University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

A Strategy Language for Bigraphs

Adam Shannon

March 21, 2024

Abstract

Bigraphs are a useful graphical modelling formalism with varied applications including in networks and cyber-physical security which can sometimes be difficult to work with because of the strict limits imposed by how they evolve using exclusively non-deterministic application of simple rewriting rules. This work presents a strategy language for Bigraphs to implement common controls on rule applications such as sequencing and conditionals, the language is defined with a minimal and extended syntax along with semantics describing how it operates and is implemented in an interpreter compatible with other popular Bigraph tools. Evaluation across several example models has shown how strategies can encapsulate or work in tandem with other control extensions and how we can simplify bigraphical systems by reducing the number of rules or simplifying their definition to improve usability through the natural expressiveness of strategies.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Adam Shannon Date: 02 February 2024

Contents

1 Introduction

Bigraphs are an abstract graphical modelling formalism originally proposed by Milner and defined rigorously in [Milner (2009)] which are used across numerous contexts such as wireless network modelling [Calder et al. (2014)] or in expressing evolutionary systems like the game of life [Damgaard (2005)] as well as in other systems which have a duality in the types of connectivity between entities. Bigraphical reactive systems (BRS) combine bigraphs with a rewriting logic where simple reaction rules $L \rightarrow R$ can be used to match and replace sub-entities within a bigraph and are applied either non-deterministically, stochastically or probabilistically to model the evolution of a given system.

Modelling complex systems with rewriting logic can be difficult because of a lack of fundamental control over the context of how rules are applied, for example, there is no way to explicitly apply two rules in sequence or to condition rules based on the application of others. Other rewriting logic's typically solve this lack of control by the creation of a minimalist domain-specific programming language referred to as strategy language which uses popular imperative methods like iteration and conditional statements on rules to provide a more natural means to express evolutionary models without relying on independent extensions or enforcing design limitations which are hard for users to follow.

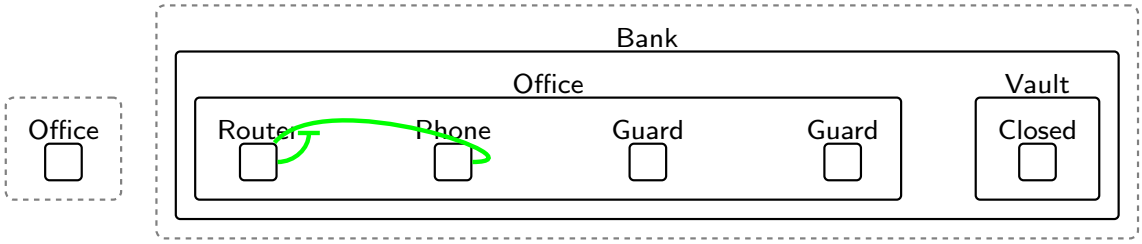


Figure 1.1: *Simple Bank and office Bigraph model*

The bigraph in Figure 1.1 provides a model of a bank containing **Vault** and **Office** entities along with a second office which exists externally to the bank in parallel. To reason about the physical security of the bank reaction rules are created to build a BRS such as in Figure 1.2 which moves a guard entity from the office to the vault if applicable to the current local state. The graphical notation incorporates both types of possible relation in a bigraph using the relative positioning of entities combined with explicit lines drawn between them and is completely equivalent to the formal algebraic representation.

In a standard BRS, the set of all available rules is applied non-deterministically to evolve the system step by step which does not allow for sequential or iterative rule applications that are often highly desirable in a general model. For example, moving all guards into the vault simultaneously in Figure 1.1 would in practice require careful design of all rules to ensure only the relevant one is applicable during the guards' movement phase, additional rules could also be required to handle guards outwith the office or a more general situation where the number of guards is not fixed which would inflate the model and add complexity even when using extensions like priority classes [Sevegnani and Calder (2016)] which enable some weak ordering.

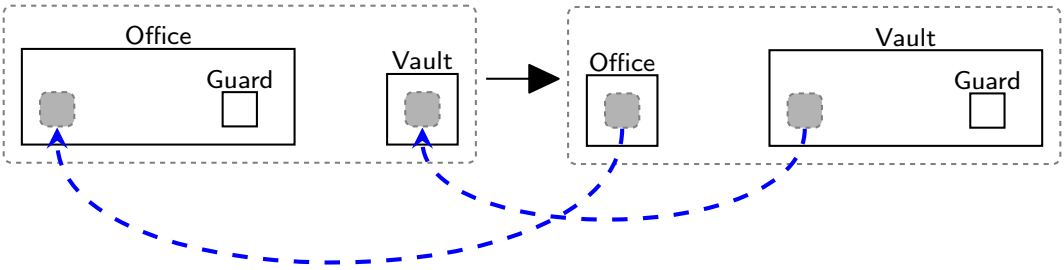


Figure 1.2: *Rule to move guard from the office to the vault*

Conversely, explicit sequencing of generalised rules such as in Figure 1.2 or use within a conditional iterator which enables exhaustive application could be deployed to model the complex dynamics of the model in a far more natural way without imposing any restrictions on the rest of the system. When combined with other rule control strategies like conditions on successful applications, both the model itself as well as its creation process can be simplified by abstracting complexity away from simple rewrite rules to common programming language constructs, justifying the creation of a minimalist strategy language for providing control over how rules are applied.

1.1 Contributions

Motivated by the deficiencies of the existing BRS infrastructure the desired contributions of a strategy language for bigraphs were clear.

- **Language definition** - A concrete definition of the strategy language with a minimalist syntax describing the available constructs and the language functionality.
- **Semantics** - Small step operational semantics defining how each strategy is executed in terms of a series of intermediate reduction steps.
- **Implementation** - A full implementation of an interpreter for the extended strategy language per the semantics and testing infrastructure to demonstrate correctness.
- **Usage examples** - exemplar BRS represented in the strategy language with a discussion of common program design patterns when using strategies and the benefits of strategies over simple rules in each example.

The interpreter was created to apply strategies in a linear sequence to restrict the scope of the implementation to be achievable within the available timeframe. This reduces the tree of possible evaluations at each execution step in the implementation to a deterministic sequence of applications but does not prohibit any strategy design in the language from existing in a full non-deterministic style BRS and reasoning about strategy functionality and design is equivalent in either case.

2 Background

2.1 Bigraphs and their properties

A Bigraph $B :< k, X >\rightarrow< m, Y >$, is a pair of graphs over the same set of nodes, a place graph represented by a forest which is a disjoint union of trees and a link graph represented by a hypergraph where each edge can join any number of nodes. The place graph is typically used to model the physical aspects of a system such as a series of rooms within a building while the link graph is intended for relations which may be temporal such as each room's connection to a shared wifi network providing a natural separation of concerns for the different types of connectivity in a system which can be seen in a more mathematically intuitive form in Figure 2.1.

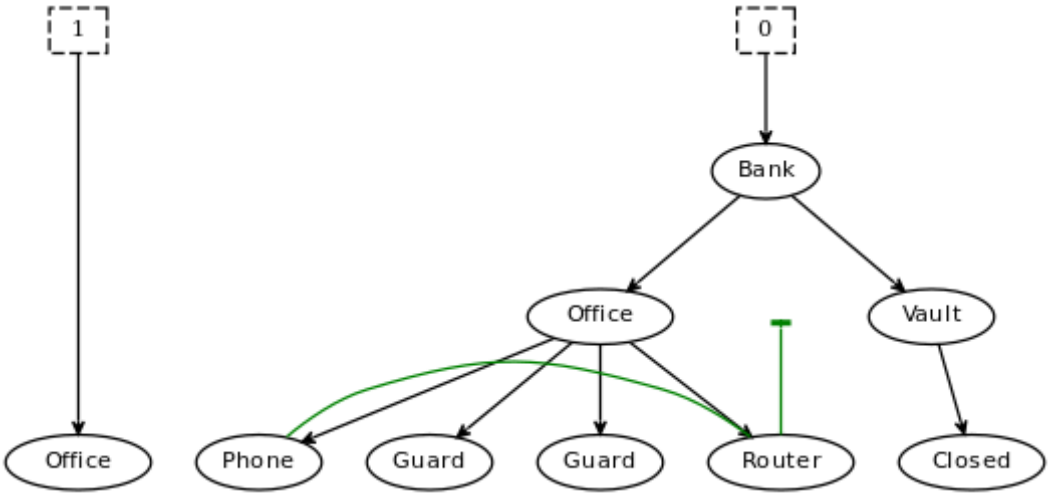


Figure 2.1: *Mathematically familiar representation of the bank bigraph*

Bigraphs allow for decomposition and are inherently nested structures where each node is itself another Bigraph, they contain k **sites** representing an abstraction of one or more unspecified Bigraphs and m **regions** representing known nested Bigraphs. The decompositional properties of Bigraphs also allow for any particular instance's place graph and link graph to be treated independently in the process of applying rules.

In Figure 1.1 a Bank and adjacent office buildings are presented in Bigraphical form. Place graph entities are represented by rectangles with solid outlines drawn containing any entities which are nested within them, the dotted lines represent regions and are used to denote the top-level boundaries of a system. The office within the bank contains two links: a closed link between a Wi-Fi router and a phone and a disconnected link connected only to the router.

2.1.1 Place graphs:

The place graph is used to model spatial relations between real-world entities for example a

building and several rooms. In Figure 2.2 the algebraic notation for Bigraphs uses the infix nesting operator "Bank.(...)" to represent parent-child relationships, the merge product operator to denote sibling entities "Guard | Guard" and the parallel product to separate a bigraph into regions "Bank.(...) || Office" which are the roots of each tree in the forest.

$/a/b(\text{Bank}.(\text{Office}.(\text{Router}\{a,b\}|\text{Phone}\{b\}|\text{Guard}|\text{Guard})|\text{Vault}.\text{Closed})||\text{Office}.1$

Figure 2.2: *Equivalent algebraic form of the Bank example*

The place graph's interfaces are described firstly by the roots referred to as outer entities or regions in the forest which represent the top-level parallel components of a given modelling context such as the Bank and external Office in Figure 1.1. The other part of the place graph interface is the inner entities referred to as sites, a site within a bigraph is a generalisation denoting that one or more unspecified bigraphs exist at a location and is mainly used for writing reaction rules in a general way without having to over specify the matching instance.

There are two special place graphs, 1 which denotes an entity with no children and *id* which denotes a site in the algebraic notation, these are used exclusively for defining bigraphs and are not equivalent to the **Id** strategy in section 3.2 or identity rules which typically have a similar notation. A node with no children (aside from 1) is referred to as atomic, although several types of other entities may exist we will refer to only atomic and simple non-atomic entities in the scope of this analysis although any entity type which is valid in the standard definition of a bigraph will retain its validity in the strategy language.

