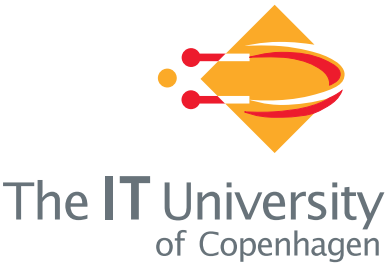


Bigraphs by Example

Søren Debois
Troels Christoffer Damgaard

This work was funded in part by the Danish Research Agency (grant no.: 2059-03-0031) and the IT University of Copenhagen (the LaCoMoCo project).

IT University Technical Report Series **TR-2005-61**

ISSN 1600–6100 **10 2005**

ISSN 1600–6100

ISBN 87-7949-090-5

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7,
DK-2300 København S
Denmark

Telephone: **+45 72 18 50 00**
Telefax: **+45 72 18 50 01**
Web `www.itu.dk`

Bigraphs by Example

Søren Debois
Troels Christoffer Damgaard

Abstract

To gain familiarity with bigraphs and to investigate their modeling capabilities, we model a switch, finite automata, the game of “life”, combinatory logic, term unification and an event-driven system as bigraphical reactive systems.

Contents

Chapter 1

Introduction

Bigraphs and bigraphical reactive systems [8, 9] are new theoretical tools that emphasize interplay between physical locality and virtual connectivity. Bigraphs have roots in process calculi, in particular π-calculus [12] and the Ambient Calculus [3]. Constructed using principles of [11], bigraphs are composible and has a bisimulation that is a congruence wrt. composition.

Both π-calculus and Petri-nets have been encoded in bigraphs [9, 13]. Work on the Lambda-calculus and the Ambient Calculus is underway [15]. There is an axiomatization of static bigraphs [14]; axiomatization of binding bigraphs was recently explored in [5]. Work on constructing a logic for static bigraphs is also underway [4].

As practical modeling tools, however, bigraphs remain virtually uncharted. In this document, we give six examples of bigraphical models, thus taking the first step in investigating bigraphs as modeling tools. Our examples serves exclusively to show what models might look like; we have made little efforts to prove theorems about our models. Incidentally, the examples might function as a companion introduction to bigraphical reactive systems.

We assume a basic familiarity with bigraphs; introductions can be found within [13, 8, 9].

Acknowledgements We would like to thank Arne Glenstrup, Martin Elsman and the rest of the IT University of Copenhagens BPL group for useful discussions.

Notational Peculiarities We allow ourself to depart slightly from the standard notation of [13, 8, 9]. We write the control of atomic nodes *within* that node. The two bigraphs below are identical.

It is customary to enclose each root of a bigraph in a dashed box. To conserve space, we draw prime bigraphs *without* such a box. The two bigraphs below are identical.

Chapter 2

Pure Bigraphs

In this chapter we model systems in pure bigraphs exclusively.

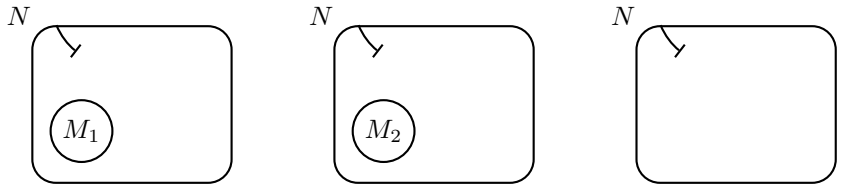
2.1 A Switch

In computer networks, a *switch* [16] is a device that multiplexes packets between physical networks. One way a switch can figure out where incoming packets should go is by noting the source address in such packets. Whenever the switch sees a packet with source address a on a network n, it registers internally the address a with the network n; subsequent packets to a are then transmitted on n. Packets to unknown addresses are broadcast, i.e., transmitted on all networks. In PC/LAN networking, switches implemented this way are also known as "learning bridges".

We model a learning bridge as a bigraphical reactive system. First, we sketch the dynamics of the system for a 3-way learning bridge. Then we give suitable reaction rules.

We use controls N (networks) of arity 1, M (machines) of arity 2 and P (packets) also of arity 2. The single port of a network N links N to the machines that are registered as located within N. The two ports of a packet P link the packet to its source and target machine, respectively. Conversely, the two ports of a machine M link the machine to its registered location, if any; and all packets referencing the machine [1], respectively.

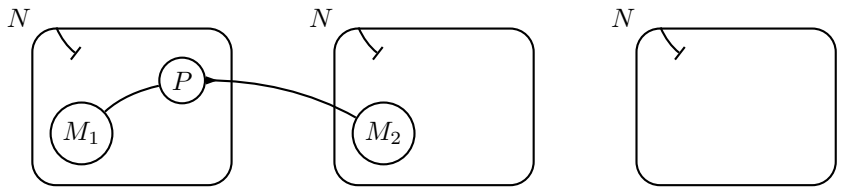
Our switch connects three networks, initially with a machine in two of the networks. (We have subscripted numbers on the M-controls to avoid repeatedly talking about the "left-most" or "right-most" machine.)



The switch has not yet associated any machines with any networks; we emphasize this by explicitly linking each network to a singleton edge. (In principle, the ports of each machine should be similarly connected to singleton edges. However, that would clutter the pictures unnecessarily.)

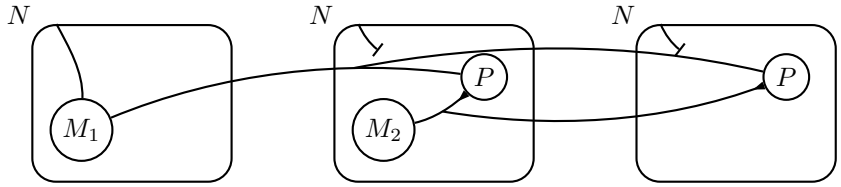
The machine M_1 transmits a packet to M_2.

[1] It is not necessary to separate links between machine and network, and links between machine and packets. We maintain the distinction solely to keep the model intuitive, since we *think* of these two types of connections as being distinct.

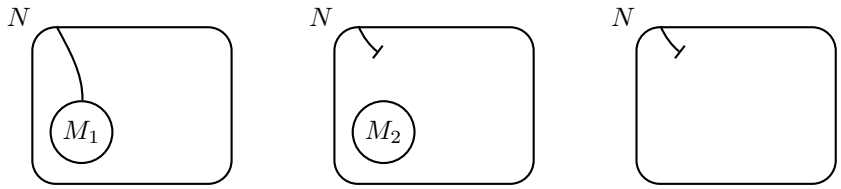


We distinguish the source and target ports of the packet by putting an arrow on the target port. (We ought to distinguish similarly between the two ports of a machine. However, in that case, no confusion is possible.)

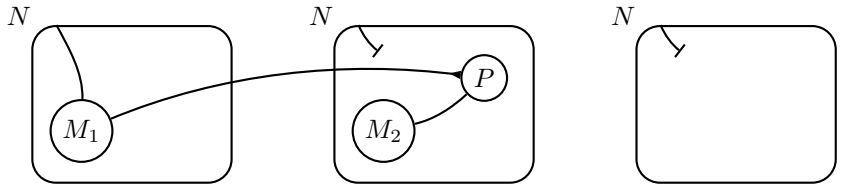
Not knowing where to put the packet, the switch broadcasts it. Also, the switch links the machine M_1 to the left-most network, thus recording the location of M_1.



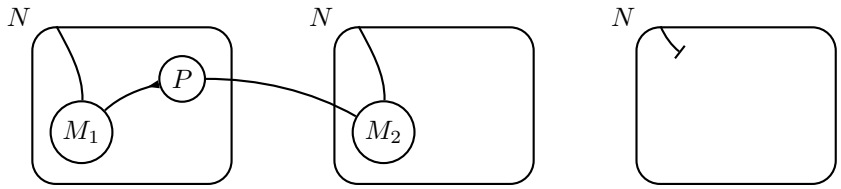
The machine M_2 receives the packet. The copy of the packet on the third network disappears, unreceived.



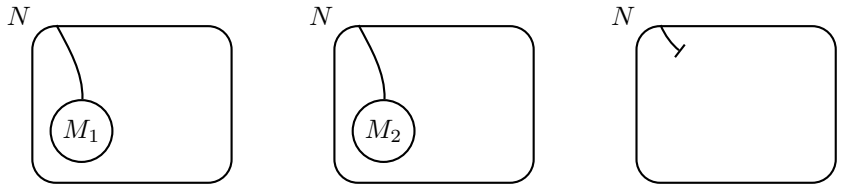
A response is sent.



