TECHNISCHE UNIVERSITEIT DELFT

MASTER OF SCIENCE THESIS IN COMPUTER SCIENCE

Program Synthesis in Causal Analysis of Biochemical Programming

Sebastien VAN TIGGELE

Supervisors:
Dr. Sebastijan DUMANČIĆ
Reuben GARDOS REID

9th September 2025

Delft University of Technology

Program Synthesis in Causal Analysis of Biochemical Programming

Master's Thesis in Computer Science

Algorithmics group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Sebastien van Tiggele

9th September 2025

Author

Sebastien Leonardo van Tiggele

Title

Program Synthesis in Causal Analysis of Biochemical Programming

MSc presentation

September 17, 2025

Graduation Committee

Dr. S. Dumančić (chair)	Delft University of Technology
J.A. Baaijens	Delft University of Technology

Abstract

Understanding how local molecular interactions give rise to global cellular outcomes is a central challenge in computational systems biology. Rule-based modeling frameworks such as Kappa provide a way to specify these interactions compactly as rules, which can be simulated to produce causal "stories" explaining how a specified event of interest (EOI) arises. However, even small changes in the conditions of a rule can change these causal structures in ways that are difficult to anticipate. This thesis addresses the question: how do modifications to rules affect the causal stories produced by a model? We propose a novel program synthesis framework that automates the generation and exploration of Kappa programs. First, we construct a grammar specialized to a given program's agent signature, extended with constraints to ensure only valid rules are produced. Second, we introduce a rule modification procedure that systematically adds structural contextual conditions to rules while preserving their core transformations. Together, these techniques allow us to enumerate the space of modified programs and retrieve subsets of this space guided by user-defined specifications over the stories. Our experiments show that the constraints reduce the search space by several orders of magnitude, making exhaustive exploration feasible for small models. Using static analysis tools, we reduce the exploration time by 45%. We demonstrate how different specifications, such as ensuring EOI reachability, can be used to retrieve a subset of modified programs that satisfy them automatically. Together, this work demonstrates the effectiveness of the program synthesis approach in modeling kappa programs and posing questions about their causal structure.

Preface

One of the first things you will read in this thesis is the last part I am writing. The last 9 months have been a sincere journey, and it was a true test of more than just expertise in coding. I have learned a lot during this time, not only about research and writing, but also about perseverance. Choosing a project that combined algorithmics with biochemical simulation programs was not what I expected to do to conclude my time studying Computer Science in Delft. Still, I am glad I had the opportunity to learn about a research field that was quite far from home.

I would like to thank my supervisors, Sebastijan and Reuben, for all the meetings where we could discuss the various directions my thesis could take. Your support and ideas have been very useful in guiding me through this process. I am also grateful to everyone in the PONY lab for the weekly meetings where we could exchange ideas, provide feedback, or share food at a potluck. I am glad to have met you all.

I want to thank Walter Fontana for the insightful discussions we had about Kappa and the stories. I am very thankful for the enthusiasm you had for the idea of combining program synthesis with Kappa.

Finally, I would like to thank all of my friends, whether they are working on their thesis or not, for helping me stay motivated by joining me in the study sessions osn campus. Without the coffee and lunch breaks, finishing my thesis would not have been possible.

Sebastien van Tiggele

Delft, The Netherlands
9th September 2025

Contents

1 Introduction **1**

2 Background **3**
2.1 Rule-based modeling 3
2.2 The Kappa Language 5
2.2.1 Agents and Interfaces 5
2.2.2 Rules 5
2.2.3 Stories 6
2.3 Program Synthesis 8
2.3.1 SyGuS 9
2.3.2 Sketch 9
2.3.3 Herb.jl 10
2.3.4 Program Synthesis with constraints 10

3 Related work **13**
3.1 KaSa: A Static Analysis tool 13
3.2 KaDE 14
3.3 Matching stories to Program traces 14

4 Problem Statement **15**
4.1 Preliminaries 16
4.2 Creating Kappa Programs 16
4.3 Modifying an existing program 17
4.4 Programs following a specification 17

5 Modeling Kappa Programs **19**
5.1 The Search Space 19
5.2 Grammar for Kappa Rules 20
5.2.1 Creating Agents 21
5.2.2 Filling Agents 22
5.2.3 Constraints 24

6 Modifying Kappa Programs **31**
6.1 Modifications to the rules 31
6.1.1 Modifying Rules: Core vs. Context 31
6.1.2 Types of modifications 32
6.1.3 Modifying RuleNodes 33
6.2 Synthesizing new programs 34

7 Results **37**
7.1 Experimental setup 38
7.1.1 KS - Processive 38
7.1.2 KS - Distributive 39
7.1.3 ABC model 40
7.1.4 early EGFR model 41
7.2 Search space of RuleNodes 42
7.3 Search space of Modified Programs 45
7.4 Kappa programs by specification 49
7.4.1 Modified programs with the same story (Robustness) . . . 50
7.4.2 KaSa false positives 51
7.4.3 New rules in the story 51
7.5 Summary and Takeaways 52

8 Conclusions and Future Work **53**
8.1 Conclusions 53
8.2 Future Work 53
8.2.1 Types of modifications to the context 53
8.2.2 Changes to the core rules 54

Chapter 1

Introduction

Understanding how complex molecular systems produce specific cellular outcomes is a central challenge in computational biology [Kitano, 2002]. Using Rule-based modeling frameworks such as Kappa [Danos et al., 2007, Boutillier et al., 2018b] or BioNetGEn [Faeder et al., 2009, Harris et al., 2016], we can model complex biochemical systems by specifying local interactions between the molecules of such a system. These local interactions are modeled as *rules* that describe how molecular components such as proteins can bind, dissociate, or undergo modification depending on their context. More importantly, we can focus on a specific reaction of interest and extract an explanation of these reactions that precede it in a simulation, called a *story* [Danos et al., 2007]. This story answers how a specific signaling outcome emerges and corresponds closely to the biological notion of a "signaling pathway."

For example, Kappa has been used to model the EGFR signaling network, a system that plays a critical role in regulating mammalian cell behaviors such as proliferation, survival, and differentiation. In the EGFR model presented by Danos et al. [Danos et al., 2007], processes such as ligand binding, receptor dimerization, and phosphorylation are modeled, resulting in triggering cascades of downstream interactions between proteins and kinase activations.

Despite the biological complexity, the entire system is captured compactly using 70 rules, which together define a model capable of generating over 10^{23} distinct molecular species. Such combinatorial complexity would be intractable using traditional differential equation models, but rule-based modeling allows these interactions to be specified locally and simulated efficiently.

Although rule-based models such as Kappa allow us to trace signaling pathways through stories, the structure of these stories can be sensitive to subtle changes in the rule context. Even minor modifications, such as adding a binding requirement or altering a phosphorylation condition, can change which reactions are possible and, in turn, produce different causal explanations for the same event of interest (EOI). This raises a significant challenge: we currently have a limited understanding of how these rule modifications affect the global story structure.

This question is especially relevant for modelers who do not know the exact mechanisms of a biological system and want to explore how robust an observed outcome is. For instance, under what changes to a rule is the EOI still reachable? Are there alternative paths to the same result that might be obtained by making slight tweaks to the conditions of a rule? Or the other way around, which specific rule changes would block the EOI? This could be useful when the EOI corresponds to a disease-related behavior that a biochemist might want to prevent.

Currently, answering these questions requires manual edits of a program and repeated simulations, which is time-consuming and hard to scale. This limits our ability to systematically investigate the relationship between local rule context and global cellular signaling behavior. This thesis aims to answer the following question:

> *How can we systematically explore the effect of rule modifications on the causal story structure of a rule-based model?*

To address this, we introduce a tool that automates the generation and exploration of Kappa rules using a program synthesis approach [Gulwani et al., 2017].

Our first contribution is the process of generating valid rules by using a grammar with constraints. Given an input program $\mathcal{P}$, our method creates a grammar tailored to $\mathcal{P}$ that will only generate rules using agents and their sites as defined in the agent signature. On top of this grammar, we impose a set of constraints [Swinkels, 2024] that exclude invalid rules that cannot be ruled out by the grammar alone.

The second contribution this work makes is to add modifications to the rules, using a sketch-based approach. These modifications involve adding more conditions to the rule's context. We show that we can traverse the space of modified programs (programs where one or more rules in $\mathcal{P}$ are modified) efficiently by statically checking if the variant still produces a story. A modeler can specify a specification over the story, and use this tool to enumerate the set of modified programs that satisfy this specification.

We demonstrate that an otherwise intractable search space can be reduced to a manageable size by combining a grammar, constraints, and a rule modification procedure, yielding reductions of several orders of magnitude. Using static analysis tools, we further improve efficiency, cutting exploration time by up to 45%. Together, these components provide a framework in which modelers can pose biologically meaningful questions about the conditions underlying reactions. By specifying the desired explanation, the algorithm automatically generates the subset of programs that adhere to it.

Chapter 2

Background

In this chapter, we introduce the field of computational systems biology in section 2.1 to give some context to the rule-based modeling paradigm, and why we use it for analyzing the causal structures in our models. The syntax of a rule-based language is explained in Section 2.2, as we need a part of it to understand the contributions made in Chapter 5. In Section 2.3, we will cover program synthesis and how it enables us to achieve the goal of systematically creating new programs. Here, we also mention some ideas from the field of Program synthesis that we implement in our method.

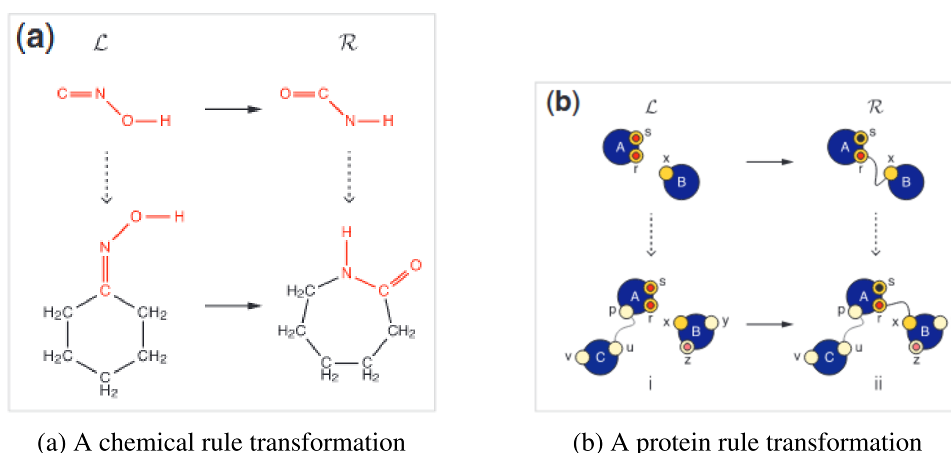
2.1 Rule-based modeling

Systems biology aims to understand complex biological phenomena by modeling the dynamic interactions of molecular components within a cell. These interactions, such as protein binding and phosphorylation, constitute networks that give rise to cellular behavior like signal transduction or apoptosis. These models enable researchers not only to simulate system behavior under various conditions but also to make predictions, test hypotheses, and gain mechanistic insights that are challenging to obtain through experiments alone. [Kitano, 2002]

Historically, many models of biochemical systems have been formulated as systems of ordinary differential equations (ODEs), capturing the time evolution of molecular species concentrations. This framework leverages the well-developed theory of dynamical systems, offering insights into phenomena such as steady states, oscillations, and other dynamical properties. However, ODE models require explicit enumeration of all chemical species and reactions in the system, making them impractical for larger problems. For example, a protein with eight modifiable sites can exist in 256 states, and a dimer of two such proteins results in over 65000 unique species. Constructing ODEs for such systems becomes infeasible [Danos et al., 2007].

To overcome these limitations, rule-based modeling languages like Kappa [] and BioNetGen [Faeder et al., 2009, Harris et al., 2016] have emerged. These languages

operate over graph structures with rules that act as graph-rewrite transformations. The key idea of a rule is to specify a transformation of a part of the graph that results in a change in specific entities in the model. This idea comes from Chemistry, where the left-hand side of a reaction rule specifies the fragment $\mathcal{L}$, and the right-hand side of the rule the change $\mathcal{R}$ (Figure 2.1a). Here, the underlying structure change not mentioned by the rule is governed by the electron pushing rules at a lower level of abstraction. In interactions between proteins, they often depend on some but not all aspects of their state. Rule-based languages take inspiration from the chemical perspective (figure 2.1b)but apply it to molecular systems biology by viewing proteins as higher-order atoms. [Boutillier et al., 2018b]



(a) A chemical rule transformation

(b) A protein rule transformation

Figure 2.1: Illustrations of rule-based transformations in chemistry and protein interactions.

Rule-based modeling languages like Kappa and BioNetGen use the agent abstraction to represent molecular components. An agent represents a biological entity, typically a protein, equipped with an interface of named sites that represent interactive capabilities, such as binding or internal site modification. These form site-graphs: the idea that an agent node is connected to its site-nodes, and the site-nodes can connect through rules. A rule $r : \mathcal{L}_r \rightarrow \mathcal{R}_r$ describes a transformation between two site graphs, a left-hand side $\mathcal{L}_r$ and a right-hand side $\mathcal{R}_r$, that specifies patterns [Boutillier et al., 2018b].

The system is a graph of a collection of disconnected site graphs, each representing one instance of a protein. This is called a mixture (M). a rule r is applied to a mixture (M) by embedding $\mathcal{L}_r$ into (M) if everything mentioned in $\mathcal{L}_r$ exists in the mixture. The rule is executed by replacing the change mentioned in $\mathcal{R}_r$ The model is a set of rules with an initial mixture of agents, and the model evolves stochastically through applying the rules through the Doob-Gillespie algorithm [Gillespie, 1977, 2007], which applies rules using Continuous time Monte-Carlo (CTMC). Each rule r is assigned a rate constant γ_r, which influences the probability rate that triggers on any given embedding of $\mathcal{L}_r$ in $\mathcal{M}$ [Boutillier et al., 2018b].

These rules provide a compact and transparent way to handle the combinatorial complexity that will arise in more traditional differential equation approaches to these models. More importantly, these systems of rules enable us to perform causal analysis more effectively than reaction networks, as we can reason about the level of rules.

2.2 The Kappa Language

Rule-based languages, such as Kappa and BioNetGen, use textual notation to represent these graph-rewrite frameworks. In this thesis, we will focus on Kappa, as the Kappa ecosystem contains many valuable and ready-to-use tools for simulating biochemical systems and analyzing the causal mechanisms we want to investigate. We will describe how Kappa declares the agents, and the syntax of the rules that denote how these agents can interact. For the purpose

2.2.1 Agents and Interfaces

In Kappa, an agent represents a molecular entity, e.g., a protein, and its interface describes the sites it exposes for interaction or modification. Each site can optionally carry an internal state.