2.1.2 Link graphs:

The link graph models any non-spatial relations in a given bigraph such as the connection between a series of devices and a wifi router. Each entity in the link graph has several ports which allow participation in connections, the number of ports is called the arity and places a fixed limit on the number of links each node in the link graph can maintain where links themselves are hyperedges each associated with a set of interconnected ports.

Much like the inner and outer interfaces of the place graph, the link graph has inner names and outer names referred to as open and closed links respectively. Open links have meaningful names and refer to a connection with some general entity outside the visible scope of the model analogous to sites in the place graph. The closed links are more common and represent well-defined links between entities such as $/a/b$ in Figure 2.2 which connects the **Router** and **Phone** entities and is closed algebraically under the region containing the Bank, the link *a* is referred to as disconnected containing only a single connection to the **Router** entity, any arbitrary link may also be idle where it exists but is not related to any entity in the bigraph. Identifiers for a link in the algebraic notation are simply placeholders which do not uniquely identify a link itself, practically this implies that no link in a bigraph can be distinguished in the process of applying reaction rules.

2.1.3 Bigraphical Reactive Systems

A Bigraphical reactive system (BRS) makes use of simple reaction rules to rewrite an initial state bigraph so it can evolve in an exploratory manner, rule application can modify both the link and place graphs of a system and in the standard BRS definition may produce several possible state transitions for the system being modelled at any given application attempt.

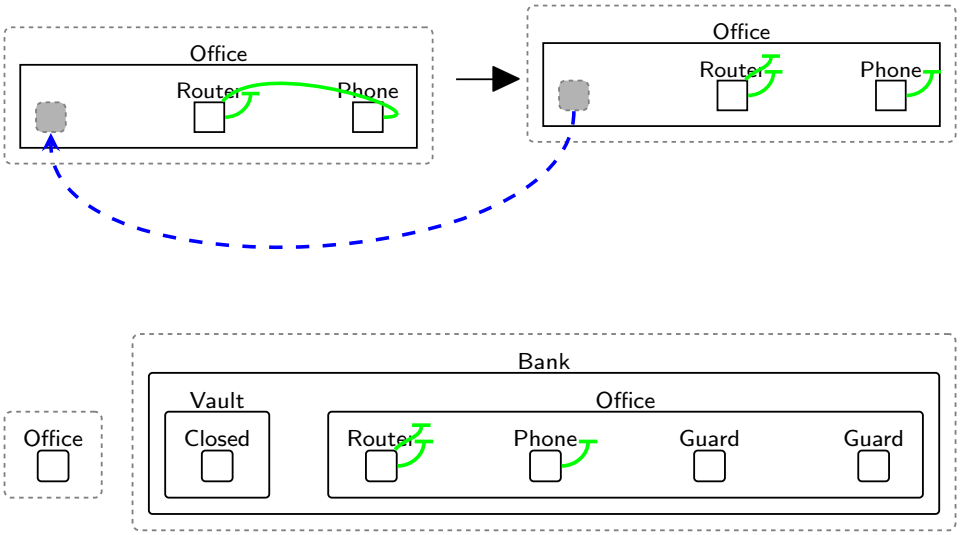


Figure 2.3: *result of a single rewriting application for a rule which disconnects the phone and router entities in the bank model.*

Reaction rules:
Figure 2.3 demonstrates the application of a simple reaction rule for the Bank analogy used to disconnect a phone from the router, the left-hand side L captures the **Office** within the bank including an additional site inserted to generalise the rule to be applicable even if other entities are present in a given **Office** instance. A successful match of L will leave the phone and router no longer connected, each with a remaining disconnected link which could be useful for modelling a scenario where both are willing to accept connections from new devices.

Formally, a bigraph reaction rule $L \rightarrow R$ is executed by matching an occurrence ($B \models L$) of an arbitrary bigraph L nested within another bigraph B. If one or more occurrences exist the reaction rule will return a set of possible re-writes which substitute L with R. An occurrence is defined when B can be decomposed into $B = C \circ (L \otimes id_I) \circ D$ and C, D are an arbitrary context and parameter for the match representing the internal and external environment where L has been found. The algebraic form of bigraphs is included here to provide intuition about the mathematical form of relationships in a BRS but is mostly ignored in the definition of strategies which work at a level of abstraction above rules, thus the completely equivalent graphical form of bigraphs will be used throughout this work for simplicity.

2.1.4 BRS Extensions

There are a number of extensions to the standard BRS which aim to implement additional behaviours for simplifying the creation of bigraphical models including some controls on rules. The strategy language is designed to either maintain compatibility with or completely encapsulate and expand upon the behaviour of these extensions particularly those which implement weaker rule control mechanisms to avoid the extra limitations which they may impose on the BRS design process.

Priority classes:
In BigraphER [Sevegnani and Calder (2016)] rule priorities allow partial ordering of rule application in the evolution of a BRS. $\{A, B\} < \{C, D\}$ enforces that rules in priority class $\{A, B\}$ should only be applied if neither C nor D applies to the current state, this ordering is rigid so that rules in the second class will always try to execute before those in the first, does not allow for nested

classes and is constant at runtime so only provides a weak sequencing mechanism for rules.

Conditional reaction rules
The use of application conditions [Archibald et al. (2020)] allows for a rule to be applied based on the occurrence or absence of an arbitrary bigraph P in either the context C or parameter D of its decomposition. Each conditional reaction rule is a reaction rule coupled with a set of application conditions $\langle t, P, L \rangle$ where every application condition has to be true for the rule to execute. Conditional rules are useful for providing simple control of whether a rule should be applicable at the level of a specific decomposition but cannot be nested and often still require a restrictive priority structure to provide simple conditional context to a BRS at the expense of inflating the model with additional rules.

Other extensions
Other extensions to Bigraph behaviour don't focus on rule control so were not a direct influence on the strategy language's design but are nonetheless fully compatible with the final language specification and give a useful insight into other directions for improving BRS beyond the goals of strategies.

Bigraphs with sharing [Sevegnani and Calder (2015)] lets entities have several parents, this means that place graphs are represented by a directed acyclic graph rather than a forest and would allow for example a **Guard** entity in Figure 1.1 to be represented as having vision over multiple rooms within the bank. BigraphER also allows for entity and rule definitions to be parameterised by an integer to define a family of similar instances in a single command which is useful in combination with the iterative strategies presented in Figure 3.3 to reduce the complexity of models from the creators perspective.

2.2 Strategy Languages

Rewriting logic is a common paradigm for modelling complex systems using a set of rules which describe how abstract entities such as graphs or terms are transformed to determine the evolution of a local state, rules are often applied non-deterministically which can make modelling complex systems challenging in terms of both expressive capability and computational efficiency. To remedy the problems with simple rules many rewriting systems implement strategy languages which are small domain-specific programming languages designed to control how rules may be applied, strategies are generally analogous to common control structures in imperative programming such as conditionals and iterators which enable localised avoidance of non-determinism in re-writing systems and can greatly simplify models by allowing for a more natural expression of evolutionary actions within a given context.

2.2.1 Common functionality and behaviour

The strategy languages which inspired the creation of a strategy language for bigraphs such as PORGY [Fernández et al. (2012)], Maude [Eker et al. (2023)], Stratego [Visser (2001)] and ELAN [Castro and Borovanský (2000)] all perform either graph or term rewriting and hence share some common pre-defined strategies which were also integrated or adapted for usage with bigraphs.

Identity and fail
The simplest of these strategies are `Id` and `Fail` which always produce the same result and are most useful for the definition of other strategies, their existence results from the foundational idea of strategies as a way to respond to the success or failure of rewrite applications and consequently they are typically terminal at least for a particular branch in the derivation tree of a strategy program.

Sequences
Another core strategy idea present in every language is the concatenation operator defined with an infix "**;**" which is used to explicitly sequence strategies always failing to apply if any of the strategies in a sequence fail which is a desirable property for the creation of more advanced behaviour by locally suspending the natural non-determinism of rewriting logics. Other types of sequencing with nuanced behaviour are also popular and were a major focus when developing strategies for bigraphs, for example, the regular expression strategies in Maude which enable a sequence to be executed zero or more times until exhaustion or "`orelse`" in the extended syntax of PORGY which can be used to create partially executing sequences.

Conditional strategies
Traditional if-else style conditionals were also ubiquitous among the other languages and are an important building block for other strategies, especially in languages which allow user-defined strategies. In any model, this style of conditional is the most natural way to dynamically respond to the success of other strategies which is a particularly powerful behaviour for allowing a rule to only occur in a desirable context without placing limits on the design of the system as a whole.

Strategy level non-determinism
In contrast to explicit sequencing, each language also contains a strategy to execute a random applicable strategy from a set of arguments denoted by "`dc(S1, S2)`" in ELAN and an infix "`S1 | S2`" in Maude. The consistent presence of constructs which explicitly enable sequential or deterministic behaviour balanced with more ambiguous strategies emphasises a key design principle in the creation of strategy languages where the usefulness of strategies is impeded by an overly deterministic or linear approach which was helpful when considering which strategies were eligible for the minimum language for bigraphs.

Iteration strategies
Iterative strategies are also prevalent across different styles of strategy languages and are present in both conditional and unconditional forms usually denoted by `While` and `repeat`. Despite usually being defined as syntactic sugar for sequences or recursive conditional strategies, iteration is a valuable inclusion to the minimal syntax, much of the focus of strategies is to simplify rule or entity definitions but iterators provide simplification at the strategy level which is easy to ignore when designing strategy languages but particularly important in applied usage.

2.2.2 Strategy language Syntax

Much of the standard syntax adopted in section 3.2 is shared with other popular strategy languages and in particular with PORGY which is designed for general graph rewriting so closely matches the design goals of a strategy language for bigraphs. Generally, strategy language syntax adopts familiar constructs from imperative programming languages such as `repeat` and `while` for iteration and the common conditional structure of `if(...)then(...)else(...)` which is advantageous to usability by promoting intuitive familiarity but also an inherently natural format where only rules can mutate state and strategies are analogous to functions in popular programming languages defining the only the order of application.

A functional paradigm in the syntax could have been used instead to support the declarative nature of strategies however the prototype interpreter approaches strategy application as a strictly sequential process which makes an imperative design more natural and maintains consistency with other strategy languages.

