BigraphIDE: An Integrated Development Environment for Bigraph Modeling and Evolution Using Rewrite Rules

Manuel Krombholz[1], Dominik Grzelak[1] and Uwe Aßmann[1]

[1]Software Technology Group, Technische Universität Dresden, Nöthnitzer Str. 46, 01187 Dresden, Germany

Abstract

Bigraphs provide a versatile mathematical formalism for modeling systems characterized by both locality and connectivity. While established tool support exists for modeling and simulation, support for visual modeling remains limited. We present BigraphIDE, an integrated development environment for bigraph modeling, rewriting, and interactive, pattern-based reachability checking, delivered as a Visual Studio Code extension. The tool is built on the Graphical Language Server Platform (GLSP) and uses the Bigraph Framework as its computational backend, which was extended with mutable bigraphs to enable incremental, in-place editing. BigraphIDE supports visual construction of all bigraph constituents (i. e., nodes, edges, inner and outer names, sites, and multiple roots) through a dedicated diagram editor with signature management and automatic layout. Rewrite rules are specified as pairs of bigraphs and organized into named rule sets via drag-and-drop. An evolution manager supports rule application through play, step, and pause controls, generating an interactive evolution tree with checkpoint-based branching and replay. Reachability checking is provided by allowing bigraphs to be used as match patterns that stop the evolution when satisfied. Together, these capabilities may provide researchers and students with a practical, integrated platform for modeling and rewriting bigraphical reactive systems.

1. Introduction

Bigraphs [1] provide a unifying formalism for reactive systems that exhibit both locality and connectivity across spatial regions. Despite their expressive power, tool support for bigraphs is mostly focused on textual specification and verification; visual modeling is rarely supported. This is surprising, since bigraphs are a diagrammatic formalism by design. Some of the existing approaches separate modeling, rule application, and model checking into different tools or command-line workflows, so users have to switch between representations and do much of the glue work by hand. Visual tooling would match the formalism more closely and lower the entry barrier, especially for students.

Bigraphs combine spatial nesting with non-local connectivity, yielding a non-trivial structure to construct and interpret. With regard to the related literature (Section 2.4), modelling and bigraph rewriting are often performed without diagrammatic support, which may increase the likelihood of inconsistencies. Visual feedback both during model construction and system evolution can mitigate these difficulties by making structural changes directly observable. Similarly, a visual interface for constructing graphs and transition systems, as exemplified by tools such as AGG [2] and GROOVE [3], can support users in systematically defining and exploring system behaviour.

Therefore, BigraphIDE[1] aims to provide modelling and interactive, pattern-based reachability checking within a unified environment, without requiring programming. The central artifacts—the workspace bigraph, rewrite rules (redex and reactum), evolution traces, and bigraph patterns (i. e., correctness properties for verification)—all appear as inspectable diagrams, so structure and dynamics stay visible throughout modeling, rewriting, and analysis. This paper is a tool contribution: its novelty lies in integrating visual modeling, rewriting, and interactive pattern-based reachability checking into a single environment that requires no programming. The contributions of this paper are as follows:

1. A VS Code-based bigraph modeling editor built on GLSP [4] and the Bigraph Framework [5], with

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

manuel.krombholz@tu-dresden.de (M. Krombholz); dominik.grzelak@tu-dresden.de (D. Grzelak); uwe.assmann@tu-dresden.de (U. Aßmann)

© 2026 Copyright 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

[1]https://github.com/bigraph-toolkit-suite/bigraphs.ide

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-26

CEUR-WS.org/Vol-4338/GCM-paper2.pdf

mutable-bigraph support for incremental diagram updates, that supports all bigraph constituents: nodes with controls and arities, edges, inner and outer names, sites, and multiple roots.
2. A visual rewrite rule builder in which users model redex and reactum as bigraphs and assemble them into named rule sets via drag-and-drop.
3. An evolution manager that simulates a bigraphical reactive system by driving rewrite-rule application (play, step, pause) under user control and generating checkpoints; it supports interactive, pattern-based reachability checking with user-defined bigraph patterns as stopping conditions.

2. Background and Related Work

2.1. Bigraphs

A bigraph [1] is a mathematical structure that represents a system in terms of two orthogonal aspects: *locality* and *connectivity*. Concretely, it pairs two graphs that share a common set of nodes but capture independent structural dimensions:

- The *place graph* is a forest that captures spatial nesting. Its roots represent top-level regions, its interior nodes represent entities of the modeled system, and its *sites* act as holes indicating where further structure may be inserted. Every node belongs to exactly one parent (a root or another node), forming a strict hierarchical containment.
- The *link graph* is a hypergraph that captures connectivity independently of locality. Every node has a fixed number of *ports*, defined by the arity of its control. Ports on nodes and inner names are connected by *links*, which form hyperedges. A link is either an *edge* (an internal, closed hyperedge) or an *outer name* (an open link exposed at the interface). Inner names and outer names together constitute the bigraph's *interface*, through which bigraphs compose.