Agent types are declared using the `%Agent` statement. For example:

```
%Agent: A(x{u p} y)
```

This declaration specifies that agent `A` has two sites: `x` and `y`. Site x can be in one of the internal states u or p (e.g., unphosphorylated or phosphorylated), and y has no internal states.

In a rule or mixture, agent instances are written with their current state. The binding state of a site is specified using square brackets, where `x[.]` indicates that site x is free, `x[_]` says it is bound to something, while a non-negative number `x[1]` denotes it is bound to a unique other site in the same expression whose site is also bound with identifier `1`.

For example:

```
K(x[1]), S(a[1])
```

This represents a `K` agent whose site `x` is bound via bond `1` to site `a` on agent `S`.

2.2.2 Rules

Rules describe transformations applied to agent instances when specific patterns are matched in the current system state, referred to as the mixture. Each rule consists of:

- a left-hand side (LHS): the pattern to match in the current system

- a right-hand side (RHS): the result of applying the rule,
- and a rate constant, which represents the probability rate of the reaction.

Here is an example of two rules:

```
'K.S' K(x[.]), S(a[.])      -> K(x[1]), S(a[1])      @ 1
'b++' K(x[1]), S(a[1],b{u})-> K(x[1]), S(a[1],b{p})@ 1
```

These rules specify:

- 'K.S': agent K binds to agent S at their respective x and a sites (both initially free)
- 'b++': Once bound, K can phosphorylate site b on S, changing its internal state from 'u' to 'p'.

The square brackets in x[1] and a[1] indicate that the x site of K is linked to the a site of S via bond identifier 1.

2.2.3 Stories

The input to KaSTOR is a trace τ, which is the complete sequence of events produced during a stochastic Kappa simulation:

$$\tau = e_1, e_2, e_3, \ldots, e_N$$

Each event in the trace corresponds to a single rule application, recording not only the rule used but also the specific agent instances and sites that were tested or modified.

The event of interest (EOI), denoted as e_N in this example, is chosen by the modeler. It typically corresponds to a biologically significant outcome, such as the formation of a key molecular species, the activation of a receptor, or an undesirable event like the onset of a tumorigenic state. The goal is to understand how this specific event arose, given the whole history of the simulation.

Causal lineage and precedence Given a trace $s = e_1, e_2, \ldots, e_N$ and a designated EOI e_N, KaSTOR first identifies which earlier events were causally relevant to producing that outcome. Because Kappa simulations are stochastic and concurrent, temporal order alone does not imply causality.

Two events e_i and e_j are said to be concurrent if they could occur in either order without affecting the outcome. By contrast, we say that e_i precedes e_j if applying the rule in e_j requires a condition established by e_i. This notion of precedence serves as a proxy for causality in KaSTOR's analysis.

It is essential to note that precedence is not always causal. For example, suppose a kinase can phosphorylate a substrate, and the substrate can independently

be degraded. If degradation follows phosphorylation in a trace, this ordering does not imply that phosphorylation *caused* degradation; it merely indicates that phosphorylation happened first. KaSTOR takes a conservative approach: it initially treats all precedence relations as potential causal dependencies, and later refines this structure during compression.

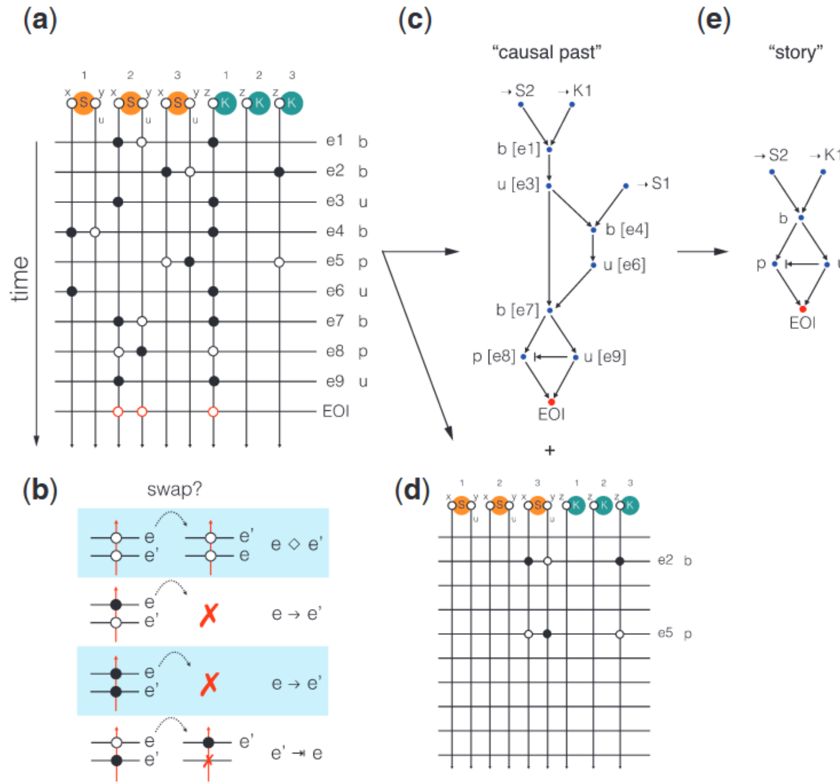
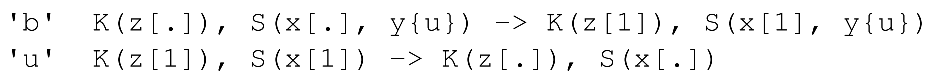


Figure 2.2: Causal Analysis of Event Traces (adapted from Boutillier et al. [Boutillier et al., 2018b]). Panel (a): A trace represented as worldlines that track modifications (filled circles) and tests (open circles) at each site. Panel (b): Rules for sliding tests and modifications backwards in time to reconstruct causal relations. Panel (c): The causal past of the EOI extracted from the trace. Panel (d): Events that had no bearing on the EOI are discarded. Panel (e): Causal compression yields the minimal subset of events sufficient to explain the EOI, referred to as a *story*.

To illustrate how KaSTOR extracts causal explanations from stochastic simulations, consider the example adapted from Boutillier et al. [Boutillier et al., 2018b]. The model encodes a Michaelis–Menten type mechanism with two agents, a kinase K and a substrate S, governed by the following rules:



```
'b'  K(z[.]), S(x[.], y{u}) -> K(z[1]), S(x[1], y{u})
'u'  K(z[1]), S(x[1]) -> K(z[.]), S(x[.])
```

```
'p'  K(z[1]), S(x[1], y{u}) -> K(z[1]), S(x[1], y{p})
'*'  S(x[.], y{p}) -> S(x[.], y{p})
```

Here, the query rule `'*'` is used to mark the event of interest (EOI): the production of a free phosphorylated substrate. Suppose we simulate a mixture containing three substrates and two kinases, and obtain the trace $s = bbubpuppu$, where each letter denotes the rule applied.

Panel (a) of Figure 2.2 shows the trace represented as worldlines: vertical threads for each site, with filled circles for modifications and open circles for tests. KaSTOR reconstructs the *causal past* of the EOI (panel c) by sliding tests backward in time according to the rules in panel (b). This yields a directed acyclic graph in which arrows encode precedence relations between events. Events that have no bearing on the EOI are discarded (panel d). Finally, the causal past is subjected to causal compression, which searches for a minimal subset of events sufficient to produce the EOI. The resulting compressed DAG, shown in panel (e), is called a *story*.

This example highlights the principle of causal analysis: from a long, redundant trace, KaSTOR isolates only those events necessary to account for the event of interest, yielding a concise, interpretable causal explanation.

2.3 Program Synthesis

Program synthesis is the task of automatically generating a program that satisfies a given specification or set of requirements [Gulwani et al., 2017]. Typically, the building blocks of this program are defined in a grammar, from which programs are constructed and verified against the specifications. Program synthesis consists of:

- A grammar, the building blocks that create the program space
- An intent, the specification of the desired behavior of a program
- The search, how the program space is explored to find a program that meets the requirements.

A grammar could look something like:

```
1: Number = 1
2: Number = 2
3: Number = 3
4: Number = x
5: Number = Number + Number
6: Number = Number - Number
7: Number = Number * number
```

The search algorithm will start from a root, e.g., a Number, and iteratively apply rules from the grammar and replace the nonterminal symbols in the program with potential right-hand side applications of the rule. The program tree that represents

a program is called an abstract syntax tree (AST) and represents a program constructed from the grammar. Figure 2.3 shows a possible program built from the grammar.

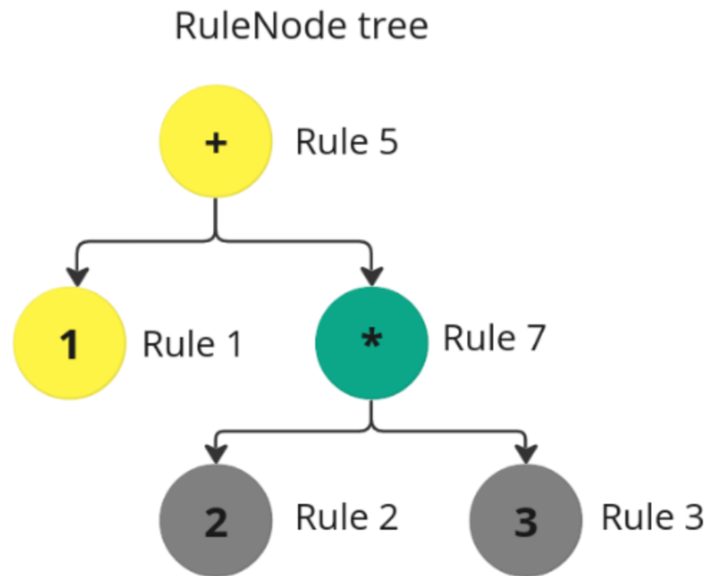


Figure 2.3: Abstract syntax tree representation of a program

2.3.1 SyGuS

Syntax-Guided Synthesis (SyGuS) is a program synthesis paradigm in which the search for a satisfying program is guided by both a semantic specification and a syntactic grammar. The grammar limits the space of candidate programs to those that are syntactically meaningful, while the semantic specification ensures that the synthesized programs meet the desired behavior. This modeling paradigm is well-suited for our purposes because the grammar of a Kappa program can be automatically extracted from agent declarations, and constraints, such as disallowing multiple site declarations, can be enforced during search.

2.3.2 Sketch

The sketch paradigm [Solar-Lezama, 2008] takes a different approach: the user provides a partial program with ”holes” and a specification, and the synthesizer fills in the holes such that the complete program satisfies the specification. In our case, the ”holes” are found in the modifiable parts of Kappa rules, parts of the rule that can be modified without changing the core transformation. We use this idea to

define a constrained search space over possible rule modifications. Each candidate modification is a completion of the original rule with additional conditions.

2.3.3 Herb.jl

For the synthesis process, we use Herb.jl, an efficient Julia-based program synthesis library. Herb.jl supports enumeration-based synthesis guided by grammars, and can be extended with custom constraints. It represents programs as abstract syntax trees (ASTs) that can be constructed using a grammar.

Herb.jl uses the concept of a *RuleNode*, which represents a node in the abstract syntax tree (AST) of a candidate program. Each RuleNode corresponds to a production rule in the grammar and holds references to its child nodes (subtrees). A complete program is thus represented as a rooted tree of RuleNodes. These trees are constructed top-down by choosing, at each position (or "hole"), a production rule from the grammar and recursively expanding its children.

To perform synthesis, Herb.jl traverses the space of all such trees systematically, typically breadth-first or depth-first. At each step, partial programs (trees with unexpanded holes) are incrementally refined until either a complete program is found that satisfies the given specification or the search space is exhausted. Because the number of possible completions grows exponentially, Herb.jl allows constraints to be integrated directly into this traversal: constraints can prune infeasible branches of the search tree before they are fully expanded, significantly improving efficiency.

This combination of grammar-guided enumeration and constraint propagation makes Herb.jl a suitable framework for our work. In our setting, the grammar is automatically derived from a Kappa model's agent declarations, while the constraints encode admissible modifications of rules (see Section 5.2.3). By leveraging Herb.jl, we can systematically generate candidate rule modifications while ensuring that all synthesized programs remain valid Kappa models.

2.3.4 Program Synthesis with constraints

A central challenge in program synthesis is the enormous size of the program space: even small grammars generate exponentially many candidate programs. Many of these are redundant or trivially invalid, so pruning this space is essential. Recent work [Swinkels, 2024] approaches this problem by incorporating constraints directly into the synthesis process, ensuring that only programs consistent with these constraints are ever explored.

Instead of offloading constraints to external SAT/CP solvers, the constraint framework posed by this work lets Herb.jl propagate grammar-level constraints natively during top-down enumeration of the abstract syntax trees (representing candidate programs). This allows constraints (e.g., forbidding specific subtrees, enforcing uniqueness) to be checked and propagated as the program tree is being built. The result is that large portions of the search space can be pruned without ever instantiating invalid candidates. This work shows that constraint propagation can filter

out up to 99% of the search space, yielding up to 50-fold improvements in runtime. This constraint framework will be used to formulate constraints that shape the program space of creating valid Kappa programs (Chapter **??**).

Chapter 3

Related work

Exploring the connection of rule changes to the story of a program is a novel idea, and there is no direct work to compare it to. There is, however, e.g., direct, weak work that also concerns kappa stories, which could help in guiding this research.

3.1 KaSa: A Static Analysis tool

KaSa is another tool within the Kappa language ecosystem [Boutillier et al., 2018a]. It is designed to perform static analysis on Kappa models. This means KaSa analyses the Kappa program's rules and structure without needing to run simulations of the program. The core of KaSa is a fix-point engine that identifies patterns that may never occur during the running of a system, regardless of the initial state. By abstracting the set of reachable states, KaSa can gather various insightful properties about a model.

The goal of KaSa, the static analysis tool of the Kappa ecosystem, is to assist the modeler by warning about potential issues and providing a quick overview of the model. For example, it can detect *dead rules*, i.e., rules that may never be applied in any simulation; it can identify irreversible transformations of proteins, where a rule changes a site to a state that can never be undone; and it can warn about the potential formation of unbounded molecular compounds or polymers, which signal uncontrolled growth of complexes. Unlike our approach, which requires running simulations and extracting causal stories dynamically, KaSa provides static guarantees without executing the model.

In this thesis, we also make use of KaSa's functionality, in particular its *reachability analysis*—to determine whether a rule can fire at all in the system. This analysis is directly relevant to our work, since it also concerns the dependencies and relationships between rules in a Kappa program.