3 Strategy Language - Heist

Heist is a domain-specific programming language designed with a minimalist approach to provide simplified and effective BRS modelling by placing controls on reaction rules referred to as strategies. The language is defined by the combination of its Syntax in section 3.2 denoting available commands and a small-step operational semantics in section 3.3 which determines how syntactic constructs are evaluated in the execution of a program and allows for reasoning about language properties.

3.1 Language design goals

The language itself is designed for abstraction and hence concrete non-functional requirements were difficult to define early in the development process and mostly resolved through experimentation, the main motivation for core language requirements was to take a minimalist approach to enabling as much expressivity and abstraction as possible.

- The core language should encapsulate or maintain compatibility with as many BRS control extensions as possible.
- The minimal syntax should abstract design complexity away from reaction rules and user-defined strategies should not restrict the functionality of others.
- Strategies should focus on enabling control flow by providing explicit sequencing, conditional branching and iteration over rules compatible with a standard BRS.
- The Language should be easily extensible to account for common design patterns being introduced as core strategies in future.

3.2 Syntax

In the definition of the syntax n denotes an integer in $\mathbb{N}$, *name* denotes a user-defined identifier, and α,β denote reaction rules and conditional reaction rules respectively.

Strategy (S)::=	Definition (D) ::=
| **Id** | **Fail**	| **strat** *name* := S
| S; S | S?S	
| α | β	
| **if**(S_1)**then**(S_2)**else**(S_3)	Application (A) ::=
| **while**(S_1)	| **do** *name*
| **Any**($S_1, \ldots, S_n$)	| **do** S

Figure 3.1: *Minimal strategy language syntax*

The syntax was designed to be intuitive for general users through the use of popular constructs like **While** and **for** which are ubiquitous in general programming languages as well as other strategy languages promoting usability.

A minimalist approach to achieving the core design goal of control flow was taken where the strategies in Figure 3.1 are intended to represent foundational behaviours used to create other more useful strategies as required. Originally the syntax was defined such that strategies and explicit applications like rules were separated, however, a more natural approach was to treat rules themselves as strategies so that all possible executions could be treated equivalently and associated with a name to allow for modular user-defined strategies which promote code re-use and other generally positive design practices. Core strategies in the minimal syntax include sequencing, strategy-level conditionals and conditional iteration as well as partial execution and a non-deterministic `any` wrapper for a set of executable strategies, each was included to provide distinct core functionality and to combine with the others to facilitate advanced strategy design patterns for easy expression of behaviour difficult to implement in a standard BRS.

3.3 Small step operational semantics

The language's small-step operational semantics describe how each syntactic construct is evaluated by the interpreter in the process of executing a program. Small-step semantics are syntax-oriented semantics written to represent transitions between intermediate steps in the reduction process for the execution of a computation. They were chosen because although the complexity of other semantic styles such as denotational semantics may make it easier for language properties to be proved mathematically, the constructs in Heist are designed for simplicity and are intended to demonstrate the utility of strategies where only simple properties like correctness and termination are useful.

$$\text{exampleRule}\ \frac{\textbf{predicates}}{\textbf{< expr, Env >} \rightarrow \textbf{< expr', Env' >}}$$

Figure 3.2: *Sample semantic rule syntax*

The inference rule in Figure 3.2 demonstrates the general form of the logic in small-step semantics where we have a relation between two states called configurations via the execution of an expression. The general rule should be read such that the transition between configurations occurs given that the predicates hold, the predicates are other reduction rules or logical conditions reasoned on the result of applying the initial expression, where inference rules with no predicates are axiomatic and can be chained together with other rules to prove language properties once the semantics have been defined.

A configuration $\langle S, (\sigma, B)\rangle$ describes the application of a strategy S to an initial bigraph B with a global lookup table σ associating user-defined strategies with string identifiers, since only calls and definitions in subsection 3.3.9 explicitly work with σ and evaluations may function independently a simplified notation of $\langle S, B\rangle$ is used to define individual strategies behaviour. A program in the strategy language sequentially evaluates a set of configurations invoked with the **do** command where the reduction $\rightarrow \subseteq$ **Configuration** $\times$ **Configuration** is a mapping between configurations for each step in a program.

3.3.1 Identity and Fail

The identity and failure strategies always denote that either execution along the current path has been completed successfully and no more sub-strategies exist be executed or that a failed application has occurred at an arbitrary point in the evaluation process so the initial state B

shouldn't be modified, **Id** and **Fail** both create terminal states in a given strategy evaluation but rather than an empty sequence with no more evaluations **Fail** returns a special constant denoting that a strategy was not applicable which is key to defining the behaviour of other strategies.

$$\text{Id}\ \frac{}{< \mathbf{Id}, B > \rightarrow < \quad , B >}$$

$$\text{Fail}\ \frac{}{< \mathbf{Fail}, B > \rightarrow < \ \text{fail} , B >}$$

Fail and identity are the language's most fundamental strategies included mostly for use in simplifying definitions of the reduction rules for the other strategies. The inclusion of identity and fail in the minimal syntax rather than as backend auxiliary functions was based partly on ubiquitous inclusion in the other strategy languages but also on the iterative nature of strategy development where exploring useful design patterns to form emergent strategies for future inclusion often relied on combinations of **Id** and **Fail** to create new behaviours.

3.3.2 Bigraph Reaction rules

Bigraph reaction rules are treated as strategies and generally may return a set of possible transforms or fail if not applicable. We let all bigraph reaction rules be implicitly non-deterministic so that rather than returning all possible rewrites a single evaluation is immediately executed and any strategy will be finite, terminating with either identity or failure. To clarify the notation of the semantics, a rule $L \rightarrow R$ will be written as α, a successful rule application will be written as $\rightarrow^{\alpha}$ and a non-applicable rule which does not modify B is written $/\rightarrow^{\alpha}$.

$$\text{RuleApplied}\ \frac{B \rightarrow^{\alpha} B'}{< \alpha, B > \rightarrow < \mathbf{Id}, B' >}$$

$$\text{RuleFailed}\ \frac{B /\rightarrow^{\alpha}}{< \alpha, B > \rightarrow < \mathbf{Fail}, B >}$$

3.3.3 Sequential application

$S_1; S_2$ is the sequential application of two strategies which may be the same, the configuration $\langle S_1, B \rangle$ is first evaluated to produce either **Id** or **Fail**, then the second strategy is applied to the result where the sequencing of the identity or failure strategy with any other strategy S_2 is either equivalent to applying S_2 alone or to failure.

$$\text{SequenceId}\ \frac{}{< \mathbf{Id}; S, B > \rightarrow < S, B >}$$

$$\text{SequenceFail}\ \frac{}{< \mathbf{Fail}; S, B > \rightarrow < \mathbf{Fail}, B >}$$

$$\text{Sequence}\ \frac{< S_1, B > \rightarrow < S', B' >}{< S_1; S_2, B > \rightarrow < S'; S_2, B >}$$

Since strategies are designed to be inherently compositional these rules can be extended to any number of strategies with the axiom that any composition of strategies containing a **Fail** must always evaluate to **Fail** which is a positive for efficiency allowing a reduction to stop prematurely when an intermediate step fails. This style of explicit sequencing derives from the common scenario where contextual actions in a model should occur simultaneously but the combination of multiple rules for each object into a single rule is over specific and limits reuse in a more general version of the model, sequences where every sub-strategy is expected to succeed are also fundamental to creating user-defined strategies warranting the inclusion of **;** in the minimal language.

3.3.4 Conditional Strategy

The conditional strategy $\mathbf{if}(S_1)\mathbf{then}(S_2)\mathbf{else}(S_3)$ functions by first applying S_1 then sequentially applying either S_2 or S_3 on the resulting production, this requires the use of an implicit auxiliary conditional operator in the language backend to check whether S_1 has failed which will be omitted from this semantic notation. The strategy effectively treats **Id** and **Fail** as analogous to true and false boolean results to explicitly condition a sequence of applications which provides the language's main way to achieve contextual applications within a model.

$$\text{CondPasses}\ \frac{}{< \mathbf{if}(\mathbf{Id})\mathbf{then}(S_2)\mathbf{else}(S_3), B' > \rightarrow < S_2, B' >}$$

$$\text{CondFail}\ \frac{}{< \mathbf{if}(\mathbf{Fail})\mathbf{then}(S_2)\mathbf{else}(S_3), B' > \rightarrow < S_3, B' >}$$

$$\text{If}\ \frac{< S_1, B > \rightarrow < S', B' >}{< \mathbf{if}(S_1)\mathbf{then}(S_2)\mathbf{else}(S_3), B > \rightarrow < \mathbf{if}(S')\mathbf{then}(S_2)\mathbf{else}(S_3), B' >}$$

These conditional strategies do not serve the same function as conditional reaction rules and are designed to be a more general control flow foundational to the definition of more complex strategies. The main function of the conditional is to provide a rigid method for handling non-application of arbitrary S_1 which can not be achieved by priority classes alone.

3.3.5 Conditional reaction rules

Conditional reaction rules as described in 2.1 are introduced to Heist as a primitive strategy since their ability to contextualise application on the occurrence of an arbitrary Bigraph cannot be recreated at the strategy level, this is because checking for matches which satisfy application conditions requires checking each possible decomposition where a rule can apply and so cannot be abstracted away from a rule definition. Conditional reaction rules are denoted by β and each has a set of possible application conditions which must all be satisfied for a successful application $\rightarrow^{\beta}$ to occur, as with standard reaction rules $/\rightarrow^{\beta}$ represents an unsuccessful application where no possible decomposition satisfies all of the application conditions attached to β.

$$\text{CondRuleApplied}\frac{B \rightarrow^{\beta} B'}{< \beta, B > \rightarrow < \mathbf{Id}, B' >}$$

$$\text{CondRuleFailed}\frac{B /\rightarrow^{\beta}}{< \beta, B > \rightarrow < \mathbf{Fail}, B >}$$

Each application condition is a tuple $\beta_i = \{\pm, P, \uparrow\downarrow\}$ where P is an arbitrary Bigraph to be matched $\pm$ indicates the positive or negative occurrence of P and $\uparrow\downarrow$ determines whether to look outwith or within the context of each possible decomposition of the attached rule.

