Matching Causality Hypotheses to Simulations of Biological Systems

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

Citable link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:38811458

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA
Acknowledgements

I want to express my immense gratitude for my thesis advisor Professor Stephen Chong, who has helped me through every stage of this process, from suggesting that I look into Kappa last summer to reading a thesis draft recently. I could not have done this without his advice and mentorship.

I also want to thank Professor Walter Fontana and Jean Yang for introducing me to Kappa and teaching me about the interesting problems the language could help tackle. I am incredibly grateful for the mentorship they have provided.

I want to thank Jerome and Pierre for offering their expertise with the KaSim implementation, Jonathan and Ioana for sharing ideas on the story formalism with me, and Hector for discussing the biological insights we want from stories. I want to thank John Bachman for showing me the apoptosis Kappa system and helping me with its setup.

Also, I would like to express my thanks for my readers Professor Radhika Nagpal, Professor Chong, and Professor Fontana for generously taking the time to look through this thesis.
Abstract

Computational models for studying chemical reaction networks are indispensable tools for analyzing the intertwined reactions involved in complex biological systems. Indeed, executable models such as the programming language Kappa provide useful insights by allowing users to specify a system of chemical rules, and then simulating plausible sequences of reactions that might execute in our cells given this rule system. Kappa thus provides the opportunity to analyze the sequence of steps that lead to a chemical event of interest. Formal models for the causal structure of these sequences have been developed based on Kappa, describing the chain of events that directly or indirectly contribute to the production of an event of interest. These models, known as stories, attempt to capture the notion of a biological pathway. In this thesis, we develop a novel framework for matching stories to Kappa simulations, ensuring that we report a match if and only if the story is a reasonable model for the underlying causality of the simulation. We then present algorithms to efficiently compute these matches, and apply these algorithms in two settings: we study the prevalence of competing pathways in cellular death, and we analyze the dynamics of a common biological pattern, three-membered ring formation. Our insights on these systems align with those presented in literature, demonstrating that these matching algorithms have applicability for analyzing biological systems.
# Table of contents

List of figures xi

1 Introduction 1

2 Background and Related Work 5

2.1 Approaches to Modeling Protein Networks 5
2.1.1 Mathematical Models 6
2.1.2 Executable Models 6
2.1.3 Executable Models with Molecular State 8
2.2 Kappa Background and Prior Work 8
2.3 Prior Work on Pathways in Kappa Simulations 11
2.4 Overview of Contribution 13