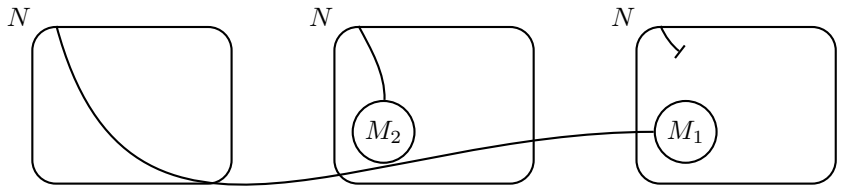
This time, the switch knows the location of the target machine M_1 and transmits the packet accordingly. Also, the location of M_2 machine is recorded.



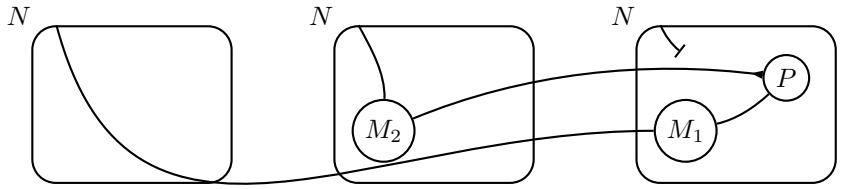
The packet is received.



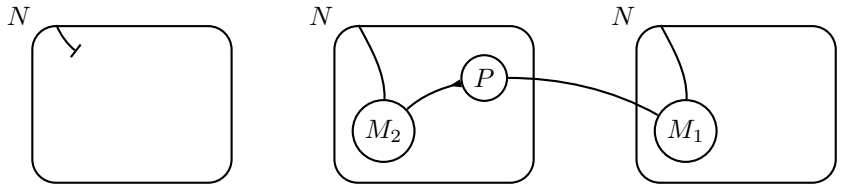
Suppose that M_1 is plugged out, moved to some other office and plugged back in. The recorded location of M_1 is now wrong.



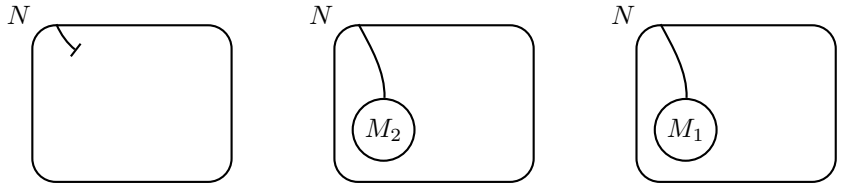
(Obviously, transmissions from M_2 to M_1 will now be lost. A real learning-bridge resets its switching tables every 100 milliseconds or so, thus limiting the damage.) A packet is sent from M_1.



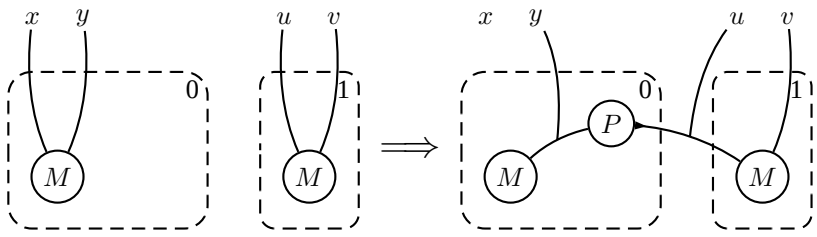
The packet is transmitted to the known location of M_2. In the process, the location of M_1 is updated.



Finally, the packet is received.



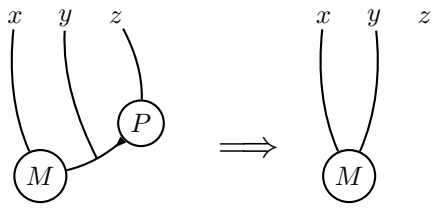
We now give reaction rules enabling the system to evolve as outlined. First, a machine can send a packet to another machine.



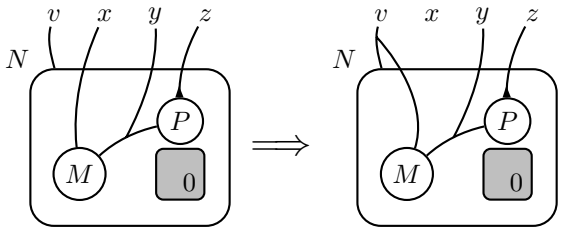
(2.1)

This rule applies whether or not the two machines are co-located. If the machines were not placed in different roots on the left-hand side, the rule would apply to co-located machines only. Conversely, composing the left-hand side with a context with sibling sites will co-locate the two machines. Thus, we retain both options. (The rule does not allow a machine to send a packet to itself.)

A machine can receive a packet addressed to it, provided the machine and the packet are co-located.

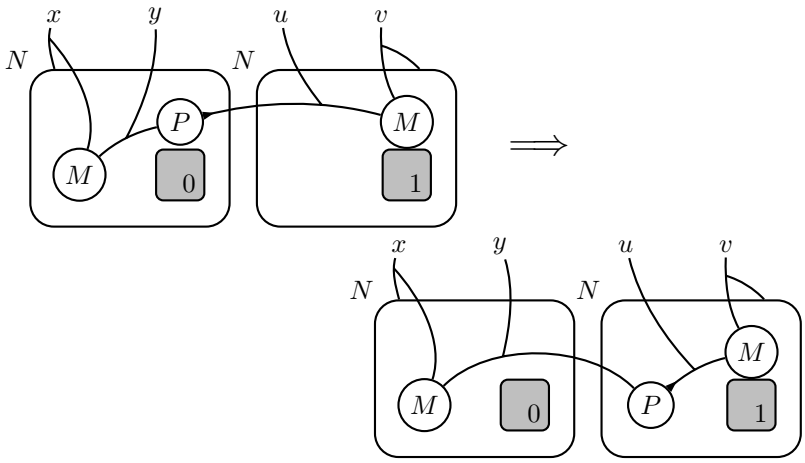


If a machine emits a packet, the switch associates the source address found in the packet with the packets enclosing network. Note the site; it is necessary to make the rule applicable when other machines and packets are co-located with the emitting machine.

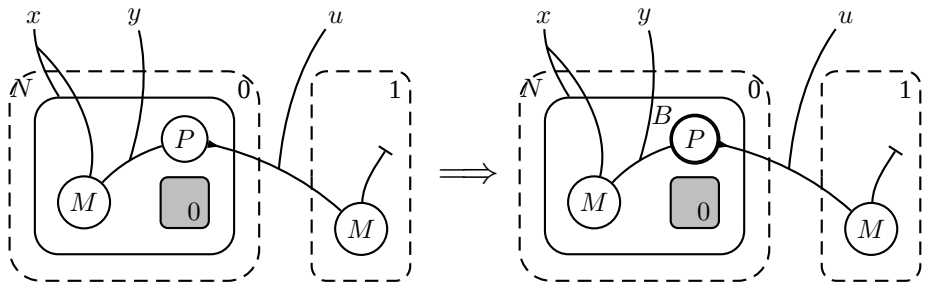


(2.2)

The switch can move a packet to the network associated with the packets target address. This rule cannot apply unless the source machine has known location.



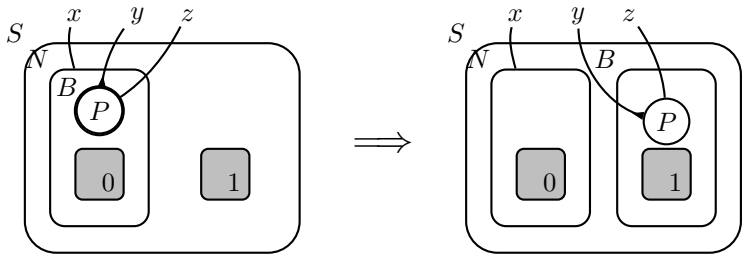
Now all we need are rules for broadcasting packets to machines that have not been located. If we satisfy ourselves with modeling a 3-way switch, writing such a rule is straightforward. Instead, we give rules for broadcasting on an arbitrary number of networks. First, we recognize when to broadcast. We mark packets that need broadcasting as such by enclosing them in a control B of of arity 0.



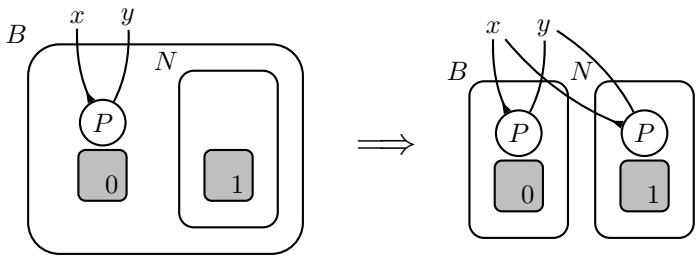
The singleton edge on the machine in the root 1 implies that the location of said machine is not known by the switch. We are required to use a wide reaction rule; otherwise, the rule would apply only when the

network and the target machine are co-located. Conversely, this rule stipulates that the two machines are not co-located.