3.3.6 Conditional iteration

Iteration may be derived recursively from the conditional strategy to continuously apply the strategy S_1 while it does not produce **Fail**. For a real program to be valid any instances of the conditional iteration strategy should have finite S_1 or else an infinite loop may occur which will raise a runtime error in the implementation. Conditional iteration in this context is mainly intended to exhaust the application of an arbitrary strategy which is particularly useful for implementing the common pattern where all similar entities should be affected equivalently by a strategy such as in setup or cleanup steps within a model.

$$\text{While}\frac{}{< \mathbf{while}(S_1), B > \rightarrow < \mathbf{if}(S_1)\mathbf{then}(\mathbf{while}(S_1))\mathbf{else}(\mathbf{Id}), B >}$$

3.3.7 Choice Strategy

The choice strategy is another sequencing strategy implemented as an infix operator which allows users to execute either S_1 or S_2, the strategy works by first attempting to apply S_1 evaluating S_2 only if **Fail** is produced. As with other sequences the choice strategy evaluates with precedence to the leftmost strategy but typically parentheses should be used to avoid ambiguity.

$$\text{ChoiceFirst}\frac{< S_1, B > \rightarrow < \mathbf{Id}, B' >}{< S_1 \ ? \ S_2, B > \rightarrow < \mathbf{Id}, B' >}$$

$$\text{ChoiceSecond}\frac{< S_1, B > \rightarrow < \mathbf{Fail}, B >}{< S_1 \ ? \ S_2, B > \rightarrow < S_2, B >}$$

Choices were referred to in early language versions as priority strategies and do partly encapsulate the behaviour of priority classes where the parameters S_1 and S_2 are sets of rules, the difference is that the choice strategy may be nested freely and that the strategy parameters are executed

deterministically rather than as a non-deterministic choice although this can be achieved at the strategy level with **Any**.

3.3.8 Any Strategy

The **any** strategy is defined to act on the result of a set of independent applications by making a random choice of one successful strategy in the group to be executed, this solves the problem of full priority class encapsulation and may be useful elsewhere for implementing behaviour to circumvent rigid sequencing and nesting of conditions to return some non-determinism to strategy execution.

$$\text{AnyFail}\frac{< S_i, B > \rightarrow < \textbf{Fail}, B >}{< \textbf{Any}(S_1, S_2, \ldots, S_n), B > \rightarrow < \textbf{Any}(S_2, \ldots, S_n), B >} \quad \text{for random integer i} \in [1,\text{n }]$$

$$\text{AnySuccess}\frac{< S_i, B > \rightarrow < \textbf{Id}, B' >}{< \textbf{Any}(S_1, S_2, \ldots, S_n), B > \rightarrow < \textbf{Id}, B' >} \quad \text{for random integer i} \in [1,\text{n }]$$

$$\text{AnyExhausted}\frac{< S, B > \rightarrow < \textbf{Fail}, B >}{< \textbf{Any}(S), B > \rightarrow < \textbf{Fail}, B >}$$

The necessity for **any** in the minimal syntax was initially uncertain since in a BRS, strategies would already be executed non-deterministically making the strategy's most obvious function redundant. However, **any** emerged as an important tool to make user-defined strategies less rigid when a strict sequence was not desirable providing an important balance to deterministic behaviour as well as enabling faithful recreation of a simple rule-based BRS in Heist which justified its inclusion.

3.3.9 Callable Strategies

Every Strategy in Heist can be associated with a string identifier using the **Definition** syntax where in the interpreter (section 4.4) strategies are exclusively invoked with the **do** keyword in a Heist program. This style is mainly a consequence of the non-deterministic interpreter but generally, programs with modular strategies promote good design practices like code re-use and pose no theoretical issue since they may be treated as equivalent to evaluating a strict sequence of each strategy preceded by a **do** statement.

$$\text{definitionSuccess}\frac{(name, S) \notin \sigma}{< (\textbf{strat}\ \ name := S), (\sigma, B) > \rightarrow < \textbf{Id}, (\sigma', B) >} \text{where } \sigma' = \sigma \cup (name, S)$$

$$\text{definitionFail}\frac{(name, S) \in \sigma}{< (\textbf{strat}\ \ name := S), (\sigma, B) > \rightarrow < \textbf{Fail}, (\sigma, B) >}$$

$$\text{callSuccess}\ \frac{(name, S) \in \sigma}{< (\mathbf{do}name), (\sigma, B) > \rightarrow < S, (\sigma, B) >}$$

$$\text{callFail}\ \frac{(name, S) \notin \sigma}{< (\mathbf{do}name), (\sigma, B) > \rightarrow < \mathbf{Fail}, (\sigma, B) >}$$

Callable strategies naturally enable recursion assuming that a parser for the language can capture the commands of each user-defined strategy prior to attempting evaluation so that an potentially recursive strategy exists in the environment before being called using **do** keyword.

3.4 Language extensions

A subset of the semantic constructs included in the language prototype are not minimally required since they can be defined simply in terms of the minimal syntax but remain useful in applied work and reinforce the notion of improved expressivity for complex models. Additional designs which merit their own named strategy will continuously arise from practical use and the ease of extension using the minimal language helps to demonstrate Heist's flexibility and malleability for future development.

Strategy (S)::=
| S_{minimal}
| **for**(n)**use**(S)
| **upto**(n)**use**(S)
| **Check**(P)

Figure 3.3: *Extended strategy language syntax*

3.4.1 Unconditional iteration

Unconditional iteration explicitly enables an integer number of repeated executions of a given strategy, this behaviour is not available with the **while** strategy which may only exhaust the application of a given strategy but is not part of the minimal language since it simply represents syntactic sugar for n repeated sequencing's of a given strategy.

$$\text{ForExhausted}\ \frac{}{< \mathbf{for}(1)\mathbf{use}(S), B > \rightarrow < S, B >}$$

$$\text{For}\ \frac{}{< \mathbf{for}(n)\mathbf{use}(S), B > \rightarrow < \mathbf{for}(n-1)\mathbf{use}(S;S), B >}\ \text{integer n such that } n \geq 1$$

3.4.2 Partial unconditional iteration

Another useful addition to the extended syntax is a form of the unconditional iterator defined with the choice sequence " **?** " rather than the more rigid ";" sequences to allow a strategy to be executed at most $n \in \mathbb{N}$ times rather than failing if less than n applications are possible.

$$\text{UptoExhausted}\ \frac{}{< \mathbf{upto}(1)\mathbf{use}(S), B > \rightarrow < S\ \mathbf{?}\ \mathbf{ID}, B >}$$

$$\text{Upto}\ \frac{}{< \mathbf{upto}(n)\mathbf{use}(S), B > \rightarrow < \mathbf{upto}(n-1)\mathbf{use}(S\ \mathbf{?}\ \mathbf{ID}), B >}\ n \geq 1$$

Together the two strategies for unconditional iteration enable evaluation with a fixed number of repetitions or as many times as possible up to a maximum, this syntactic sugar is an important enabler in applied modelling for dealing with scenarios where groups of similar entities should be dealt with in the same way and the exhaustive application of **While** is too general.

3.4.3 Matching Strategies

As a consequence of the prominence of conditional checks in Heist, a common usage pattern involves checking for general entity existence which is only partly possible using conditional rules where existence checking is limited to either the context or parameter of a match instance. Hence, a more general existence strategy **Check(P)** is introduced for matching on any occurrence of P without modifying the local state equivalent to an identity rule $P \rightarrow P$.

$$\text{MatchSuccess}\ \frac{B \rightarrow^{\alpha} B'}{< \mathbf{Check(P)}, B > \rightarrow < \mathbf{Id}, B' >}\ \text{where } \alpha := P \rightarrow P$$

$$\text{MatchFailed}\ \frac{B /\rightarrow^{\alpha}}{< \mathbf{Check(P)}, B > \rightarrow < \mathbf{Fail}, B >}\ \text{where } \alpha := P \rightarrow P$$

Further matching infrastructure is desirable but is severely limited by the nature of the core BRS, as a consequence of the non-deterministic nature of BRS matching must typically be coupled to a specific rule application which will by definition cause a state transition prohibiting the creation of non-mutating checking and matching infrastructure at the strategy level.

3.5 Language properties

Formal mathematical proof of extensive mathematical properties was outside the scope of this development but the choice of small-step operational semantics allows for the natural intuition of simple language properties like correctness and termination.

Correctness
Correctness for each strategy in Heist can be proved by demonstrating each strategy evaluates with the expected behaviour based on its description. In small-step semantics, this can be demonstrated

using a proof tree such as in Figure 3.4 which presents an evaluation of the conditional strategies else branch based on the result of an assumedly terminal S_1, small-step rules enumerate the intermediate steps of computation and the axiomatic secondary inference rules in each strategy definition generally make correctness proofs tautological thus they will be omitted here.

$$\text{CondFail}\ \frac{\text{If}\ \dfrac{\overline{< S_1, B > \rightarrow < \mathbf{Fail}, B >}}{< \mathbf{if}(S_1)\mathbf{then}(S_2)\mathbf{else}(S_3), B > \rightarrow < \mathbf{if}(\mathbf{Fail})\mathbf{then}(S_2)\mathbf{else}(S_3), B >}}{< \mathbf{if}(\mathbf{Fail})\mathbf{then}(S_2)\mathbf{else}(S_3), B > \rightarrow < S_3, B >}$$

Figure 3.4: *Proof tree for the conditional strategies else branch assuming S_1 terminates which correctley evaluates to $\langle S_3, B \rangle$ as desired*

Termination

Termination is verifiable as a general property of all strategies by definition except for **while** and recursive strategies defined by the user. In practice, any interpreter should be responsible for handling exceptions caused by recursion errors or infinite while loops however the nature of BRS provides some inherent protection where the occurrence of these errors mostly requires tautological conditions like identity rules which should not be common in general usage.

To avoid infinite loops when using the **while** strategy or recursion there are several pitfalls in the choice of the argument S which users should avoid. Firstly S should never be an identity-type strategy that will always succeed, including rules which always apply within a given system and **Check** instances in models where few states exist and success is near guaranteed, another pattern with similar effect is the over-specification of S where for example numerous conditionals are used to handle every possible case within a system which contextually may never allow **Fail** to occur. The other means of generating a non-terminating strategy is the use of recursion without any base case or nesting within an arbitrary control strategy. User-defined strategies are executed linearly so tail recursion should be adopted as good practice to avoid infinite calls, generally, it is also suggested that recursive calls should typically be wrapped within a conditional or some other strategy capable of producing a **Fail** since the other strategies in a recursively defined sequence are likely to apply in the vast majority of cases by nature.

4 Implementation

OCaml was chosen as a development language for the implementation of Heist mainly due to the existing suite of Bigraph tools packaged alongside BigraphER which allowed for a focus on novel strategy implementation without having to create supporting infrastructure from scratch. Furthermore, OCaml is a popular choice for programming language development with well-documented tooling readily available and as a hybrid functional language is naturally able to represent the broader idea of a strategy presented in the theoretical semantics so was the clear choice of language to build Heist.