3 Kappa Formalism 15

3.1 Kappa Programs 16
3.2 Kappa Program Formalism 17
3.2.1 Agents 17
3.2.2 Rules 18
3.2.3 State 19
3.2.4 Partial Valuations 20
3.2.5 Rule Modifications 21
<table>
<thead>
<tr>
<th>3.2.6</th>
<th>Kappa Rates .................................................. 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Kappa Simulations .............................................. 22</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Kappa Simulation State ........................................ 23</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Events ............................................................. 24</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Trace ............................................................... 26</td>
</tr>
<tr>
<td>3.4</td>
<td>Causal Analysis of Kappa Simulations .......................... 28</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Concurrent Events ............................................... 29</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Precedence Relation over Traces ................................ 30</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Trace Equivalence ............................................... 31</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Stories ............................................................. 32</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Motivating Further Compression for Stories .................... 34</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Direct and Weak Compression for Stories ....................... 38</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Working with Stories ............................................. 41</td>
</tr>
<tr>
<td>4</td>
<td>Story-Trace Matching ............................................. 43</td>
</tr>
<tr>
<td>4.1</td>
<td>Motivation for Story-Trace Matching ............................ 44</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Introducing the Question of Story Prevalence in a Kappa System 44</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Why Use Story-Trace Matching for Finding Story Prevalence .... 47</td>
</tr>
<tr>
<td>4.2</td>
<td>Story-Trace Matching with Direct Compression ................... 49</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Goal for Direct Matching Algorithm .............................. 50</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Direct Matching Algorithm ....................................... 51</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Correctness Condition for Direct Compression Matching Algorithm 52</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Correctness for Direct Compression Matching Algorithm .......... 55</td>
</tr>
<tr>
<td>4.3</td>
<td>Story-Trace Matching with Weak Compression ..................... 57</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Goal for Weak Matching Algorithm ............................... 57</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Additional Complexity of Weak Matching Algorithm: Parallel Exploration 59</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.3.3 Weak Matching Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>4.4 Describing Undone Story-Trace Embeddings</td>
<td>65</td>
</tr>
<tr>
<td>4.4.1 Undesirable Story-Trace Matches: Undone Story-Trace Embeddings</td>
<td>65</td>
</tr>
<tr>
<td>4.4.2 Defining Undone Story-Trace Embeddings</td>
<td>67</td>
</tr>
<tr>
<td>4.4.3 Properties of Undone Story Embeddings</td>
<td>68</td>
</tr>
<tr>
<td>4.5 Refined Story-Trace Matching with Weak Compression</td>
<td>69</td>
</tr>
<tr>
<td>4.5.1 Goal for Refined Matching Algorithm</td>
<td>69</td>
</tr>
<tr>
<td>4.5.2 Impact of Story Undoneness on Story-Trace Matching Algorithm</td>
<td>70</td>
</tr>
<tr>
<td>4.5.3 Refined Matching Algorithm</td>
<td>73</td>
</tr>
<tr>
<td>4.6 KaSim Implementation</td>
<td>76</td>
</tr>
<tr>
<td>5 Applications</td>
<td>79</td>
</tr>
<tr>
<td>5.1 Determining Prevalence of Apoptosis Pathways</td>
<td>79</td>
</tr>
<tr>
<td>5.1.1 Apoptosis Pathway Modeling</td>
<td>80</td>
</tr>
<tr>
<td>5.1.2 Type I and Type II Apoptosis Pathways</td>
<td>81</td>
</tr>
<tr>
<td>5.1.3 Apoptosis Pathway Analysis Approach</td>
<td>83</td>
</tr>
<tr>
<td>5.1.4 Apoptosis Pathway Analysis Results</td>
<td>88</td>
</tr>
<tr>
<td>5.2 Characterizing System Dynamics in Ring Formation</td>
<td>90</td>
</tr>
<tr>
<td>5.2.1 Investigating Dynamics with Story-Trace Matching</td>
<td>90</td>
</tr>
<tr>
<td>5.2.2 Test System: Ring Formation</td>
<td>91</td>
</tr>
<tr>
<td>5.2.3 Ring Formation Dynamics Analysis Approach</td>
<td>92</td>
</tr>
<tr>
<td>5.2.4 Ring Formation Dynamics Analysis Results</td>
<td>95</td>
</tr>
<tr>
<td>5.3 Discussion of Story-Trace Matching Applicability</td>
<td>97</td>
</tr>
<tr>
<td>6 Future Work</td>
<td>101</td>
</tr>
<tr>
<td>6.1 Future Applications of Story-Trace Matching</td>
<td>101</td>
</tr>
<tr>
<td>6.1.1 Assessing Stories Generated without Simulation</td>
<td>101</td>
</tr>
</tbody>
</table>
6.1.2 Efficient Story Generation .............................................. 102
6.1.3 Parameter Sensitivity .................................................. 103

6.2 Further Improving Story-Trace Matching ............................ 103
  6.2.1 Exploring Multiple Story to Trace Matching ....................... 103
  6.2.2 Story Pattern to Trace Matching .................................... 103
  6.2.3 Additional Forms of Compression .................................... 104

7 Conclusion ........................................................................... 105

References ............................................................................... 107

Appendix A Story-Trace Embedding Undoneness Properties ............ 109
  A.1 At Least One Reasonable Story-Trace Match ......................... 110
  A.2 At Most One Reasonable Story-Trace Match .......................... 113
  A.3 Settings with More than One Story-Trace Match ....................... 119

Appendix B Apoptosis System .................................................. 123
  B.1 Kappa System for Apoptosis .............................................. 123
List of figures