3.2 KaDE

KaDE [Camporesi et al., 2017] is a tool within the Kappa system that compiles Kappa rules into reduced Ordinary Differential Equation (ODE) models. By interpreting rules as an implicit reaction network, KaDE produces a system of ODEs representing species concentrations over time. A distinctive feature of KaDE is its use of static analysis: it identifies equivalence classes of sites within agents and computes so-called bisimulations to reduce the reaction network. This allows KaDE to generate compact ODE systems without enumerating the whole network. Unlike tools like BioNetGen [Harris et al., 2016], which require manual specification of site equivalence, KaDE automates this process, and its reductions remain valid across different parameter values. KaDE's ODE perspective abstracts away the fine-grained causal dynamics, which our work aims to uncover and analyze.

3.3 Matching stories to Program traces

One of the challenges in understanding causal mechanisms in Kappa is in validating whether or not the generated stories actually match the trace from which they are constructed. This is important when we want to study not just what causal paths are possible, but also how frequently particular mechanisms arise during simulations. In his thesis, [Rangan, 2016] developed a formal framework for *story-trace matching*, which aims to determine whether a given story is a valid explanation of a trace produced by simulating a Kappa program.

In the framework, stories are formalized as equivalence classes of event sequences, where equivalence is defined based on a precedence relation over events and a notion of concurrency. The framework defines forms of compression as introduced in [], e.g., direct, weak, and strong, and allows for efficient matching of an existing story to its trace. The main contribution of the work is that it presents an efficient method for matching stories to a trace, rather than generating them from a trace. However, this thesis will also investigate potentially new stories that are produced from modified programs.

Chapter 4

Problem Statement

Understanding how cellular signaling systems give rise to specific outcomes is a central goal in computational biology. Rule-based modeling frameworks such as Kappa provide a means to encode mechanistic knowledge about molecular interactions. From a Kappa program and an initial mixture of agents, a stochastic simulation can generate a causal story: a directed acyclic graph (DAG) that captures the sequence and dependencies of rule applications culminating in a specified event of interest (EOI).

However, an important question remains unanswered: how does the structure of these causal stories change when the underlying rules in the Kappa program are modified?

Currently, there is no systematic method for investigating how perturbations to a model's rules affect the causal explanation of an EOI. Biologists who wish to test hypotheses such as alternative phosphorylation mechanisms or binding site mutations must modify rules manually and interpret simulation outputs on a case-by-case basis. It would be great if we could construct modified programs that return, for example, a set of programs that preserve the path to the EOI but with a different explanation, thus telling a different story. Or one could model the EOI in a system as some event that we do not want to happen, and return the set of programs that will block the path to the event of interest, after which the program can be investigated on which rule conditions would have had to have changed, which might be possible to recreate in a lab setting.

This thesis addresses the following central problem:

> *Given a Kappa program and a set of rule-level modification strategies, how can we systematically explore the space of rule variants and characterize the resulting changes in their causal mechanisms?*

In the sections, we will define some subproblems that we need to answer to help answer this problem.

4.1 Preliminaries

In Section 2.2, we have introduced the kappa language; however, let's define formally what parts of the kappa program constitute the input to our problem. These definitions will help in understanding the issues addressed in this section. A kappa program $\mathcal{P}$ is essentially a collection of statements. For this thesis, we will consider the *agent signature*, the set of *rules*, and the *event of interest* to be the relevant parts of the input to our problem. We assume the program contains initialization statements for the agents.

The *agent signature* consists of a list of agent declarations $\mathcal{A}$. An agent $a \in \mathcal{A}$ contains a name n, and a list of sites s. Each site in s also has a name, and optionally a list of state names. A *rule* is a declaration of the form LHS -¿ RHS. Both the LHS and the RHS consist of a pattern of one or more agents. As mentioned in 2.1, Kappa expressions denote a *site graph*. In a site graph, nodes have a set of sites, which is the interface of the node. These sites can bind to each other, and thus have states regarding their binding or their internal state. The LHS and the RHS of a rule represent a pattern, which is a site graph with agents, and can include sites in a specific state. Lastly, the *EOI* is the pattern that serves as the endpoint of the causal chain of events being modeled. Just like the LHS and the RHS of a rule, it consists of one or more agents in a particular state. This should be included in $\mathcal{P}$.

The simulation tools used to run the program and get the stories can be seen as a function over $\mathcal{P}$ that returns a set of stories $\mathcal{S}$. Each $s \in \mathcal{S}$ is a directed acyclic graph (DAG), where the final node is one representing the EOI. The nodes with depth 0 are the initializations of the agents that have to precede the EOI. The other nodes in the tree represent rules.

4.2 Creating Kappa Programs

The problem is one of construction. We have discussed the need to add conditions to a program's rules, and this can be done manually with ease. A modeler, supplied with an initial program consisting of an agent specification and a set of rules, can use the Kappa Simulation tool to change the rules by hand [Fontana et al., Boutillier et al., 2018b]. This tool utilizes KaSa, the Kappa static analysis tool, to provide live feedback on the Kappa that is written, aiding a modeler in creating Kappa programs. By examining the agent specification and familiarizing oneself with the Kappa syntax, it is possible to create new rules or modify existing ones easily. When a rule is made that violates the syntax, the tool provides feedback. For example, the rules can only specify agents that are initialized; rules that are not syntactically valid will result in an error.

When we create a mechanism to construct rules, those rules must adhere to the syntactic and semantic constraints that Kappa has. One could attempt to make these rules without much regard for these constraints and run them through the analysis tools the Kappa ecosystem provides, but this would be quite infeasible. It would be

much better if we could systematically construct the rules from the building blocks, the kappa syntax, and the agent specification of a particular program imposes. This leads to the problem:

> *Given a program $\mathcal{P}$, enumerate the set of all syntactically valid rules $\{\mathcal{R}_{\mathcal{P}}\}$ that refer to agents and their sites as specified in $\mathcal{P}$*

What rules exactly match the requirements as specified in the statement above, and what it means to be syntactically valid, will be discussed in chapter 5.

4.3 Modifying an existing program

Once we can generate valid rules for a given agent specification, the next step is to explore how modifications to the rules influence a program's behavior. Instead of constructing new rules entirely from scratch, we often wish to perturb an existing program by modifying its rules in a controlled way. Such modifications have biological explanations, for example, an alternative phosphorylation requirement or a more stringent binding context.

To formalize this process, we consider modifications as localized changes to the context of a rule, specifically the conditions that determine whether the rule applies, which remain unchanged after the rule is applied. These include changes to binding states or internal site states. Our goal is to produce variants of the original program in which one or more rules have been modified in such a way.

> *Given a program $\mathcal{P}$ generate the set of modified programs $\{\mathcal{P}'\}$ such that each $\mathcal{P}'$ differs from $\mathcal{P}$ by one or more rule modifications, and each modified rule differs from its original by one rule modification.*

4.4 Programs following a specification

By only considering programs that can be seen as modifications of the original program $\mathcal{P}$, we not only decrease the search space, but also stay in the context of finding programs that are close to the original program. A modeler might want to know how the story changes for programs that contain modified rules by asking for a particular specification that a modified program must have. For example, given a program $\mathcal{P}$, what programs with m modifications to n rules still preserve the path to the event of interest in the stories? In other words, in which modified programs model a system where the event of interest can still be produced? Or a specification about the stories themselves: Which modified programs contain an agent or a rule not present in the original story? Which modified programs have the same story? Which modified programs have the most inhibition arrows?

Someone with domain knowledge about modeling these biochemical programs will be able to give out any specification that can be applied to the information on

the story, and should get a set of programs that conform to that specification. The problem thus can be formulated as:

> *Given a Program $\mathcal{P}$, its corresponding stories $\{S_{\mathcal{P}}\}$ and a specification ϕ, give all modified programs $\mathcal{P}'$ such that $\phi(\mathcal{P}) = 1$*

The specification can be any function given over the story S. A modeler should be able to provide some specification about either the program or its corresponding stories and receive the set of programs that satisfy these requirements.

Chapter 5

Modeling Kappa Programs

The first contribution of this thesis is a grammar for constructing Kappa rules. We show how a grammar, specialized to the agent specification of a given program $\mathcal{P}$, can be used to generate syntactically valid rules. We then extend this grammar with constraints to enforce properties that cannot be captured by a grammar alone, such as site uniqueness, bond pairing, and structural symmetry of a rule. Finally, we introduce constraints to eliminate redundancy by utilizing the canonical ordering of a rule. Together, these techniques model the search space of valid Kappa rules that can be constructed.

5.1 The Search Space

To solve the problem of being able to create modifications of a Kappa program $\mathcal{P}$ in the form of changes to rules, we first need to be able to construct valid Kappa rules, and have them form a "modification" of the original program $\mathcal{P}$.

The first challenge is to be able to create rules that adhere to the Kappa syntax. We now do this manually, typing out the rules or using tools like RuleVis [Abramov et al., 2019] to visually construct them. A Kappa file consists of declarations, which can include rules, variables, agent signatures, etc. For this thesis, we can only examine rules as specified in Section 2.3 of the Kappa manual [Fontana et al.]. Each of these declarations follows a formal grammar in Backus-Naur form (BNF). Program synthesis uses a grammar to create programs, so it could be an idea to reproduce the Kappa grammar in a Program synthesis framework such as Herb.jl []. However, this grammar allows for any label, name, and combination of agents (with their corresponding names and sites), as the program space is intractably large.

The parser of the Kappa Simulator will not allow for referring to agents and sites that are not declared in the agent specification, and capturing this in the grammar will not only limit the search space to only syntactically valid rules corresponding to a program $\mathcal{P}$, but also eliminate the need of using an external validation tool to check if the rule is valid.

By creating our own grammar that is limited to creating rules only corresponding to a particular program $\mathcal{P}$, we reduce the complexity of the Kappa syntax as defined by their grammar, and remove the need for checking the validity of the Kappa rule by an external tool. In section 5.2.3, we will explain that our grammar is not able to capture all requirements to be able to be parsed, a problem we will solve by adding constraints to the grammar.

5.2 Grammar for Kappa Rules

We will use Herb.jl [] to create a grammar, as it allows us to create a grammar that can be extended with custom constraints, which can be very efficiently enumerated. As described in Section 2.3.3, programs in Herb are created by starting from a non-terminal symbol called the start symbol. By applying the rewrite rules of the CFG, we can represent all possible programs in the language. We thus create a grammar to capture rules that adhere to a specific program $\mathcal{P}$, as specified by the agent specification.

We consider a Kappa program that models the interaction of a kinase and a substrate to demonstrate how our program synthesis grammar is constructed with respect to a program, as shown in Listing 5.1.

```
//agent specification
%agent: K(x)
%agent: S(a b{u p} c{u p})

//initialize agents
%init: 100 K()
%init: 100 S()

//rules
'K.S'   K(x[.]), S(a[.]) -> K(x[1]), S(a[1]) @ 1
'K..S'  K(x[1]), S(a[1]) -> K(x[.]), S(a[.]) @ 1
'b++'   K(x[1]), S(a[1], b{u}) -> K(x[1]), S(a[1], b{p}) @ 1
'c++'   K(x[1]), S(a[1], c{u}) -> K(x[1]), S(a[1], c{p}) @ 1
'b--'   S(b{p}) -> S(b{u}) @ 10
'c--'   S(c{p}) -> S(c{u}) @ 10

//event of interest
%obs: 'S++' |S(b{p} c{p})|

//notify KaSTOR to track this event
%mod: [true] do $TRACK 'S++' [true];
```

Listing 5.1: Kinase & Substrate Kappa File

This Kappa program introduces two agents, K and S, representing a kinase and a substrate, respectively. The lines starting with `%init:`, `%obs:`, and `%mod:` are not relevant for this section as they are only used in the simulation of the program. Mainly, the agent specification is used for creating the grammar, and in listing 5.2 we show our program synthesis grammar corresponding to this program.

```
1    1: Rule = AgentPattern --> AgentPattern
2    2: AgentPattern = Agent
3    3: AgentPattern = (Agent, Agent)
4    4: Agent = K(K_site)
5    5: Agent = S(S_site)
6    6: Agent = S(S_site, S_site)
7    7: Agent = S(S_site, S_site, S_site)
8    8: K_site = K_site_x
9    9: K_site_x = x{empty}[Bond]
10   10: S_site = S_site_a
11   11: S_site = S_site_b
12   12: S_site = S_site_c
13   13: S_site_a = a{empty}[Bond]
14   14: S_site_b = b{empty}[Bond]
15   15: S_site_b = b{S_site_b_state}[empty]
16   16: S_site_b = b{S_site_b_state}[Bond]
17   17: S_site_b_state = u
18   18: S_site_b_state = p
19   19: S_site_c = c{empty}[Bond]
20   20: S_site_c = c{S_site_c_state}[empty]
21   21: S_site_c = c{S_site_c_state}[Bond]
22   22: S_site_c_state = u
23   23: S_site_c_state = p
24   24: Bond = 1
25   25: Bond = .
26   26: Bond = _
27   27: empty = ϵ
28   28: emptyagent = ∅
```

Listing 5.2: Grammar

The choice of the production rules as shown in 5.2 will be explained and motivated in this section.

5.2.1 Creating Agents

The first rule in every possible grammar is the starting rule creating the left-hand side and the right-hand side,

```
Rule = AgentPattern -> AgentPattern
```

representing the left-hand side and the right-hand side. Depending on the number of maximum agents per side in the provided rules, each `AgentPattern` can contain one or more `Agent` Types,

```
AgentPattern = Agent
AgentPattern = (Agent, Agent)
AgentPattern = (Agent, Agent, Agent)
```

etc.

This way of constructing the LHS and RHS is different from the grammar in the Kappa manual, where a rule can grow around the -> by the production rule

```
1    ⟨rule-expression⟩ ::= ⟨agent⟩ ⟨more⟩ ⟨agent⟩
2    ⟨more⟩ ::= ,⟨agent⟩ ⟨more⟩ ⟨agent⟩, | ->
```

This way of constructing agents ensures that the number of agents per side stays the same, which is one of the requirements for valid rules. It also allows for an arbitrary depth of the program, which can be limited by the iterator searching through the space. Since the production rules are written in Julia, we cannot copy the exact syntax without throwing errors. Instantiating each side of the rule with a set amount of agents is chosen, as for modifying rules, we will not be adding to the total number of agents per side of the rule.

5.2.2 Filling Agents

The Agent

```
%agent: S(a b{u p})
```

The following grammar productions are created:

```
Agent = S(S_site)
Agent = S(S_site, S_site)
S_site = S_site_a | S_site_b
```

In Kappa, an agent Each site variant (`S_site_a`,`S_site_b`) includes productions for both empty and non-empty internal and binding states. For example:

```
S_site_a = a{empty}[Bond]
S_site_b = b{Empty}[Bond]
S_site_b = b{S_site_b_state}[Empty]
S_site_b = b{S_site_b_state}[Bond]
S_site_b_state = u | p
```

Kappa follows the principle of underspecification or don't care, don't write [Danos et al., 2007], meaning that a rule only needs to mention the aspects of an agent that are relevant to its applicability. Sites omitted from a rule are implicitly unconstrained, meaning they can be in any state (e.g., internal site or bonding site). In our grammar, we enforce this convention structurally. Each site production must contain either an internal state, a bond annotation, or both. This prevents the generation of sites that do not mention anything.