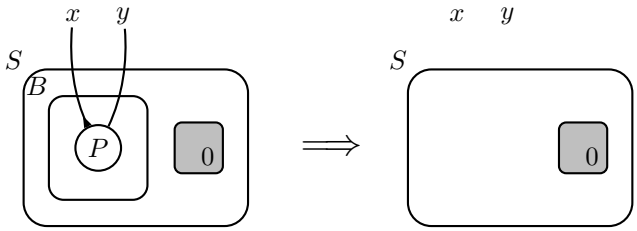
We then start copying packets marked for broadcast onto the networks by enclosing networks that have yet to receive the packet in a B control. To get hold of the networks, we introduce a new control S of arity 0; S simply contains the networks known by the switch. (The actual copying is done by the next rule. The present rule merely starts the process.)



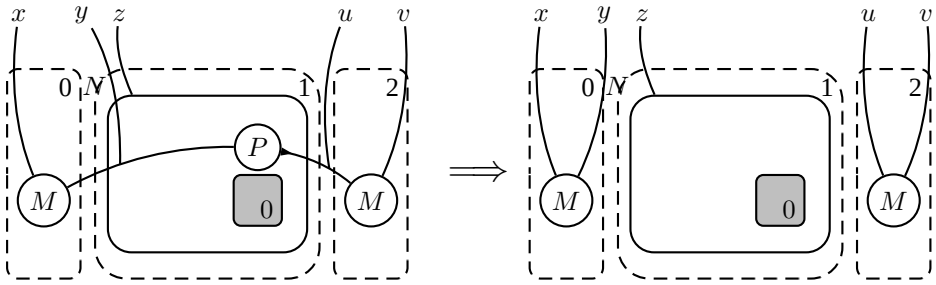
The copying itself is a recursive process, transmitting the packet on one network, while recursively copying the packet onto the remaining networks.



Once the B node contains no more networks, the packet have been broadcast. We can now safely remove the B node and its copy of the packet.



Finally, broadcasting will usually transmit packets on networks where no-one will receive them. This rule removes such packets.



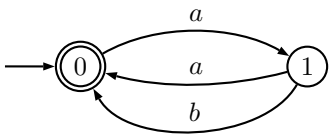
In reaction rule (2.1), we saw how to capture co-locality in a reaction rule: Simply put two nodes below the same root. However, to capture the property that two nodes are *not* co-located, it is *not* enough to put the two nodes below different roots, since composing with a context with sibling sites would then co-locate the two nodes.

Thus, we must either put one of the two nodes somewhere else than below a root, or stipulate that the bigraphical reactive system in question has no active contexts with sibling sites (i.e., all active contexts must have monomorphic place graphs). In the former case, we are abandoning the property "the nodes a and b are not co-located" in favor of the more specific "a is located at the node c, whereas b is not located at the node c". It is unclear whether this specificity will be a hindrance in practical modeling work. In the latter case, we lose the ability to specify that two nodes may or may not be co-located. Recall that in rule (2.1), we put two nodes below different roots precisely to make it possible for the redex to match whether or not the nodes are co-located.

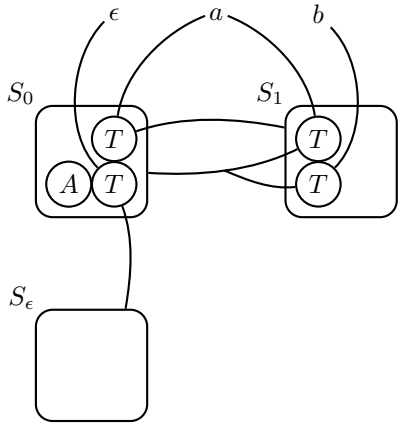
The need for wide reaction rules means in particular that we need redexes that are not prime, and hence neither basic nor simple. (A pure concrete bigraph is simple if it is discrete, prime, guarding, mono and epi. It is basic if it is simple and every node has a root as parent.) Thus, it appears that the characterizations of simple and basic transition systems in [9, Chapters 13 & 14] will only rarely apply to bigraphical models of real-life systems.

2.2 Finite Automata

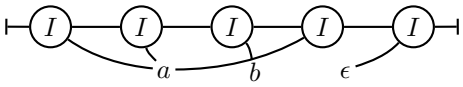
Here is a finite automaton $\mathcal{A}$ over the alphabet $\Sigma = \{a, b\}$.



This automaton recognizes the language $(a(a|b))^\star$. We encode this automaton as a bigraph by taking the alphabet to outer names, the states to nodes with control S, and each transition $x \xrightarrow{\sigma} y$ to a control t enclosed in the node for x and linked to the outer name σ and the node for y. We mark the starting node by putting a node of control a (an "active" token) within it, and we mark the accepting nodes by giving them an artificial ϵ transition to an empty state; we assume in general that $\epsilon \notin \Sigma$. The above automaton becomes the following bigraph. (We have subscripted the S-controls exclusively for the readers convenience.)



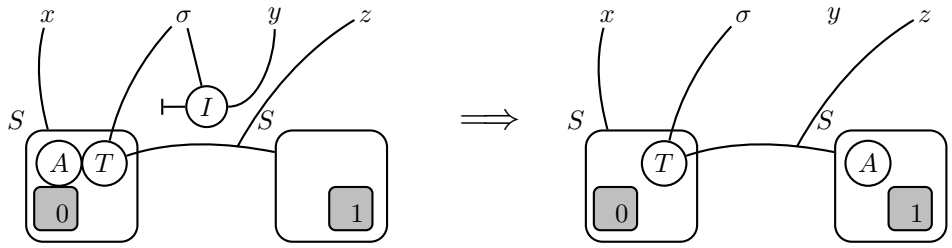
We encode the input sequence as a linked list using atomic controls i with arity 3. The first two ports are linked to the next and previous elements of the list, respectively, and the third port is linked to an inner name representing a symbol of Σ. Finally, we append the symbol ϵ to the input sequence. Altogether, the input sequence $aaba$ becomes the following bigraph.



Even though the singleton edges linked to the first and last i-control respectively seems to be indistinguishable, they are not: The leftmost edge is linked to the "previous" port of the first i control, whereas the rightmost edge is linked to the "next" port of the last i control.

Once we specify reactions, we can apply an automaton to an input sequence by composing their respective bigraphical encodings: $[\![aaba]\!] \circ [\![\mathcal{A}]\!]$.

The reaction rule is straightforward: If the active state has a transition linked to the leftmost input symbol, delete the input symbol and move the a token.



It is instructive to compare our bigraphical representation of $\mathcal{A}$ with the formal definition of $\mathcal{A}$. An automata is formally defined [10] as a 5-tuple $(\Sigma, S, \rightarrow, A, s_0)$, in this case $\Sigma = (a, b)$, $S = \{0, 1\}$, $\rightarrow = \{(0, a, 1), (1, a, 0), (1, b, 0)\}$, $A = \{0\}$ and $s_0 = 0$. The names of the states disappear in the bigraphical representation. The interesting properties of states, e.g., their transitions, is preserved using links and locality.

Dispensing with names is an essential feature of modeling in bigraphs. In the formal definition of $\mathcal{A}$, we see the symbol 0 many times, each time understanding it as denoting some abstract entity. In the bigraphical representation we draw this abstract entity only once: it is the upper-left S-node.

On a theoretical note, we believe that automata equivalence is recovered as bisimulation equivalence:

Conjecture 1 *Bisimulation induced by this reaction rule coincides with automata equivalence, i.e., for all automata A_0 and A_1.*

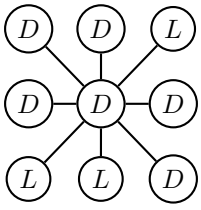
$$\mathcal{L}(A_0) = \mathcal{L}(A_1) \qquad \text{iff} \qquad [\![A_0]\!] \simeq [\![A_1]\!].$$

2.3 The Game of Life

Invented by John Conway, "Life" [7] is played on an infinite grid of square cells, each cell being either live or dead. Cells are simultaneously transformed according to these rules:

1. A live cell with exactly 1 or 4–9 live neighbors becomes dead.
2. A dead cell with exactly 3 live neighbors becomes live.