Formally, a bigraph $B : \langle m, X \rangle \to \langle n, Y \rangle$ maps an *inner face* with m sites and inner names X to an *outer face* with n roots and outer names Y. These two faces serve as typed interfaces that govern how bigraphs may be plugged together through composition.

A *signature* $\mathcal{K}$ defines the vocabulary of a bigraph model. Each element of $\mathcal{K}$ is called a *control* and carries a *name*, an *arity* $\mathrm{ar}(K) \in \mathbb{N}_0$ specifying the number of ports, and a *kind*: *active* (children are reachable for matching), *passive* (children present but not matchable), or *atomic* (no children permitted). Every node is assigned exactly one control, so the signature determines the set of permissible node types and link capacities.

2.2. Reaction Rules and Bigraphical Reactive Systems

A *reaction rule* is a pair $\rho = (L, R)$ of bigraphs called the *redex* and the *reactum*, sharing the same outer face. Application of ρ to a host bigraph B involves finding an embedding of L in B (respecting both place and link structure) and replacing the matched occurrence with R, yielding a new bigraph B'. If no embedding exists, the rule cannot fire; if several exist, any may be chosen, introducing nondeterminism.

Because reaction rules act on structure instead of a fixed program counter, they fit naturally to concurrent and distributed processes—for example, agent movement, communication events, or network reconfigurations expressed as local structural transformations.

A *bigraphical reactive system* (BRS) pairs a signature $\mathcal{K}$ with a set of reaction rules $\{\rho_1, \ldots, \rho_k\}$. Applying these rules repeatedly produces a sequence of successor states, or a branching tree in the nondeterministic case; both are referred to as the *evolution* of the system and together form its labeled transition system (LTS). The process stops when no rule is applicable, i. e., a deadlock state, or when an external stopping criterion, e.g., a property check, is satisfied.

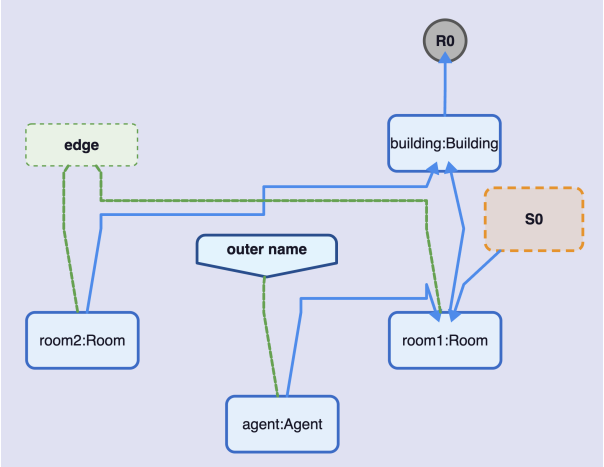


Figure 1: Bigraph notation primer. The grey circle represents a root; rounded blue rectangles represent nodes with labeled controls and ID; the orange dashed rectangle represents a site; the green dashed rectangle represents an edge; the arrow-shaped rectangle represents an outer name; green lines represent links; blue containment shows nesting.

2.3. Composition

Bigraphs support *composition* ($B_0 \circ B_1$), which fills the sites of B_0 with the roots of B_1 and fuses matching names, and *tensor product* ($B_0 \otimes B_1$), which places two bigraphs side by side with disjoint regions and names. Composition underpins rewriting: applying a reaction rule decomposes the host around the matched redex and reassembles it with the reactum. In BigraphIDE, both operations are directly accessible via a composition button in the Bigraph Explorer (Section 3.2).

2.4. Related Work

Prior work on graphical bigraph modeling includes *Bigellor*, a web-based editor that goes back to Tobies's diploma thesis [6], that is available as open-source software[2] but not actively maintained.

Besides Bigellor, several tools have been built for constructing, rewriting, and analyzing bigraphs, each one addressing shortcomings of its predecessors: BigMC [7] (2012), BigRED [8] (2013), BigraphER [9] (2016), and the Bigraph Framework [5] (2024). They cover different concerns—graphical editing, command-line analysis, programmatic modeling—and we compare them with BigraphIDE to make the gap that motivates our work visible: no existing tool combines visual modeling with interactive, pattern-based reachability checking in a single environment.

BigRED. BigRED [8] is a graphical development environment for bigraphs, implemented as a set of Eclipse plugins on top of the Eclipse Graphical Editor Framework (GEF). It provides a visual editor for all standard bigraph constituents—nodes with controls and arities, edges, inner and outer names, sites, and multiple roots—a signature editor, and a reaction-rule editor in which redex and reactum are drawn as a bigraph pair. BigRED is, however, only a modeling front-end: it cannot apply reaction rules or simulate evolution itself. It instead translates models into the term language of BigMC [7], runs BigMC as a subprocess, and parses the results back for display. Reachability analysis is thus delegated; BigRED itself does not provide interactive model checking.