Finally, the grammar is decorated with some rules generating the different Bond types:

```
Bond = 1
Bond = .
Bond = _
```

For this research, we only use the bond values of `[.]`, denoting an unbound site, `[_]`, denoting a bound site, and `[1]`, where the number can be seen as the bond label between two sites.

To further clarify how our grammar represents a Kappa, we illustrate the translation of the rule

```
'b++' K(x[1]), S(a[1], b{u}) ->
      K(x[1]), S(a[1], b{p}) @ 1
```

into an abstract syntax tree (AST). The grammar ensures that each valid rule is represented as a tree, where the root corresponds to the start symbol `Rule`, and subsequent levels correspond to productions applied to nonterminals.

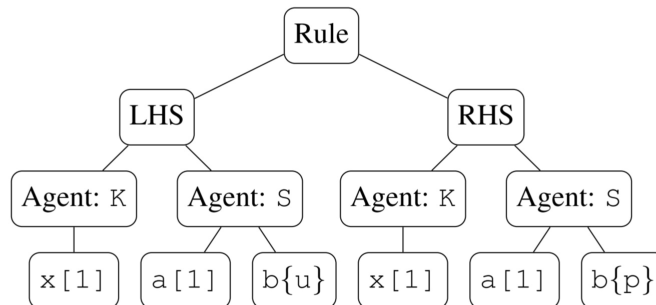


Figure 5.1: High-level AST representation of the `b++` rule. Each agent is represented directly with its sites and states. This diagram abstracts away intermediate grammar productions and highlights the structural correspondence between the left-hand side (LHS) and right-hand side (RHS) of the rule.

Figure 5.1 presents a high-level view of the abstract syntax tree (AST). The tree is divided into a left-hand side (LHS) and a right-hand side (RHS), corresponding to the two sides of the rule. Each side consists of one or more agents, and each agent in turn contains one or more sites. For every site, the grammar specifies either an internal state, a bond state, or both.

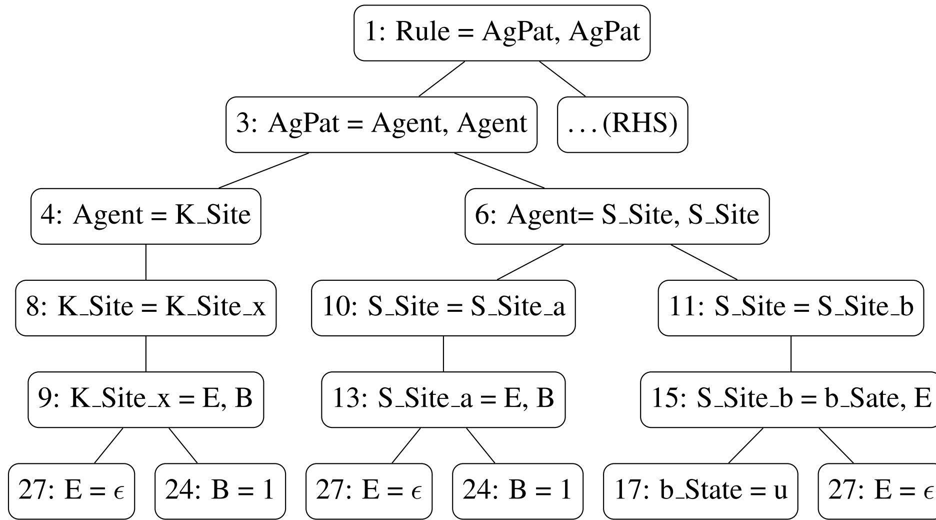


Figure 5.2: Exact AST of the left-hand side of rule `b++` rule using grammar production numbers (cf. rules 1–28). Each node corresponds to the application of a production, with integers denoting the rule used. This diagram demonstrates how the high-level tree in Figure 5.1 is systematically derived from the grammar. Note that E stands for the production rule Empty, and B stands for the production rule Bond.

Figure 5.2 provides the precise derivation tree, where each node is annotated with the index of the production rule applied. For example, the root expands via production 1 (`Rule = AgentPattern -> AgentPattern`), to produce the LHS and the RHS of the rule.

This algorithm can generate a grammar tailored explicitly to the input program $\mathcal{P}$, to create rules that consist of the agents and their sites as specified in the agent signature of $\mathcal{P}$.

5.2.3 Constraints

The context-free grammar used in the synthesis process is capable of generating rules that adhere closely to the syntax of the Kappa language. In particular, it only uses agent and site names and internal states as defined in the agent signature of the source program $\mathcal{P}$. This ensures that the grammar can only generate rules that pass some of the key syntactic checks performed by the Kappa parser.

However, not all requirements for creating valid rules can be encoded in the grammar. Some constraints either require information about other parts of the rule (e.g., a bond label may only appear twice in each side of the rule, as it identifies a bond between exactly two sites), which is beyond the expressiveness of a context-free grammar. We can add constraints to the grammar to prune out rules that the Kappa parser also would flag. By filtering out rules that will not pass through the Kappa parser, we avoid having to call it externally if done correctly.

We divide these into two categories: constraints that are essential for semantic validity (to ensure the Kappa simulator accepts the rule), and constraints that eliminate semantically redundant variants to optimize the search process.

Constraints Ensuring Semantic Validity of Kappa Rules

These constraints are necessary to produce rules that conform to the Kappa language specification:

Structural symmetry Kappa requires that the agents appearing on the left-hand side (LHS) and right-hand side (RHS) of a rule are structurally symmetric. That is, the arrow notation of a rule requires an explicit mapping between agents on the left and agents on the right. Kappa sees it as each side requiring the same number of comma-separated ”slots”, which are occupied by an agent. A rule that does not satisfy the constraint would look like:

```
K(x[.]) S(a[.]) -> S(b[1]) S(a[1]) @ 1
  #1L     #2L         #1R     #2R
```

The Kappa parser maps the agents on both sides from left to right as numbered above, mapping #1L to #1R and #2L to #2R. As it sees a mismatch between #2L and #2R, it throws an error.

Another type of structure preservation is in the sites that are mentioned in the slots of the agents. For example:

```
A(x{p}) -> A() @ 1
A(x[1]) A(x{u}) @ 1
```

The first rule is invalid as it omits x entirely on the RHS. Though this rule appears to “do nothing”, it is invalid and must thus be pruned not to cause errors when parsing the Kappa. The second rule is invalid, in the same fashion as the first rule; we cannot represent a transformation between the LHS and the RHS if we do not mention either the state or the label of a site in both the LHS and the RHS. This is also enforced by the same constraint that dictates the “slots” the agents must occupy.

The underlying propagator enforcing this constraint functions by trying to make each node on the left-hand side and its corresponding symmetric node on the right-hand side equal, except for the leaf nodes. It does this locally by the propagator knowing the “path” through the AST to the node and using that to find the symmetric equivalent node on the other side of the AST. Since every AST created from this grammar starts at the top with :Rule, with the first child being the LHS and the second child being the RHS, we can flip the first value in the path to find the corresponding symmetric node. It attempts to make these values equal so that the structure of the LHS and RHS remains the same. The main assumption behind this constraint is that each program always has a root node and two subtrees with the same depth. This is ensured by the way we structured the grammar, so it works.

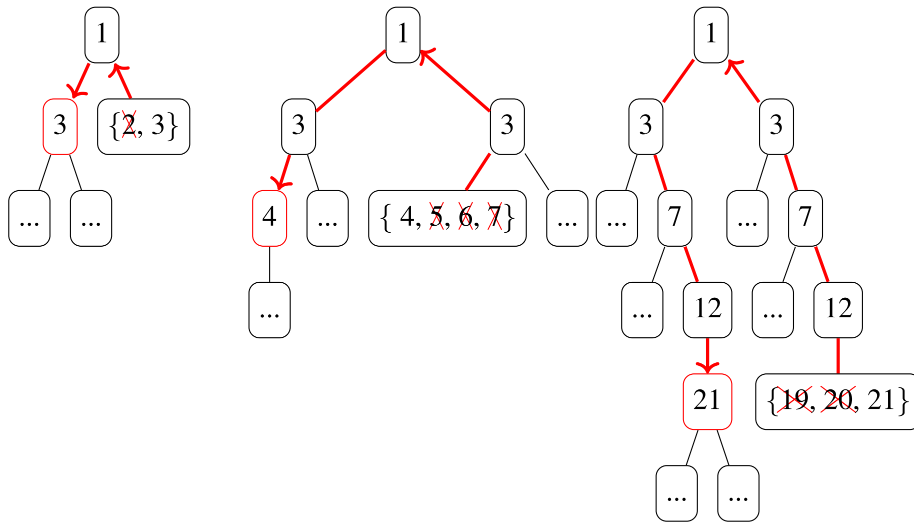


Figure 5.3: Three examples of the workings of the symmetric constraint. Each node gets checked with its corresponding node on the other side of the rule, by using the path that is propagated locally in the constraint.

Note that this not only ensures the same number of agents per side of the rule and the same number of sites per agent, but also the same type of site mentioned. The grammar distinguishes between sites only mentioning a bond, sites mentioning an internal state, and sites mentioning both. This captures the semantic constraint that when an internal site state or a bond state is mentioned on the left-hand side of the rule, it should also be mentioned on the right-hand side of the rule.

Site uniqueness Each agent must specify each site name at most once. Without this constraint, a grammar might generate agents like `A(b{u} b{p})`, which are invalid in Kappa. We enforce a site uniqueness constraint so that each site appears only once per agent instance. We do this by creating a modified Unique constraint that only activates from a certain depth. Since all programs created by our grammar have a set depth (and grow in width for more agents and sites), we can set this depth to 2, therefore setting the unique constraint as implemented in the library from all the subtrees representing agents. Figure 5.4 shows that from depth 2, it posts the unique constraints for the rules that specify the site names: 10: S_Site = S_Site_a, 11: S_Site = S_Site_b, and 12: S_Site = S_Site_c.

Bond pairing Bond labels such as the 1 in `K(x[1])` refer to named links between exactly two sites. Kappa requires that each such bond label appear exactly twice in the LHS or RHS of a rule, once on each site it connects. The grammar cannot enforce this global constraint, so we impose our own created paired occurrence constraint during the synthesis. It sets the solver to infeasible when a side of the rule contains more than two occurrences of a rule representing a bond label, and

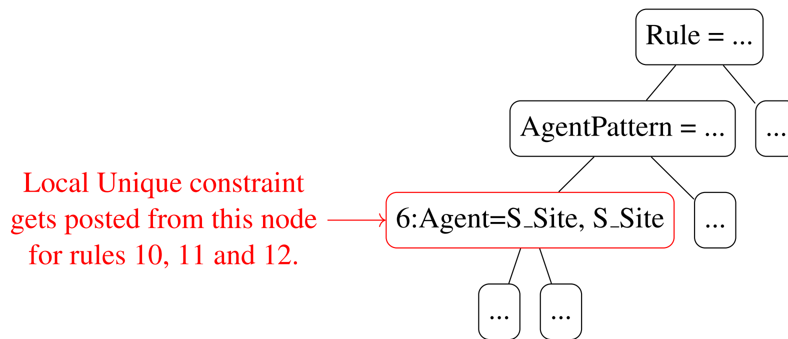


Figure 5.4: The Unique constraint is posted from the depth level where agents and their sites are declared, and from this point on, the rules representing the sites must be unique.

also does not allow programs where the rule representing a bond label appears only once on one side of the rule.

This constraint is posted on the left-hand side and the right-hand side of the rule separately. It keeps a count of the occurrences of the rule, and when it exceeds two, it sets the solver to infeasible. If the count is exactly two, the constraint is considered satisfied, and all unfilled holes have that rule pruned from its domain. If the count is one and there is no place left where that rule could be filled in, the solver is set to infeasible.

No underspecified bond formation Kappa rules must not allow ambiguous or underspecified bond creation. For example, the rule: `A(x[.]) -> A(x[_]) @ 1` Would ask the simulator to change a bond from a free site `x[.]` to an unspecified partner `x[_]`, which is invalid. The simulator cannot resolve which agent or site should be involved on the other end of the bond.

Another semantic requirement is that a rule cannot have all forms of transformations between bonds. for example, `K(x[_])` → `K(x[.])` denotes a bond degrading, but a transformation is under specified when we would write `K(x[.])` → `K(x[_])`. Kappa does not know which bond must form, so we need to make sure that these rules are not allowed.

Assuming the program adheres to the symmetry constraints as forced in the symmetry constraint, the propagator checks all rules containing rule Bond as a child, uses the path to this node to retrieve the counterpart in the right-hand side of the rule, and removes the rule number that corresponds to Bond = _ when the node on the left-hand side contains either rule corresponding to a bond label (in this case rule 24: Bond = 1, or the rule corresponding to unbound (rule 26: Bond = .). Figure 5.5 shows an example of the rule being removed from the domain of the corresponding node.

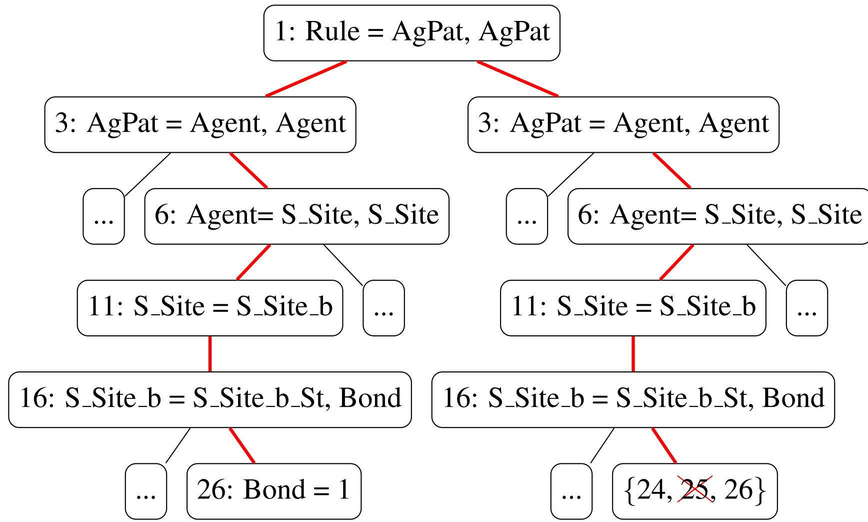


Figure 5.5: Program tree showing rule 25: Bond = _ in the right-hand side to exclude it from the domain once the counterpart in the left-hand side of the tree is filled with a specified bond label.