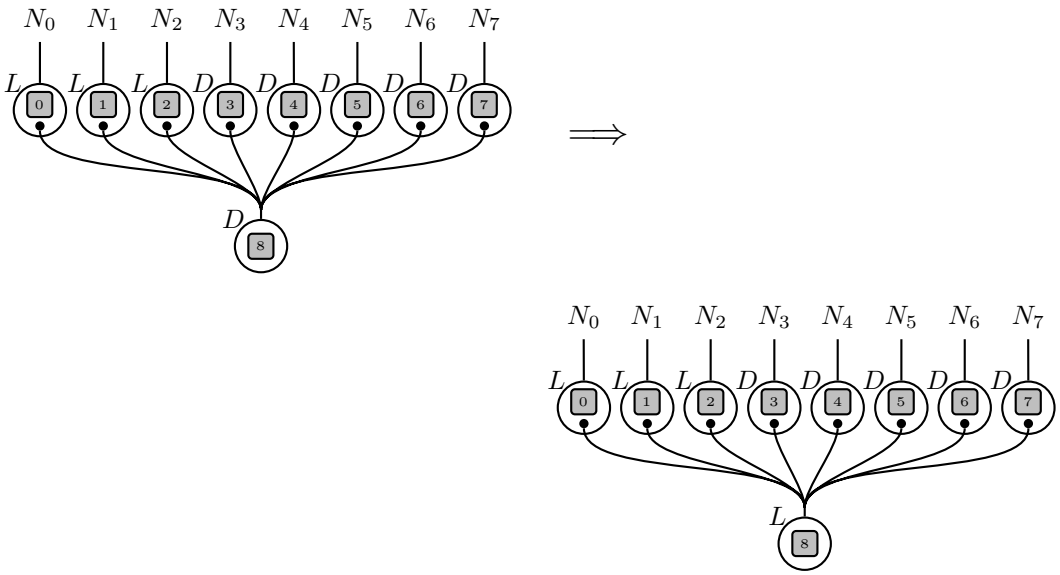
In the straightforward bigraphical representation, each cell is represented by a control L or D, both of arity 8, neighboring cells being linked. Here is a dead cell with exactly 3 live neighbors.



(We have ignored links not including the center node.)

Besides the information "the center node has 3 live neighbors", this representation also tells us *which* neighbors are live, since ports are ordered. Unfortunately, we cannot construct a redex abstracting away this extra information, and will thus have to write $\binom{8}{3}$ reaction rules to implement the second rule of the game. The game's first rule can apply in an additional $\binom{8}{1} + \sum_{k=4}^{8} \binom{8}{k}$ ways, in total requiring us to write 227 reaction rules. We would rather not.

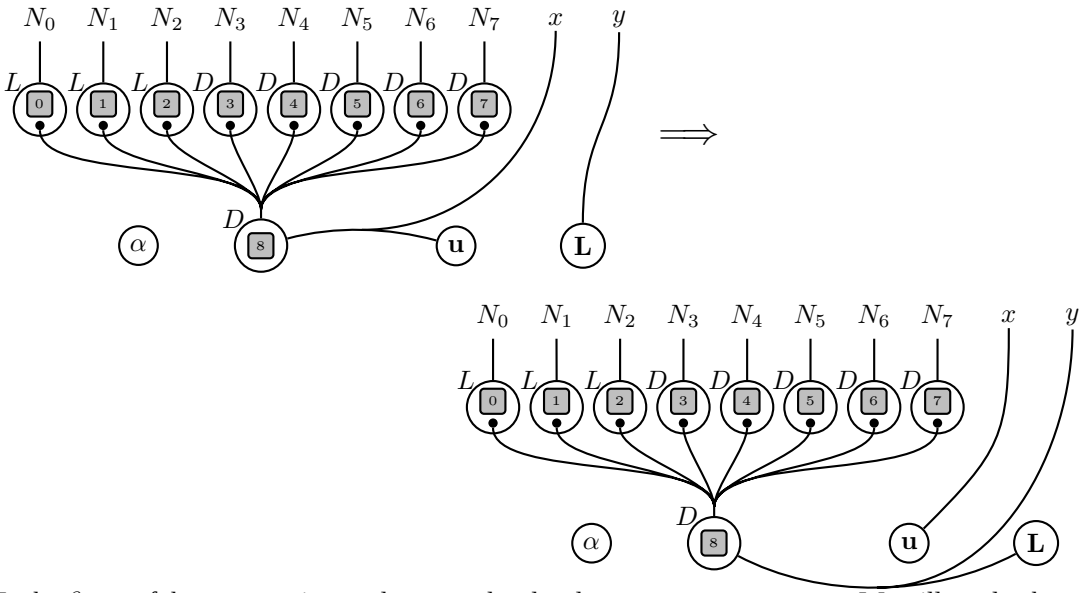
To abstract away the ordering, we introduce an anonymous control • (pronounced "dot") of arity 1. Each cell will contain 8 dots, the dots serving as unordered ports. We can now write a single reaction rule capturing the condition "a dead node has exactly three live neighbors".



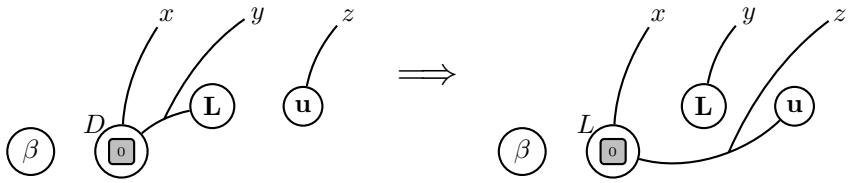
The game's first rule requires 6 reaction rules; one for each of the numbers 1, 4–8 of live neighbours. These reaction rules are similar to the one above; we do not give them explicitly.

As yet, our 7 reaction rules do not fulfill the synchronicity requirement of the game. To alleviate this, we split the computation into two parts: First, compute the next state for each cell, but do not alter the cell. Second, once all next states have been computed, update the cells.

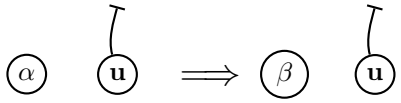
We introduce controls α and β, both of arity 0, to represent which part of the computation we are in; and we introduce controls **L**, **D** and **u** of arity 1. When in the α part, each cell is linked to either **L**, **D** or **u** depending on whether the next state of the cell is live, dead or uncomputed. The birth rule is now predicated by the precence of an α node and the next state being as yet uncomputed:



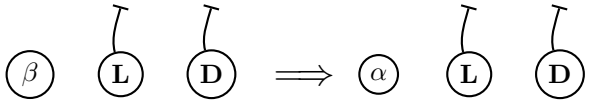
In the β part of the computation, nodes are updated and next states are reset to **u**. We will need to have 4 reaction rules, one for each of the 4 combinations of previous and next state. We give only the one in which a dead cell becomes live.



Finally, we need to change the state. We change from the α to the β state when there are no more next states to compute, i.e., then the **u** node is linked to a singleton edge.



Conversely, we change the state from β to α when there are no more cells to update, i.e., when the **L** and **D** nodes are linked to singleton edges.



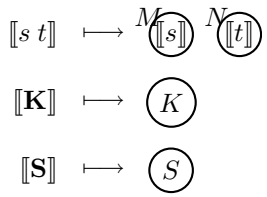
As the game of "life" is played on an *infinite* grid, the above rules are not quite sufficient: On an infinite grid, the process of computing next states does not terminate. However, at any point in the play of the game, only a finite number of nodes are live, so we can simulate an infinite grid by representing only the necessary nodes. We leave this final fix as a non-trivial exercise[2].

2.4 Combinatory Logic

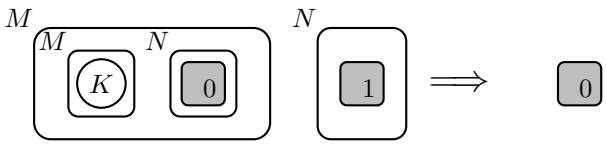
Recall the syntax of combinatory logic [2, Chapter 7]: $M, N ::= \mathbf{S} \mid \mathbf{K} \mid M\ N$. Semantics is given by the following reduction rules.

$$\mathbf{K}\ M\ N \rightarrow_w M \qquad \mathbf{S}\ M\ N\ U \rightarrow_w (M\ U)(N\ U)$$

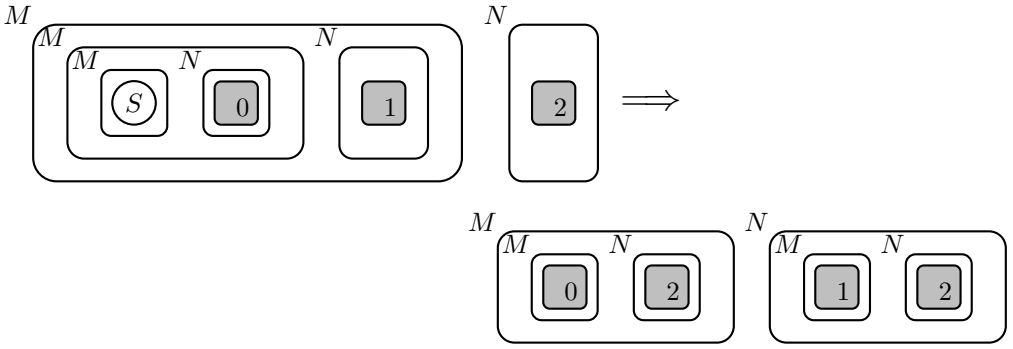
Since the place graph is a tree, the only difficulty in encoding abstract syntax is maintaining the distinction between the terms $M\ N$ and $N\ M$. In previous examples, we have imposed ordering using links, however, in this particularly simple case, it is convenient to impose order by enclosing the function part of an application in a node with control M, and the argument part in a node with control N. Otherwise, the translation is straightforward:



The reaction rules are translated almost verbatim, sites taking the place of meta-variables.



