[2]Research Center of the Portuguese Air Force Academy, Portugal.
[3]Department of Computer Science, University of Salzburg, Austria. Email: ck@cs.uni-salzburg.at.
[4]School of Engineering, Porto University, Portugal. Email: jtasso@fe.up.pt.

Section VI shows the BigActor model is able to support this semantics of adaptation. The controller is restricted to being a bigActor (or a set of bigActors) and the plant a bigraph. Definition 5 is the semantics of the feedback loop.

One can find several approaches in the literature for providing semantics of feedback loops. For example, Simulink[1] has a loop detection mechanism (known as algebraic loop detection). When a loop is found Simulink tries to solve it [5]. Edwards and Lee [7] provide semantics for feedback loops in a synchronous block-diagram language as the unique least fixed point of the function of all the blocks. Ramadge and Wonham [15], [16] provide feedback loop semantics for Discrete Event Systems (i.e. discrete, asynchronous, and possibly nondeterministic processes). The plant (process being controlled) is described as the generator of a formal language, while the controller is constructed as per a specified target language that incorporates the desired feedback loop behavior. Such controllers are also known as supervisors. The existence of a supervisor is reduced to finding the largest controllable language contained in a given legal language. The composition of plant and controller is *blocking*, in the sense that when the controller is computing the plant is blocked.

Our approach is motivated by [15], [16] although in our case, the controllers are specified as (possibly concurrent) bigActors while the plant is a bigraph. Our composition is asynchronous which allows the introduction of blocking and non-blocking semantics.

BigActors may request control actions that can never be executed or messages that can never be received. We call these incorrect executions. We address correctness issues of bigActor executions using two orthogonal approaches: require bigActors to not request unexecutable commands regarding their observations; and prevent executable commands to become unexecutable due to concurrency. We define a feedback bigActor analogous to a feedback controller in Definition 12. Theorem 1 shows that in a system with one bigActor, being a feedback bigActor is a sufficient condition for correctness. Theorem 2 presents a sufficient condition for correctness of concurrent bigActors by preventing undesirable interleaving.

Theorem 3 show another sufficient condition for correctness of concurrent bigActors based on requiring bigActors to always operate in disjoint areas of the bigraph. Theorems 1, 2, and 3 formalize the sense in which the bigActor model supports the programming of concurrent agents working in a world with changing structure by adapting to it.

We situate the BigActor model in the Actor and Bigraph literatures as follows. The contributions of this model to the Actor literature are to provide the actor with a reflection of the structure of the system. This approach has been adopted to reflect back to actor systems low-level properties of the system. Nielsen, Ren, and Agha [13], [14] provide a real-time semantics to actor systems using timed graphs to constrain the execution of actors. Agha introduces locality to actors semantics [17] for modelling multiple mobile agents. The location model used by Agha is simply a host relation between actors and hosts. The location model is flat in the sense that there is no information of where a host stands in relation to another host. Moreover, there is no sense of connectivity between hosts. Using bigraphs in the BigActor model we provide a richer model of the structure, capable of modelling nested locality of components, their connectivity and also the way the structure evolves with time.

From a bigraph research stand-point, the BigActor model provides means for embedding computation into bigraphs that model the structure of the world. This is addressed in the literature by [6], [4] which model the structure of the world and computation using different Bigraph Reaction Systems (BRS) which are composed together. It is known that querying bigraphs exclusively using bigraphs reactive systems is difficult [4]. This is addressed by [4] by introducing three different BRSs: context, proxy, and agents. Our approach differs from [6] and [4] in the sense that we model computation using the actors model, which is coupled from an observation stand point using a query language. This removes the burden of modelling queries using BRSs. Moreover, by combining the Actor model with bigraphs we intend to leverage the use of bigraphs together with the large spectrum of actor languages and frameworks (e.g. Erlang [3], Scala [9], Cloud Haskell [8], Dart, Akka, etc.).

II. Bigraphical Framework

In this section we introduce Bigraphs. A reader familiar with Bigraphs should be able to skip this section. The examples, however are used throughout the paper. This explanation follows [12].

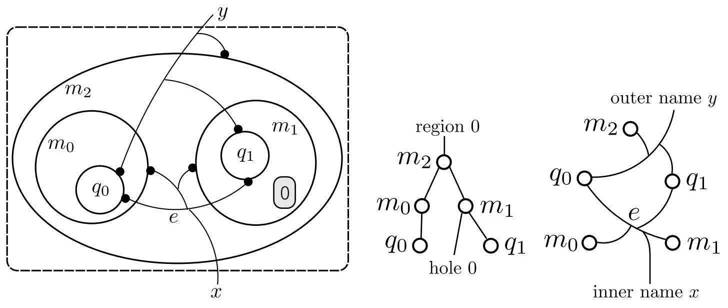
As the name suggests a bigraph is a mathematical structure with two graphs, the place graph - a forest that represents nested locality of components and a link graph - a hypergraph that models connectivity between components. Figure 2 presents an example of a bigraph.

Place graphs are contained inside *regions* (dashed rectangles) and may also contain *holes* (dark grey empty rectangles) Regions and holes enable composition of placing graphs, i.e. a hole of a given bigraph can be replaced by a region of another bigraph using the composition operator. We explain composition later.

A link graph may contain hyperedges and *inner names* and *outer names*. Names are graphically represented by a line connected at one end to a port or an edge and the other end is left loose. Just as one can fit regions inside holes, one can also merge inner names and outer names using the composition operator.

A node can have *ports* (black dots) which are points for connections to edges or names. The kinds of nodes and their number of ports (arity) are the signature of the bigraph.

[1]Simulink is a registered trademark of MathWorks, Inc.



(a) Bigraph. (b) Placing and Linking graphs.

Fig. 2. Example of a bigraph and the corresponding place and link graphs.

The *signature* takes the form $(\mathcal{K}, ar)$ where $\mathcal{K}$ is a set of kinds of nodes called *controls* and $ar : \mathcal{K} \rightarrow \mathbb{N}$ assigns an arity (i.e. a natural number) to each control. Each node in the bigraph is assigned a *control*. For example, the bigraph of Figure 2(a) has the following signature: $\mathcal{K} = \{M : 1, Q : 2\}$ where m_i has kind M with arity 1 and q_i has kind Q with arity 2. By convention we start kind names with upper-case characters and node names with lower-case characters.

A bigraph B is called *concrete* when each node and each edge is assigned a unique identifier (known as *support*). We denote the set of node identifiers of B as V_B and the set of edge identifiers as E_B (e.g. $V_B = \{m_0, m_1, m_2, q_0, q_1\}$ and $E_B = \{e\}$ in the example of Figure 2).

A bigraph without support is called *abstract*. In abstract bigraphs, nodes are denoted by their control while edges are kept anonymous. Abstract bigraphs are defined using an algebra. In this paper we almost exclusively use concrete bigraphs and thus we devote this section to present them formally. For a formal introduction to abstract bigraphs see [12], Chapter 3.

In order to define bigraphs formally we need to introduce the concept of a bigraph interface. An **interface** is a pair $\langle n, X\rangle$ where n in the interface denotes the set $\{0, 1, \ldots, n-1\}$ of holes (or regions) and X denotes the set of inner names (or outer names). Holes and inner names are collectively called an *inner face* while regions and outer names are called an *outer face*.

Definition 1. A **bigraph** is a 5-tuple of the form:

$$(V, E, ctrl, prnt, link) : \langle m, X\rangle \rightarrow \langle n, Y\rangle$$

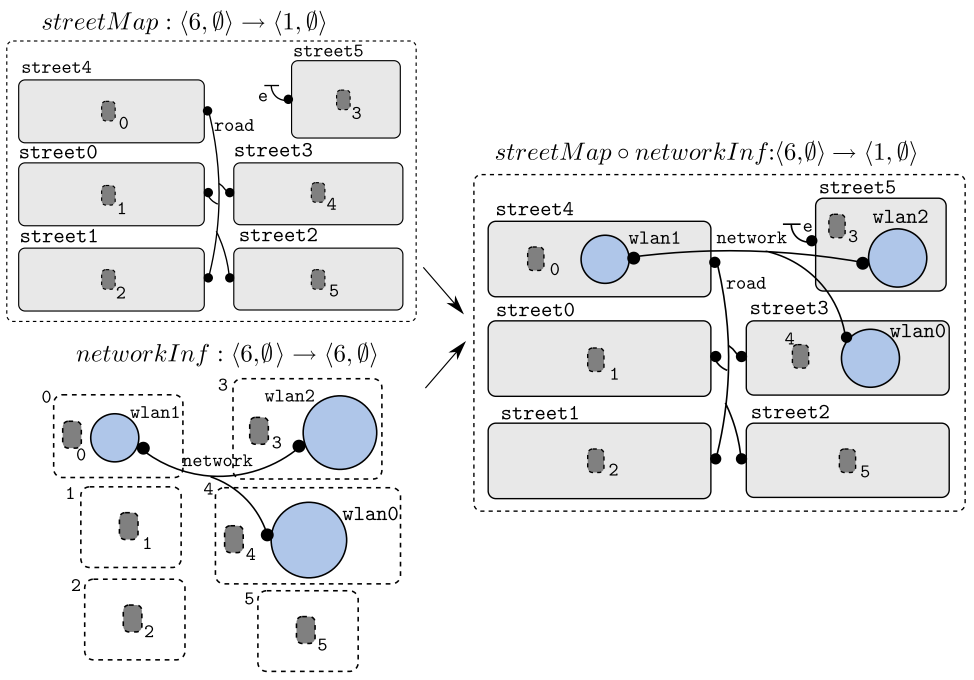
where V is a set of nodes, E is the set of hyperedges, $ctrl : V \rightarrow \mathcal{K}$ is a control map that assigns controls to nodes, $prnt : m \uplus V \rightarrow V \uplus n$[2] is the parent map and defines the nested place structure, $link : X \uplus P \rightarrow E \uplus Y$ is the link map and defines the link structure. P denotes the set of ports of the bigraph and is formalized as $P = \{(v, i) \mid i \in \{0, 1, \ldots, ar(ctrl(v)) - 1\}\}$. For convenience we introduce a map $Pts : V_B \rightarrow \mathcal{P}(\mathbb{N})$ that takes a node and returns the set of ports of that node. $\langle m, X\rangle \rightarrow \langle n, Y\rangle$ provides the inner face and outer face of the bigraph, i.e. $\langle m, X\rangle$ is the inner face and $\langle n, Y\rangle$ is the outer face.