The implementation of Heist is split into a Library and executable interpreter which are bundled as separate packages using the OCaml build system *Dune*. The choice to split the implementation in this way was made partly due to the limited scope of the project enforced by the timeframe so that the library strategies could be integrated separately into a more mature bigraph toolkit if desired, the sequential interpreter also benefits from the separation functioning as a standalone executor of strategies which could not be easily unit tested in a full BRS implementation as a consequence of non-determinism.

4.1 Implementation requirements

The requirements for the implementation are closely related to those of the general language but it was decided early in the development that a fully non-deterministic BRS simulation at the strategy level would be too complex to complete within the allotted timeframe, instead a linear sequential evaluation model was adopted rather than the standard tree of possible derivations at each step which simplified the functional requirements of the implementation to make development feasible within the scope of a 20 credit project.

- An executable interpreter for the minimal syntax should be produced which can execute programs written using the Heist syntax.
- The language should provide syntax for importing Bigraphs and rules from **JSON** compatible with the format defined by BigraphER.
- A Test suite to confirm that strategies work as expected both in isolation and when combined.
- Developer-focused documentation to enable the extension and refinement of the strategy language.

4.2 Syntax and grammar

The implementation of Heist began by translating the extended semantics from Figure 3.3 to an identical Menhir grammar defining the syntax of the interpreted language, aside from each strategy expression the grammar also defines a separate expression type for rule and bigraph definitions which are all grouped as sub-expressions under `command` the top-level construct captured by the parser. In Listing 4.1 the syntax defined by the grammar is used to create three callable strategies with sequences, iteration, and a choice of rule applications demonstrating how a typical strategy program should look within a *.heat* file used to store Heist programs.

```
let first_rule = [filepath/Rule.json];;
letcond cond_rule = [filespath/CondRule.json] appcond ( (! in + using
    [filepath/bigraph.json]) );;

strat example_strategy := { first_rule;
                 cond_rule
                 };;

strat example_strategy_2 := {
                  while{first_rule};
                  second_rule
                  };;


strat example_strategy_3 := { any(first_rule, cond_rule, example_strategy_2);
                   first_rule
                   };;

do cond_rule;;
do example_strategy_3;;
do example_strategy_1;;
```

Listing 4.1: *code file implementing strategies in Heist*

Heist's constructs were partly based on the algebraic notation for Bigraphs [Milner (2009)] and the standard notation for application conditions [Archibald et al. (2020)], but general strategy syntax was created to resemble a typical imperative programming language promoting usability through familiarity.

4.3 Library

The Heist library contains the evaluation mechanisms and the abstract types derived from the grammar including each type of strategy expression, rule imports, and application condition definitions used by the evaluation methods like Listing 4.2 to determine how strategies should be evaluated.

```
 and step_conditional (s1,s2,s3) globs b0 =
 let (_,res_1) = eval_strategy (Strat s1) globs b0 in
  match res_1 with
    | Some b' -> eval_strategy (Strat s2) globs (Some b')
    | None -> eval_strategy (Strat s3) globs (b0)
```

Listing 4.2: *Sample backend evaluation function used recursivley to evaluate the conditional stragegy*

The library implements all of the methods performing strategy evaluation recursively where a general `evaluate_strategy` function is used as an entry point to match with each of the possible foundational strategies in Heist with a call to the corresponding `step` function for specific behaviour. The types which `evaluate_strategy` uses to invoke the correct strategy behaviour are OCaml's variant types defined as part of the Heist library and derived directly from the language grammar, each strategy is represented as a tuple of strategy expressions `strategy_expr` except in a call to a user-defined strategy which is read as a string used to lookup its definition

and immediately execute it before the current strategy can continue, rule and conditional rule invocations also work via a user assigned identifier and are executed linearly within a given sequence.

```
evaluate a rule import
evaluate a rule import
parse a strategy definition
parse a strategy definition
evaluate a strategy application (top-level)
lookup good
executing reaction rules
finished
```

Listing 4.3: *Sample console output from strategy execution*

To ensure Correctness in the library functions each evaluation in a program logs its execution to standard output as in Listing 4.3, the correctness of individual strategy output is captured in unit testing but because the interpreter raises a runtime error on unchecked **Fail** demonstrating a program can run through logging provides a relatively strong notion of validity. Simple logging also helped to debug the lexer and parser in the incremental addition of each strategy to the library but is not strict enough for full BRS correctness, a full BRS would instead be verified with a formal model checker like *PRISM* but since the interpreter may only execute library strategies deterministically the combination of logging and unit testing of strategies in isolation achieved a similar notion of certainty about correctness.

4.4 Interpreter

The Interpreter makes use of the library's evaluation methods by walking the parse tree and evaluating each top-level command depth first. The extended syntax (section 3.2) for Heist was directly translated to an EBNF grammar split into a lexer (A.1) specification using *ocamellex* and a parser specification (A.2) compatible with the Ocaml parser generator *Menhir*. The parser generated from the specification is then used on the lexed tokens of program to generate a parse tree such as in Figure 4.1 where the leaves are tokens and the nodes sub-expressions, the tree is walked by the interpreter to evaluate strategies.

Menhir is typically used to build simple imperative programming languages so the imperative nature of the language constructs enabled easy translation, however, unlike language design tools such as *ANTLR* it does not automatically generate a Listener or Visitor pattern to walk the parse tree efficiently which could be harmful to efficiency in a full BRS definition for complex models.

The prototype interpreter carries out two passes of the parse tree to enable strategies to be called out of order and nested within others to simplify the addition of recursion to Heist. The first pass only captures the scope of each strategy definition to create a global lookup table of `(id,strategy_expr)` pairs which together with an initial state Bigraph B_0 define the environment for execution, the second pass then evaluates strategy expressions in the order that they are called using the backend functions for each syntactic construct defined in the Heist library effectively ignoring all statements which aren't `do` calls invoking a given strategy.

```
$ ./heist.exe BankStrategies.heat Bank.json -o results/final_state.json
```

Listing 4.4: *Sample command line call for the Heist interpreter*

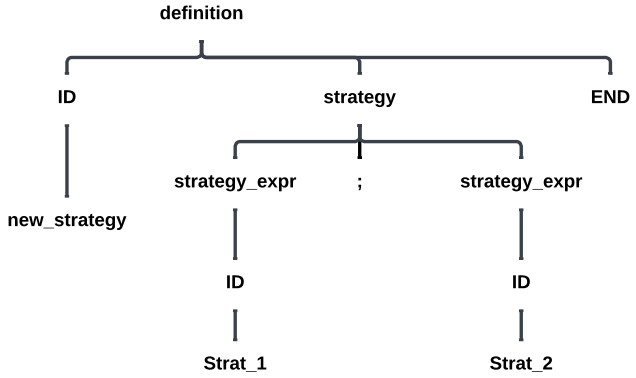


Figure 4.1: *Sample parse tree for defining a strategy new strategy composed of two others*

The interpreter exists as a standalone executable dependent on the library which is invoked on the command line with a *.Heat* file, the relative path to the initial state B_0 in JSON format and optionally the `-t` or `-js` flag specifying an output file for the final state in either tikz or JSON format as defined in *BigraphER*. Ocaml is a niche language so approaching the building of the main interpreter as a standalone executable benefited accessibility, the main dependency for usage is access to the JSON format which can be written from scratch although the process would be time-consuming, most users are likely to work with other tools capable of automatic generation from the algebraic syntax but this remains a notable drawback of the interpreter's design which can only be fixed by the introduction of the syntax for bigraph definition into the language or by integration of Heist into more mature tools like BigraphER.

4.5 Testing

Testing was implemented in the prototype language using AlcoTest within the Heist Library module to perform unit testing on each strategy, unit testing was the preferred mode of testing since the goal of the implementation was to demonstrate working strategies within a simplified execution mechanism as a proof of concept to be iterated upon. Integration testing was conducted entirely manually as a continuous process during and after development, the nature of using Bigraphs as a modelling tool meant that working with a known Bigraph and ruleset allowed strategies to be manually checked as they were implemented and then combined to ensure appropriate behaviour in more complex models.

Heist implements a simple but extendable error reporting mechanism, since Heist is interpreted rather than compiled syntax errors are reported at runtime with distinct error messages for each syntactic construct. In this version of Heist uncaught failing strategies $< \textbf{Fail}, B >$ will also raise an error which satisfies the **Fail** strategy literally by preventing further execution , in a full simulation **Fail** should not be raised an error at runtime and should instead be treated as the end of a path on the derivation tree.

4.5.1 Unit Testing

It was decidedly more important to focus on rigorous unit testing of the library functions determining how strategies are evaluated rather than the language interpreter infrastructure

which relied more on application to known BRS to determine its validity. Alcotest was chosen as a unit testing framework for its seamless integration with the build system and its focus on informative pretty printing for added clarity rather than extensive logging since the suite of unit tests was relatively small.

```
Testing `Strategy evaluation suite'.
This run has ID `XL3NOCLK'.

  [OK]          Successfull Strategy Evaluation          0   Simple sequence.
  [OK]          Successfull Strategy Evaluation          1   Conditional then...
  [OK]          Successfull Strategy Evaluation          2   Conditional else...
  [OK]          Successfull Strategy Evaluation          3   While iterator.
  [OK]          Successfull Strategy Evaluation          4   Priority.
  [OK]          Successfull Strategy Evaluation          5   For iterator.

Full test results in `/mnt/c/Users/adams/Desktop/Bigraphs/repos/BigraphStrategies/Heist/_build/
 evaluation suite'.
Test Successful in 0.097s. 6 tests run.
```

Figure 4.2: *Snippet of foundational test suites checking fundamental execution correctness*

Heist's unit testing infrastructure also provides a novel environment for testing individual rule and strategy evaluations which is not typically achievable in standard BRS solvers due to the non-deterministic nature of BRS where possible results vary too much for unit testing. Use as a sandbox is an important emergent use case for the interpreter supported by lightweight and colourful test infrastructure, unit testing was particularly valuable during development and when combined with the logging provided by the library enables an iterative design approach to creating BRS improving user confidence in the models they are creating.