2.1 Example of Rules in a Kappa System (Image reproduced from [12].) . . . . 9
2.2 A Graphical Representation of Rule Applications in a Biological System
(Image reproduced from [12].) . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 An Example of a Kappa Simulation and its Corresponding Pathway . . . 12
3.1 Simple Kappa Program and Corresponding Formalism . . . . . . . . . . . 16
3.2 Example Rule with Multiple Instantiation Possibilities . . . . . . . . . . . 27
3.3 Example Story and Weak-Compression for System Modeling Substrate
Double-Phosphorylation by Enzyme . . . . . . . . . . . . . . . . . . . . . 37
4.1 Multiple Stories in Trimer Formation Kappa System . . . . . . . . . . . . 46
4.2 Multiple Stories for a Single Event of Interest Appearance in a Trace . . . 48
4.3 Multiple Agent Identifier Isomorphisms in Weak Compression Story-Trace
Matching: Within Rule . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.4 Multiple Agent Identifier Isomorphisms in Weak Compression Story-Trace
Matching: Greedy Assignment Failure . . . . . . . . . . . . . . . . . . . . 61
5.1 Apoptosis Pathway Diagram . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.2 Stories Representing Type I and Type II Apoptosis . . . . . . . . . . . . 85
5.3 cPARP Production via Type I and Type II Apoptosis . . . . . . . . . . . . 89
5.4 Ring Formation Kappa System . . . . . . . . . . . . . . . . . . . . . . . . 93
5.5 Ring Formation Dynamics with Various Intermediate Binding Affinities . . 96

A.1 Two Stories Match Trace Without Undone Embedding . . . . . . . 120
Chapter 1

Introduction

Complex interactions between molecular entities like proteins are responsible for the diverse phenotypes that emerge in different cell types, whether in normal or disease states. Our understanding of interactions between these molecules has flourished over time, with literature now rich with information describing the function of specific proteins in various cellular systems. For instance, the Gene Ontology contains hundreds of thousands of annotations detailing the function of proteins and indicating the cellular systems in which they function [5].

However, as we build on our knowledge of biological systems, it is essential to understand more than these proteins’ immediate actions - we must learn how one molecular interaction influences others, leading to cascades of actions in larger networks. Large systems controlling complex phenomena like cellular death often involve hundreds or thousands of interactions, creating network dynamics that are impossible to understand without additional tools. We must build analytical tools to understand these network interactions, finding how phenotypes like cellular death respond to sometimes subtle perturbations that reflect different cell types, disease states, or drug candidate trials.
The Kappa language has been developed to provide such a tool, aiming to elucidate how large sets of chemical reactions may work together in our cells [12]. The programming language provides a method to express chemical interactions in a single executable system, and to then simulate the sequence of reactions that may occur when these interactions can act together. Thus, Kappa simulations provide a platform for asking questions about a network of interactions. Without further analysis, however, running these simulations can only answer the most basic biological questions, such as how often an event occurs in these simulations.

If we build further analytical tools on top of Kappa simulations, the set of biological questions that we can address grows much richer. We can begin to ask questions about causal explanations, which specify the chain of events that either directly or indirectly cause a final product we care about to appear in the simulation. For instance, we might learn whether some interaction is necessary for the generation of a dangerous end product. If this product contributes to a disease state, finding necessary interactions for its generation can accelerate the process of finding therapeutic targets. Alternatively, we may be able to determine whether one causal explanation for a final event occurs more frequently than another, and moreover, we might view differences in these frequencies across small perturbations in the system’s conditions. Understanding the prevalence of causal mechanism in this way can help us better understand and target disease. In fact, the question of how a causal chain of events changes with perturbation is especially relevant to cancer, in which the relevant protein networks are especially dense and can malfunction due to a small set of genetic changes [19].

Through this thesis, we develop further analytical tools on top of Kappa simulations, allowing us to answer questions about the causal explanations present in a system. In particular, we design and apply algorithms for a primitive operation that lies at the core of various biological questions: the task of efficiently and accurately discovering when a causal explanation
matches a Kappa simulation.