BigMC. BigMC [7] is a model checker for bigraphical reactive systems, written in C and released under the GPL. It reads bigraphs and reaction rules in a textual term language and performs explicit-state, breadth-first exploration of the induced transition system. Properties are boolean combinations of two predicates: `matches(T)`, which checks whether a bigraph term T has an embedding in the current

[2] https://github.com/bigraph-toolkit-suite/bigraphs.ide.bigellor

state (a reachability query), and `terminal()`, which detects deadlocks. When a property is violated, BigMC returns a counter-example trace leading to the offending state. To mitigate state-space explosion, BigMC uses a static analysis based on a *causation* predicate that decides whether applying one rule can enable another; this lets the tool prune branches that are statically unreachable. It supports only pure bigraphs—no sharing, no stochastic rules—and its term language is incomplete with respect to inner names and edges [7].

BigraphER. BigraphER [9] is an OCaml-based rewriting and analysis engine for bigraphs, shipped as a library and a command-line tool. Its specification language follows the standard algebraic notation for bigraphical expressions: signatures are declared with `ctrl`, bigraphs with `big`, and reaction rules with `react`, where `->` separates redex and reactum. BigraphER natively supports *bigraphs with sharing* (a node may reside under multiple parents), *stochastic reaction rules* simulated via Gillespie's Stochastic Simulation Algorithm (SSA), and *priority classes* that order rule application. Matching is encoded as a SAT problem and solved by MiniSat [9]. BigraphER performs exhaustive state-space exploration and checks predicates expressed as pattern matches at every discovered state; for quantitative verification, it exports the resulting labeled transition systems and continuous-time Markov chains to the PRISM probabilistic model checker. Like BigMC, BigraphER is a command-line tool; it has no graphical editor, no interactive evolution control, and no checkpoint navigation.

Bigraph Framework. The Bigraph Framework [5] is a Java library with a programmatic API for constructing, composing, rewriting, and verifying bigraphs. Bigraphs are built through a fluent builder API; the underlying metamodel is based on the Eclipse Modeling Framework (EMF), with instance models serialized as XMI and metamodels as Ecore files [5]. Matching and rewriting are delegated to jLibBig, and the framework includes a dedicated sub-hypergraph isomorphism solver for link-graph matching. On top of rewriting, it offers built-in model-checking capabilities: breadth-first, depth-first, random, and simulated-annealing exploration of the state space, automatic construction of labeled transition systems, and predicate-based property checking. It also provides converter modules for interoperability with BigMC, BigraphER, and BigRED file formats. The Bigraph Framework serves as the computational backend of BigraphIDE, which delegates bigraph modeling, matching and rewriting to it. For the purpose of this work, we extended the framework with *mutable bigraphs* so that structural updates can be applied in place instead of rebuilding the whole bigraph after each operation. Despite its rich API, the framework provides no graphical construction editor and no interactive evolution controls; all modeling and analysis has to be done in Java code.

Summary. Existing bigraph tools tend to separate modeling from execution: users write specifications in one tool and run the analysis in another, or they build bigraphs programmatically and invoke rewriting from code. Bigellor provides a web-based visual editor without rewrite-rule modeling or model checking; BigRED adds rewrite-rule authoring and delegates execution to BigMC, but offers no evolution control in the IDE itself. BigMC and BigraphER provide command-line model checking and rewriting, but no visual modeling. The Bigraph Framework offers a rich programmatic API with built-in model checking, but requires Java code and has no graphical front-end. No existing bigraph tool combines visual rewrite-rule authoring (by drag-and-drop), interactive evolution with play/step/pause control, checkpoint-based branching rendered as an evolution tree, and on-the-fly reachability checking with user-defined stopping patterns—this is the gap BigraphIDE fills. Table 1 compares the tools on four features.

Several tools are marked as supporting verification, but they do so in different ways: BigRED delegates checking to BigMC; BigMC, BigraphER, and the Bigraph Framework explore state spaces exhaustively (BigraphER also supports stochastic simulation); BigraphIDE matches patterns interactively along the user-guided evolution path.

Table 1
Feature comparison of bigraph tools. *Visual Editor*: graphical construction of bigraphs. *Rewrite Rules*: definition and application of reaction rules. *Evolution Control*: interactive play/step/pause execution. *Verification*: model checking or property checking. A checkmark (✓) indicates native support; a dash (–) indicates the capability is absent; (∗) indicates support only through delegation to an external tool.

Tool	Visual Editor	Rewrite Rules	Evolution Control	Verification
BigRED [8]	✓	✓	–	(∗)
Bigellor [6]	✓	–	–	–
BigMC [7]	–	✓	–	✓
BigraphER [9]	–	✓	–	✓
Bigraph Framework [5]	–	✓	–	✓
BigraphIDE	✓	✓	✓	✓

3. BigraphIDE: Architecture and Workflow

This section presents the design goals of BigraphIDE, its system architecture, and the workflows for modeling bigraphs, defining rewrite rules, and running evolution projects.

3.1. Design Goals

The design of BigraphIDE is guided by three groups of goals.

Modeling. The editor must support visual construction of all bigraph constituents—nodes with controls and arities, edges, inner and outer names, sites, and multiple roots—so that every bigraph expressible in the Bigraph Framework [5] can be built graphically, and all model artifacts must remain file-based and transparent for reproducibility.