[2]It is necessary to reintroduce the ordering on neighbour nodes, in order that the board may grow. One way is to make the dot-nodes non-atomic, and place within them nodes whose control indicate to which of the eight directions the dot corresponds. Thanks to Martin Elsman for pointing out this solution.



This encoding does not entail an interesting coding of the lambda-calculus as a bigraphical reactive system. Although there is a straightforward translation of lambda-terms to terms of Combinatory Logic, that translation does not commute with β- and w-reduction. At best, we can have the equality relations induced by β- and w-reduction (i.e., the reflexive, symmetric, transitive closure of the respective relations) commute with translation, provided we add a handful of ground reductions to Combinatory Logic. Refer to [2, Chapter 7], in particular to Corollary 7.3.15.

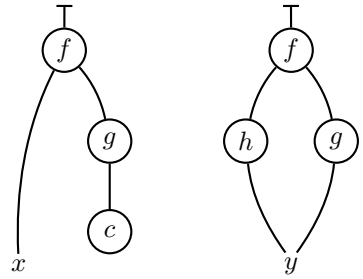
2.5 Unification

Here are two terms over the signature $\Sigma = \{f^2, g^1, h^1, c^0\}$:

$$f(x, g(c)) \qquad f(h(y), g(y))$$

These terms have a most general unifier $\theta = \{x \mapsto h(c), y \mapsto c\}$.

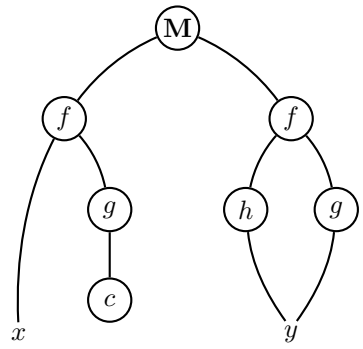
We assign to each function symbol F of arity n a control F of arity $n + 1$. The above terms become the following bigraphs. To maintain a semblance of terms in the diagrams, we take the liberty of writing outer names at the bottom.



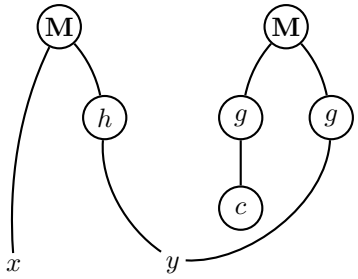
The extra port is used to link a node to its parent. Root nodes are linked to a singleton edge.

We now outline the dynamics of the unification process in the three cases where unification succeeds, fails, or loops (the "occurs check"). Afterwards we sketch the reaction rules.

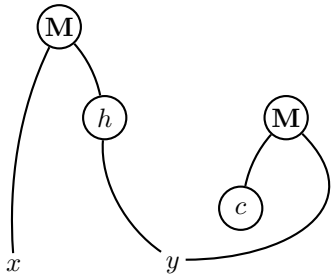
We introduce a control M of arity 2, used to represent a matching requirement. Unification of two terms begins by linking both of them to an M node.



If two nodes with the same control are linked by a matching node, we delete the two nodes, and distribute the matching requirement over their children.



The left M-node is now linked to the variable x and the subtree $h(y)$, indicating that the most general unifier, if it exists, must take x to $h(y)$. This concludes matching of the left tree; matching proceeds only in the right tree.

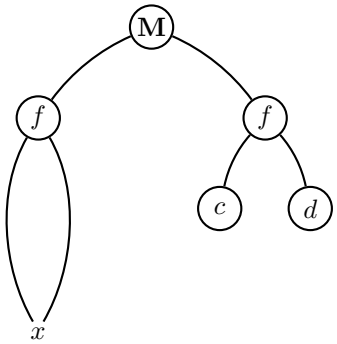


The right M-node is now linked to the variable y, indicating that a most general unifier, if it exists, must take y to c. Since both the M nodes are linked to a variable, no matching requirements are imposed, so unification is now complete. The substitution $\{x \mapsto h(y), y \mapsto c\}$ can be read directly off the final bigraph. (We leave the obvious flattening of the y variable as an exercise.)

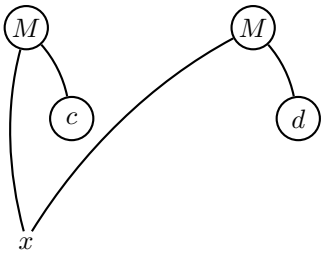
Add a nullary symbol d to the signature, and consider the terms

$$f(x, x) \qquad f(d, c).$$

These terms do not have a unifier. Bigraphically:



The M-node distributes over the f nodes.

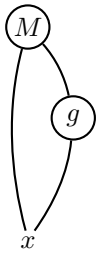


The above matching is contradictory since two M nodes are linked to the same variable, but to different function symbols. In general, a matching is contradictory if there is a path from a constructor node to a different constructor node such that the intermediate nodes are all variables or M nodes. In the above case, the path is $c \to M \to x \to M \to d$.

Finally, consider the terms

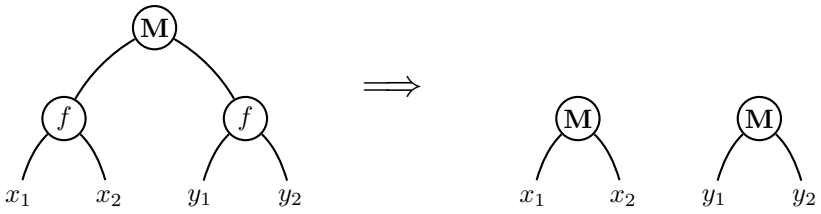
$$g(x) \qquad x.$$

These terms fail the "occurs check". Bigraphically:



Matching is complete in zero steps. We see the problem: Once matching is complete, the final bigraph cannot contain non-trivial loops, i.e., a path from a variable to a variable that includes a constructor. In the above case, $x \to M \to g \to x$ is such a path, the constructor g giving non-triviality.

To write reaction rules for the above system, we have to express the condition "nodes linked to the same M-node must themselves have identical controls". We cannot express this condition generally, so we have to write a separate rule for each symbol in the signature. Here is such a rule for the f symbol:



It is possible to capture identity of function symbols by factoring the nodes of the bigraph into placeholder nodes linked to nodes representing function symbols. The reaction rule in the finite automaton example of Section 2.2 uses the same trick to capture identity of an input symbol and the symbol guarding a transition: On the left-hand side of that reaction rule, both the t and the i are linked to σ.

Chapter 3

Binding Bigraphs

In this chapter, we develop a bigraphical model of simple event-driven systems, which illustrate the additional modeling capabilities we acquire when using *binding* bigraph reaction rules. Incidentally, this exercise is also a tour of "Design Patterns in Bigraphs". Throughout, we remark on similarities and connections with the patterns of Gamma et. al. [6].

3.1 Event-driven Systems

We build a bigraphical *framework* for modeling event-driven systems – in the sense that we intend the model to be an *extensible* specification, that allows a user to model a concrete event-driven system. In the present paper, we concern ourselves with expressing event-models that underlie standard graphical user interfaces.

Although we motivate the framework with a GUI application, we will abstract away from specific applications. We intend the framework to be usable also for modeling other event-driven applications, for instance XML-parsers (see e.g., [1]).

We proceed incrementally, simultaneously adding components to our system and giving suitable reaction rules. Components will be modelled by nodes of a particular control – signifying the type of that component. We use place graph nesting to model *exclusive* association between components, e.g., that a window *has* a button (and no other window has that button). We shall mainly use the link graph to model *shared* assocation.