The composition of bigraphs is defined by matching interfaces. Like function composition, a bigraph $A : I \rightarrow J$ composed with a bigraph $B : K \rightarrow I$ is a bigraph $C : K \rightarrow J$.

Definition 2. Let $A : \langle m_a, X_a\rangle \rightarrow \langle n_a, Y_a\rangle$ and
$B : \langle m_b, X_b\rangle \rightarrow \langle n_b, Y_b\rangle$. The **composition** of bigraphs A and B, denoted by $A \circ B$, is a bigraph $C : \langle m_b, X_b\rangle \rightarrow \langle n_a, Y_a\rangle$ where $V_C = V_A \uplus V_B$, $E_C = E_A \uplus E_B$, $ctrl_C = ctrl_A \uplus ctrl_B$. $prnt_C$ is obtained by "filling" the holes of A with regions of B while $link_C$ is obtained by "merging" the inner names of A with the outer names of B. By convention, the regions are matched to holes with the same indices and inner names are matched with outer names with the same name. For a formal definition of $prnt_C$ and $link_C$ see [12], page 17. The composition $A \circ B$ is well defined iff the outer face of B is equal to the inner face of A, i.e. $\langle n_b, Y_b\rangle = \langle m_a, X_a\rangle$.

Example 1. Consider the bigraphs of Figure 3. The bigraph $streetMap : \langle 6, \emptyset\rangle \rightarrow \langle 1, \emptyset\rangle$ models a street map where the grey nodes represent streets and links represent physical adjacency between street nodes. Bigraph $networkInf : \langle 6, \emptyset\rangle \rightarrow \langle 6, \emptyset\rangle$ models a network infrastructure where the blue nodes model wireless hotspots and links represent connectivity which is linked to the edge `network` modelling a network. The bigraph resulting from the composition of both bigraphs is $streetMap \circ networkInf : \langle 6, \emptyset\rangle \rightarrow \langle 1, \emptyset\rangle$. Note that since the $networkInf$ keeps the holes inside street nodes one could compose $streetMap \circ networkInf$ with other features (e.g. cars, pedestrians, utilities network, etc.). The signature for this example and subsequent ones in this paper is $\mathcal{K} = \{\texttt{Street} : 1, \texttt{Wlan} : 1, \texttt{Comp} : 1, \texttt{Feat} : 0\}$. In Figure 3, `streeti` are of kind `Street`, and `wlani` are of kind `Wlan`. The kinds `Street` and `Wlan` denote respectively streets on a city and locations with wireless connectivity. The kinds `Comp` and `Feat` are going to be used in examples later and represent, respectively computational devices and features of the world to be observed.

[2]The symbol $\uplus$ denotes the exclusive union operator for sets.



$streetMap : \langle 6, \emptyset \rangle \rightarrow \langle 1, \emptyset \rangle$
street4
street5
street0
street3
street1
street2
road
e
$networkInf : \langle 6, \emptyset \rangle \rightarrow \langle 6, \emptyset \rangle$
wlan1
wlan2
wlan0
network
$streetMap \circ networkInf : \langle 6, \emptyset \rangle \rightarrow \langle 1, \emptyset \rangle$

Fig. 3. Composition of the bigraph $streetMap$ with the bigraph $networkInf$.

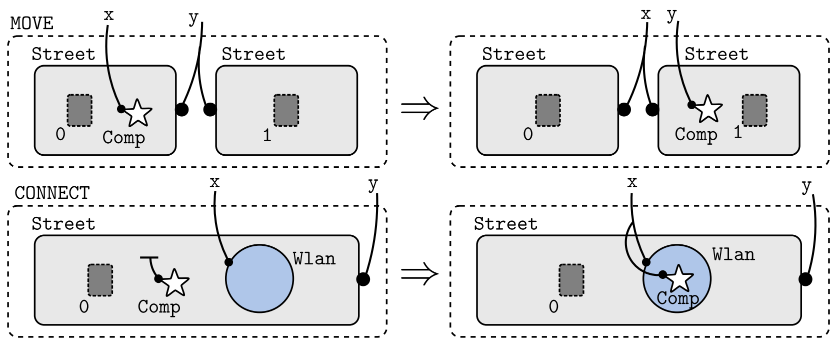


Fig. 4. Abstract BBRs MOVE that moves a Comp node from one street to another, and CONNECT that connects the Comp node to the network infrastructure.

A. Dynamics of bigraphs

Milner defines Bigraph Reaction Rules (BRR) to create dynamics on bigraphs [12] .

A *bigraph reaction rule* is a tuple (R, R', η) where R and R' are bigraphs called respectively *redex*[3] and *reactum*. The redex is the portion of the bigraph to be matched and the reactum is the bigraph that replaces the matched portion. η is called the instantiation map and indicates how holes in R correspond to holes in R'. If η is the identity map, then we represent the rule as $R \rightarrow R'$. For the reminder of the paper we always assume η is the identity map, i.e. hole i in R matches with hole i in R' for all $i \in \mathbb{N}$. If R and R' are abstract bigraphs, then $R \rightarrow R'$ is an abstract BRR. If R and R' are concrete then the BRR is concrete. It is often convenient to define BBRs to be abstract even when working with concrete bigraphs. This defines rules that can be applied to several contexts.

Let $r = R \rightarrow R'$ be a BRR and B a bigraph. In order to perform the reaction r in B we need first to decompose B into $C \circ R \circ d$ where C represents the context and d represents the parameters inside the holes of R. Assuming that η is the identity map, we compose C with the reactum R' and with d to get the resulting B', i.e. $B = C \circ R \circ d \Rightarrow B' = C \circ R' \circ d$. This application of BRRs works both for abstract and concrete BRRs.

Example 2. Consider the two abstract BRR of Figure 4. The first rule, MOVE, models a computational device (e.g. a user with a smartphone) denoted by a star with kind Comp) moving from one street to another. The second rule, CONNECT, models computational device to move inside a hotspot with network connectivity (denoted by a node with kind Wlan) and connecting to the network. Note that the rules are parametric, i.e. they can be applied regardless the nodes inside the street nodes. For example, this rule allows modelling a computational device to move with or without other users in a street. Figure 5 shows an

[3]Redex stands for "reducible expression".

example of the application of BRR `MOVE`. The bigraphs in this example are concrete. Thus, we need to first concretize `MOVE`. Since we want to move `sp0` of kind `Comp` from `street2` to `street3` we denote the concrete BRR as `MOVE(street2, street3)`. Figure 5 depicts the context C, and parameters d where the rule is applied. Note that the parameters allow the rule to be applied with `sp1` inside `street3`. The upper part of Figure 5 shows the bigraph B transitioning to B'. Under B and B' we

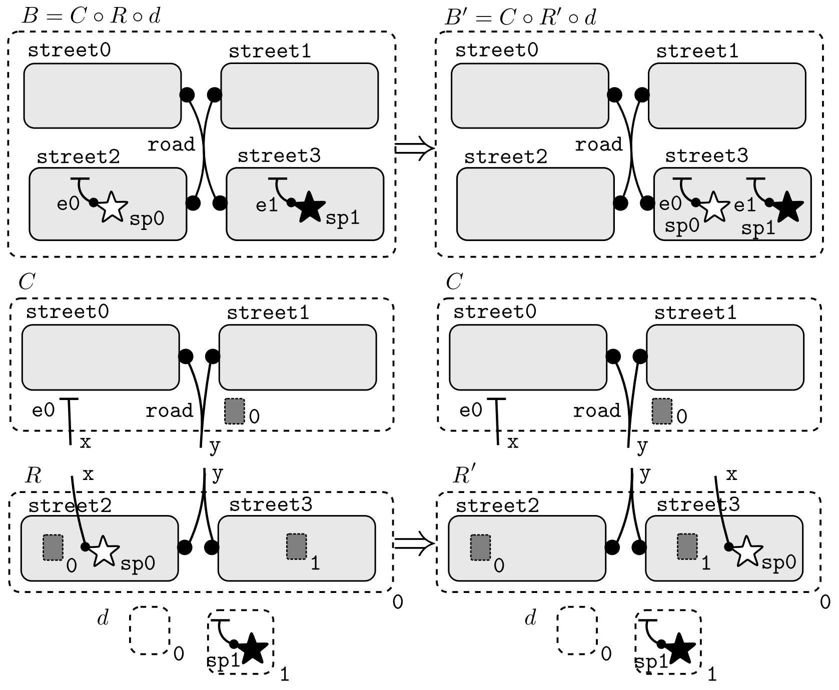


Fig. 5. Example of the application of the `MOVE` reaction rule.

show the decomposition of each bigraph in the context C, redex R, reactum R', and parameters d.