Rewriting and evolution. Rewrite rules should be visually modeled bigraph pairs (L, R) organized in named rule sets, and evolution runs must be reproducible—with configurable rule sets, verification patterns, and play/step/pause controls—and fully traceable through an operation history and checkpoint-based replay.

Integration and usability. Visual modeling, rewriting, evolution, and pattern-based reachability checking should live in one environment to avoid tool switching; delivering BigraphIDE as a VS Code extension reuses a familiar ecosystem and keeps onboarding low for researchers and students.

3.2. System Architecture

BigraphIDE is implemented as a VS Code extension that registers a dedicated view container in the VS Code activity bar. Diagram editing is provided by the Eclipse Graphical Language Server Platform (GLSP) [4], whose server component runs as a separate Java background process and communicates with the extension over a TCP socket, the default transport for GLSP. The computational backend is the Bigraph Framework [5], a Java library that handles bigraph operations, rewrite-rule matching, and evolution execution. For BigraphIDE, we extended the framework with *mutable bigraphs*. The original framework represents bigraphs as immutable objects, so every modification rebuilds the whole bigraph. The mutable variant applies single user operations—adding a node, creating a link—in place, without reloading the diagram after each change. All model artifacts are based on the Eclipse Modeling Framework (EMF) [10]: bigraph models, signatures, and metamodels are stored as XMI and Ecore files.

The view container exposes five sidebar panels. The *Bigraph Explorer* lists workspace bigraph files; a "+" button creates a new one, and each entry offers a *verification* button (assigning the bigraph as a verification pattern) and a *composition* button (computing the composition or tensor product with the bigraph open in the editor). The *Rewrite Rule Explorer* manages rule sets and rules, and the *Rewrite Rule Editor* is a control panel with two slots (redex and reactum) into which bigraphs are dragged to assemble a rule pair. The *Evolution Manager* configures and controls evolution runs (Section 3.5), and the *Evolution Project List* lists evolution projects, with a "+" button to create one.

Each workspace bigraph is represented by four files. Three are produced by the Bigraph Framework following its EMF-based persistence scheme—a `.xmi` file (the bigraph model), a `.signature.xmi` file (the signature: controls with names, arities, and kinds), and a `.signature.ecore` file (the EMF metamodel)—while the fourth, a `.bigraph-meta` file introduced by BigraphIDE, stores visualization metadata such as canvas positions and relative coordinates for nested nodes.

3.3. Modeling Bigraphs

All modeling operations described in this subsection are executed against the Bigraph Framework running inside the GLSP server. Each user action on the canvas (e.g., creating a node, connecting ports, nesting elements) is translated by the GLSP connector into a framework-level mutation on the mutable bigraph instance, which ensures that the diagram always reflects a valid bigraph state.

To begin modeling, the user opens a workspace folder in VS Code. The BigraphIDE view container appears in the activity bar, and the Bigraph Explorer lists any existing bigraph files. Clicking the "+" button in the explorer opens a name dialog; upon confirmation, a blank bigraph (four files) is created and immediately opened in the GLSP diagram editor. The editor displays a single prepared root together with the modeling toolbar.

The toolbar is organized into five sections:

- **Header:** provides general tools, including selection, node deletion, resetting the bigraph layout, and fitting the view to the canvas.
- **Place Graph:** offers available controls as selectable buttons, along with a site element; a "+" button enables the definition of new controls.
- **Link Graph:** contains tools for creating connections, including edges, outer names, and inner names.
- **Layout:** provides five automatic layout strategies: Std (layered according to bigraph conventions), Layer, Tree (MrTree), Force, and Stress.
- **Export**: supports exporting the current diagram as an SVG.

Before placing nodes, the user defines controls: clicking "+" in the Place Graph section opens a dialog for a control's name, arity, and kind (atomic, active, or passive), after which the control appears as a placement button. Nodes are then placed by selecting a control in the toolbar and clicking the canvas; nesting is done by dragging a node into a parent node or site; and links are drawn with the Link Graph connection tool, with sites, outer names, and inner names placed analogously. Place-graph (nesting) connections are rendered in blue and link-graph (linking) connections in green, and all named elements can be renamed via a right-click context menu. The Bigraph Framework validates model consistency at each operation, so invalid constructions (e.g., arity violations or illegal nesting) are rejected with an error message surfaced to the user.

3.4. Rewrite Rules

Rewrite rules in BigraphIDE are first-class artifacts. The user models the left-hand side (redex) and right-hand side (reactum) as separate bigraphs using the standard GLSP editor described above, and saves them.