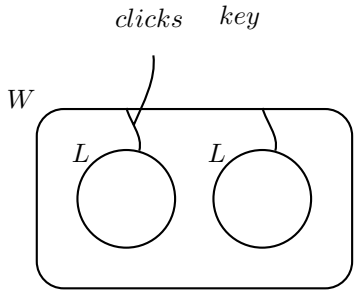
3.1.1 A Model with Atomic Events

We build a model in two steps; initially, we build a model where events are simple markers, that have no associated information. In the next section we sketch, how the initial model needs to be altered, in order to add events with associated information.

Events and Listeners

The two basic components of an event-driven model are *events* and *listeners*.

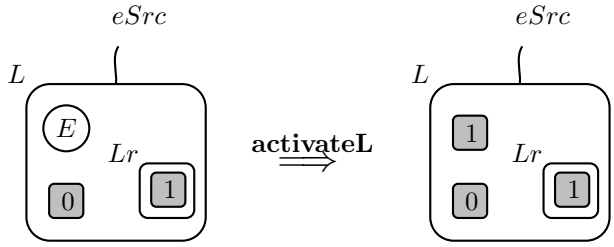
A listener L is a component which upon the receival of an event E initiates one or more actions. (For that reason, listeners are also referred to as event-handlers.) Event-receival is modelled by the appearance of an E node within an L node. Listeners could, for example, handle mouseclick events and keyboard events associated with a particular window.



Registering a listener as an event-handler, for a particular kind of event associated with some component, amount to connecting a port associated with that event of the component (e.g., the *key* port for a window) with the port of the listener. This will allow us to let a single listener handle different kinds of events.

As all events are equal and have no associated information in the simple system, we assign E atomic control and arity 0.

A listener contains a repository Lr of commands C (see below) that upon event receival are *enabled*. We enable the commands by copying them from the repository.



To ensure that no reactions can occur with commands in the repository, we assign Lr *passive* control – while L itself will have *active* control.

(Note, that we overload the graphical notation for bigraphs slightly in drawing the reaction rule above. Formally, the reactum in $\mathbf{activeL}$ is not a binding bigraph, since there are two different sites numbered equally. We intend, of course, that both sites numbered 1 of the reactum are copies of site 1 of the redex. Formally, we should specify the *instantiation*-components of the reaction rule, to achieve this – but we find that the overloading improves readability (see [9, Chapter 12] for the formal details).)

Commands

As we are building a framework for doing event-driven systems, we wish to make the framework extensible. Therefore, we must allow the user to add components (nodes of possibly new controls) and dynamics (reaction rules), that must be able to interact with the existing components without rewriting the reaction-rules, we give. We can only specify a set of basic rules that model the kind of general behavior, that we would expect from all event-driven systems.

In particular we need a unified method of requesting the execution of some action in the system, which in certain circumstances we can treat without specifying explicitly, *what* is to be done.

If we wish to enforce the event-driven paradigm, it might be very sensible to restrict the user to write only reaction rules that specifies the behavior of command requests. We leave it as further work to investigate and specify more precisely, how extensions of bigraph models like this could be sensible restricted.

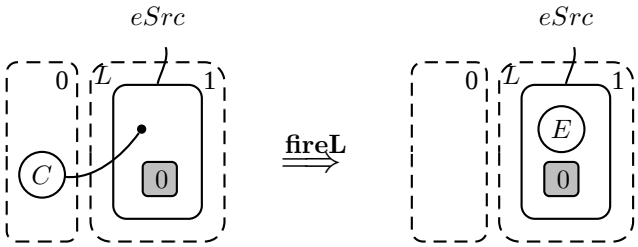
The abstraction we wish to provide, can be seen as an instance of the Command behavioral pattern in [6, Page 233]. Hence, we name the component, that models and encapsulates the information related to execution requests, *command*.

Command components will be modelled by nodes of control C with arity 1. The single port of a C node is connected to its receiver. A C node that is enabled, that is appears in an active context, models a

request for some action to be taken in a receiver. The semantics of a command request will be modelled in reaction rules.

Our first rule for commands illustrates the pattern we use for specifying command semantics; an enabled command linked to a receiver in a certain state makes it possible for the system to execute a particular reaction rule.

The following rule describes what a command linked to a listener means. It enables a listener to defer the handling of an event to another listener. (Incidentally, this can be seen as an instance of the Chain of Responsibility pattern [6, Page 223].) The listener does this by containing a command linked to another listener.



Note that this rule does not exclude the possibility of the first listener enabling several commands based on the (initial) event receival.

We use nodes of the anonymous • control (see "The Game of Life" example, p. 10) to model association with a command's receiver port. We cannot just connect each command node to a new port of the receiver, as a priori we cannot know the number of commands that would like to associate themselves with that particular receiver. We could have assigned each component in the system a single request port, but this would go against our stated aim of a loose coupling in the execution request structure. By abstaining from creating direct linkage between a Command and its receiver, we are able to write reaction rules that depend on a C-node having exactly one receiver, without specifying explicitly the kind of that receiver. We shall see an example of this in the **fireStateCond** rule.

State

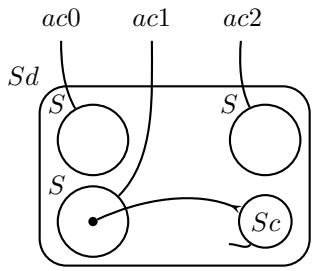
As a base component in the framework, we also model *state*, allowing us to model for example maintenance of a list or the state of a toggle button. The following collection of components and reaction rules handles *state* of components in a style similar in flavour to the State pattern of [6, Page 305].

We localize state information in a node of control Sd (a *state diagram*), which will contain a number of concrete states S. The intention is that a component can associate itself with a state diagram by nesting an Sd node within itself. If several components share an association with a state diagram – if for example, a toggle button and a dropdown box can affect the same underlying state diagram, the assocation is modelled by linkage.

The chosen representation of association will have no consequence on our representation of state, though, as we intend to handle change of state by our execution request structure – that is by Command reaction rules.

To this end, in each state diagram, we add a *state controller* node of control Sc which have arity 2. The invariant for our model is that the first port will always be linked to S node(s) that represents the *current* state, while the second port can be linked to one or more states, which have been *requested*.

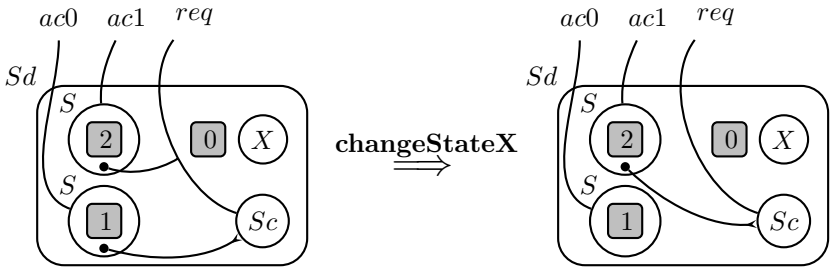
Below we see a state diagram that is *stable*, e.g., there are no requests for changing the current state. To distinguish the ports on the Sc node, we mark the *current* port with an arrow.



The arity of an S-node is 1, allowing us to associate a component with each state in a state diagram. This will allow us to let behavior of the system *depend* on state (see the end of this section). This port will be linked to a discrete outer name in all rules except **fireStateCond**.

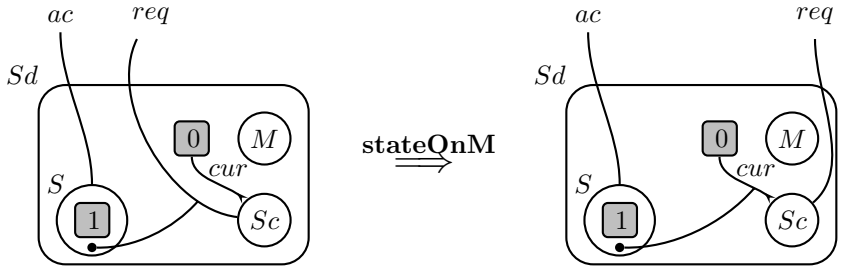
Before specifying the dynamics, we are faced with a design choice: Are we going to model mutually *exclusive* state diagrams or *multiple* state diagrams? Each choice may be easily modelled by either disallowing or allowing the current-state port to be connected to several S nodes. We could, of course, encode multiple state diagrams in exclusive state diagrams. We shall model both directly, though, as in doing this we treat subtly different modelling issues, and we will investigate to what extent we can parameterize our reaction rules on the type of state diagram. To that end, we shall mark Sd nodes with atomic marker nodes, X or M, signifying whether a state diagram is exclusive or multiple state, respectively.

In exclusive state diagrams we can associate a state with the current link, exactly when another state has disassociated itself with the current link. We model this state-change semantics by making a single rule, thus ensuring that there is no intermediate phase violating this invariant.