III. Actor Model

The Actors model of computation is a model for distributed concurrent computing entities [1], [2]. An actor system is composed of autonomous objects called actors. Actors communicate using asynchronous message passing. Messages that have been sent but not yet received are queued up in the receiver actor's mailbox. The receiver eventually removes the message and processes it. An actor encapsulates a state and a thread. Each actor has a mail-address used by other actors to send it messages.

As a response to a message an actor may: **compute** and change state; **send** a message; or **create** new actors.

The Actor model has been adopted as the concurrency model in several programming languages such as Erlang [3], Scala [9], Dart, Could Haskell [8], and in embedded systems design [10].

Next we present an operational semantics (using the contextual style) for the actors model. We follow the operational semantics approach taken in [13], [2], [17].

The Actor model operational semantics is formalized as a transition relation over the set of actors configurations.

Definition 3. The Actor model configuration is a tuple $\langle \alpha \mid \mu \rangle$ where α is a set of actor names, and μ is the set of pending messages. An actor $a \in \alpha$ can be either *busy* or *inactive*. An actor is busy if it is currently executing. $[E \vdash R \sqsubset e \sqsupset]_a$ denotes a **busy actor** a, with environment E (i.e. local state), and reduction context R filled with expression e which is currently being executed. An actor is *inactive* if it is waiting for a message. $(E \vdash b)_a$ denotes an **inactive actor** a with environment E and next behaviour b. When a message arrives, the actor's behaviour b is applied to the corresponding message. $\langle a \Leftarrow v \rangle$ denotes a message to a with content v.

The operational semantics is defined using the five rules[4] of Figure 6. The rule $\langle \mathbf{fun} : \mathbf{a} \rangle$ models the execution of an expression e of an actor a. $\rightarrow_\lambda$ denotes the semantics of a given language that specifies the local computation of the actor. Note that the execution of e to e' can produce side-effects in the actor's local environment E. This rule is also responsible for removing nil and making the actor ready to be terminated by rule `term`. The rule $\langle \mathbf{new} : \mathbf{a}, \mathbf{a'} \rangle$ models the spawn of a new actor a' from a. The new actor a' becomes inactive after creation. The rule $\langle \mathbf{term} : \mathbf{a} \rangle$ models the termination of the execution

[4]The original semantics includes two more rules for handling communication with external actors, i.e. actors not defined in the configuration. External actors are not an essential feature for bigActors and thus we decided to remove them for the sake of simplifying the exposition.

$$\langle \mathbf{fun} : \mathbf{a}\rangle \frac{[E \vdash e]_a \rightarrow_\lambda [E' \vdash e']_a}{\langle \alpha, [E \vdash e]_a \mid \mu\rangle \rightarrow \langle \alpha, [E' \vdash e']_a \mid \mu\rangle}$$

$$\langle \mathbf{new} : \mathbf{a}, \mathbf{a}'\rangle$$
$$\langle \alpha, [E \vdash R \sqsubset \mathtt{new}() \sqsupset]_a \mid \mu\rangle \rightarrow \langle \alpha, [E \vdash R \sqsubset \mathit{nil} \sqsupset]_a, (E \vdash b)_{a'} \mid \mu\rangle$$
$$\langle \mathbf{term} : \mathbf{a}\rangle$$
$$\langle \alpha, [E \vdash R \sqsubset \sqsupset]_a \mid \mu\rangle \rightarrow \langle \alpha, (E \vdash b)_a \mid \mu\rangle$$
$$\langle \mathbf{rcv} : \mathbf{a}, \langle \mathbf{a} \Leftarrow \mathbf{m}\rangle\rangle$$
$$\langle \alpha, (E \vdash b)_a \mid \mu, \langle a \Leftarrow v\rangle\rangle \rightarrow \langle \alpha, [E[FV(b) \mapsto v] \vdash b \sqsubset \mathit{nil} \sqsupset]_a \mid \mu\rangle$$
$$\langle \mathbf{snd} : \mathbf{a}, \langle \mathbf{a}' \Leftarrow \mathbf{m}\rangle\rangle$$
$$\langle \alpha, [E \vdash R \sqsubset \mathtt{send}(a', m) \sqsupset]_a \mid \mu\rangle \rightarrow \langle \alpha, [E \vdash R \sqsubset \mathit{nil} \sqsupset]_a \mid \mu, \langle a' \Leftarrow v\rangle\rangle$$

Fig. 6. Actor operational semantics.

of a command, making the actor changing from busy to inactive. The rule $\langle \mathbf{rcv} : \mathbf{a}, \langle \mathbf{a} \Leftarrow \mathbf{v}\rangle\rangle$ models an inactive actor a removing a message from its mailbox and updating its local environment with the contents v. The rule $\langle \mathbf{snd} : \mathbf{a}, \langle \mathbf{a}' \Leftarrow \mathbf{v}\rangle\rangle$ models an actor a sending a message v to a'. The new message is included in the set of pending messages.

IV. BigActor Model

This section formalizes the BigActor Model. The operational semantics is presented in Section IV-A. We explain the semantics by deriving a trace of the program in Example 3.

Example 3. Consider the pseudo-code in Figure 7. The pseudo-code is in the style of actor languages. The program models

```
app@sp
  method start()
     control(MOVE(street2,street1));
     observe(children.parent.host);
  method rcvObs(B_obs: Bigraph)
     control(MOVE(street1,street3));
     control(CONNECT(wlan0));
     send(server@srv,B_obs);
```

Fig. 7. Pseudo-code for the appBA BigActor.

the following scenario. An app is hosted on a smartphone (app@sp) in street2. It wants to photograph the green boxes (Figure 10) in street1. Hence the smartphone (bigraph node sp) moves. It specifies the picture to be taken by a query and waits till the picture is bound to its free variable. In order to send the picture to server@srv the app moves to one of the blue circles modeling wireless hotspots, and connects to the hyperlink network. It then sends the message to the server.

A. Operational semantics

The operational semantics is entirely in Figure 8. We use the same contextual style as the actor semantics in Section III. A BigActor model executes by transitioning through a sequence of configurations. Definition 4 defines a configuration.

Definition 4. A BigActor **configuration** is a tuple $\langle \alpha \mid \mu \mid \eta \mid B\rangle$ where α is a set of bigActors, μ is a set of pending messages, η is a set of pending requests, and B is a bigraph. A BigActor in α is of the form $a@h$, with a an actor, and $h \in V_B$ a node in the bigraph B. h is called the host of a. The set of pending messages μ is as in the actor semantics in Section III. A request in the set of requests η is a tuple $(a@h, r)$ meaning that the operation r is requested by $a@h$. The operation r can be any one of the expressions $\mathtt{send(a'@h',m)}$, $\mathtt{observe(q)}$, $\mathtt{control(u)}$, $\mathtt{migrate(h')}$. These are the premises of semantic rule $\langle \mathtt{req} : \mathtt{a@h}, \mathtt{r}\rangle$ in Figure 8. $\mathtt{a'@h'}$ can be any BigActor in α, m any actor message, q a query resulting in an observation by a BigActor of the bigraph B as expressed by Equation 1, u any concrete bigraph reaction rule as described in Section II, and h' another node in the bigraph B.

The interpretation of a query into an observation of a given bigraph B relative to a host h is specified by the map:

$$[\![\cdot]\!]_h^B : \mathtt{query} \rightarrow \mathbb{B} \tag{1}$$

Queries and observations are addressed in Section V.

Of the four elements in the BigActor configuration, α is as in the actor model except we write the $a@h$ instead of just a for the elements of α, μ is exactly as in the actor model, and η and B are new. Definition 5 defines an execution of the BigActor model as a sequence of configurations.