The Rewrite Rule Explorer, located below the Bigraph Explorer, manages rule sets: a "+" button creates a new rule set and a nested "+" a new rule, which opens a rule editor webview with two slots

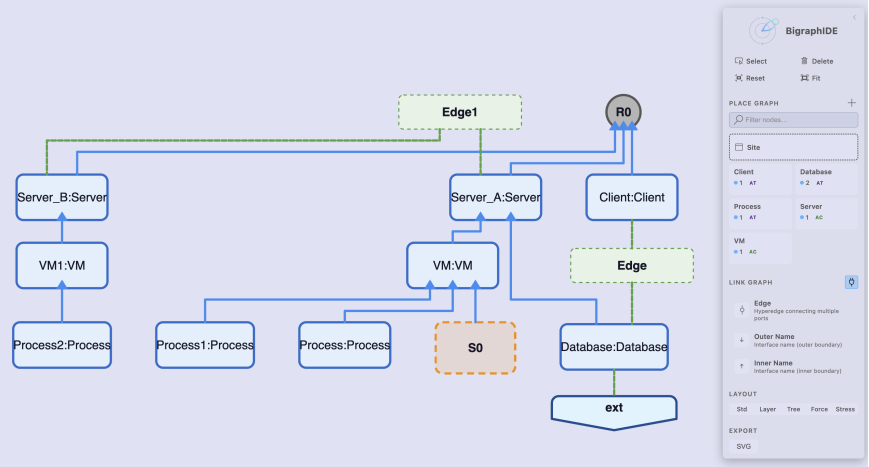


Figure 2: The BigraphIDE modeling editor showing a bigraph with multiple nodes, nesting, and links. The toolbar on the left provides access to all modeling operations.

into which the user drags the redex and reactum bigraph files from the Bigraph Explorer. After naming the rule and saving, the workspace configuration records the (L, R) pair; clicking a rule's entry later reopens both diagram tabs as well as the rule editor. When a rule is applied during evolution, the Bigraph Framework performs the matching and rewriting, replacing an embedding of the redex in the host bigraph with the reactum.

3.5. Evolution Projects

An evolution project drives the repeated application of rewrite rules to a workspace bigraph. To create one, the user clicks "+" in the Evolution Project List, which opens the Evolution Manager in the sidebar.

The Evolution Manager provides the following controls: a text input for the project label; the workspace bigraph path, which is auto-populated from the currently focused bigraph tab; a rewrite-rule set selection with per-rule active/inactive toggles; a rule-application strategy selector (*first-first* or *round-robin*); a verification zone for assigning a bigraph pattern as stopping condition (detailed in Section 4); a checkbox with numeric input for a maximum-operations limit; a checkbox for checkpoint file generation; a checkbox to visualize intermediate steps; *play* and *step* buttons; a *pause* control (available while a *play* run is active) that suspends the evolution process until the user resumes; an evolution history panel; and an evolution tree visualization.

BigraphIDE supports *play* and *step* execution, while a *play* run can be *paused* and resumed. In *step* mode, each user click performs exactly *one* evolution step: a single rule application under the current configuration of active rules (i. e., one transition). Although bigraph rewriting is nondeterministic in general (Section 2.2), BigraphIDE resolves this nondeterminism deterministically so that each step follows a single, reproducible path. When several active rules are applicable, the engine scans the rule set in its configured order and applies the first matching rule; the rule-application strategy governs the scan order, where *first-first* always restarts the scan from the first rule, whereas *round-robin* rotates the starting rule after each application so that rules are tried in turn. When the redex of the chosen rule has more than one embedding in the current bigraph, the first match reported by the matcher is rewritten. A step thus applies exactly one rule at exactly one match site; branching is never introduced automatically and arises only when the user continues evolution from an earlier checkpoint. In *play* mode, the engine runs a loop: it keeps applying rules until the user *pauses*, no active rule can match, a configured *verification* bigraph matches the current state (on-the-fly pattern match and stop), or the optional maximum-operations limit is reached. While *play* is running, the user can *pause* at any time; resuming continues from the current state. After a pause, the evolution can be continued. Between steps, users may toggle individual rules active or inactive via a dedicated chip in the rule list, or remove rules entirely, adapting the strategy on the fly.

The evolution tree is rendered as a visual tree in the sidebar. Each node represents a bigraph state corresponding to a checkpoint. Clicking a tree node restores the workspace bigraph to that state in the editor. From any earlier checkpoint, the user can continue evolution and so create new branches in the tree, which supports exploring alternative evolution paths.

When checkpoint generation is enabled, each evolution step produces checkpoint files in a dedicated subfolder. This subfolder also contains a copy of the original workspace bigraph and an evolution-project JSON file that stores the full operation history, the current checkpoint cursor, the rewrite-rule index, and the verification-bigraph index. The evolution history panel displays a chronological list of all executed operations, recording the applied rule and a timestamp for each entry. Entries are shown in blue when inactive and highlighted in yellow for the currently selected checkpoint.