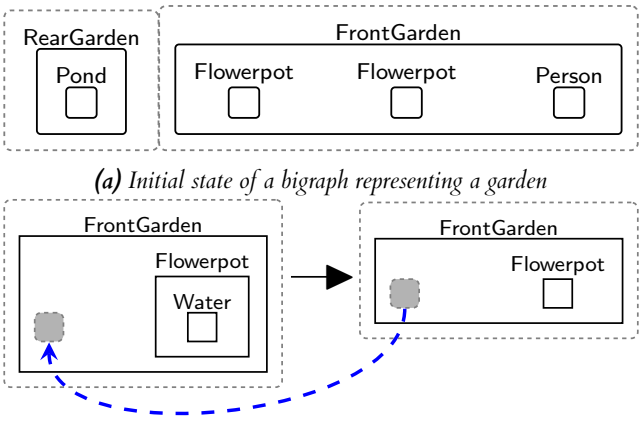
5 Evaluation and Usage

5.1 Strategy Design Evaluation

The inclusion of each strategy in Heist was based not only on existing strategy languages for rewriting logic's but also on whether each would be useful within a BRS. This was often difficult to determine with certainty due to a lack of widespread BRS usage however a problem-driven approach based on deficiencies in the examples which did exist was successful in avoiding redundancy and adhering to minimalist design principles.

5.1.1 Conditional Control flow

The justification for the design of the conditional strategy was to overcome the lack of an ability to act based on the result of reaction rules which is fundamental in almost any modelling context. Priority classes partly enabled a similar behaviour but were heavily restrictive on expressivity by forcing a fixed ordering over the entire BRS evaluation and working at the level of a global ordering overall rules, so Heist needed to create a fundamental way to handle non-application of an arbitrary strategy without any of the disadvantages of priorities.



(a) Initial state of a bigraph representing a garden

(b) Rule to allow flowers to lose water over time

Figure 5.1: *Garden Bigraph where actions like watering flowers eventually require a backtracking rule to continue evolution.*

The model in Figure 5.1a represents a garden designed so that the **Person** entity may perform various tasks such as watering the flowerpots, a frequent design pattern in the creation of BRS was to have the highest priority class contain a series of cleanup rules such as in Figure 5.1b where to allow evolution to continue watered plants must eventually revert to their initial state. Cleanup rules can be problematic since they must always be applied before the application of any rule in a lower priority class placing strict limits on how rules can be ordered and making it

difficult to model systems which require several rules. The conditionals in Heist can be used to update this design pattern to a more natural form which places fewer limitations on the model as a whole, this can be achieved by using a singular reset strategy which can check exactly whether cleanup is necessary using conditional strategies predicated on the current state. Furthermore, Heist's conditionals grant users the ability to perform smaller tidying steps dynamically during the evaluation process which can improve efficiency for example with BRS built to solve puzzles where backtracking rules can be applied when an incorrect move is made to avoid exploring redundant states.

The design of the conditional strategy is not without potential downsides resulting mainly from the difficulty of creating predicate-checking strategies within a BRS. The design of the conditional implies that the first parameter S_1 should be able to check for existence without actually mutating the current state, this is partly possible when combined with the **Check** strategy but more complex checks are generally prohibited since repeated entities aren't distinguishable leading to convoluted combinations of conditional-type strategies which unavoidably inflate complexity.

5.1.2 Sequencing and Iteration

BRS generally lack the ability to perform any iterative application steps and cannot model systems with fixed repeated patterns easily. The choice for the **while** strategy to be conditioned on the successful execution of a strategy was made so that an arbitrary strategy can be applied exhaustively, this is a useful pattern for applying a rule to all instances of the same entity simultaneously rather than affecting only one at a time such as in Figure 5.2 where the **Person** entity can water all adjacent plants within a location simultaneously rather than randomly watering across the model with each step.

The definition of **While** in Heist deviates from the typical format of similar conditional iterators in other languages where the traditional form of "`while(S1)do(S2)`" is frequently adopted. In the process of creating Heist a similar style was initially used, however, in practice this style of conditional iterator proved awkward to apply where S_1 typically attempted to avoid mutating the current state simply to match the style of the construct rather than the construct itself matching usage patterns. Although the typical form of the **While** strategy is not present in Heist it can be recreated naturally with infix sequencing if desired using **While**$(S_1; S_2)$ which remains a simple expression for the more traditional behaviour further justifying the use of a single predicate version of the strategy.

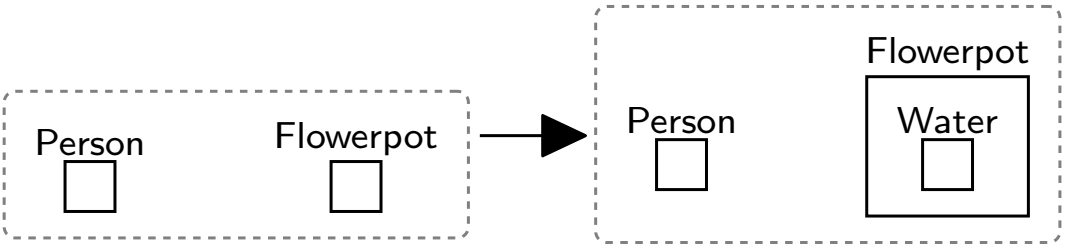


Figure 5.2: *General plant watering rule to be applied to all flowerpot entities*

Sequencing
Explicitly Applying a strategy an integer number of times is also not possible within the standard BRS framework and is typically achieved by adding repeated instances of extra "tagging" entities which can be acted upon by rules without changing the state of the necessary entities, this can lead to a drastic increase in the number of entity and rule instances in a model which is a concern in terms of both efficiency and readability. Again partial sequencing is possible with priority classes but at the strategy level explicit sequences defined using ";" and the "**for**" strategy provide the same behaviour without forcing users to work around any imposed limitations, this takes the

control of how rules interact with each other away from rules themselves and allows the user to simplify rule definitions to the most fundamental state changes in a model which is always advantageous for helping to intuit correctness during the creation process.

5.1.3 Options and partial execution

Inclusion of the S_1 **?** S_2 strategy was inspired by the need for a way to balance the rigidity of strategy sequences by giving users the choice of whether to continue with partial execution rather than relying on the successful application of every strategy in a sequence to provide a non-failing result. Alongside **Any** the choice operator re-introduces the possibility of additional ambiguity in the outcome of a strategy and can be particularly useful for halting sequences prematurely when a desired state is reached which is made easier by the inclusion of **upto** in the extended syntax.

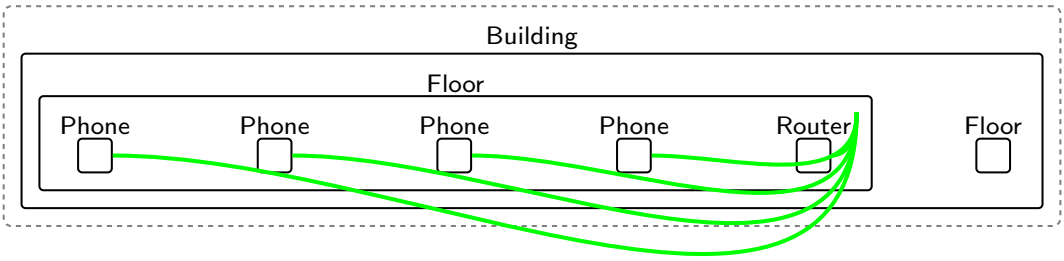


Figure 5.3: *Bigraph modelling a series of phone entities connected to a router*

The choice strategy was initially added to Heist as a flawed attempt to mimic the behaviour of priority classes but emerged as the simplest way to enable strategies to partially execute which was a notable deficiency discovered when attempting to use the interpreter to create illustrative BRS models. Consider Figure 5.3 which models a series of phone entities connected to a network router, a BRS implementing a network protocol may want to interact with only a fixed number of phones at any given step which we would typically model with the **For** strategy, naturally, phones should disconnect from the router when out of range via a rule application however this poses a problem where the fixed iteration strategy becomes too strict and if too few phones are connected then our strategy will never execute which is less desirable than a partial execution. To overcome the lack of flexibility in other strategies we can chain the choice operator together with identities as in Listing 5.1 to create sequences which apply as many sub-strategies as possible and always return **ID** unless everything fails, other combinations of ";" and "?" can be used to create particularly intricate sequence behaviours which demonstrate the success of the minimal languages ability to capture a broad spectrum of behaviour without unnecessary inflation.

```
strat partial_sequence := { (S1 ? ID) ; (S2 ? ID) ... };;
```

Listing 5.1: *Partially executing sequence in Heist*

Exploring choices and partial execution enforced that newly designed strategies should not require complete knowledge of a Bigraphs state at each point in the execution to be useful, other strategies in Heist are somewhat absolutist so **upto** and "**?**" are particularly valuable for keeping BRS less absolutely deterministic and enforcing the exploratory paradigm of the originally intended modelling style.

5.1.4 Beyond priorties

Many of the strategies in Heist supersede the behaviour of priority rules and priority classes which despite key deficiencies remain a useful way to model some systems, hence strategies were designed to encapsulate their behaviour completely rather than eliminate it.

Priority classes operate by checking whether any of a collection of rules is executable at each level and choosing one of the applicable rules non-deterministically moving down the hierarchy defined prior to execution. The structure $\{X, Y\} > Z$ works by applying either X or Y and never both before attempting to apply Z and will always attempt to apply rules in the higher priority section before any application of a lower priority rule, this ordering is rigid for the entire execution of a BRS and does not allow nesting.

Capturing priority behaviour requires firstly the ambiguous choice of a strategy in the execution of an arbitrary class $\{A, B, C, \dots\}$ achieved using the **Any**(A, B, C) strategy which executes only one applicable strategy in any given transition, this can then be explicitly sequenced with another **Any** call to re-create a priority class assuming that the entire rule-set for the model is only invoked from within this sequence of strategies. Less natural is the representation of a global ordering fixed for the entire execution which is only possible at the level of individual programs by avoiding the creation of other strategies which would break the order but is nonetheless possible should the complete behaviour of priority classes be desired.

```
# equivalent to {X,Y} > {Z}
strat priority_strat := { any(X,Y) ; Z } ;;
```

Listing 5.2: *Equivalent representation of priority class instance in Heist*

The main justification for the existence of the **Any** strategy is to completely encapsulate the original behaviour of priorities such as in Listing 5.2 but inclusion in the minimal syntax also overcomes the deterministic nature of the other strategies to support the inherent ambiguity in the evolution of BRS. The imperative nature of the other strategies in Heist supports a deterministic style of application where for example an entire sequence will always fail if one inner strategy fails, this arises partly from design choices to simplify the prototype but is also the fundamental reason for strategies to exist in an entirely non-deterministic simulation environment thus the addition of **Any** is another important way to avoid abstracting too far from the original purpose of a BRS.