Definition 5. A BigActor execution is a sequence $c_0, c_1, c_2, \ldots$ where each $c_i \xrightarrow{\lambda_i} c_{i+1}$ and $\xrightarrow{\lambda_i}$ is derived by a semantic rule labelled λ_i. The permissible λ_i are those in Figure 8.

$$\langle \mathbf{fun : a@h} \rangle, \langle \mathbf{new : a@h, a'@h} \rangle \qquad \langle \mathbf{term : a@h} \rangle, \langle \mathbf{rcv : a@h, m} \rangle$$

$$\langle \mathbf{req : a@h, r} \rangle \quad \frac{\mathtt{r} \in \{\mathtt{send(a'@h', m)}, \mathtt{observe(q)}, \mathtt{control(u)}, \mathtt{migrate(h')}\}}{\langle \alpha, [E \vdash R \sqsubset \mathtt{r} \sqsupset]_a@h \mid \mu \mid \eta \mid B \rangle \rightarrow \langle \alpha, [E \vdash R \sqsubset \mathit{nil} \sqsupset]_a@h \mid \mu \mid \eta, (a@h, \mathtt{r}) \mid B \rangle}$$

$$\langle \mathbf{obs : a@h, q, o} \rangle \quad \frac{B_{obs} = [\![\mathtt{q}]\!]_h^B \quad \mathtt{q} \in \mathcal{Q}_a \quad E' = E[FV(b) \mapsto B_{obs}]}{\langle \alpha, (E \vdash b)_a@h \mid \mu \mid \eta, (a@h, \mathtt{observe(q)}) \mid B \rangle \rightarrow \langle \alpha, E' \vdash b \sqsubset \mathit{nil} \sqsupset]_a@h \mid \mu \mid \eta \mid B \rangle}$$

$$\langle \mathbf{ctr : a@h, u} \rangle \quad \frac{\mathtt{u} = (R \rightarrow R')\, h \in V_R \cap V_{R'}\, B = C \circ R \circ d\, B' = C \circ R' \circ d}{\langle \alpha \mid \mu \mid \eta, (a@h, \mathtt{control(u)}) \mid B \rangle \rightarrow \langle \alpha \mid \mu \mid \eta \mid B' \rangle}$$

$$\langle \mathbf{mgrt : a@h, h'} \rangle \quad \frac{\mathtt{h'} \in V_B \;\; \exists pt \in \mathit{Pts}(h).\exists pt' \in \mathit{Pts}(h').\mathit{link}(pt) = \mathit{link}(pt')}{\langle \alpha, a@h \mid \mu \mid \eta, (a@h, \mathtt{migrate(h')}) \mid B \rangle \rightarrow \langle \alpha, a@h' \mid \mu \mid \eta \mid B \rangle}$$

$$\langle \mathbf{snd : a@h, \langle a'@h' \Leftarrow m \rangle} \rangle \quad \frac{\exists pt \in \mathit{Pts}(h).\exists pt' \in \mathit{Pts}(h').\mathit{link}(pt) = \mathit{link}(pt') \vee (h = h')}{\langle \alpha \mid \mu \mid \eta, (a@h, \mathtt{send(a'@h', m)}) \mid B \rangle \rightarrow \langle \alpha \mid \mu, m \mid \eta \mid B \rangle}$$

Fig. 8. BigActors operational semantics.

The semantics can be understood as follows. A BigActor can do actor computations. This is asserted by the inclusion of the rules $\langle \mathbf{fun : a@h} \rangle$, $\langle \mathbf{new : a@h, a'@h} \rangle$, $\langle \mathbf{term : a@h} \rangle$, and $\langle \mathbf{rcv : a@h, m} \rangle$. This rules are the same as their actor counterparts in Figure 6. The augmentation of an actor a to $a@h$ and the expansion of the configuration to include η and B has no significance in the rules `fun`, `term` and `rcv`. In the rule `new`, the $@h$ is used to assert that a new actor will have the same host as its creator. Thus to an actor programmer `new` has the usual actor semantics, though in the BigActor model `new` where the $@h$ has the semantics of a host, the rule prevents the creation of remote bigActors. The environment E in Figure 8 is the same as the E in the actor model in all rules.

In addition to executing expressions by `fun`, `term`, `new` and `rcv` as an actor would, a BigActor can also execute the expressions $\mathtt{send(a'@h', m)}$, $\mathtt{observe(q)}$, $\mathtt{control(u)}$ and $\mathtt{migrate(h')}$ as syntactically formalized in Definition 4. The semantics of these expressions are unique to this model and given first by the rule $\langle \mathbf{req : a@h, r} \rangle$ and then by the rules labelled as `snd`, `obs`, `ctr` and `mgrt`, respectively. The `req` rule asserts that when any of the expressions $\mathtt{send(a'@h', m)}$, $\mathtt{observe(q)}$, $\mathtt{control(u)}$ or $\mathtt{migrate(h')}$ execute in a BigActor program, the program advances exactly as an actor program by the rule `fun` in Figure 6 but with a side-effect. The side-effect is the addition of the request $(a@h, r)$ to the set η. This creates a new configuration. When η contains a request with an expression $\mathtt{send(a'@h', m)}$, $\mathtt{observe(q)}$, $\mathtt{control(u)}$ or $\mathtt{migrate(h')}$ a BigActor model can further advance by application of the rules `snd`, `obs`, `ctr` or `mgrt` as described next.

To make the rest comprehensible we analyze an execution trace of the program in Figure 7 from Example 3.

The `control`, `observe` and `send` instructions have the syntax and semantics in Figure 8 and Definition 4. These instructions are encapsulated in two behaviors named the method `start` and method `rcvObs`. The syntax of the method construct and sequentiality are not formalized. We informally assume instructions execute in the order in which they appear in the text and the BigActor `app@sp` starts with the method `start`. The rest follows formally from the semantics in figure 8. Figure 9 is a formal execution of this program in the sense of definition 5. For the sake of space we abbreviate `children.parent.host` as `q`, `streeti` as `sti`, `server@srv` as `s@srv`, `app@sp` as `a@sp`, `CONNECT` as `CONN`, and `wlan0` as `w0`

All configurations in Figure 9 follow the notation in Definition 4, Definition 5, and two short-hands for brevity. The Figure has 9 lines. Each line has two configurations. Each line has two transitions except the last one which has only one. In each configuration the reduction context, denoted by R in the semantic rules, is abstracted with a .. The $app@sp$ program instruction executing at each step is explicitly typed in the square brackets in each configuration. The environment E in the semantic rules is also abstracted by . in the configurations in the first 3 lines and the first configuration in line 4. The environment is a valuation of the variables of $app@sp$ just as in the actor model. $app@sp$ has only one variable `B_obs`. This has no valuation in the first three lines and acquires a value in the second configuration in line 4. It is denoted with an E thereafter. B_0 through B_3 are bigraphs. These are visualized in Figure 10. This figure visualizes a projection of the trace in Figure 9. It projects away all transitions representing applications of `fun`, `term` and `req`. The `fun` and `term` transitions advance the program `app@sp` just like an actor program and do nothing particular to this model. The `req` transitions advance like `fun` but also have the side-effect of adding a request to the set η. Then other semantic rules can be applied to consume the elements in η as mentioned earlier to produce the configurations visualized in Figure 10.

The execution trace in Figure 9 can now be understood as follows. We assume the BigActor `app@sp` starts with the behavior `start` and advances to the first instruction by the rule `fun`. The next transition is by the `req` rule and puts $\mathtt{control(move(street2, street1))}$ in η. $\mathtt{MOVE(street2, street1)}$ is a concretization of the abstract BRR `MOVE` in Figure

1. $\langle [\cdot \vdash \cdot \sqsubset \texttt{start}() \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_0 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow} \langle [\cdot \vdash \cdot \sqsubset \texttt{control}(\texttt{MOVE}(\texttt{st2}, \texttt{st1})) \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_0 \rangle \overset{\langle \texttt{req:a@sp,MOVE}(\cdot,\cdot) \rangle}{\longrightarrow}$

2. $\langle [\cdot \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid (a@sp, \texttt{MOVE}(\texttt{st2}, \texttt{st1})) \mid B_0 \rangle \overset{\langle \texttt{ctr:a@sp,MOVE}(\cdot,\cdot) \rangle}{\longrightarrow} \langle [\cdot \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_1 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow}$

3. $\langle [\cdot \vdash \cdot \sqsubset \texttt{observe}(\texttt{q}) \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_1 \rangle \overset{\langle \texttt{req:a@sp,observe(q)} \rangle}{\longrightarrow} \langle [\cdot \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid (a@sp, \texttt{observe}(\texttt{q})) \mid B_1 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow}$

4. $\langle [\cdot \vdash \cdot \sqsubset \sqsupset]_a @h \mid \emptyset \mid (a@sp, \texttt{observe}(\texttt{q})) \mid B_1 \rangle \overset{\langle \texttt{term:a@sp} \rangle}{\longrightarrow} \langle (E \vdash \texttt{rcvObs})_a @h \mid \emptyset \mid (a@sp, \texttt{observe}(\texttt{q})) \mid B_1 \rangle \overset{\langle \texttt{obs:a@sp,q,B}_{\texttt{obs}} \rangle}{\longrightarrow}$

5. $\langle [E[FV(\texttt{rcvObs}) \mapsto B_{obs}] \vdash \texttt{rcvObs}]_a @h \mid \emptyset \mid \emptyset \mid B_1 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow} \langle [E \vdash \cdot \sqsubset \texttt{control}(\texttt{MOVE}(\texttt{st1}, \texttt{st3})) \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_1 \rangle \overset{\langle \texttt{req:a@sp,MOVE}(\cdot,\cdot) \rangle}{\longrightarrow}$