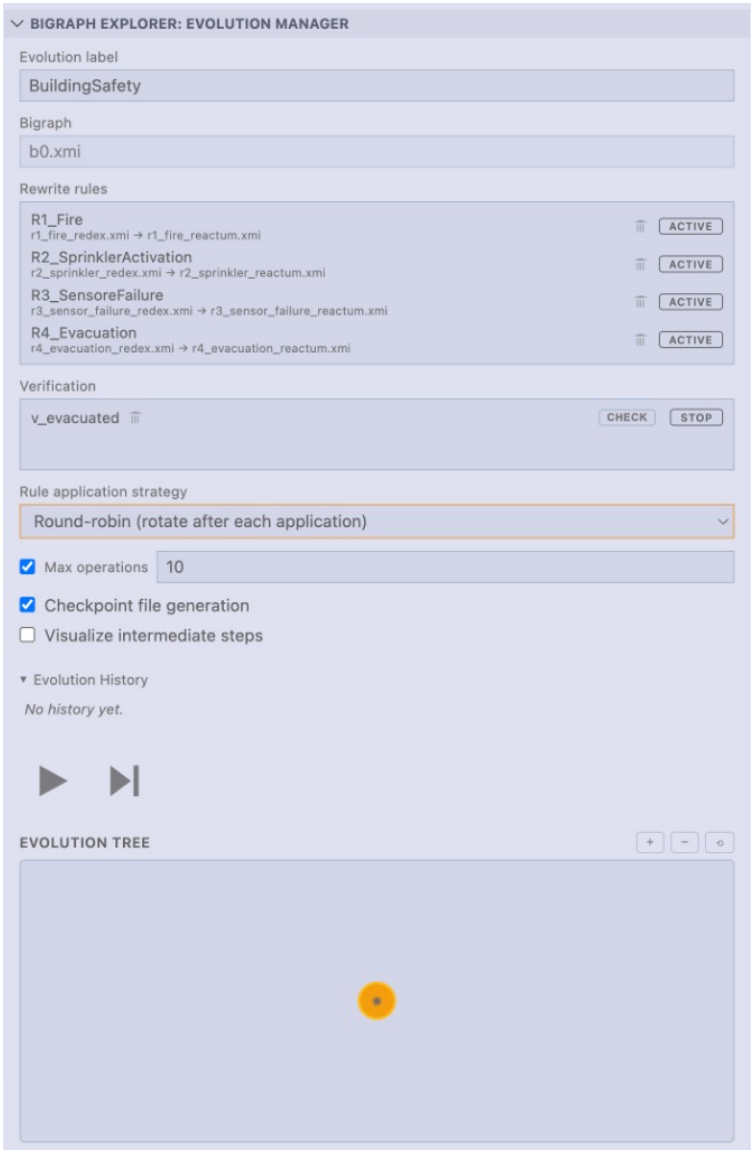


Figure 3: The Evolution Manager showing the configuration panel, evolution history, and the interactive evolution tree. Tree nodes represent bigraph states; clicking a node restores that state in the editor.

4. Interactive, Pattern-Based Reachability Checking

A central feature of BigraphIDE is that property checking runs inside the evolution loop rather than as a separate, offline step. Correctness properties are expressed as *bigraph patterns*, which are matched against the evolving system, enabling on-the-fly checking along the explored evolution path as well as

on-demand checking at individual states. We describe this as *guided, pattern-based reachability checking*: BigraphIDE does not perform an exhaustive exploration of the state space, but checks user-defined patterns against the states actually visited during a guided evolution run. Exhaustive state-space exploration remains available through the Bigraph Framework backend [5], but is not the focus of the interactive workflow presented here.

A bigraph pattern for verification purposes is a pure bigraph that acts as a redex, i. e., a pattern matched against the current workspace state. It is created with the same GLSP modeling workflow as any other bigraph; no separate specification language or formalism is required. Semantically, a verification check is a bigraph matching operation: *does this pattern have an embedding in the current bigraph state?*

This matching operation connects to correctness properties as follows. A bigraph pattern P characterizes the set of states in which P has an embedding, so checking P along an evolution run expresses a property over the explored states. A target pattern such as $V_{evacuated}$ (Section 5.1) encodes a *reachability* property—a desired configuration can be reached—and the run halts at the first witnessing state. Dually, a pattern describing an undesirable configuration, such as the hazard pattern V_{danger}, encodes a *safety* property: the absence of a match along the explored path witnesses that the hazard was avoided, whereas a match exhibits a concrete counter-example state.

To assign a bigraph pattern to an evolution project, the user clicks the verification button on the corresponding entry in the Bigraph Explorer. The bigraph then appears in the verification zone of the Evolution Manager. Multiple bigraph patterns can be assigned to a single project.

Once assigned, the same bigraph pattern can be evaluated in two complementary ways. During evolution (in play or step mode) only *active* bigraph patterns are considered after each rewrite step: the verification zone provides a per-pattern toggle, and evolution stops if any active pattern matches (i. e., an embedding of the redex is found in the workspace bigraph). Inactive patterns are not used for automatic stopping but remain available for on-demand checking. This realizes a form of on-the-fly reachability checking: evolution continues until a specified target pattern is reached, at which point the tool reports the witnessing state. The evolution tree, the operation history, and any generated checkpoint files reflect the state at which the stop occurred, so traceability is preserved up to the point of termination. In this mode, bigraph patterns act as reachability-based stopping conditions that steer the evolution towards (or away from) states of interest.

Outside of an active evolution run, the same bigraph patterns can be checked on demand. The prerequisite is that a bigraph is open in a VS Code tab and an associated evolution project is selected. Each bigraph pattern listed in the verification zone offers a dedicated *check* button. Clicking it runs the pattern match and shows the result as an icon: a green check mark if the pattern matches (the redex has an embedding), a red cross otherwise. The result of each on-demand check is persisted in the evolution-project JSON file, linked to the current checkpoint. When the user navigates back to that checkpoint later, the stored result is displayed immediately without re-running the check. This caching lets the user inspect the verification status across the entire evolution tree without recomputing matches.