Constraints to Reduce Redundancy and Optimize Search

These constraints are not required for semantic validity but serve to reduce the search space by eliminating redundant rule variants:

Canonical ordering Rules that differ only by the order of agents or sites are semantically equivalent in Kappa due to the unordered nature of pattern matching. For example, the two rules:

```
A(x[.]), B(y[.]) -> A(x[1]), B(y[1]) @ 1
B(y[.]), A(x[.]) -> B(y[1]), A(x[1]) @ 1
```

are considered equivalent by the simulator, as the relative order of A and B does not affect the match or transformation. To avoid enumerating such permutations, we impose a canonical ordering constraint on agents and their sites, sorting according to the rule ID. This reduces duplicates during synthesis. The grammar is searched for all AgentPattern and Agent rules, and adds an ordered constraint to that rule and its children. This captures both the ordering of the agents themselves and the sites within each agent.

pruning non-firing rules Rules that cannot be applied in any reachable state are meaningless within the model. Although the grammar might generate such rules, we apply pruning based on static analysis or prior simulation data to eliminate rules that never fire. This constraint We implement this constraint by checking the original set of kappa rules in a program. If, in the RHS of a rule, a site never forms

a bond, we can add the ForbiddenSequence constraint to forbid the Bond rule from appearing after the rule that specifies that site.

Chapter 6

Modifying Kappa Programs

6.1 Modifications to the rules

Given a Program $\mathcal{P}$, we can now construct a context-sensitive grammar g that can generate all rules that pass through the Kappa parser, addressing the first problem in Chapter 4. The space of all possibilities is still enormous, and in the second problem of Chapter 4, we want to have the notion of a modified program P with one or more modifications to one or more rules. In this section, we will outline what we define as being a "modification" of a rule and our method, inspired by Sketch [Camporesi et al., 2017] of creating these modifications.

6.1.1 Modifying Rules: Core vs. Context

In the example program of Listing 5.1, the model consists of 6 rules. Consider the phosphorylation rule

```
'b++'   K(x[1]), S(a[1], b{u}) ->
        K(x[1]), S(a[1], b{p}) @ 1
```

This rule states that the substrate agent `S` undergoes phosphorylation at site `b`, provided that it is bound via site `a` to site `x` of kinase `K`. The distinction here lies between the part of the rule that changes, namely the internal state of site `b` in agent `S`, and the part that remains unchanged, i.e. the bonding between `K(x)` and `S(a)`.

We refer to the transition from `b{u}` to `b{p}` as the *core rule change*. By contrast, the surrounding requirements, in this case the binding configuration that enables this modification, constitute the *context*. This distinction is essential: biologists are often less interested in altering the fundamental transformation (e.g. phosphorylation itself) than in probing how the *conditions* under which it occurs affect the causal structure of the system.

Altering the core transformation would redefine the very set of possible events in the system, effectively creating a different biochemical model rather than a variant of the original. In contrast, modifying only the context preserves the identity of the

event while changing the circumstances in which it arises. It is in this restricted sense that we use the term *rule modification*: throughout this work, modifications refer exclusively to adjustments of contextual conditions, not to the core transformation itself. This ensures that we remain within the system we modify, where observed differences in stories can be attributed to conditional modifications rather than whole changes in system dynamics.

6.1.2 Types of modifications

In line with the distinction between the core and context of a rule described in Section 6.1.1, we define rule modifications as changes that add further contextual conditions while leaving the core transformation intact. This choice ensures that modifications preserve the identity of the biochemical event (e.g., phosphorylation) while probing how additional requirements alter the causal structure of the system. We deliberately exclude modifications that remove conditions, as such changes may compromise the biological relevance of the model.

We consider two types of modifications to a rule:

i. **Adding an internal site state.** This introduces an additional condition on a site's phosphorylation state.

ii. **Adding a bonding condition.** This introduces the condition that the rule requires that a site be either explicitly unbound, or bound to another site.

Let's look at the example

```
K(x)
S(a b{u p} c{u p})

'b++'   K(x[1]), S(a[1], b{u}) ->
        K(x[1]), S(a[1], b{p}) @ 1
```

we can apply type (i) by requiring that site c on agent S is unphosphorylated:

```
'b++'   K(x[1]), S(a[1], b{u}, c{u}) ->
        K(x[1]), S(a[1], b{p}, c{u}) @ 1
```

Biologically, this means that phosphorylation at b only occurs if c is also in the unphosphorylated state.

Similarly, a type (ii) modification can enforce that site b must be unbound:

```
'b++'   K(x[1]), S(a[1], b{u}[.]) ->
        K(x[1]), S(a[1], b{p}[.]) @ 1
```

Modifications must affect the same structural location on both sides of the rule. For example:

```
'invalid rule'  K(x[1]), S(a[1], b{u}[.]) ->
                K(x[1]), S(a[1], b{p}, c[_]) @ 1
```

is rejected by the Kappa parser, as the left- and right-hand sides no longer align structurally. Likewise, the following modification

```
'change core'  K(x[1]), S(a[1], b{u}[.]) ->
        K(x[1], S(a[1], b{u}[_])
```

remains syntactically valid but introduces a *new core change* (bond creation on site b), and is therefore outside the scope of our definition of contextual modifications.

6.1.3 Modifying RuleNodes

To implement the modifications of the rules, we translate each rule in the original program $\mathcal{P}$ into a RuleNode, the abstract syntax tree (AST) representation used by Herb.jl. This is done with a custom parser that processes each line of the Kappa file and constructs the corresponding RuleNode structure. Once in this representation, we can systematically scan the left-hand side (LHS) of each rule for sites where additional contextual conditions may be inserted.

For example, consider the rule fragment

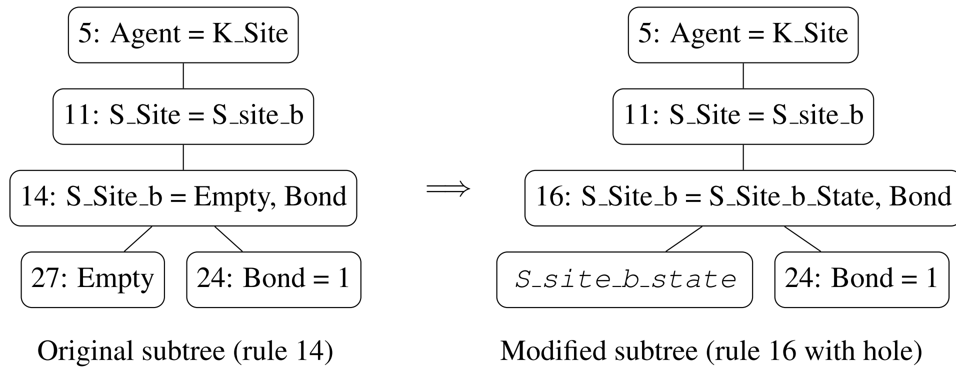
```
K(x[1]), S(a[1], b{u})
```

Here, four distinct modifications are possible, all affecting the substrate agent S:

1. Adding an internal state condition to site a.
2. Adding a bonding condition to site b.
3. Expanding S to include site c with an internal state.
4. Expanding S to include site c with a bonding condition.

Each such modification opportunity is represented by inserting a *hole* into the RuleNode. This follows the principle introduced in Sketch [Solar-Lezama, 2008]: a hole is a nonterminal symbol left unspecified, which can later be filled in by the grammar during program enumeration. In our case, a hole placed at site b corresponds to the nonterminal Bond. In contrast, a hole at site c corresponds to either an internal state nonterminal or a bond nonterminal, depending on the type of modification. The Herb.jl enumerator then explores all completions of the RuleNode, generating a set of candidate rules that instantiate the hole in different ways.

To preserve the structural symmetry of Kappa rules, any modification introduced on the LHS must also be copied to the corresponding location on the RHS. This ensures that the contextual modification does not inadvertently introduce a new core transformation, but rather strengthens the conditions under which the existing transformation can occur.



Original subtree (rule 14) Modified subtree (rule 16 with hole)

6.2 Synthesizing new programs

Now we are equipped with a tool that can load in a Kappa program $\mathcal{P}$, create a Program synthesis grammar that capture the building blocks and constraints to create rules that are valid Kappa with respect to the agent signature of $\mathcal{P}$, load in the rules of a Kappa program into the RuleNode program representation of Herb.jl and generate modified rules of a program.

We refer back to the problem:

> *Given a program $\mathcal{P}$ generate the set of modified programs $\{\mathcal{P}'\}$ such that each $\mathcal{P}'$ differs from $\mathcal{P}$ by one or more rule modifications, and each modified rule differs from its original by one rule modification.*

Using the method described in Section 6.1.3, we can now begin the process of generating the rules that semantically refer to program $\mathcal{P}$. By applying a rule modification to one or more rules of a program, we create a modified program. Algorithm 1 shows our algorithm for exploring modified programs. The algorithm takes as input the starting Kappa program $\mathcal{P}$, its set of rules, and index i (beginning in 1), and calls itself with $i{+}1$ per variant of rules[i]. As you can see, this process will exhaustively search for all programs with combinations up to $max_changed_rules$, which is set to the total amount of rules, but can be changed to a lower value.

Algorithm 1: ExploreRecursively (recursive exploration of modified Kappa programs)

```
Function ExploreRecursively(P, rules, i, max_changed_rules):
    if i > max_changed_rules then
        reachable ← RunKaSa(P);
        if IsDead(reachable, "goal") then
            RecordDead(P);
            return
        RunKaSimKaStor(P);
        S ← StoryFiles();
        if S = ∅ then
            RecordReachableNoStory(P);
        else
            RecordSurvivor(P, S);
        return
    r ← rules[i] ;
    if NotInStory(r, P) then
        ExploreRecursively(P, rules, i+1);
    V ← GetRuleModifications(r);          // modified
     variants of r
    foreach v ∈ {r} ∪ V do
        old ← rules[i]; rules[i] ← v; P.rules ← rules;
        ExploreRecursively(P, rules, i+1, changed + 1[v ≠ r],
         max_changed);
        rules[i] ← old ;                     // backtrack
```

At rule i, the algorithm uses the rule modification algorithm to expand the rule to all possible variants, as described in Section 6.1.3. For each variant v, the original program is updated by replacing rule i with v. If the resulting program no longer reaches the EOI, exploration of this branch is halted. Otherwise, the algorithm proceeds recursively to the following rule index.

Because of the causal dependencies between rules and stories, we can prune the search space. If a single modified rule already prevents the original program from reaching the EOI, then any program that contains this rule will also fail. So, these branches don't need to be explored further.

To exploit this property, we introduce a preprocessing step. Before the search, we evaluate all single-rule variants of the original program, record which ones make the EOI unreachable, and store them in a `forbidden_variants` list. During exploration, whenever the algorithm encounters a rule variant contained in this list, it immediately discards that branch, avoiding redundant exploration.

Chapter 7

Results

The goal of this thesis is to explore the causal structure of a program with sharpened rule conditions, utilizing a novel program synthesis framework. Using the grammar and constraints, it can generate rules that are valid with respect to the given program $\mathcal{P}$. Furthermore, using a sketch-based approach, we can add modifications to the rules of a program, where a modification means adding a condition to the context of that rule. A modified program is a program where one or more rules are modified using this method. In this section, we first want to evaluate how well the grammar and the constraints can capture this search space. We aim to answer the following experimental research questions:

1. How much do the constraints reduce the search space of possible rules?
2. How much do we reduce the search space of Programs to Modified programs?

The second part of this section is dedicated to demonstrating the power of automatically exploring the space of modified programs. Following the program synthesis by specification paradigm, we can ask for programs that adhere to a user-specified specification. Since adding conditions to the context of a program's rules can make the specified event of interest unreachable, we want to find only programs that preserve the path to the EOI. Furthermore, we want to test whether or not the method can efficiently return programs that adhere to any user-given specification. To answer this, the following questions are posed:

3. Can we find modified programs that preserve the path to the event of interest?
4. Can we find one or more modified programs that adhere to a specification?

We first show how we have set up the experiments in the next section, and afterwards we will answer the posed experimental research questions.

7.1 Experimental setup

The experiments are done on a system running an AMD Ryzen 5 5600H processor and 16 GB RAM. The methods were implemented in Julia using the Herb.jl program synthesis library. For simulation of the Kappa programs, we made use of the following three tools from the Kappa Ecosystem [1]:

- **KaSa** (static analysis) to verify the reachability of the event of interest (EOI)
- **KaSim** (simulation) to run the Kappa programs and generate an event trace
- **KaSTOR** (causal analysis) to extract the stories from the simulation trace

All Kappa programs used in the experiments have a `%obs: 'EOI' | <eoi_state> |` line in the file, which can be used to track the specified event of interest (EOI) during the simulation.

To evaluate the rule modification algorithm of section 6.1, we use the following Kappa programs:

7.1.1 KS - Processive

This Kappa program models two agents, agent K representing a kinase that can bind to and phosphorylate agent S representing a substrate[2]. In this processive variant of the kinase-substrate system, site x of agent K can only bind to site k of agent S. The phosphorylation at sites x and y of agent S can only occur once this bond is formed.

```
1     %agent: K(x)
2     %agent: S(x{u p},y{u p},k)
3
4     'K.S' K(x[.]), S(k[.]) -> K(x[1]), S(k[1]) @ 1
5     'x++' K(x[1]), S(x{u}, k[1]) -> K(x[1]), S(x{p}, k[1]) @ 10
6     'y++' K(x[1]), S(y{u}, k[1]) -> K(x[1]), S(y{p}, k[1]) @ 10
7
8     'K..S' K(x[1]), S(k[1]) -> K(x[.]), S(k[.]) @ 1
9     'x--' S(x{p}) -> S(x{u}) @ 1
10    'y--' S(y{p}) -> S(y{u}) @ 1
11
12    %obs: |S(x{p} y{p})|                              // EOI
```

Listing 7.1: KS Processive

Running this program gives one story, shown in Figure 7.1. The goal is reached once rules `y++` and `x++` have occurred, which can happen concurrently. They both depend on rule `K.S`; however, this can happen after the introduction of agent K and S.