6. $\langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid (a@sp, \texttt{MOVE}(\texttt{st1}, \texttt{st3})) \mid B_1 \rangle \overset{\langle \texttt{ctr:a@sp,MOVE}(\cdot,\cdot) \rangle}{\longrightarrow} \langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_2 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow}$

7. $\langle [E \vdash \cdot \sqsubset \texttt{control}(\texttt{CONN}(\texttt{w0})) \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_2 \rangle \overset{\langle \texttt{req:a@sp,CONN.}(\cdot) \rangle}{\longrightarrow} \langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid (\texttt{a@sp}, \texttt{CONN}(\texttt{w0})) \mid B_2 \rangle \overset{\langle \texttt{ctr:a@sp,CONN}(\cdot) \rangle}{\longrightarrow}$

8. $\langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_3 \rangle \overset{\langle \texttt{fun:a@sp} \rangle}{\longrightarrow} \langle [E \vdash \cdot \sqsubset \texttt{send}(\texttt{s@srv}, \texttt{B\_obs}) \sqsupset]_a @h \mid \emptyset \mid \emptyset \mid B_3 \rangle \overset{\langle \texttt{req:a@sp,send}(\cdot) \rangle}{\longrightarrow}$

9. $\langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \emptyset \mid (a@sp, \texttt{send}(\texttt{s@srv}, \texttt{B\_obs})) \mid B_3 \rangle \overset{\langle \texttt{snd:a@h,m} \rangle}{\longrightarrow} \langle [E \vdash \cdot \sqsubset nil \sqsupset]_a @h \mid \langle \texttt{s@srv} \Leftarrow \texttt{B\_obs} \rangle \mid \emptyset \mid B_3 \rangle$

Fig. 9. Execution of BigActor program from Example 3.

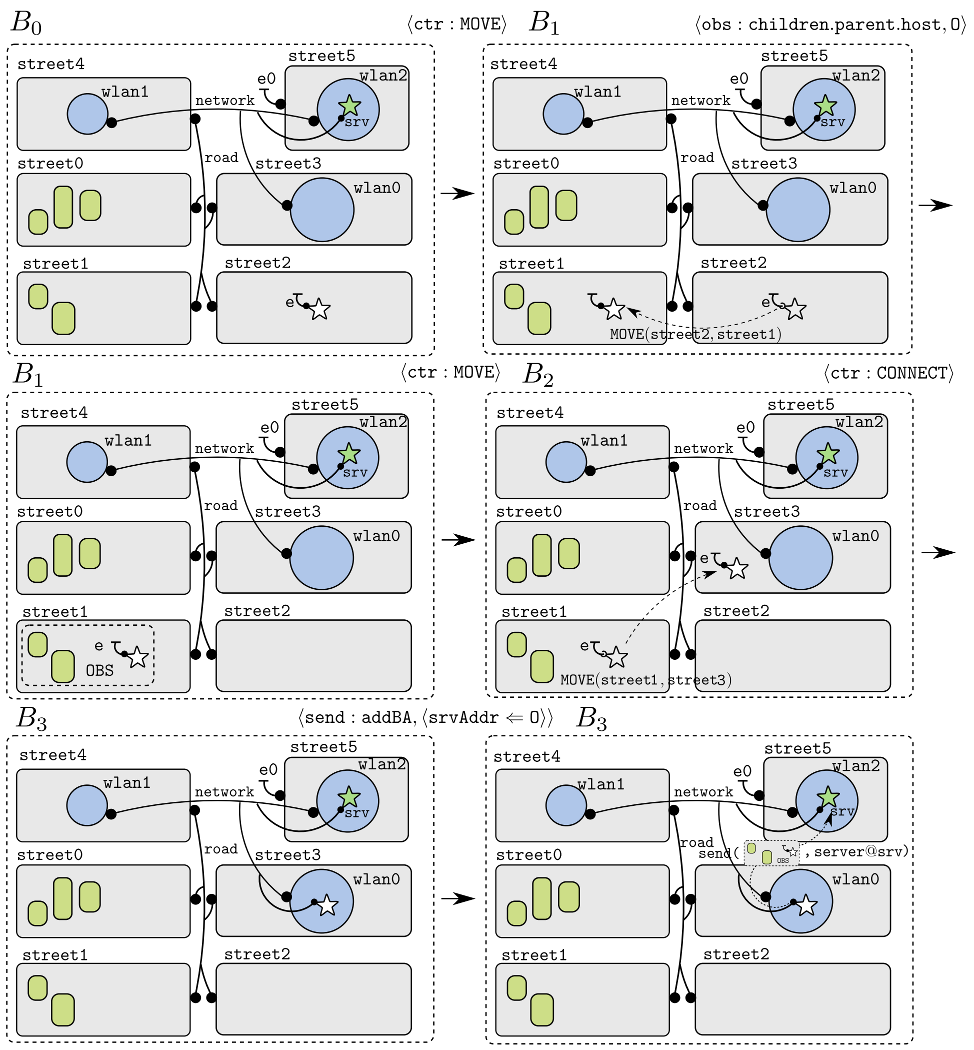


Fig. 10. A projection of the execution trace of the BigActor app@sp.

4. This enables use of the rule ctr to change the bigraph B_0 to B_1. This is visualized in the top row of Figure 10. One can see the white star move. Everything else in the two bigraphs is the same. The second transition in line 2 simply advances

the actor program by rule fun to the instruction observe(children.parent.host). The first transition in line 3 puts this imperative into η by the rule req. The first behavior is now exhausted and the actor become inactive by application of the rule fun and then term resulting in the two configurations in line 4. In the second configuration in line 4 the BigActor app@sp has advanced to the second behavior called rcvObs in Figure 7.

The observe(q) imperative is still present in η which allows us to apply the rule obs taking us to the first configuration in line 5. The query q in the program is children. parent.host and is interpreted on bigraph B_1 with respect to sp. This transition is visualized in Figure 10 as the transition from the top row to the middle one. Since the host node of app is sp denoted by the white star, its parent is street1 and the children are the green nodes and sp itself. The interpretation of queries is formalized in Section V. The application of rule obs produces the bigraph enclosed by the dashed region in the left-hand figure of the middle row of Figure 10. The transmission of the observed bigraph to the BigActor in the rule obs is handled much like an actor receives a message. The inactive actor become busy and the observation is bound to one of its free variables, FV(rcvObs) in line 5 of Figure 9. The behavior has only one free variable named B_obs in Figure 7. The environment E is now the observed bigraph being the valuation of B_obs. η once again becomes empty. The rest of line 5, lines 6 and 7 in Figure 9 are derived like the prior lines. The rule CONNECT(wlan0) is a concretization of the abstract BRR in Figure 4.

Lines 8 and 9 illustrate the semantics of the BigActor send. In the second configuration in line 8 the BigActor is busy at the expression send(server@srv, B_obs). Like the actor model server@srv is intended to denote another BigActor supposed to receive the message. The message is the value of B_obs. The actor semantics would put this message in μ. We put the expression in η just like the other requests and then apply the snd rule to complete message delivery. We do this because the premises of the snd rule requires a link in the bigraph between the hosts of the sending and receiving bigActors. If the existence of this link can be proven, the message is transferred from η to μ. It can in this case due to prior execution of the expression connect(wlan0) as illustrated by the transition connecting the middle and last rows of Figure 10, It would then be received by the other actor by application of the rule rcv exactly as in the actor model.

The BigActor rule snd is the counterpart of the actor rule with the same name, and the BigActor expression send(a′@h′, m) the counterpart of the actor send(a′, m). However, the semantics is different since it requires the existence of a link between two hosts in the bigraph. This means a BigActor sending a message to a correct mailbox address could fail to communicate. This is a violation of actor semantics but not of BigActor semantics. A good analogy might be accessing a webpage in the internet, it is not sufficient to know the URL, one must also be connected to the internet by some means (e.g. wireless connection, 3G, etc.).

The example does not illustrate the rule mgrt for the expression migrate. This rule behaves much like the others. A BigActor can request a migration by the expression migrate(h′) where h' is a node in the bigraph. If the host of the BigActor and h' are connected by a link in the bigraph, the request can be consumed by the rule mgrt and the host of the BigActor will be changed. Note the bigraph remains unchanged. There is only a change in the hosting relationship between the actors and the bigraph. Once again, we are thinking of the link as the physical network required for the flow of a nomadic program.

We conclude this section with a few remarks.

Remark 1. The program in Figure 7 is precise in the BigActor model. We see the conciseness of the program as a contribution of the model.

Remark 2. We see the relationship between BigActor and bigraph like the relationship between a program and its machine. Actor programs are concurrent programs and so the machine can be a distributed one. The actors model the programs and the bigraph a shared distributed machine. The programs can observe the machine (rule obs) and change the machine (rule ctr). Control can move, add, or remove nodes and do the same with connections (links). The hosting relationship is the distribution of the program over the machine. The programs can change their distribution (rule mgrt). The programs can exchange messages (rules snd, rcv) but only up to the machine (bigraph links). The programs can compute (rules fun, term) and create new programs (rule new) and locate them in the machine (locally). They could migrate thereafter.