5.2 Vault opening problem

Consider the scenario where a Vault or some other secure system requires a login procedure to be executed exactly n times to be successfully opened originally presented in Section 8 of [Archibald et al. (2024 (preprint)]. The rules from the original solution are presented in Figure 5.4 where the sequenced application of a login rule is not universally enforceable so the **Vault** must contain n additional entities (**LoginT**) to count the number of interactions, determining how many interactions have occurred also requires an extra entity named **login** which is checked for using a conditional rule to place boundaries on the beginning and the end of the login process. Furthermore, priority classes are necessary to restrict the order in which these rules can apply, this means that the inclusion of the vault system within a wider model is generally impractical further motivating the use of strategies which don't place limits on the rest of the system.

In Heist, we can forgo the necessity to generate n entities inside the vault to track how many people have logged in by explicitly defining a strategy which attempts to give each **Person** entity a token where success guarantees that the correct number of users have simultaneously logged in.

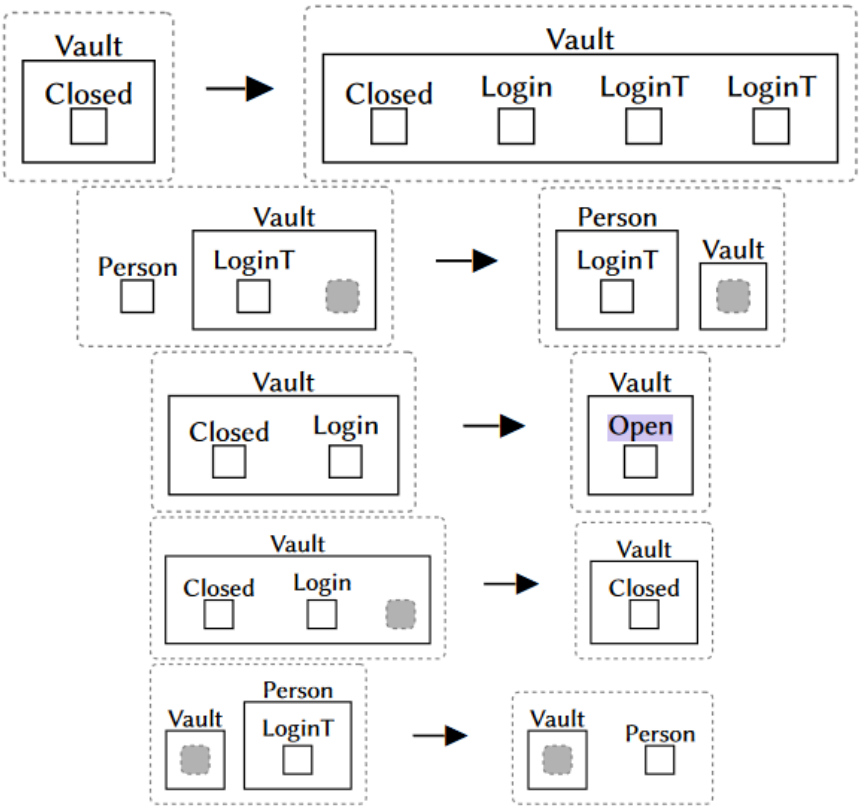
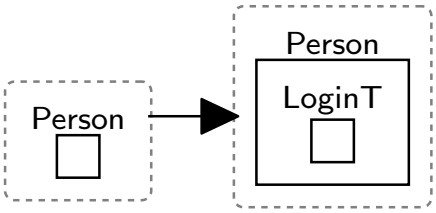
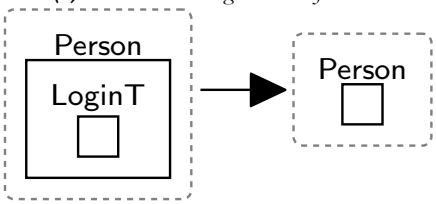


Figure 5.4: *Rules for the original BigraphER solution to the n=2 vault problem using an extended BRS [Archibald et al. (2024 (preprint)]*

This mechanism adds only one additional entity and allows for the problem to be solved with only three reaction rules which are much simpler than the conditional rules necessary for the tagging method.



(a) *Rule to take login token from Person.*



(b) *Rule to give login token to Person.*

Figure 5.5: *Simplified reaction rules to model the vault opening scenario in Heist to control the distribution of keys and cleanup of keys.*

One way to solve the problem of simultaneous logins with strategies is shown in Listing 5.3 where a login strategy is created to execute the rule in (5.5b) exactly *n* times using a **For** strategy where upon failure we may be certain that an incorrect number of Person entities have logged in and act accordingly. We can then define a second strategy to exhaustively perform cleanup with (5.5a) to be used as required before and after the login procedure has been completed to remove the keys given to each **Person**.

Heist enables the simplification of the reaction rules used in the model to only fundamental entity replacements and eliminates the need for any conditional rules, furthermore, abstraction of the control flow into strategies allows for the introduction of other rules or strategies which may be desired in a wider model context without being limited by a strict priority structure. The additional benefit of simplifying the reaction rules is a natural consequence of the strategy paradigm which wasn't necessarily anticipated, simplifying rules in more complex models could lead to inflation in the number of rules and simply shift the complexity to the strategy level but this may be desirable since the functional nature of strategies allows for the adoption of a modular style of re-writing strategies which can be re-used as desired.

```
let give_key = [test/Bigraphfiles/Vault/give_key.json];;
let take_key = [test/Bigraphfiles/Vault/take_key.json];;
let close_vault = [test/Bigraphfiles/Vault/close_vault.json];;
let open_vault = [test/Bigraphfiles/Vault/open_vault.json];;

strat clean := {
   if(take_key)then(clean)else(close_vault)
};;

strat vault_opening := {
   if( for(2)use(give_key) ) then ( open_vault ) else (close_vault)
};;

do vault_opening;;
```

Listing 5.3: *Simple solution for vault problem with strategies*

5.3 River crossing puzzle

The river crossing puzzle is a classic puzzle where a group of objects must be moved between the banks of a river with rules dictating which moves are possible in each step, the problem was used to demonstrate the capability of other term re-writing languages like Maude [Martí-Oliet et al. (2009)] and has been implemented using BigraphER as a sample BRS so represents an interesting opportunity for Heist's capabilities to be compared in a popular context.

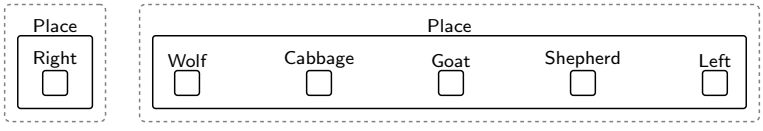


Figure 5.6: *Initial state of river crossing problem*

The rules of the puzzle are that the goat may not be left alone with the cabbage, the wolf may not be let alone with the goat and the shepherd must only carry one entity from left to right in any given turn. Simulation of this problem with Bigraphs requires a priority structure to check

whether a given move has broken the rules which can be removed with controls at the strategy level, however, the main advantage of strategies in puzzle-solving type problems is the ability to add some complexity at the strategy level guiding exploration to improve efficiency.

```
# import rules
let move_cabbage = [test/Bigraphfiles/RiverCross/move_cabbage.json];;
let move_goat = [test/Bigraphfiles/RiverCross/move_goat.json];;
let move_wolf = [test/Bigraphfiles/RiverCross/move_wolf.json];;
let move_shepard = [test/Bigraphfiles/RiverCross/move_shep.json];;

# Victory and Failing conditions
strat wolf_fail := { check(test/Bigraphfiles/RiverCross/wolf_eats.json) } ;;
strat goat_fail := { check(test/Bigraphfiles/RiverCross/goat_eats.json) } ;;
strat success := { check(test/Bigraphfiles/RiverCross/goal.json) } ;;

# Undo explicitly failing moves
strat try_wolf := { move_wolf ; if(any(wolf_fail,
    goat_fail))then(move_wolf)else(ID) } ;;
strat try_cabbage := { move_cabbage ;
    if(any(wolf_fail,goat_fail))then(move_cabbage)else(ID) } ;;
strat try_goat := {move_goat;
    if(any(wolf_fail,goat_fail))then(move_goat)else(ID)} ;;
strat try_shepard := { move_shepard;
    if(any(wolf_fail,goat_fail))then(move_goat)else(ID)} ;;

# driver
strat driver := { success ? (any(try_wolf, try_cabbage, try_goat, try_shepard) ?
    FAIL) };;

do driver;;
```

Listing 5.4: *Simple solution for River crossing problem with strategies*

The simplest contextual knowledge we can derive from the puzzle rules is the existence of two failure conditions which can be explicitly checked for to avoid needless exploration and that the first move should always be a goat move because anything else leads to an immediate failure. In the BigraphER implementation application of rules leading to a failure condition does not end state evolution thus each possible failure state will continue to evolve until deadlock when no rule is applicable, conversely the simple checking logic in 5.4 ensures that any move leading to a failure state is immediately undone before evolution can continue significantly pruning the number of possible states and thus improving evaluation efficiency without major assumptions about the puzzle's solution.

This rudimentary Heist implementation is able to remove two conditional rules and a priority class relative to the sample BigraphER solver at the expense of adding more callable strategies to the system. This tradeoff is amenable because strategies can more naturally express guided evolution and provide more opportunities for refinement relative to only rules.

6 Conclusion

Heist, a minimal strategy language for Bigraphical modelling has successfully enabled the simplification of complex BRS with a focus on abstraction and usability by controlling how reaction rules are applied. By drawing inspiration from several existing strategy languages a minimum justifiable language is specified with a syntax containing common control structures designed to promote code reuse by acting as fundamental building blocks for user-defined strategies which more naturally express advanced behaviour in bigraphical models. Full operational semantics were created to describe how each strategy in the core language is computed in terms of a series of intermediate reductions which can be used to formally verify the language properties and to create a more advanced interpreter in future if desired. Common design practices which emerged via usage during development were implemented as an extension to the language each with its own semantics based on the minimal language, the ease of extension demonstrates the language's adaptability and successful ability to capture the core desirable functionality required for Heist to serve as a strong foundation for the exploration of sophisticated BRS.

To implement Heist a mutually dependent library and interpreter were created in OCaml incorporating both the minimal and extended language semantics. Although the Heist interpreter executes strategies strictly deterministically rather than as a fully non-deterministic BRS the **any** strategy allows for a faithful recreation of a BRS within the interpreter. Beyond simplifying the implementation to a reasonable scope a long-term functionality for the sequential interpreter is to provide a unique facility to unit-test individual rules and strategies which is not typically possible in a full BRS and helps to provide users with confidence in the correctness of each part of a model during the creation process.