[1] `https://kappalanguage.org/download`

[2] `https://www.di.ens.fr/~feret/teaching/2024-2025/MPRI.2.19/activities/causality/causality_03_proc.ka`

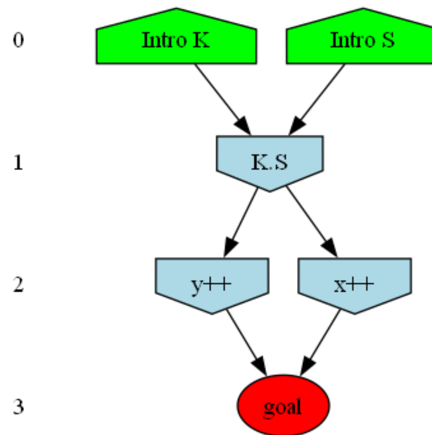


Figure 7.1: The story of KS_Processive

7.1.2 KS - Distributive

This model[3] differs slightly from the previous in that two bonds can form between the agents: one connecting to site x and the other connecting to site y of agent S. The phosphorylation rules `x++` and `y++` are also changed in that sites x and y can only phosphorylate when that specific site is bound to agent K.

```
%agent: K(x)
%agent: S(x{u p},y{u p},k)

'K.Sx' K(x[.]), S(x[.]) -> K(x[1]), S(x[1]) @ 1
'K.Sy' K(x[.]), S(y[.]) -> K(x[1]), S(y[1]) @ 1

'x++' K(x[1]), S(x{u}, x[1]) -> K(x[1]), S(x{p}, x[1]) @ 10
'y++' K(x[1]), S(y{u}, y[1]) -> K(x[1]), S(y{p}, y[1]) @ 10

'K..Sy' K(x[1]), Sy[1]) -> K(x[.]), S(y[.]) @ 1
'K..Sx' K(x[1]), Sx[1]) -> K(x[.]), S(x[.]) @ 1

'x--' S(x{p}) -> S(x{u}) @ 1
'y--' S(y{p}) -> S(y{u}) @ 1

%obs: |S(x{p} y{p})|                                  // EOI
```

Listing 7.2: KS Distributive

Figure 7.2 shows the story of the KS distributive model. It is similar to the story of the KS processive model, differing in the separate binding rules needed for the phosphorylation rules `y++` and `x++`. Both “paths” of events leading to the event of interest are completely concurrent, only needing the introduction of the agents. Note that two K agents are required for the formation of the event of interest.

[3] `https://www.di.ens.fr/~feret/teaching/2024-2025/MPRI.2.19/activities/causality/causality_04_distrib.ka`

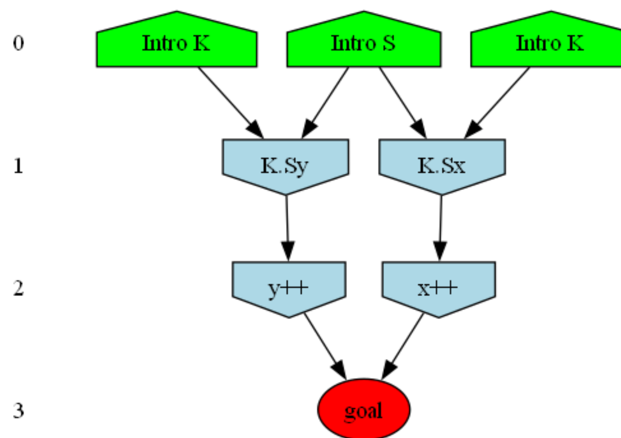


Figure 7.2: The story of KS_distributive

7.1.3 ABC model

This program [4] differs from the KS models described above because it involves three agents. As shown in Figure 7.3, the story illustrates how the rules lead to the event of interest, specifically an agent C with both its sites phosphorylated. Note that this program does not include any concurrency in the rules mentioned in the story, except for the introduction nodes. First, agents A and B bind together at both of their sites named x. Only when site x of agent A is bound can it form a bond from its site c to site x1 of agent C. Site x1 can then be phosphorylated when that bond is present, and also when site x2 remains unbound. Afterward, this rule dissolves the bond. The rule `a.c` can then occur, leading to the final `mod x2` rule, which phosphorylates site x2.

```
%agent: A(x,c)
%agent: B(x)
%agent: C(x1{u p},x2{u p})

'a.b' A(x[.]), B(x[.]) -> A(x[1]), B(x[1]) @ 1
'a..b' A(x[1]), B(x[1]) -> A(x[.]), B(x[.]) @ 1

'ab.c' A(x[_], c[.]), C(x1{u}[.]) -> A(x[_], c[2]), C(x1{u}[2]) @
    1
'mod x1' A(c[1]), C(x1{u}[1], x2[.]) -> A(c[.]), C(x1{p}[.], x2
    [.]) @ 1
'a.c' A(x[.],c[.]),C(x1{p}[.],x2[.]{u}) -> A(x[.],c[1]),C(x1{p
    }[.],x2[1]{u})@ 1
'mod x2' A(x[.],c[1]),C(x1{p},x2{u}[1]) -> A(x[.],c[.]),C(x1{p},x2
    {p}[.]) @ 1

```

[4] https://www.di.ens.fr/~feret/teaching/2024-2025/MPRI.2.19/activities/causality/causality_01_abc.ka

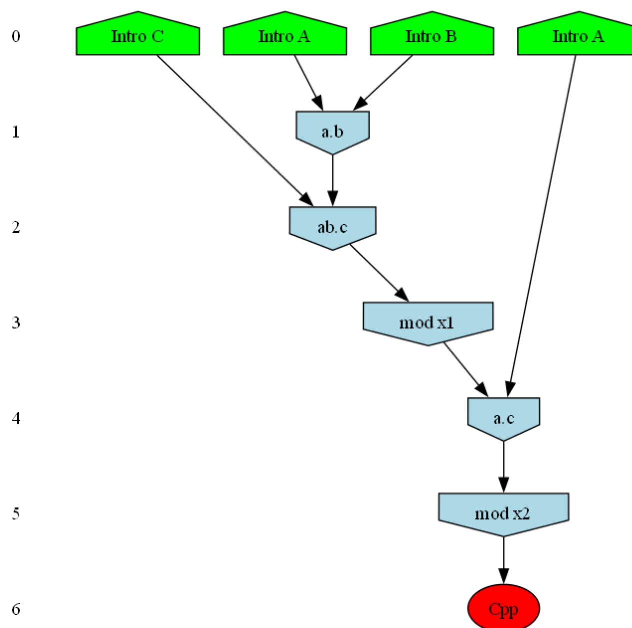


Figure 7.3: Story of PKT Program

```
13 %obs: 'Cpp' |C(x1{p},x2{p})|                    // Event of
    Interest
```

7.1.4 early EGFR model

This program[5] models a part of the EGFR signalling network. In this program, the event of interest is modeled as `Grb2(SH2[_], SH3n[1]), SoS(PR[1])`, the recruitment of SoS. There are five different kinds of agents interacting in the model, governed by 20 rules.

```
1  %agent:   EGF(r)
2  %agent:   EGFR(CR,C,N,L,Y1016{u p},Y1092{u p},Y1172{u p})
3  %agent:   SoS(PR,S{u p})
4  %agent:   Shc(Y{u p},PTB{u p})
5  %agent:   Grb2(SH3c,SH3n,SH2{u p})
6
7  'EGFR.EGFR'
8  EGF(r[2]), EGFR(L[2],CR[.],N[.],C[.]), EGF(r[3]), EGFR(L[3],CR[.],
       N[.],C[.]) ->
9  EGF(r[2]), EGFR(L[2],CR[1],N[.],C[.]), EGF(r[3]), EGFR(L[3],CR[1],
       N[.],C[.]) @1
10
11 'EGFR/EGFR'
```

[5]https://www.di.ens.fr/~feret/teaching/2024-2025/MPRI.2.19/activities/causality/causality_05_sos.ka

```
12 EGF(r[2]), EGFR(L[2],CR[1],N[.],C[.]), EGF(r[3]), EGFR(L[3],CR[1],
       N[.],C[.]) ->
13 EGF(r[2]), EGFR(L[2],CR[.],N[.],C[.]), EGF(r[3]), EGFR(L[3],CR[.],
       N[.],C[.]) @1
14
15 'EGF.EGFR' EGF(r[.]), EGFR(L[.],CR[.]) -> EGF(r[1]), EGFR(L[1],CR
       [.]) @ 1
16 'EGF/EGFR' EGF(r[1]), EGFR(L[1],CR[.]) -> EGF(r[.]), EGFR(L[.],CR
       [.]) @ 1
17 'Shc.Grb2' Shc(Y[.]{p}), Grb2(SH2[.]) -> Shc(Y[1]{p}), Grb2(SH2
       [1]) @ 5
18 'Shc/Grb2' Shc(Y{p}[1]), Grb2(SH2[1]) -> Shc(Y{p}[.]), Grb2(SH2
       [.]) @ 1
19 'EGFR.Grb2' EGFR(Y1092{p}[.]), Grb2(SH2[.]) ->
20     EGFR(Y1092{p}[1]), Grb2(SH2[1]) @ 1
21 'EGFR/Grb2' EGFR(Y1092{p}[1]), Grb2(SH2[1]) ->
22     EGFR(Y1092{p}[.]), Grb2(SH2[.]) @ 1
23 'EGFR.Shc' EGFR(Y1172{p}[.]), Shc(PTB[.]) ->
24     EGFR(Y1172{p}[1]), Shc(PTB[1]) @ 1
25 'EGFR/Shc' EGFR(Y1172{p}[1]), Shc(PTB[1]) ->
26     EGFR(Y1172{p}[.]), Shc(PTB[.]) @ 1
27 'Grb2.SoS' Grb2(SH3n[.]), SoS(PR[.],S[.]{u}) ->
28     Grb2(SH3n[1]), SoS(PR[1],S[.]{u}) @ 1
29 'Grb2/SoS' Grb2(SH3n[1]), SoS(PR[1]) -> Grb2(SH3n[.]), SoS(PR[.])
       @ 1
30 'EGFR.int' EGFR(CR[1],N[.],C[.]), EGFR(CR[1],N[.],C[.]) ->
31     EGFR(CR[1],N[2],C[.]), EGFR(CR[1],N[.],C[2]) @ 1
32 'EGFR/int' EGFR(CR[1],N[2],C[.]), EGFR(CR[1],N[.],C[2]) ->
33     EGFR(CR[1],N[.],C[.]), EGFR(CR[1],N[.],C[.]) @ 1
34 'pY1092@EGFR' EGFR(N[1]), EGFR(C[1],Y1092{u}[.]) ->
35     EGFR(N[1]), EGFR(C[1],Y1092{p}[.]) @ 1
36 'pY1172@EGFR' EGFR(N[1]), EGFR(C[1],Y1172[.]{u}) ->
37     EGFR(N[1]), EGFR(C[1],Y1172[.]{p}) @ 1
38 'uY1092@EGFR' EGFR(Y1092[.]{p}) -> EGFR(Y1092[.]{u}) @ 1
39 'uY1172@EGFR' EGFR(Y1172{p}[.]) -> EGFR(Y1172{u}[.]) @ 1
40
41 'pY@Shc' EGFR(Y1172{p}[1]), Shc(PTB[1],Y{u}[.]) ->
42     EGFR(Y1172{p}[1]), Shc(PTB[1],Y{p}[.]) @ 1
43
44 'uY@Shc' Shc(Y{p}[.]) -> Shc(Y{u}[.]) @ 1
45
46 %var: 'sos recruited'   |Grb2(SH2[_],SH3n[2]),SoS(PR[2])|
        // EOI
```

Listing 7.3: early_EGFR

This program has two different stories leading to the event of interest, as shown in Figures 7.4 and 7.5.

7.2 Search space of RuleNodes

The first research question we address is: *How much do the constraints reduce the search space of possible rules?* To evaluate this, we measure the size of the search

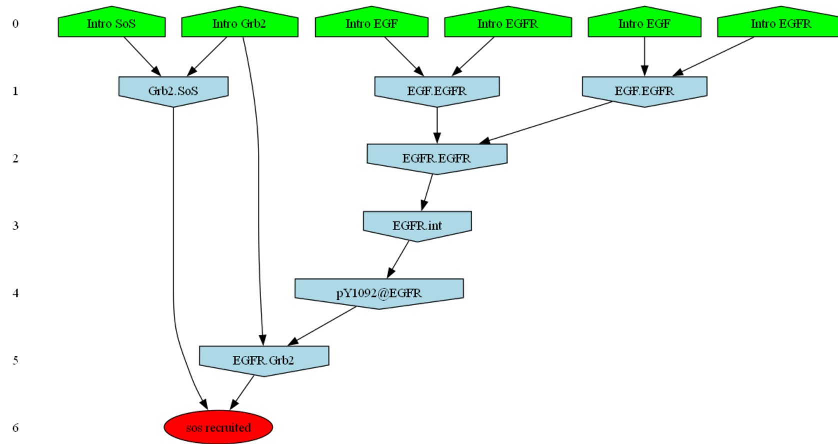


Figure 7.4: Early Egfr Story 1

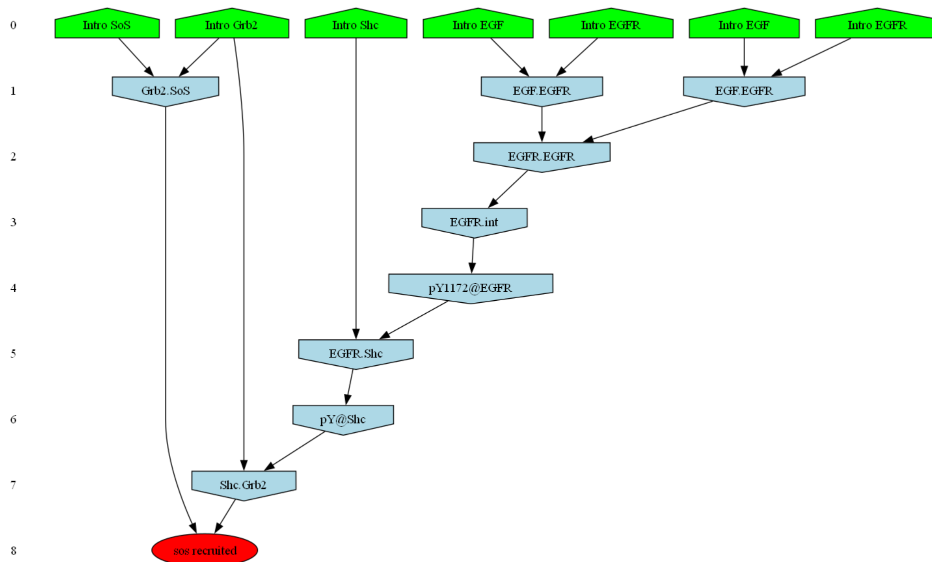


Figure 7.5: Early EGFR Story 2

space generated by our grammar both with and without constraints.