Remark 3. In this semantics the coupling of BigActor and bigraph is asynchronous. Program execution puts requests in η. The requests affect the bigraph, hosting relationship, or observations at a later instant of model time. This means a BigActor can make requests executable in the bigraph at request time but unexecutable at the later time. Section VI addresses this issue.

Remark 4. Control is local. If a BigActor seeks to replace component R of a bigraph with R' then the rule ctr requires the BigActor to be located (hosted) in R. We show in Section V that observation is also local.

Remark 5. In the actor model state is local. The memory of an actor influences another only up to the messages flowing between them. In the BigActor model the Bigraph is a shared structure, and some might argue this globalizes state. For example, one may communicate between bigActors by placing and removing nodes in the bigraph and never send any inter-actor messages at all. We would consider this misuse of the model, but have nevertheless elected to provide the bigraph as a shared structure to keep the model expressive. The BigActor formalism is a model for actors operating in the physical world, which may in many applications be a shared space to the actors.

Thus the bigraph can be used as a back-channel for communication between actors. There are two ways to close this channel. The first is rather trivial. One may simply make the bigraph into an actor. Then all `req`, `obs`, `ctr`, and `mgrt` actions would appear as the flow of messages between actors and the entire physical world would become the local state of one actor. This solves the problem but does not really address any of the fundamental difficulties of coordinating observing and changing a shared physical machine. Therefore we have presented a semantics in which the bigraph is not an actor, because by making the coordination difficulties visible in the model, the programmer gets the opportunity to program for them.

However, the model does support a sufficient condition, derived from its semantics closing the bigraph as a backchannel for communication. Basically, one requires two bigActors never control and observe the same region of the bigraph. The semantics of a BigActor permits us to derive the area of influence of a BigActor as expressed in Definition 6.

Definition 6. Let $a@h$ be a BigActor and $\mathcal{R}_a = \{R_0 \to R'_0, R_1 \to R'_1, \ldots, R_n \to R'_n\}$ the set of all concrete BRRs used by it. The **area of influence** of $a@h$ over a bigraph B is the set of nodes A such that $v \in A \Rightarrow \exists i.B = C \circ R_i \circ d \wedge v, h \in V_{R_i}$.

Example 4. Consider Example 3 where the host of the app BigActor is inside `street2` (bigraph B_0 of Figure 10). The area of influence under the current bigraph is:

$$A^{appBA} = \{\texttt{street0}, \texttt{street1}, \texttt{street2}, \texttt{street3}, \texttt{street4}\}$$

Note that due to the current location of the host the only rule that can be applied is `MOVE`.

Let $c_0 \to c_1 \to \ldots$ be an execution trace where $c_i = \langle \alpha_i \mid \mu_i \mid \eta_i \mid B_i \rangle$ is a BigActor configuration and $\to$ is given by the BigActor semantics. Let $A_i^{a@h}$ be the area of influence of $a@h$ over the bigraph B_i and $B_{obs_j}^{a@h}$ be an observation of $a@h$ over the bigraph B_j. If for any two bigActors $a@h$ and $a'@h'$, $A_i^{a@h}$ and $V_{B_{obs_j}^{a'@h'}}$ never overlap for any time $i < j$ then the bigActors cannot use the bigraph as a back channel for communication, e.g. $a@h$ might place a node in the bigraph but $a'@h'$ would not observe it because the sets $A_i^{a@h}$ and $V_{B_{obs_j}^{a'@h'}}$ would be disjoint.

V. Query language and observations

In this section introduce a query language for querying the bigraph for local observations.

The query language syntax is specified by the following grammar:

```
query ::= node|children.node|linkedTo.node
node  ::= host|parent.node
```

Observations are local with respect to the host node of the BigActor that is querying. We think of “local” in terms of the placing graph (parents and children of a given node) and in terms of the linking graph (nodes linked to a given node). Our intention is to introduce a query language that is expressive enough for the examples in the paper and that entails the property of providing observations that are local with respect to a given host. Our aim is not to introduce a general purpose query language for Bigraphs.

The interpretation of a query q over a bigraph B with respect to a host h is defined such that a given observation $B_{obs} = [\![q]\!]_h^B$ is composable with the remaining context of B, i.e. there exists a context C and parameters d such that $B = C \circ B_{obs} \circ d$.

Example 5. Figure 11 presents an example of querying the bigraph B_0 of Figure 10 with respect to sp. On the right-hand side of Figure 11 one can see the interpretation of the queries `host`, `parent.host`, `children.parent.parent.host`.

Let B be given by $(V_B, E_B, ctrl_B, prnt_B, link_B) : \langle m_B, X_B \rangle$
$\to \langle n_B, Y_B \rangle$. The interpretation of the query language goes as follows.

$$[\![\texttt{host}]\!]_h^B \mapsto (\{h\}, \emptyset, ctrl_h, prnt_h, link_h) : \langle 1, \emptyset \rangle \to \langle 1, Y_h \rangle$$

where $ctrl_h(h) = ctrl_B(h)$,

$$prnt_h(v) = \begin{cases} 0 & \text{if } v = h \\ h & \text{if } v = 0 \end{cases}$$

$$link_h(pt) = \begin{cases} link_B(pt) & \text{if } link_B(pt) \in Y_B \\ y_e & \text{if } e = link_B(pt) \in E_B \end{cases}$$

$$\begin{aligned} Y_h = &\{y_e \mid \forall pt.e = link_h(pt) \wedge e \in E_B\} \uplus \\ &\{y \mid \forall pt.y = link_h(pt) \wedge y \in Y_B\} \end{aligned}$$

Edges are abstracted by unique names to make the composition $B = C \circ B_{obs} \circ d$ valid. The edges are included in the context and are also connected to the same names. Composition would bind the names in the context and in the observation restoring the edges. Also note that the contents of node h are abstracted with a hole 0.

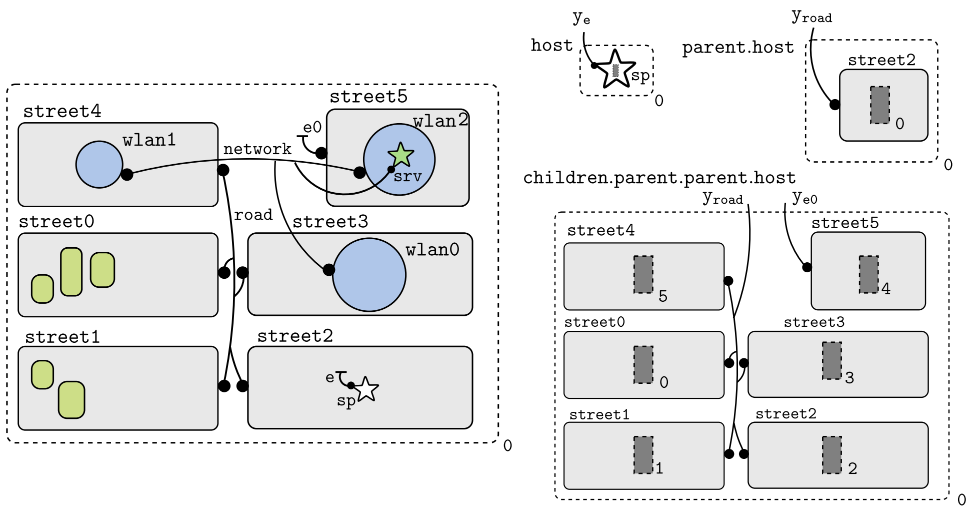


Fig. 11. Interpretation of several queries over bigraph B_0 with respect to host sp.

Let the interpretation of $[\![\texttt{node}]\!]_h^B$ be denoted as the bigraph B_n. Note that B_n only has one node which we denote as n, i.e. $V_{B_n} = \{n\}$. The interpretation if parent.node is given by:

$$[\![\texttt{parent.node}]\!]_h^B \mapsto (\{p\}, \emptyset, ctrl_p, prnt_p, link_p) : \langle 1, \emptyset \rangle \to \langle 1, Y \rangle$$

where $p = prnt_B(n)$. The remainder of the bigraph definition is provided by replacing p for h in the definition of $[\![\texttt{host}]\!]$.

The interpretation of children.node is given by:

$$[\![\texttt{children.node}]\!]_h^B \mapsto (V_C, \emptyset, ctrl_C, prnt_C, link_C) : \langle | V_C |, \emptyset \rangle \to \langle 1, Y_C \rangle$$

where $V_C = \{v \mid v \in V_B,\, n = prnt_B(v)\}$ and

$$prnt_C(v) = \begin{cases} 0 & \text{if } v \in V_C \\ v' & \text{if } v = holeOf(v') \end{cases}$$

$holeOf : V_C \to \mathbb{N}$ is a function that assigns to each children node a unique hole to abstract its contents. $ctrl_C$, $link_C$, and the set of external names Y_C are given as before although now ranging over the set of children nodes and respective ports.