5. Evaluation, Discussion, and Future Work

5.1. Case Study: IoT Building Safety Monitoring

To evaluate BigraphIDE, we model an IoT-enabled building safety system in which smoke detectors, sprinklers, and a network controller interact to protect building occupants. We pick this scenario because it combines deep spatial nesting with cross-boundary connectivity between IoT devices, so it covers all major features of the tool.

Signature. The signature defines seven controls (Table 2). All controls are defined interactively through the BigraphIDE toolbar. With seven controls, the Place Graph section of the toolbar is fully populated and each control is available as a one-click placement button.

Table 2
Signature for the IoT building safety case study.

Control	Kind	Arity	Represents
Building	active	0	Top-level container
Room	active	1	Room; port connects to IoT network
Sensor	atomic	1	Smoke detector; port connects to IoT network
Sprinkler	atomic	1	Fire sprinkler; port connects to IoT network
Person	atomic	0	Building occupant
Smoke	atomic	0	Presence of smoke/fire
Controller	atomic	1	Central IoT controller; port connects to IoT network

Initial bigraph B_0. The initial model B_0 is a building with a single, unprotected room—no sprinkler—containing two persons and a smoke sensor, plus a central controller placed in the building outside the room. The sensor and controller share an IoT-network edge (green link) for communication. Without a sprinkler, fire cannot be suppressed automatically, forcing the system to rely on evacuation. This setup was built in BigraphIDE in under two minutes.

Rewrite rules. Four rewrite rules model the reactive behavior of the system:

- **R1 (Fire outbreak)**: A room that contains a site (allowing existing occupants) gains a *Smoke* node. The rule is not one-shot: a later match can still add another *Smoke* in the same room.
- **R2 (Sprinkler activation)**: In a room containing both *Smoke* and a *Sprinkler* connected to the IoT network, the *Smoke* node is removed. Models automatic fire suppression.
- **R3 (Sensor failure)**: A *Sensor* connected to the network is removed from a room. Models device malfunction.
- **R4 (Evacuation)**: A *Person* inside a room that contains *Smoke* is moved out of the room (placed in the building, outside any room). Models emergency evacuation.

Each rule was modeled as a separate redex–reactum pair in the GLSP editor and assembled via drag-and-drop in the Rewrite Rule Explorer into a single rule set named *BuildingSafety*.

Bigraph Patterns for Verification. Two verification patterns are defined to encode safety properties:

- V_{danger}: A *Room* containing both a *Person* and *Smoke* simultaneously. This pattern represents an immediate threat to occupant safety. It is configured as a non-stopping check, used for manual inspection at individual checkpoints.
- $V_{evacuated}$: Two *Person* nodes placed directly inside the *Building* (outside any room). This pattern detects that all occupants have been evacuated and is configured as a stopping condition.

Both bigraph patterns were modeled using the standard editor and assigned to the evolution project via the verification panel in the Evolution Manager.

Evolution run. The evolution project is configured with all four rules active, $V_{evacuated}$ as the stopping condition, and checkpoint generation enabled, and is driven in *play* mode. Starting from B_0, pressing *play* produces the following five-step trace:

- **Step 1 (R1—Fire)**: Smoke appears in the room (now holding two persons, a sensor, and smoke).
- **Step 2 (R3—Sensor failure)**: The sensor is removed; R2 (sprinkler activation) is skipped, as the room has no sprinkler.
- **Step 3 (R4—Evacuation)**: The first person leaves the room into the building; $V_{evacuated}$ does not yet match (it needs both persons outside).
- **Step 4 (R1—Fire persists)**: A further *play* cycle reaches R1 again, adding a second smoke event.

- **Step 5 (R4—Evacuation)**: R2 and R3 are skipped, and R4 evacuates the second person into the building; $V_{evacuated}$ now matches and **evolution stops**.

The evolution tree shows six nodes ($B_0 \longrightarrow\!\!\triangleright \ldots \longrightarrow\!\!\triangleright B_5$), with B_5 marked as the termination point. The trace illustrates a cascading failure scenario: fire breaks out in an unprotected room, the sensor fails, and repeated evacuation is the only remaining response until all occupants are safe.

Pattern-based Reachability Checking. We focus on standalone pattern matching (Section 4): at the final checkpoint B_5, $V_{evacuated}$ *matches* (green) as expected, since it was the stopping condition, while V_{danger} *does not* (red cross), confirming the hazard is absent in the evacuated state. Re-checking at an earlier checkpoint such as B_1 would instead show V_{danger} matching and $V_{evacuated}$ not yet, illustrating that any pattern can be re-evaluated at any checkpoint.