Examples from existing BRS literature were used to demonstrate the value of strategies as the best means to control rule application and eliminate problems with the basic BRS along with its extensions. I show that Heist can reduce the number of rules or simplify their definitions, furthermore that priority classes are totally encapsulated and extended in Heist and that conditional rules may be introduced as a language primitive to allow users to be as expressive as possible without the harsh limitations imposed by individual extensions. Finally, it can be demonstrated that adding complexity at the strategy level can be used to enable BRS to be more efficient by reducing the available state space to eliminate unnecessary exploration, gains in efficiency are very much user-controlled and using strategies to reduce randomness allows for fine-grain control of system exploration.

6.1 Future work

The scope of the current implementation was limited to the linear application of strategies rather than a fully non-deterministic BRS at the strategy level, although the simplified interpreter is useful for iterative testing and demonstrating the feasibility of strategies a more advanced interpreter integrated with model checking tools like PRISM is necessary for full BRS with strategies to be made available.

Furthermore, application to more practically interesting and complex systems will inevitably lead to the emergence of strategies which should be added to the extended syntax for simplifying

routine and complex tasks. This includes strategies which may rely on the mathematical properties of bigraphs to be implemented, ideas like "zooming" strategies to allow rules to be applied on a specific sub-bigraph were explored early in the development process but were too complex to implement within the scope of the minimal language despite their possible utility suggesting they should be revisited in future work.

The intended work in the immediate future will initially revolve around small quality-of-life improvements, this includes more thorough documentation for the language beyond the developer-level docs mainly useful for maintenance rather than for learning the language, usage examples are available but do not cover some advanced strategy combinations which could also help users to better understand intended usage patterns when working with Heist. Heist depends significantly on more mature bigraph tools such as BigraphER and independence could be desirable in future, at a minimum this means the addition of syntax for defining rules and bigraphs directly from the algebraic representation but compatibility with other tools is useful and should not be removed, Heist could, however, benefit from full integration into a more mature software package by taking advantage of stable bigraph infrastructure which implements the behaviour outwith its scope and this possibility should certainly be explored before rushing to independence.

A Appendices

A.1 lexer syntax for implementation (ocammelex syntax)

```
{
   open Parser
}

let white = [' ' '\t']+
let newline = '\n' | '\r'
let digit = ['0'-'9']
let int = digit+
let letter = ['a'-'z' 'A'-'Z' '/' '.' '_']
let id = letter+


rule read =
  parse
  | newline {Lexing.new_line lexbuf; read lexbuf}
  | white { read lexbuf }
  | ";" { SEMICOLON }
  | ";;" { DOUBLESEMICOLON }
  | "(" { LPAREN }
  | ")" { RPAREN }
  | "let" { LET }
  | "letcond" { CONDLET }
  | "strat" { STRAT }
  | "=" { EQUALS }
  | ":=" { BEQUALS }
  | "do" { APPLY }
  | "if" { IF }
  | "then" { THEN }
  | "else" { ELSE }
  | "}" { RBRACE }
  | "{" { LBRACE }
  | "[" { LSQUARE }
  | "]" { RSQUARE }
  | "?" { CHOICE }
  | "use" { USE }
  | "any" { ANY }
  | "while" { WHILE }
  | "appcond" { APPCOND }
  | "using" { USING }
  | "for" { REPEAT }
  | "upto" { UPTO }
  | "ID" { IDENTITY }
  | "FAIL" { FAIL }
  | "check" { CHECK }
```

```
  | "in" { IN }
  | "+" { PARAM }
  | "-" { CTX }
  | "!" { BANG }
  | "," { COMMA }
  | "#" { read_single_line_comment lexbuf }
  | id { ID (Lexing.lexeme lexbuf) }
  | int { INT (int_of_string (Lexing.lexeme lexbuf)) }
  | eof { EOF }
  | _ as c { failwith (Printf.sprintf "unexpected character: %C" c) }

and read_single_line_comment = parse
  | newline { Lexing.new_line lexbuf; read lexbuf }
  | eof { EOF }
  | _ { read_single_line_comment lexbuf }
```

Listing A.1: *Complete Heist Grammar definition used by Menhir to generate the parser*

A.2 Full Menhir Grammar

```
%{
   open Ast
%}

%token <string> ID
%token LPAREN
%token SEMICOLON
%token DOUBLESEMICOLON
%token REPEAT
%token RPAREN
%token LET
%token APPLY
%token STRAT
%token CONDLET
%token EQUALS
%token BEQUALS
%token EOF
%token IF
%token THEN
%token LBRACE
%token RBRACE
%token WHILE
%token CHOICE
%token ELSE
%token ANY
%token <int> INT
%token APPCOND
%token USING
%token IN
%token PARAM
%token CTX
%token BANG
%token COMMA
%token UPTO
```

```
%token USE
%token LSQUARE
%token IDENTITY
%token CHECK
%token RSQUARE
%token FAIL

%start file
%type <Ast.toplevel_cmd list> file

%left SEMICOLON CHOICE

%%

file:
  | EOF { [] }
  | command DOUBLESEMICOLON file { $1 :: $3 }
  | command EOF { [$1] }

command:
  | definition { Def $1 }
  | STRAT ID BEQUALS LBRACE strategy { Strategy($2,$5)}
  | APPLY; e1 = ID { Application(e1) }

strategy:
  | RBRACE { Identity }
  | strategy_expr RBRACE { $1 }

strategy_expr:
  | s=ID { CallStrategy(s) }
  | IDENTITY { Identity }
  | FAIL { Fail }
  | CHECK; LPAREN; b=ID; RPAREN; { Check(b) }
  | IF; LPAREN; s1=strategy_expr; RPAREN; THEN; LPAREN; s2=strategy_expr; RPAREN;
      ELSE; LPAREN; s3=strategy_expr; RPAREN {Conditional(s1,s2,s3)}
  | WHILE; LPAREN; s1=strategy_expr; RPAREN; {CondLoop(s1)}
  | s=separated_pair(strategy_expr, CHOICE, strategy_expr) { Choice(s) }
  | s=separated_pair(strategy_expr, SEMICOLON, strategy_expr) { Sequence(s)}
  | ANY; LPAREN; ss=separated_nonempty_list(COMMA, strategy_expr); RPAREN; {
      Any(ss) }
  | REPEAT; LPAREN; i = INT; RPAREN; USE ;LPAREN; s1=strategy_expr; RPAREN; {
      ForLoop(i,s1) }
  | UPTO; LPAREN; i = INT; RPAREN; USE ;LPAREN; s1=strategy_expr; RPAREN; {
      UptoLoop(i,s1) }
  | LPAREN; s=strategy_expr; RPAREN; { s }

cond:
  LPAREN option(is_bang) IN place USING LSQUARE ID RSQUARE RPAREN {
      CondExp($2,$4,$7) };

is_bang:
  | BANG { true }

place:
  | PARAM { Cond_Param }
  | CTX { Cond_Ctx }
```

```
definition:
  | LET; e1 = ID; EQUALS; LSQUARE; e2 = ID; RSQUARE { Ruledef(e1,e2) }
  | CONDLET; e1=ID; EQUALS; LSQUARE; e2=ID; RSQUARE; APPCOND; LPAREN;
      a1=separated_nonempty_list(COMMA, cond); RPAREN; { CondRuledef(e1,e2,a1) }
```

Listing A.2: *Complete Heist Granner definition used by Menhir to generate the parser*

Bibliography

B. Archibald, M. Calder, and M. Sevegnani. Conditional Bigraphs. In F. Gadducci and T. Kehrer, editors, *Graph Transformation*, volume 12150, pages 3–19. Springer International Publishing, Cham, 2020. ISBN 978-3-030-51371-9 978-3-030-51372-6. doi: 10.1007/978-3-030-51372-6_1. URL https://link.springer.com/10.1007/978-3-030-51372-6_1. Series Title: Lecture Notes in Computer Science.

B. Archibald, M. Calder, and M. Sevegnani. Practical Modelling with Bigraphs. *Author provided preprint*, 1(1), 2024 (preprint).

M. Calder, A. Koliousis, M. Sevegnani, and J. Sventek. Real-time verification of wireless home networks using bigraphs with sharing. *Science of Computer Programming*, 80:288–310, Feb. 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.08.004. URL https://www.sciencedirect.com/science/article/pii/S0167642313001974.

C. Castro and P. Borovanský. The use of a strategy language for solving search problems. *Annals of Mathematics and Artificial Intelligence*, 29(1):35–64, Feb. 2000. ISSN 1573-7470. doi: 10.1023/A:1018900617693. URL https://doi.org/10.1023/A:1018900617693.

D. Damgaard. Bigraphs by Example. *ITU*, 2005. URL http://en.itu.dk/Research/Technical-Reports/Technical-Reports-Archive/2005/TR-2005-61.

S. Eker, N. Martí-Oliet, J. Meseguer, R. Rubio, and A. Verdejo. The Maude strategy language. *Journal of Logical and Algebraic Methods in Programming*, 134:100887, Aug. 2023. ISSN 23522208. doi: 10.1016/j.jlamp.2023.100887. URL https://linkinghub.elsevier.com/retrieve/pii/S235222082300041X.

M. Fernández, H. Kirchner, and O. Namet. A Strategy Language for Graph Rewriting. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and G. Vidal, editors, *Logic-Based Program Synthesis and Transformation*, volume 7225, pages 173–188. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-32210-5 978-3-642-32211-2. doi: 10.1007/978-3-642-32211-2_12. URL http://link.springer.com/10.1007/978-3-642-32211-2_12. Series Title: Lecture Notes in Computer Science.

N. Martí-Oliet, J. Meseguer, and A. Verdejo. A Rewriting Semantics for Maude Strategies. *Electronic Notes in Theoretical Computer Science*, 238(3):227–247, June 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.05.022. URL https://www.sciencedirect.com/science/article/pii/S1571066109001443.

R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, Cambridge, 2009. ISBN 978-0-521-49030-6. doi: 10.1017/CBO9780511626661. URL https://www.cambridge.org/core/books/space-and-motion-of-communicating-agents/267A0C3F2DB68EF43E7158DB5A7016C3.

M. Sevegnani and M. Calder. Bigraphs with sharing. *Theoretical Computer Science*, 577:43–73, Apr. 2015. ISSN 03043975. doi: 10.1016/j.tcs.2015.02.011. URL https://linkinghub.elsevier.com/retrieve/pii/S0304397515001085.

M. Sevegnani and M. Calder. BigraphER: Rewriting and Analysis Engine for Bigraphs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 494–501, Cham, 2016. Springer International Publishing. ISBN 978-3-319-41540-6. doi: 10.1007/978-3-319-41540-6_27.

E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.