Finally, the interpretation of linkedTo is given by:

$$[\![\texttt{linkedTo.node}]\!]_h^B \mapsto (V_L, \emptyset, ctrl_L, prnt_L, link_L) : \langle | V_L |, \emptyset \rangle \to \langle | V_L |, Y \rangle$$

where $V_L = \{v \mid \exists pt \in ports(v). \exists pt' \in ports(n). link_B(pt) = link_B(pt')\}$,

$$prnt_h(v) = \begin{cases} regionOf(v) & \text{if } v \in V_L \\ v' & \text{if } v = holeOf(v') \end{cases}$$

Note that for each node in V_L we create a unique region using the function $regionOf : V_L \to \mathbb{N}$. By putting each node in a unique region we can restore the parenthood by composing again with the context. The remainder of the bigraph definition is as per the prior cases.

In this paper we use the query language query to define bigraphs which are local under a given bigraph and with respect to a host node.

Definition 7. Given two bigraphs B and B' and a host node h. B' is *local* with respect to h if there exists a q such that $[\![\texttt{q.h}]\!]_h^B = B'$ or $[\![\texttt{q.h}]\!]_h^B = C \circ B' \circ d$ for an arbitrary context C and parameters d. In other words, B' is local with respect to h if one can find a query with an interpretation containing B'.

An observation B_{obs} obtained by a BigActor $a@h$ querying a bigraph B using the local query language is, by Definition 7, a local bigraph over B with respect to h ($[\![\texttt{q.h}]\!]_h^B = C \circ B_{obs} \circ d$ where C and d are empty).

VI. Correctness

In this section we investigate the correctness of bigActors execution. We address correctness from two orthogonal perspectives:

- ensure that a bigActor by itself does not requests unexecutable commands, and
- ensure that concurrency does not make an executable request at request time becoming unexecutable

We start by formally defining what we mean by a correct execution.

Definition 8. Let $c = \langle \alpha \mid \mu \mid \eta \mid B \rangle$ be a BigActor configuration. c is **terminal** if $\mu = \emptyset$, $\eta = \emptyset$, and all bigActors in α are inactive.

Definition 9. Let $c = \langle \alpha \mid \mu \mid \eta \mid B \rangle$ be a BigActor configuration. c is **correct** if $\exists \lambda . c \xrightarrow{\lambda} c'$ or c is terminal.

Definition 10. $c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_{n-1}} c_n$ is a **correct execution** if c_i is correct for all $i \in \{0, 1, \dots, n\}$.

In order to address correctness problems we must define a scheduling discipline for serving requests. We choose a First-Come First-Served scheduling discipline.

Let Λ denote the set of labels as per the semantics stated in Figure 8. Let Λ_η denote the set of request labels and Λ_π denote the set of obs, ctr, mgrt, and snd labels. Consider a function $schd_t : \Lambda_\pi \to \Lambda_\eta$ where, for a given execution $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \dots$, $schd_t(\lambda_\pi)$ returns the request served by λ_π. Let $\prec_t \subseteq \Lambda \times \Lambda$ be a total order induced by the index i of λ_i in t.

Definition 11. An execution $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \dots$ follows a **First-Come First-Served discipline** (FCFS) if:

$$\lambda_i \prec_t \lambda_j \Rightarrow schd(\lambda_i) \prec_t schd(\lambda_j)$$

where $\lambda_i, \lambda_j \in \Lambda_\pi$.

In other words, FCFS discipline requests that requests are served in the order that are requested.

A. *Correctness of single BigActor executions*

In this section we introduce a sufficient condition for correctness of single bigactor executions.

This condition is named **feedback**. A bigActor is feedback if it only requests commands that are coherent with its last observation of the bigraph. Moreover, a feedback bigActor always observe the bigraph before a command (and in between two commands). Thus, the name *feedback*.

Definition 12. Let $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \dots$ be a bigActor execution. For each λ_i the operational semantics identifies a unique actor $\mathtt{a@h}$. For clarity we augment λ_i to $\lambda_i^{a@h}$. $\mathtt{a@h}$ is called **feedback** for all $\mathtt{r}$ and $\mathtt{r}'$ that are either `control`, `migrate`, or `send` the following are true:

1) $\forall j . \lambda_j = \langle \mathtt{req} : \mathtt{a@h}, \mathtt{r} \rangle \Rightarrow \exists i < j . \lambda_i \in \{\langle \mathtt{obs} : \mathtt{a@h}, \mathtt{q}, \mathtt{B_{obs}} \rangle, \langle \mathtt{rcv} : \mathtt{a@h}, \langle \mathtt{a@h} \Leftarrow \mathtt{B_{obs}} \rangle \rangle\}$
2) $\forall i . \forall k > i . \lambda_i = \langle \mathtt{req} : \mathtt{a@h}, \mathtt{r} \rangle \wedge \lambda_k = \langle \mathtt{req} : \mathtt{a@h}, \mathtt{r}' \rangle \Rightarrow \exists j . i < j < k \wedge \lambda_j = \{\langle \mathtt{obs} : \mathtt{a@h}, \mathtt{q}, \mathtt{B_{obs}} \rangle, \langle \mathtt{rcv} : \mathtt{a@h}, \langle \mathtt{a@h} \Leftarrow \mathtt{B_{obs}} \rangle \rangle\}$
3) For any λ_i let $B_{obs}^{a@h,\lambda_i}$ denote the last observation of $a@h$ preceding λ_i. $\forall i . \lambda_i = \langle \mathtt{req} : \mathtt{a@h}, \mathtt{r} \rangle \Rightarrow \exists \mu . \exists B . \langle \alpha \mid \emptyset \mid \{r\} \mid B_{obs}^{a@h,\lambda_i} \rangle \xrightarrow{(a@h,\mathtt{r})} \langle \alpha \mid \mu \mid \emptyset \mid B \rangle$

The first requirement states that before any request $\mathtt{r}$ by $\mathtt{a@h}$ that is not an observation, there must be an observation by $\mathtt{a@h}$ (which can be obtained by either receiving an observation from a scheduled query or receiving a message from another actor with the observation as message value). The second requirement states that between two requests $\mathtt{r}$ and $\mathtt{r}'$ by $a@h$ that are not observations there must be an observation by $\mathtt{a@h}$. The last requirement states that, given a configuration with $B_{obs}^{a@h,\lambda_i}$ as the bigraph and one request $\mathtt{r}$ then there is a new configuration obtained by consuming $\mathtt{r}$. This means that $\mathtt{r}$ does not make a configuration with the observed bigraph incorrect.

Next we prove that *feedback* is a sufficient condition for correctness of single bigActors executions.

Theorem 1. *Consider a BigActor execution $t = c_0 \xrightarrow{\lambda_0} c_1 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_n} c_n$ that follows a FCFS discipline (Assumption 1.1). Assume that α_i contains a single BigActor* $\mathtt{a@h}$ *for all i and* $\mathtt{a@h}$ *is a feedback BigActor (Assumption 1.2). Assume that η and μ are empty at c_0 (Assumption 1.3). Then c_n is correct and t is a feedback semantics.*

Proof: By mathematical induction in the the number of transitions.

Base case Given $c_0 \xrightarrow{\lambda_0} c_1$, c_1 is correct.

By Assumption 1.3 $a@h$ must be busy otherwise c_0 is terminal (Definition 8) and $\xrightarrow{\lambda}$ can not be triggered. If $a@h$ is busy then it can trigger any of the following rules leading to c_1: `fun`, `new`, `term`, and `req`. Since the overall execution is assumed to have only one bigActor `new` is excluded. By Assumption 1.3, the rules `ctr`, `mgrt`, `obs`, `snd` can not be executed since η is still empty. `fun`, and `term` are internal to the actor and can not generate a transition to an incorrect configuration. By

Definition 12 item 1, the rule `req` can only request an observation because a control, migrate, or a send request require previous observations.

Inductive step Give $c_{i-1} \overset{\lambda_{i-1}}{\rightarrow} c_i$ where c_i is correct then $c_i \overset{\lambda_i}{\rightarrow} c_{i+1}$ and c_{i+1} is correct. Since c_i is correct then λ_i can be given by one of the following rules: `fun`, `rcv`, `term`, `req`, `ctr`, `mgrt`, `obs`, and `snd` (`new` is excluded by Assumption 2).