In the unconstrained case, the grammar enumerates all rules that can be constructed from the grammar given an agent signature, which quickly leads to an exponential growth in the number of candidates as the number of agents per rule side increases. For example, with three agents carrying multiple sites, the number of unconstrained rules can already exceed tens of millions. Many of these candidates are syntactically invalid and would be rejected by the Kappa parser. Figure 7.6 shows this explosion of possibilities. For the line representing the agent signature `K(x) S(a b)`, getting the length of the grammar did not finish within an hour. For agent signatures with increased complexity, the grammar length could only be calculated for programs with one agent per side of the rule could be calculated, looking at the trend of the three lines present in the figure they will be extremely big[6]

Applying the constraints introduced in Section 5.2.3, we can see how effective the reduction in search space becomes. Figure 7.7 shows, for a range of agent signatures, how the number of generated rules increases as we allow more agents per side of a rule. As expected, the unconstrained grammar exhibits exponential growth: already for three agents with multiple sites, the number of candidate rules can exceed tens of millions.

Introducing constraints drastically prunes this space. For instance, in the case of `K(x) S(au p)` with three agents per side, the unconstrained grammar produces 2,414,916 rules. Adding structural constraints reduces this number to 26,612, and further applying an *ordering constraint* reduces it to 3,866, which is an additional 85.5% reduction relative to the constrained space.

The effect is even more pronounced for more complex agents. In `K(x) S(a b c)`, the unconstrained grammar already yields 675,684 rules for a single agent per side. With constraints this grows to 906,674 because additional agents per side are allowed, but applying the ordering constraint cuts this space by 97.7%, leaving 20,957 rules at two agents per side.

Finally, for `K(x) S(au p bu p c)`, the unconstrained grammar size could not be computed, but with constraints we obtain 17,846 rules. All generated rules are valid by construction, since the constraints ensure that only valid rules are produced. To confirm this, we additionally checked every rule with KaSa, all were accepted without error.

---

[6]The the size of the unconstrained grammar includes all combinations of sites (and their internal states or bond states) across all agents, as well as all possible arrangements of these agents in all positions specified by the rule. It can be calculated with the formula $a_n = \left(\sum_{k=1}^{n} a^k\right)^2$, where a is the number of distinct agents that can appear in the rule and n is the number of agents allowed per side. For the simple signature `K(x) S(a)`, the possible concrete agents are `K(x[.])`, `K(x[_])`, `K(x[1])`, `S(a[.])`, `S(a[_])`, `S(a[1])`, i.e., six in total. Indeed, the math matches the observed values: $36 = (6)^2$, $1764 = (6+6^2)^2$, $66564 = (6+6^2+6^3)^2$.. Using this math, the agent signature `K(x) S(au p b c` with two agents per side would lead to $\sim 7.43 \times 10^{14}$ total programs to be enumerated. Running every rule through the Kappa parser to check its validity is not reasonable, and with a number exceeding the trillions, we have to prune out faulty rules before the parser can catch them.

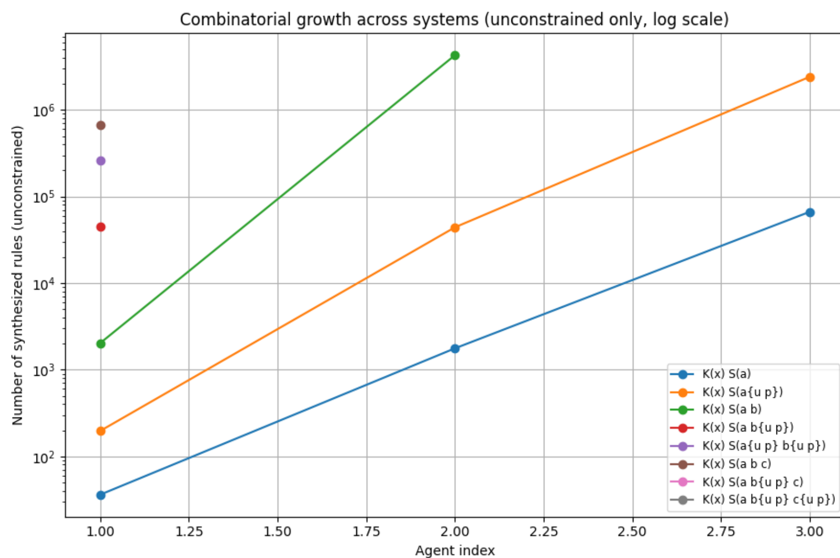
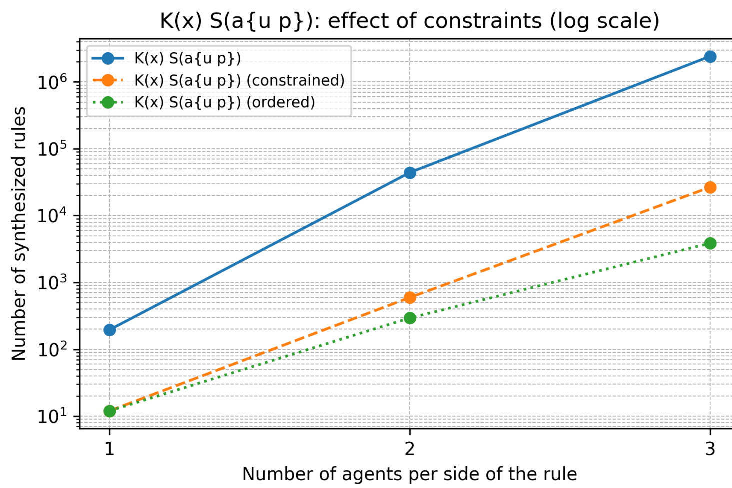


Figure 7.6: Unconstrained growth of possible Kappa rules before applying constraints. Each curve corresponds to an agent specification with increasing complexity of the `S` agent. The x-axis shows the number of agents per rule side, corresponding to expansions of the `AgentPattern` nonterminal. The y-axis shows the total number of distinct rules generated.

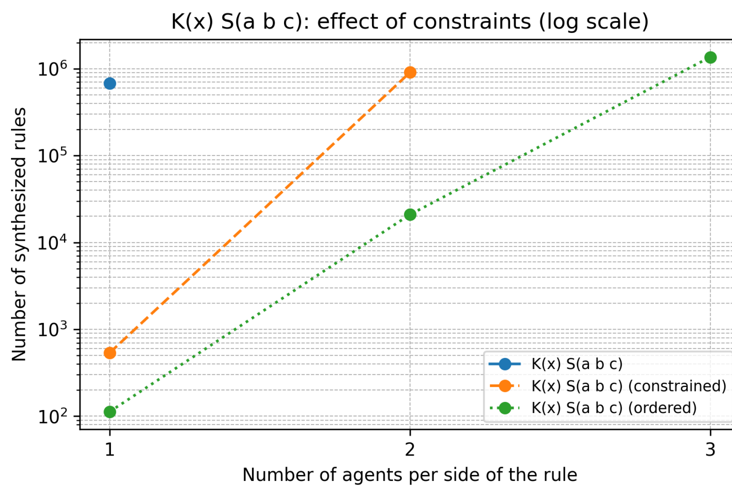
This answers the first research question posed in this chapter: the constraints not only make explicit parsing unnecessary but also drastically shrink the search space of possible programs.

7.3 Search space of Modified Programs

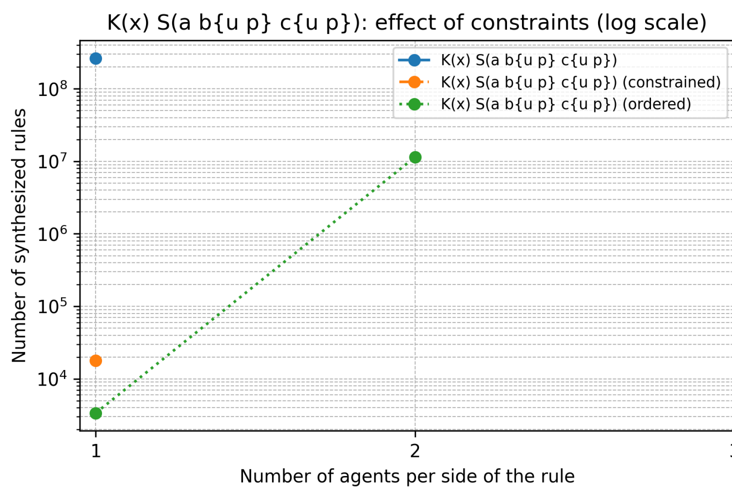
Recall that the modifications we can make to the rules involve either adding an internal site state condition to the context or a bonding condition of a site. Table 7.1 shows how many rule modifications are possible per rule. The first column indicates the number of rules that ultimately can be constructed when we keep applying the modification algorithm to the rule until it is fully specified. The second column shows the number of modified rules that can be obtained when using only one modification step. Both numbers include the original rule. The values under "Forbid seq mods" represent the number of possible rule modifications when implementing the forbidden sequence constraint from 5.2.3, excluding the addition of bonding conditions to sites that will never form a bond in the system's core ruleset. Lastly, running KaSim and KaSTOR on the original program reveals that the only rules that appear in the story are K.S, x++, and y++. For the purposes of analyzing stories, we are only interested in modifying the rules that appear in one or more stories generated by the original program.



(a) K(x) S(a{u p}



(b) K(x) S(a b c)



(c) K(x) S(a b{u p} c{u p})

Figure 7.7: Effect of constraints on rule counts for different agent signatures. Each plot shows the unconstrained, constrained, and ordered search spaces (log scale).

Table 7.1: Number of modification variants per rule in the KS processive model. Columns show counts under two constraint settings (with and without forbidden sequence bonds) and whether all modifications or only single modifications are considered.

Rule	**No forbid seq bonds**		**Forbid seq bonds**	
	All mods	1 mod	All mods	1 mod
K.S	81	9	9	5
x++	27	7	3	3
y++	27	7	3	3
K..S	81	9	9	5
x–	81	9	9	5
y–	81	9	9	5

Table 7.1 summarizes the number of possible modifications for each rule in the KS processive model. If all six rules are allowed to vary independently, the program space expands to

$$81 \times 81 \times 27 \times 27 \times 81 \times 81 \;\approx\; 2.6 \times 10^{11},$$

whereas restricting to a single modification per rule yields

$$9 \times 9 \times 7 \times 7 \times 9 \times 9 \;=\; 2.9 \times 10^{5}.$$

Introducing the "forbid sequence bonds" constraint (Section 5.2.3) shrinks this space considerably, down to

$$9 \times 9 \times 3 \times 3 \times 9 \times 9 \;=\; 5.9 \times 10^{4} \quad \text{or} \quad 5 \times 5 \times 3 \times 3 \times 5 \times 5 \;=\; 5.6 \times 10^{3},$$

depending on whether multiple or only single modifications are admitted.

An additional observation is that only three rules of the original processive model (`K.S`, `x++`, and `y++`) appear in any causal story reconstructed by KaSim and KaStor. Focusing exclusively on these rules collapses the space further:

$$9 \times 9 \times 3 \;=\; 243 \quad \text{or} \quad 5 \times 3 \times 3 \;=\; 45,$$

which makes exhaustive search entirely feasible.

In contrast, the KS *distributive* model contains eight rules with comparable modification capacities (Table 7.2). Without any constraints, the space of programs is enormous:

$$81^{6} \times 27^{2} \;\approx\; 2.1 \times 10^{14}.$$

Even if each rule is restricted to only one modification, the space remains large:

$$9^{6} \times 7^{2} \;=\; 2.6 \times 10^{7}.$$

Table 7.2: Number of modification variants per rule in the KS distributive model. Columns show counts under two constraint settings (with and without forbidden sequence bonds) and whether all modifications or only single modifications are considered.

Rule	**No forbid seq bonds**		**Forbid seq bonds**	
	All mods	1 mod	All mods	1 mod
K.Sx	81	9	27	7
K.Sy	81	9	27	7
x++	27	7	9	5
y++	27	7	9	5
K..Sx	81	9	27	7
K..Sy	81	9	27	7
x–	81	9	27	7
y–	81	9	27	7

The situation improves when forbidding sequence bonds are used. The counts reduce to

$$27^6 \times 9^2 \;\approx\; 3.1 \times 10^{10} \quad \text{or} \quad 7^6 \times 5^2 \;=\; 2.9 \times 10^6.$$

However, the story of running the original program $\mathcal{P}$ shows that only a subset of these rules (`K.Sx`, `K.Sy`, `x++`, and `y++`) actually contribute to any story observed in simulation. If we only consider modifications to the rules actually present in the story, the search space shrinks dramatically:

$$81 \times 81 \times 27 \times 27 \;\approx\; 4.8 \times 10^6 \quad \text{or} \quad 9 \times 9 \times 7 \times 7 \;=\; 3{,}969,$$

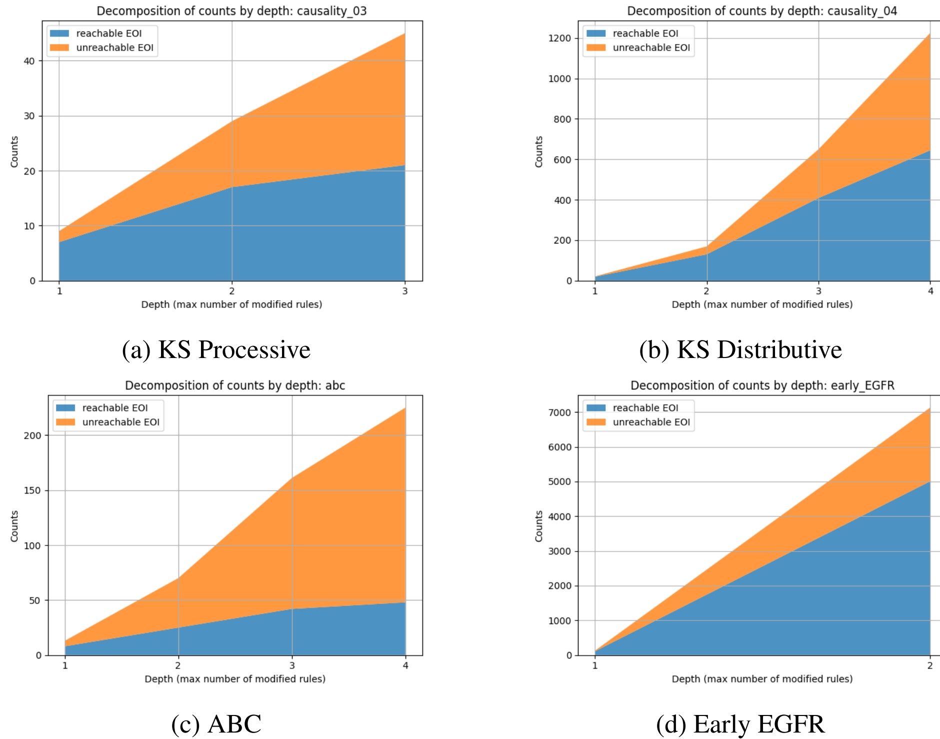
without forbidden bonds, and

$$27 \times 27 \times 9 \times 9 \;=\; 59{,}049 \quad \text{or} \quad 7 \times 7 \times 5 \times 5 \;=\; 1{,}225$$

with forbidden bonds. This last case is small enough to permit exhaustive exploration, despite the larger initial search space compared to the processive model.