Summary. This case study exercises all major features of BigraphIDE end-to-end—signature definition, visual modeling with nesting and link connectivity, drag-and-drop rule authoring, *play*/*step* evolution with checkpoint generation, evolution-tree navigation, reachability checking with persisted results—within a single integrated environment. The whole model (signature, initial bigraph, four rules, two verification patterns) was built graphically, without a single line of code or algebraic specification, and the entire study was completed in roughly ten minutes. The five-step trace exercises three of the four rules while R2 (sprinkler activation) is correctly skipped when its preconditions are not met, and on termination the evacuated state can be inspected directly on the canvas rather than parsed from textual output, never leaving VS Code.

5.2. Discussion

BigraphIDE offers several strengths over existing workflows. By integrating modeling, rewriting, evolution, and verification into a single VS Code environment, it eliminates the context switching that characterizes tool chains built from disjoint command-line utilities. Bigraph models, signatures, rewrite rules, evolution histories, and checkpoints are stored as file-based formats (XMI, Ecore, JSON), supporting reproducibility and version control. Embedding the tool in VS Code leverages a widely adopted editor, lowering the onboarding barrier for both researchers and students. The evolution tree with checkpoint navigation enables branching exploration of alternative evolution paths and deterministic replay from any recorded state. Finally, bigraph patterns for verification reuse the same graphical modeling workflow as regular bigraphs.

Several limitations remain. The current rule-application strategies are limited to first-first and round-robin over active rules, each applying a single rule at the first match found; no heuristic, random, or priority-based strategies are available yet. When a bigraph pattern matches during evolution, the entire process stops—there is no per-branch stopping or partial continuation. Furthermore, the user interface does not yet indicate *which* bigraph pattern triggered the stop, and signature compatibility between the workspace bigraph and the bigraphs used as rules or patterns is not checked automatically. For large or densely connected bigraphs, diagram layout and readability may degrade. State-space explosion in nondeterministic evolution scenarios remains a fundamental challenge, as no pruning heuristics are currently implemented.

Architecturally, the computational backend is, in principle, exchangeable: because BigraphIDE talks to the Bigraph Framework through the GLSP action protocol and a fixed set of operation handlers, replacing it with a different bigraph library would require only re-implementing these handlers.

Regarding threats to validity, evolution outcomes depend heavily on the quality of the rule set and the specificity of the verification patterns. Our evaluation is a single illustrative case study exercising the full workflow end-to-end; it emphasizes clarity and multi-checkpoint traceability rather than raw performance, and we leave systematic benchmarking and user studies to future work.

5.3. Future Work

The existing step and *play* scheduling will be complemented by additional rule-selection and exploration strategies (random, priority-based, bounded, and goal-directed), with a strategy plugin architecture for user-defined or domain-specific strategies. To mitigate state-space explosion, we plan to integrate pruning and reduction techniques, such as causation-based pruning [7] and bounded or goal-directed exploration. We also intend to support generalizations of bigraphical reactive systems studied in the literature, such as bigraphs with sharing and stochastic or probabilistic reaction rules. On the verification side, planned improvements include more granular stopping behavior, identification of which bigraph pattern triggered a stop, richer verification semantics (invariants, forbidden patterns, reachability and safety properties), and automatic signature-compatibility checking to catch mismatches before evolution begins. On the user-experience side, planned features include user-defined node icons and colors as well as richer diff views between successive states.

Declaration on Generative AI

During the preparation of this work, the authors used Cursor (with the Claude language model) for grammar and spelling checks and editorial assistance. After using these tool(s)/service(s), the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. Milner, The Space and Motion of Communicating Agents, Cambridge University Press, 2008.

[2] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: J. L. Pfaltz, M. Nagl, B. Böhlen (Eds.), Applications of Graph Transformations with Industrial Relevance, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2004, pp. 446–453. doi:10.1007/978-3-540-25959-6_35.

[3] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova, Modelling and analysis using GROOVE, International Journal on Software Tools for Technology Transfer 14 (2012) 15–40. doi:10.1007/s10009-011-0186-x.

[4] Eclipse GLSP Project, Eclipse graphical language server platform (glsp), 2023. URL: https://www.eclipse.dev/glsp/.

[5] D. Grzelak, Model-oriented Programming with Bigraphical Reactive Systems: Theory and Implementation, Ph.D. thesis, Technische Universität Dresden, 2024. URL: https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-910504.

[6] S. Tobies, Konzeption und Implementierung einer webbasierten Modellierungsumgebung für Bigraphen, Diploma thesis, Technische Universität Dresden, 2020.

[7] G. Perrone, S. Debois, T. Hildebrandt, A model checker for bigraphs, in: SAC '12: Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 1320–1325. doi:10.1145/2245276.2231985.

[8] A. Faithfull, G. Perrone, T. Hildebrandt, Big Red: A development environment for bigraphs, Electronic Communications of the EASST 61 (2013). doi:10.14279/tuj.eceasst.61.835.

[9] M. Sevegnani, M. Calder, BigraphER: Rewriting and analysis engine for bigraphs, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification (CAV 2016), Part II, volume 9780 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 494–501. doi:10.1007/978-3-319-41540-6_27.

[10] Eclipse Foundation, Eclipse Modeling Framework (EMF), 2025. URL: https://eclipse.dev/emf.