Note that this rule does not leave out the possibility of having exclusive state diagrams with one or zero S-nodes. Such state diagrams make sense, if we allow state diagrams to change dynamically – by allowing reaction rules to add or remove S nodes.

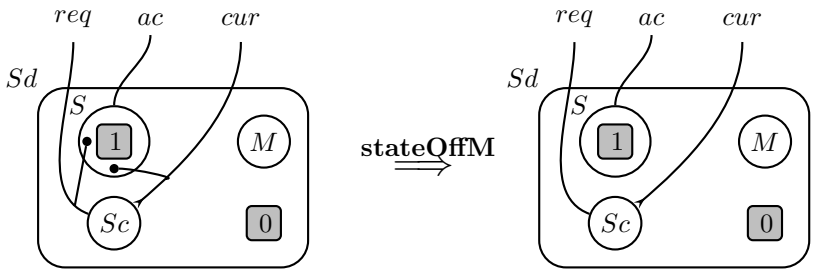
For multiple state diagrams we need to separate the mechanisms for switching states on or off. We interpret a *request* link connected to a state as a request for changing the association of the *current* link with this state.



In this rule, we see the first simple example of the usage of binding in a reaction rule. We analyze the gain in using a binding bigraph reaction rule in this case, by asking: Could we just have let the cur link be an outer name, instead of a local link to a particular site?

The answer is, yes; but, by stating the reaction rule as above, we restrict the behavior of the system – by restricting the applicability of the reaction rule. We aim simply to protect a sensible invariant of our encoding of multiple state-diagrams; namely, that state-nodes can have at maximum one associated cur-link. By making the cur-link local to site 0, we disallow any linkage from site 1 in the redex. Thus, by this rule it will not be possible to introduce multiple cur-link associations to a state-node.

In the next section, we shall see examples, where we cannot specify the behavior we want to allow, without using binding. We shall also discuss in more detail, the added modeling capability that binding bigraph reaction rules yields.

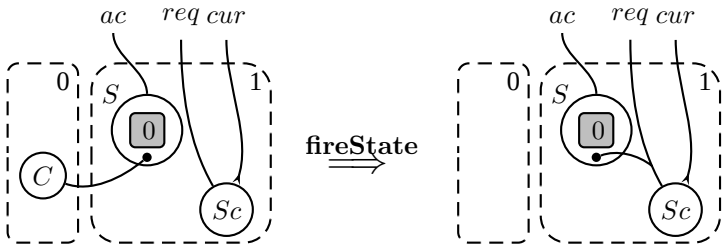


A shared property of both kinds of state diagrams is that we model resolution of unstable state diagrams, i.e. state diagrams with several state-change requests, nondeterministically. This is an inherited property of bigraphical reactive systems (BRSs). We have nondeterminism unless we model sequential behavior explicitly (as can be seen in several of the previous examples – most notably "The Game of Life").

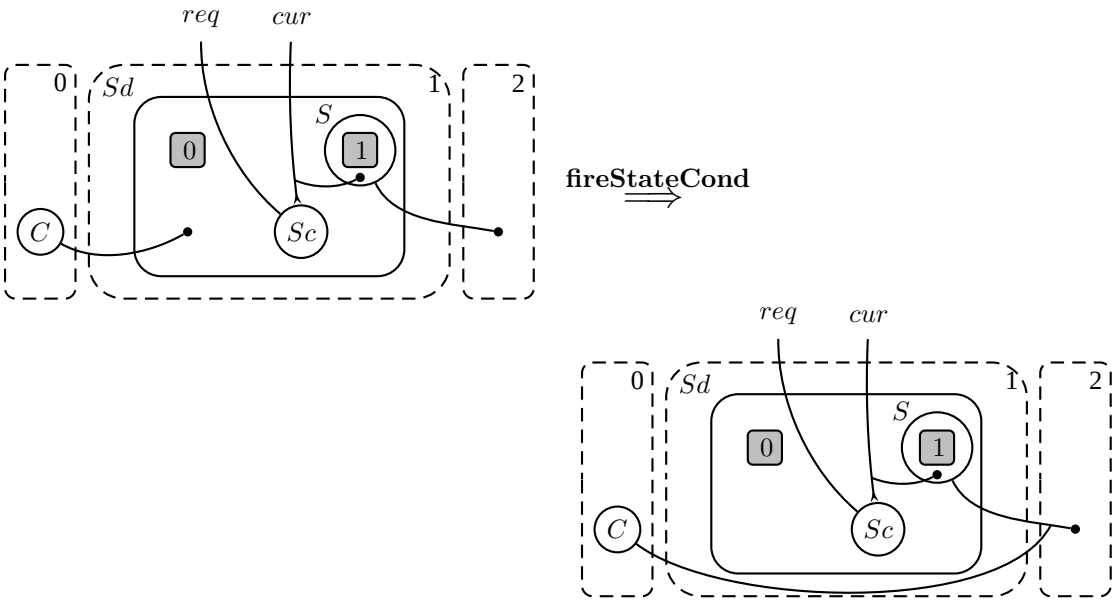
While in many cases it would make sense (at least with GUIs in mind) to explicitly model a request queue, we accept this nondeterminism for now.

Further, it would also be easy to extend these reaction rules with listeners and events that register and signal state changes to other components of the system. (The enthusiastic reader is encouraged to add request queues and statechange-listeners as an exercise.)

Having specified how the internals of states and state diagrams work, we tie the connection with commands; thus enabling events to initiate state changes. Using the same pattern that we saw for the **fireL** rule (p. 18), we specify that a command connected to a state of a state diagram is interpreted as a request for connecting that state to the *request* channel of the local state controller.



We would also like to be able to let behavior of the system *depend* on state. One way of modeling this is by associating to each S node a component, and adding the following reaction rule:



Informally, this rule says that we interpret a command associated with a state diagram (with some current state), as a request for issuing a command to the component associated with the current state. (Note, that the rule works for both kinds of state diagrams, but again we have an inherent nondeterministic choice, if there are multiple current states.)

To uphold the invariant that commands are associated with a single receiver component, we need to restrict the number of peers that a state can have on its associated component link. Instead of just letting the link be a name, we specify the reaction rule in way, which gives us more control over the link. We associate a component to a state by nesting a node of the anonymous • control inside the component. This allows us to specify, in the reaction rule above, that for the reaction rule to fire: (i) some link (to a component) must exist – i.e. the link cannot just be an edge with no peers; and that (ii) there is only *one* anonymous port (of a component) associated to the state.

Of course, the reaction rule itself does not preclude a system from actually assocating *any* kind of odd linkage to this port, and we cannot, in fact, be certain that the • node lies inside a component.

In fact, when using BRSs as a *programming* or *specification* language for an extensible framework as this, it is not immediately clear how to protect the invariants of *inbuilt* components and dynamics, that we wish a user of our framework to treat as abstract .

It is not within the scope of the current paper to delve further into this topic. We note, that work on appropriate mechanisms for protecting invariants – for instance by restricting the modeling power of a user (as hinted at in the section on commands), seems an interesting line of further work; in particular, when we wish to use BRSs as a foundation for a specification or programming language.

With these basic components and reaction rules, we claim it is quite easy to build an event-driven model of a simple graphical user interface. We could, for example, add a few buttons with associated listeners, and a state diagram with three states to the window example we saw at the start of this section to model a window with minimized, maximized, and hidden state.

For more involved systems the model seems too simple, though, as the current simple event and command structure prevents us from distinguishing events from the keyboard or the mouse – or whether a user typed `'abc'` or `'Hello world'`.