Doing the same calculations for the ABC model, we get 2187 programs when considering one modification per rule. When only modifying rules that actually appear in the original story, it's 225 programs.

For the early EGFR model, we see how the space of programs explodes: if we consider one modification to each rule, the number of possible modified programs is 1.47^{19}. We can limit the maximum number of modified rules, however, or specify which rules may be altered and which rules not.



(a) KS Processive (b) KS Distributive

(c) ABC (d) Early EGFR

Figure 7.8: Increasing the maximum number of changed rules per program increases the number of possible modified programs. The number of programs where the EOI is still reachable is given in blue, while the number of programs where the EOI is not reachable is given in orange.

7.4 Kappa programs by specification

In chapter 4 we posed the following problem: Given a program $\mathcal{P}$, return all modified programs that pass the given specification. The first specification we will use is $S(\mathcal{P}) = 1$ when EOI is reachable. We can use this specification to guide the exploration of programs, as we do not want to consider modified programs where the EOI is no longer reachable.

In Figure 7.8, a graph is shown per model that indicates the number of possible modified programs, with the x-axis showing the "depth" in the maximum number of rules changed per program. With more rules changed, the number of programs increases. The ABC model does not maintain a lot of programs that preserve the EOI, while the KS Distributive model has a lot of them. Figure 7.9 shows the runtime of the algorithm. Using KaSa to check programs on the reachability of the EOI beforehand cuts the runtime of the algorithm by up to 45%.

Having a reachable EOI is a specification that will prune out the last of the programs we do not want to consider in our search, but it is not the only possibility. In the following subsections, we will show examples of possible specifications that

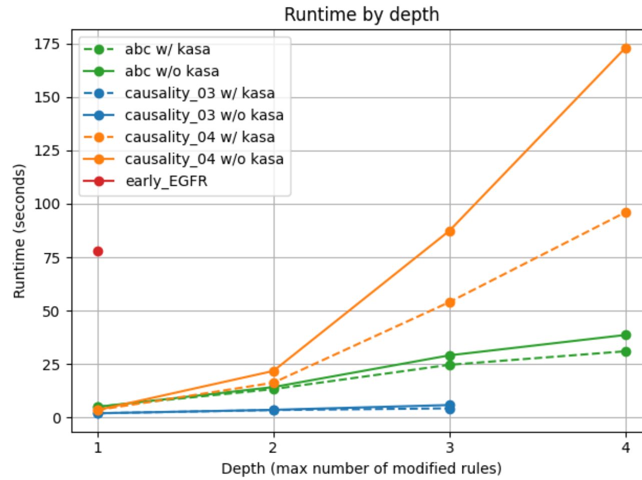


Figure 7.9: Runtime comparison of the four models. The x-axis denotes an increasing number of maximum rules that can be modified, while the y-axis is the number of seconds it took to find all programs.

a modeler could want, and show that our algorithm can return programs that follow that specification.

7.4.1 Modified programs with the same story (Robustness)

Let's take the KS distributive model. If we start exploring the modified programs, we can conclude after a few iterations that there are only three modified programs that result in the original story.

In the case where we change rule `K.Sy` to

`'K.Sy' K(x[.]), S(y{u}[.]) -> K(x[1]), S(y{u}[1]) @ 1`

or change rule `K.Sx` to

`'K.Sx' K(x[.]), S(x{u}[.]) -> K(x[1]), S(x{u}[1]) @ 1.`

or consider both modified rules, the story stays the same as the original story, from a total of 1225 possible modified programs. This indicates that the model is not robust and that even a slight change to the conditions or any rule will alter the explanation for the EOI. In the KS Processive model, something similar is observed. the rule `'K.S'`, when the same conditions are added of `x{u}` or `y{u}` in the S agent, are the only modified programs out of 45 that preserve the exact original story.

From experimentation, we can see that this specification allows for an optimization in search similar to using KaSa to prune the search space: Once a set of modified rules does not result in the story staying the same, no program that contains these rules will lead to the original story, and further exploration in this branch can be halted. This, however, is not an optimization we can make for any other specification.

Interestingly enough, for the ABC model, there are a total of 36 programs that preserve the original story, running for a total of 1 minute.

For the early EGFR model, the search space is still too big to find the programs in a reasonable time. Even with the optimization to prune out all explorations of programs that already deviate away from the original story, considering a maximum of 2 changed rules results in 938 programs that satisfy this specification, after running for 1360 seconds.

7.4.2 KaSa false positives

Another specification could be finding modified programs that KaSa would not flag as unreachable, but after running the simulation, no causal flow would be produced. In the case of the KS distributed program, among the ~ 1200 modified programs, there is only one case where this is the case. This shows how KaSa can only perform a static reachability check that is an overapproximation of actual reachability. The program in question has four changed rules, looking like

```
'K.Sx' K(x[.]), S(x[.], y{u}) ->
       K(x[1]), S(x[1], y{u}) @ 1
'K.Sy' K(x[.]), S(x{u}, y[.]) ->
       K(x[1]), S(x{u}, y[1]) @ 1
'x++'  K(x[1]), S(x{u}[1], y[.]) ->
       K(x[1]), S(x{p}[1], y[.]) @ 10
'y++'  K(x[1]), S(x[.], y{u}[1]) ->
       K(x[1]), S(x[.], y{p}[1]) @ 10
```

Inspecting the modified rules, we can see that indeed, either x++ or y++ cannot take place.

A modeler could find this particular specification interesting, as even typing out programs, KaSa can readily flag whether or not specific rules or observables are still reachable. Using our method, we can scan the space of potential modifications and not only return all modified programs that KaSa will flag, but also the modified programs that do not produce an EOI but are missed by KaSa.

7.4.3 New rules in the story

A modeler might want to be interested in what modifications to programs lead to a new event appearing in the DAG that was not present in the original story. In the case of the programs we defined at the start of this chapter, these will be the unbinding or unphosphorylation events described in the program, but not needed in the creation of the EOI in the original program.

An example in the KS distributive model is the program where `K.Sy` and `y++` are changed:

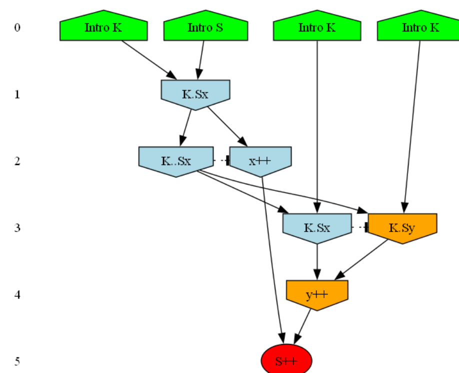


Figure 7.10: Caption

```
'K.Sy' K(x[.]), S(x[.], y[.]) ->
       K(x[1]), S(x[.], y[1]) @ 1
'y++' K(x[1]), S(x[_], y{u}[1]) ->
      K(x[1]), S(x[_], y{p}[1]) @ 10
```

This story, produced by this program, requires K and S to unbind at x before the EOI. This is one of the many examples where changing the conditions of just two rules in a program adds a “requirement” for the EOI in the form of the firing of a new rule.

7.5 Summary and Takeaways

The experiments show how the problem of inspecting programs that differ only by stricter conditions on the ruleset is made a tractable problem that we can solve. The constrained grammar, its constraints, and the rule modification process make it possible to generate valid programs to explore this space automatically.

Secondly, the specifications investigated are just an example of how this framework can incorporate a modeler’s needs and provide programs that adhere to those needs. We might not know what kinds of modified programs are biologically plausible, but this framework is built to handle a modeler’s needs.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This research offers two main contributions: first, it presents a novel method for enumerating Kappa rules that are both syntactically and semantically valid with respect to a program P. Secondly, it uses the process of rule modification to create rules that have extra conditions added to their sites. By modifying one or more rules of the original program $\mathcal{P}$, we developed an algorithm to explore the space of modified programs and return a subset of this space that adheres to a user-defined specification, like preserving the path to the event of interest or any other function over the stories.

We show that using this method, we go from a search space too big to search or filter over, to a manageable set of modified programs that can realistically be run through the simulation tool. By using the Kappa Static Analysis tool, we can speed up the process of finding valid programs, as it can check beforehand whether or not the event of interest is still reachable in a modified program.

Furthermore, using this framework, one can formulate a specification over the program and its corresponding stories to find a set of programs that lead to stories that fit the specification, reflecting the Program Synthesis by specification reasoning.

8.2 Future Work

8.2.1 Types of modifications to the context

This research has some limitations in the methods used. In our framework, a modification means adding stricter conditions to the context of an existing rule in the starting Kappa program $\mathcal{P}$. Concretely, we only allowed two types of modifications: (i) adding an internal site state (e.g., requiring a site to be `{u}` or `{p}`, and (ii) adding a bond condition, expressed by the identifiers `[.]` (free) or `[_]` (bound). These capture many common ways in which rules can be made stricter,

but they do not exhaust all possibilities. For instance, when adding the condition that a site must be bound, one could further specify which partner it must be bound to. This can be done in two ways:

- Specifying to what agent type the site is bound to. In a model like the KS Distributive program, a site can be bound to two different agents. In a pattern, this can be written as `A(y[x.B])`, which denotes that site `y` of agent `A` needs to be bound to a site `x` of some agent `B`. This distinction could result in different behavior.
- One could also add the whole agent to the rule to which the site is bound. In this case, that new agent could be further specified on the state of its sites. For example, one could add to a pattern `A(x{u})` an agent B that is connected to x like `A(y{u}[1]), B(x[1])`, which is functionally the same as `A(y[x.B])` hfrom the previous point. In this case, however, we could also add modifications to this new B agent to see the impact it has on the story.

Extending the modification algorithm to cover such cases would enhance the framework's expressive power.

8.2.2 Changes to the core rules

Another assumption we made is that the changes to the rules we consider are only to the rules' context. This assumption is made to ensure that the stories of all modified programs share the same events, but change the causal order of the events by adding extra context to the rules. Allowing changes to the core transformations of rules would significantly expand the search space and the kinds of biological questions that could be asked. Instead of only refining the conditions under which a rule can fire, such modifications could alter the actual state changes or binding events that occur, effectively introducing new mechanistic hypotheses into the model. This would open the door to exploring alternative biochemical pathways or even entirely new causal explanations for the event of interest. At the same time, it would require stronger constraints and more sophisticated synthesis strategies to keep the search tractable, since minor alterations in the core rules can quickly turn into a combinatorial explosion of possibilities. Future work in this direction could investigate how to balance the expressive power of core rule modifications with the practical need for efficiency and interpretability.

Bibliography

D. Abramov, J. Otto, M. Dubey, C. Artanegara, P. Boutillier, W. Fontana, and A. G. Forbes. RuleVis: Constructing Patterns and Rules for Rule-Based Models. In *2019 IEEE Visualization Conference (VIS)*, pages 191–195, Oct. 2019. doi: 10.1109/VISUAL.2019.8933596. URL `https://ieeexplore.ieee.org/document/8933596`.

P. Boutillier, F. Camporesi, J. Coquet, J. Feret, K. Q. Lý, N. Theret, and P. Vignet. KaSa: A Static Analyzer for Kappa. In M. Češka and D. Šafránek, editors, *Computational Methods in Systems Biology*, pages 285–291, Cham, 2018a. Springer International Publishing. ISBN 978-3-319-99429-1. doi: 10.1007/978-3-319-99429-1_17.

P. Boutillier, M. Maasha, X. Li, H. F. Medina-Abarca, J. Krivine, J. Feret, I. Cristescu, A. G. Forbes, and W. Fontana. The Kappa platform for rule-based modeling. *Bioinformatics*, 34(13):i583–i592, July 2018b. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty272. URL `https://doi.org/10.1093/bioinformatics/bty272`.

F. Camporesi, J. Feret, and K. Q. Lý. KaDE: A Tool to Compile Kappa Rules into (Reduced) ODE Models. In J. Feret and H. Koeppl, editors, *Computational Methods in Systems Biology*, pages 291–299, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67471-1. doi: 10.1007/978-3-319-67471-1_18.

V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-Based Modelling of Cellular Signalling. In L. Caires and V. T. Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, pages 17–41, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74407-8. doi: 10.1007/978-3-540-74407-8_3.

J. R. Faeder, M. L. Blinov, and W. S. Hlavacek. Rule-based modeling of biochemical systems with BioNetGen. *Methods in Molecular Biology (Clifton, N.J.)*, 500:113–167, 2009. ISSN 1064-3745. doi: 10.1007/978-1-59745-525-1_5.

W. Fontana, P. Boutillier, J. Feret, and J. Krivine. The Kappa Language and Kappa Tools.

D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, Dec. 1977. ISSN 0022-3654. doi: 10.1021/j100540a008. URL `https://doi.org/10.1021/j100540a008`. Publisher: American Chemical Society.

D. T. Gillespie. Stochastic Simulation of Chemical Kinetics. *Annual Review of Physical Chemistry*, 58(Volume 58, 2007):35–55, May 2007. ISSN 0066-426X, 1545-1593. doi: 10.1146/annurev.physchem.58.032806.104637. URL `https://www.annualreviews.org/content/journals/10.1146/annurev.physchem.58.032806.104637`. Publisher: Annual Reviews.

S. Gulwani, O. Polozov, and R. Singh. Program Synthesis Now. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, July 2017. ISSN 2325-1107, 2325-1131. doi: 10.1561/2500000010. URL `https://www.nowpublishers.com/article/Details/PGL-010`. Publisher: Now Publishers, Inc.

L. A. Harris, J. S. Hogg, J.-J. Tapia, J. A. P. Sekar, S. Gupta, I. Korsunsky, A. Arora, D. Barua, R. P. Sheehan, and J. R. Faeder. BioNetGen 2.2: advances in rule-based modeling. *Bioinformatics*, 32(21):3366–3368, Nov. 2016. ISSN 1367-4803. doi: 10.1093/bioinformatics/btw469. URL `https://doi.org/10.1093/bioinformatics/btw469`.

H. Kitano. Computational systems biology. *Nature*, 420(6912):206–210, Nov. 2002. ISSN 1476-4687. doi: 10.1038/nature01254. URL `https://www.nature.com/articles/nature01254`. Publisher: Nature Publishing Group.

R. Rangan. Matching Causality Hypotheses to Simulations of Biological Systems. July 2016. URL `https://dash.harvard.edu/handle/1/38811458`. Accepted: 2019-03-26T10:41:35Z.

A. Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

B. J. A. Swinkels. Constraint propagation in program synthesis. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2024. URL `https://resolver.tudelft.nl/uuid:35d0ed1e-fe16-43af-b32c-22770235a59f`. Faculty of Electrical Engineering, Mathematics and Computer Science.