By the semantics definition of Figure 8, the rules `fun`, `rcv`, and `term` are internal to the bigActor and do not produce nor consume requests that can put the configuration in an incorrect state. The rule `req` can produce a new request, i.e. $\eta_{i+1} = \eta_i \cup \{r\}$. If r corresponds to an observation then c_{i+1} is correct since the observations do not change the bigraph. If r is either a control, a migration or a send request then it must produce a correct configuration under $B^{a@h,r}_{obs}$. By Definition 12, $a@h$ alternates between observation and control/migrate/send then the bigraph B_i at configuration c_i can be decomposed as $B_i = C \circ B^{a@h,r}_{obs} \circ d$ where $B^{a@h,r}_{obs}$ is the last observation prior to r. By Definition 12 item 3, r can by applied to $B^{a@h,r}_{obs}$. Since $B_i = C \circ B^{a@h,r}_{obs} \circ d$ r also produces a correct configuration under B_i. The rules `ctr`, `mgrt`, and `obs` consume one request r out of η_i. By Assumption 1.1, all requests of $a@h$ are served in order. Since all requests in η_i were generated by a (single) feedback BigActor, then $\eta_{i+1} = \eta_i \setminus \{r\}$ do not make c_{i+1} incorrect. ■

B. *Correctness of concurrent BigActor execution*

In the multiple bigActor case, the feedback property is not a sufficient condition for correctness. A bigActor can request an command executable at request time by the operational semantics, but not executable when the request is served. This is a consequence of the fact that in the bigActor operational semantics (Figure 8) the time which a request is generated is necessarily different from the time which the request is served. We take two strategies to avoid these concurrency issues and ensure a correct execution of multiple bigActors: prevent undesired interleaving (time strategy); and/or prevent shared resources (spatial strategy). These lead to theorems 2 and 3.

The first strategy is called *atomic read-write semantics.*

1) Atomic read-write semantics:

Definition 13. Let Λ_θ denote the set of all `ctr`, `mgrt`, and `snd` labels and let $\Lambda^{a@h}_\theta$ be the subset of labels from Λ_θ produced by serving requests from $a@h$. An execution $t = c_0 \overset{\lambda_0}{\rightarrow} c_1 \overset{\lambda_1}{\rightarrow} \ldots$ follows an **atomic read-write semantics** if for any $\lambda_i = \langle \texttt{obs} : \texttt{a@h}, \cdot\rangle$ for which there exists a $\lambda_{i+k} \in \Lambda^{a@h}_\theta$ then $\lambda_{i+j} \notin \Lambda^{a'@h'}_\theta, j \in \{1, \ldots, \hat{k}-1\}$ where $\hat{k} = \min\{k | \lambda_{i+k} \in \Lambda^{a@h}_\theta\}$ and $a@h \neq a'@h'$.

In other words, an atomic read-write semantics requires that between any `obs` from $a@h$ and subsequent `ctr`, `mgrt`, or `snd` from the same bigActor, no other `ctr`, `mgrt`, or `snd` can occur from another bigActor $a'@h'$. This is also known as a blocking semantics.

Theorem 2. *Consider a BigActor execution* $t = c_0 \overset{\lambda_0}{\rightarrow} c_1 \overset{\lambda_1}{\rightarrow} \ldots \overset{\lambda_n}{\rightarrow} c_n$ *that follows a FCFS discipline (Assumption 2.1) and the execution follows an atomic read-write semantics (Assumption 2.2). Assume that all bigActors are feedback (Assumption 2.3). Assume that* η *and* μ *are empty at* c_0 *(Assumption 2.4). Then* c_n *is correct and* t *is the semantics of a feedback loop.*

Proof: By Assumption 2.3 all bigActors are feedback. Since the feedback property is only sufficient for correct executions of single bigActor configurations we can, without loss of generality, assume that the configurations at execution t contains only two bigActors $a@h$ and $a'@h'$. By Assumption 2.1 and Assumption 2.2 the execution has no interleaving between observations and `ctr`/`mgrt`/`send` of two different bigActors. Thus, one can partition the sequence t into subsequences $t^{a@h}_0$, $t^{a'@h'}_0$, $t^{a@h}_1$, $t^{a'@h'}_1 \ldots$ where each $t^{a@h}_k$ denotes the execution of a single BigActor $a@h$ (and possibly labels regarding rules `fun`, `term`, `new` and `rcv` which are internal to bigActors and thus do not make the configuration incorrect). Since each bigActor $a@h$ is feedback, by Theorem 1, each $t^{a@h}_k = c \rightarrow \ldots \rightarrow c'$, c' is correct. Thus, c_n is also correct. ■

The second approach is called *partitioning semantics.*

2) Partitioning semantics:

Definition 14. Consider an execution $t = c_0 \overset{\lambda_0}{\rightarrow} c_1 \overset{\lambda_1}{\rightarrow} \ldots$. Let $A^{a@h}_i$ denote the area of influence of $a@h$ at c_i. An execution follows a **partitioning semantics** if for every two bigActors $a@h$ and $a'@h'$

1) $\forall i.\forall j.A^{a@h}_i \cap A^{a'@h'}_j = \emptyset$
2) $\forall i.\forall j.\forall v \in A^{a@h}_i.\forall v' \in A^{a'@h'}_j.Lks(v) \cap Lks(v') = \emptyset$

where $Lks : V_B \rightarrow \mathcal{P}(E_B \uplus Y_B)$ is a function that returns the set of links connected to ports of a given node.

In other words, partitioning semantics requires the areas of influence of any two bigActors and their links be disjoint at every time. Partitioning semantics is non-blocking since it does not require the overall bigraph to be inaccessible between an observation and an action of a single bigActor.

Theorem 3. *Consider a BigActor execution* $t = c_0 \overset{\lambda_0}{\rightarrow} c_1 \overset{\lambda_1}{\rightarrow} \ldots \overset{\lambda_n}{\rightarrow} c_n$ *that follows a FCFS discipline (Assumption 3.1) and the execution follows a partitioning semantics (Assumption 3.2). Assume that all bigActors are feedback (Assumption 3.3). Assume that* η *and* μ *are empty at* c_0 *(Assumption 3.4). Then* c_n *is correct and* t *is the semantics of a feedback loop.*

Proof: Without loss of generality assume the execution of two bigActors $a@h$ and $a'@h'$. By Definition 14, the areas of influence and corresponding links are disjoint throughout the overall trace $t = c_0 \overset{\lambda_0}{\to} c_1 \overset{\lambda_1}{\to} \ldots \overset{\lambda_{n-1}}{\to} c_n$ then for each B_i at configuration c_i can be decomposed into $B_i = B_i^{a@h} \circ B_i^{a'@h'}$ where $B_i^{a@h}$ and $B_i^{a'@h'}$ are in two different bigraph regions without sharing any edges nor names. Thus, each configuration $c_i = \langle \{a@h, a'@h'\} \mid \mu_i \mid \eta_i \mid B_i \rangle$ can be decomposed into two configurations $c_i^{a@h} = \langle \{a@h\} \mid \mu_i^{a@h} \mid \eta_i^{a@h} \mid B_i^{a@h} \rangle$ and $c_i^{a'@h'} = \langle \{a'@h'\} \mid \mu_i^{a'@h'} \mid \eta_i^{a'@h'} \mid B_i^{a'@h'} \rangle$ where $\mu_i = \mu_i^{a@h} \cup \mu_i^{a'@h'}$ and $\eta_i = \eta_i^{a@h} \cup \eta_i^{a'@h'}$. Since $a@h$ and $a'@h'$ are feedback, then by Theorem 1, there exists two traces where $c_0^{a@h} \to \ldots \to c_n^{a@h}$ and $c_0^{a'@h'} \to \ldots \to c_n^{a'@h'}$ where $c_n^{a@h}$ and $c_n^{a'@h'}$ are correct. Thus, c_n is correct. ■

VII. Conclusions

In this paper we introduce the BigActor model for modelling structure-aware computation. The model combines the Actor model [1] and the Bigraph model [12]. We introduce an operational semantics for the BigActor model as an augmentation of the Actor model semantics. The semantics is enriched with three rules that provide means for bigActors to migrate and to observe and control the bigraph. We introduce a query language and introduce means for preventing concurrent issues that arise due to the fact that bigActors operate in a shared bigraph.

BigActor model provide means for defining agents that observe, control, and migrate over a structure of the world modelled as a bigraph. The BigActor model also introduces a model of reflection of the structure in for actor system. BigActors can also be used to embedded actors into bigraphs.

As per the future work, we are aiming at developing a theory of controllability and observability for bigActors in a control theory stand-point. We are also interested in implementing a prototype for a bigActor domain specific language for autonomous vehicles.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] J. Armstrong, R. Virding, C. Wikstr, M. Williams, et al. Concurrent programming in erlang. 1996.

[4] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, and H. Niss. Bigraphical models of context-aware systems. In *Foundations of software science and computation structures*, pages 187–201. Springer, 2006.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In *Embedded Software*, pages 84–99. Springer, 2003.

[6] S. Debois. Computation in the informatic jungle. *To appear. Draft available at http://www.itu.dk/people/debois/pubs/computation.pdf*, 2010.

[7] S. Edwards and E. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48(1):21–42, 2003.

[8] J. Epstein, A. Black, and S. Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129. ACM, 2011.

[9] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, 2009.

[10] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 2002.

[11] R. Milner. Bigraphical reactive systems. *CONCUR 2001 - Concurrency Theory*, pages 16–35, 2001.

[12] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[13] B. Nielsen and G. Agha. Semantics for an actor-based real-time language. *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, pages 223–228, 1996.

[14] B. Nielsen, S. Ren, and G. Agha. Specification of real-time interaction constraints. *Proc. of First Int. Symposium on Object-Oriented Real-Time Computing, IEEE Computer Society*, 1998.

[15] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, Jan. 1987.

[16] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan. 1989.

[17] G. Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. The MIT press, 1999.