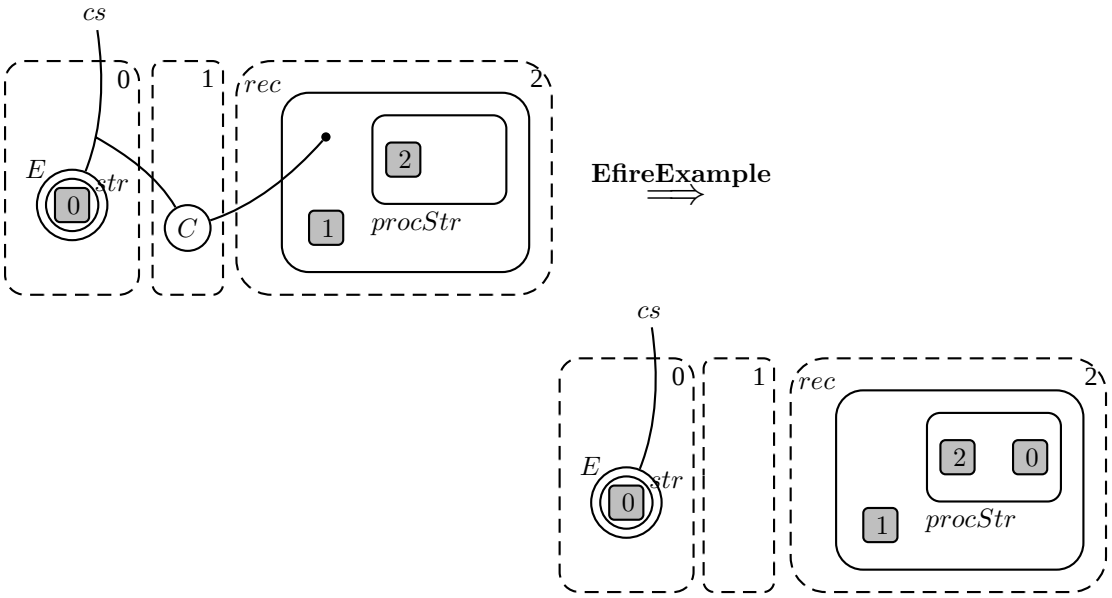
Below, we sketch how we can adopt the model given above, to work with events with associated information.

3.1.2 Associating Information with Events

We change the basic components in the following ways: The E control is now passive – allowing it to carry nested information, and have a single port, that will be connected to the commands it initiates. We increase the arity of the C control by one - to allow this connection to be made via a direct link. Commands can now refer back to their *source* event, when we specify what effect a command associated with a component should have.

(We prefix reaction rules of the extended system with an E, to distinguish them from the reaction rules in the former section.)

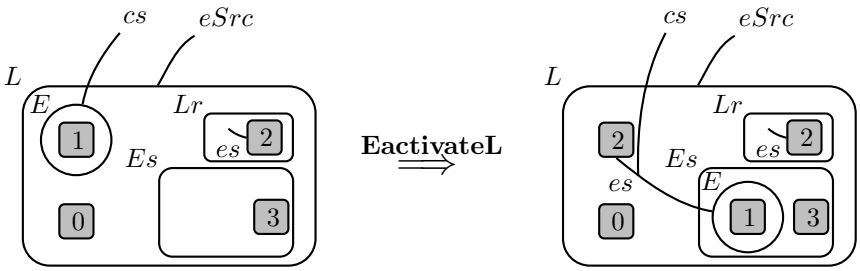
Thus, we have a new pattern for writing command reaction rules, illustrated by the following example:



In the example rule given above, we state the intended semantics of a command request, connected to an event containing a str component, upon a rec component with a $procStr$ subcomponent. We could specify another rule, allowing the same component to react in a different manner, when associated with a command linked to another kind of event.

We note that, as events carry information which can be accessed several times, we need to save events instead of removing them as the old $\mathbf{activateL}$ rule did. Hence, we add a passive event-store Es to listener-nodes where events, which are received, but not yet processed, are saved. Paired with a garbage collection rule for events with no associated commands (which we shall not bother to write), this will serve our purpose.

The new reaction rule for activating commands in listeners looks like this:

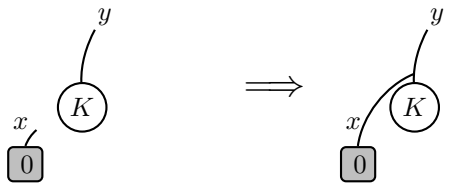


Here we see the first example, where it is actually necessary to use binding bigraphs to express the behavior that we wish to express. This is the case, because we need to explicitly change the linkage to a specific parameter and connect it to some links in the reactum. (We further extend the overloading of the graphical notation for bigraphs, since we allow the local inner name es to be located at two different sites numbered equally. The intention is (again), that both sites numbered 2 of the reactum are copies of site 2 of the redex – now with local names. Formally, the overloaded notation simply indicates how the *renaming* and *instantiation*-part of the reaction rule should be constructed which achieves the intended effect (again, see [9, Chapter 12] for the full formal details).)

We need to connect the event-source links of the copied commands from the listener repository Ls to the port of the event. This event might already be connected to other commands – therefore, we need to make this link an outer name.

It is illustrative to see a simpler reaction rule that singles out the property of the intended semantics, that requires us to use binding bigraphs.

In the reaction rule below (not related to our event-model), we wish to express, that in the reactum, some wiring in a particular parameter and some wiring in the redex, should be linked.

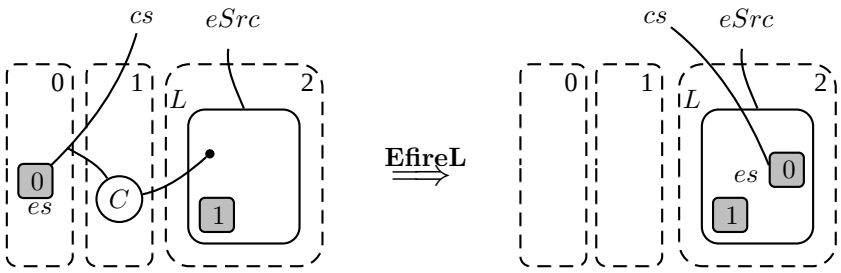


Intuitively, to stay within pure bigraphs, we would need to change the redex, to show the contents of the site *explicitly* in the redex, to make this rearrangement of linking. That would require us to write a rule for every possible content of that site; and, if, as in our event-model example, that site is supposed to be able to contain an unbounded number of nodes – that would severely restrict our system.

As we have changed the arity of C, we need to rewrite every rule involving commands, to reflect this in the extended system. As this change is quite trivial in all cases, but for one rule, we shall only consider this one:

In the rule for deferring event-handling to another listener, we have to make a nontrivial design choice: Now that we save the events in the receiving listener, and link one or more commands to them – what do we do with a command that wants to defer the handling of event, while other commands might use the event in another manner?

We choose to simply *move* the original event to the new listener, thus keeping links from the event to possible other commands intact.



3.2 Summary

We have seen examples of two kinds of usage of binding bigraph reaction rules: Restricting the possible behavior of a system as a method of protecting invariants; and examples, where binding bigraph reaction rules are necessary to model the behavior, we intend of the system.

In pure bigraph redeces and reactums, we only have one option, when we wish to leave a link open for further linkage; namely, to let the link be an outer name. The context can connect this linkage with wiring in a parameter only by an identity (the id_Z component of reaction rules), which essentially restricts us from having any control over this linkage – either for restriction purposes, or for changing that linkage.

The added power of binding bigraphs stems from the possible non-linearity of the bound linkage inside the parameter, combined with the possibility of locating linkage to one or more local inner names. This allows us to specify reaction rules like $\mathbf{stateOnM}$, where we restrict the systems that are able to match the rule; and $\mathbf{EactivateL}$ and $\mathbf{EfireL}$, where we change linkage to particular roots of a parameter.

On a final note, it seems that work on appropriate mechanisms for protecting invariants of certain parts of a system – essentially treating them as components of abstract type with only certain allowed operations – presents itself as an interesting possibility for further work, in particular in moving towards using BRSs as a foundation for a specification or programming language.

Bibliography

[1] The sax project. `http://www.saxproject.org/`.

[2] Hendrik P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, revised edition, 1984.

[3] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *FOSSACS '98: Foundations of Software Science and Computation Structures: First International Conference*. Springer-Verlag, 1998.

[4] Giovanni Conforti, Damiano Macedonio, and Vladimiro Sassone. Bilogics: Spatial-nominal logics for bigraphs. 2004.

[5] Troels C. Damgaard and Lars Birkedal. Axiomatixing binding bigraphs. Technical Report 63, IT University of Copenhagen, 2005.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison–Wesley Professional Computing Series. Addison–Wesley, 1995.

[7] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970.

[8] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–49. ACM Press, 2003.

[9] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge Computer Laboratory, February 2004.

[10] Hopcroft John E. and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison–Wesley Series in Computer Science. Addison–Wesley, 1979.

[11] James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In *CONCUR '00: Proceedings of the 11th International Conference on Concurrency Theory*, pages 243–258. Springer-Verlag, 2000.

[12] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[13] Robin Milner. Bigraphs for petri nets. To appear in Proceedings of the Advanced Course In Petri Nets (Eichstätt 03), 2003.

[14] Robin Milner. Axioms for bigraphical structure. Technical report, University of Cambridge Computer Laboratory, February 2004.

[15] Robin Milner. Bigraphs whose names have multiple locality. Technical report, University of Cambridge Computer Laboratory, February 2004.

[16] Larry L. Peterson and Bruce S. Davie. *Computer Networks — a Systems Approach*. The Morgan Kaufmann Series in Networking. Morgan Kaufmann Publishers, second edition, 2000.