We begin by providing some context on the protein network modeling techniques currently available, demonstrating that Kappa is particularly useful for understanding causal explanations in biological systems (Section 2). Next, we will move to provide a formal description of Kappa programs, which will be useful for developing our analytical tools (Section 3). We then develop a framework and tool for understanding when a causal mechanism matches a Kappa simulation, examining various criteria and algorithms for answering this question (Section 4). Ultimately, we develop a new notion for matching a causal mechanism to a simulation, eliminating false matches and allowing us to find matches efficiently. This tool gives us the opportunity to study a broader set of biological questions, which we begin to explore in Section 5. In particular, we compare the prevalence of two competing causal mechanisms in the system of cellular death, and we better understand the dynamics of a common biological pattern, ring formation. We find that in these cases, the primitive operation we explore becomes a useful tool for addressing biological questions using Kappa. We discuss alternative future applications for this method in Section 6.
Chapter 2

Background and Related Work

In the context of currently available protein network modeling techniques, Kappa presents a unique opportunity to understand the causal mechanisms involved in molecular reactions. Indeed, as a simulation language that generates sequences of steps at the molecular level, Kappa allows us to ask questions of the causal chain of reactions that leads to the production of a final event of interest. Furthermore, with formalisms based on Kappa developed to express the concepts of biological pathways, we can be equipped to carry out further analyses on the causal mechanisms in biological systems. We will discuss the context of protein network modeling techniques and provide a general overview for Kappa in this section, which we will follow with a formal description of Kappa in Section 3.

2.1 Approaches to Modeling Protein Networks

To understand how proteins interact in networks, it is useful to create computational tools which take as input the chemical reactions that participate in a system, and can then generate simulations that exhibit system behavior over time. Indeed, protein networks have been modeled through either mathematical or executable models, both of which aim to help answer questions about how a biological system behaves when specified chemical reactions occur at
Background and Related Work

particular rates [13].

Both mathematical and executable models provide a means to systematically codify this information about known protein interactions into a single model, and to view the impact of these interactions on some observable in the system. Indeed, to obtain the reactions and associated rates required to initialize such models, researchers can search literature detailing experimental data on the relevant pathway, combining data and knowledge from different studies into a common modeling framework.

2.1.1 Mathematical Models

In mathematical models, systems of chemical reactions are described as sets of differential equations, with each differential equation for instance representing the change in the concentration of a particular element as the sum of the rates of the reactions it participates in [13, 25]. These systems of equations act as a denotational representation for a system, and with empirical solvers or computational simulations, can be solved to varying levels of accuracy to obtain concentrations for agents over time. Simulators like COPASI allow users to specify biologically intuitive and researched chemical reactions representing protein interactions, automatically inferring the corresponding differential equations and generating information about concentration changes over time [18].

2.1.2 Executable Models

In contrast, executable models seek to represent the behavior of a chemical system by modeling the step by step changes of cellular state caused by the reactions in that system [13]. For instance, a system in a state in which a protein is activated can move to a state in which that protein is deactivated by a dephosphorylation event, one that removes the activating phosphorylation mark from the protein. In this way, these executable models act as a small
step operational semantics, carrying the system from state to state via particular transitions. When multiple transitions are available, the choice can be made probabilistically based on rates associated with these transitions.

We choose to focus our work on these executable models, because crucially, executable models allow the user to ask questions about the reactions that play out in the step by step simulation of a system. Furthermore, they allow researchers to study the times at which these reactions occur. Whereas mathematical models provide concentrations of agents over time, masking the rules that lead to these changes, the executable models’ primary output is a sequence of events which resembles the sequences of reactions that might carry out in our cells. The ability to ask questions about the transitions used in a simulation is critical for understanding interesting biological properties, for instance to determine the causal pathway that leads to a particular state of the system. Executable models have been used to answer such questions about causal pathways to yield useful insight, though they do not capture more complex properties that impact true reactions in our cells, such as the spatial localization of different proteins or their conformational changes [8].

Various forms of executable models have been in use, such as boolean networks and PetriNets [15, 17, 27, 1]. These models provide various levels of granularity for modeling cellular state, with some capturing transitions between a small set of broad, general states [14], and others capturing transitions between highly specific states defined on the molecular level [12]. BioNetGen [11] and Kappa [12] are two instances of executable models that take the more specific molecular approach to modeling cellular state, with states for instance capturing the modification of particular domains on different proteins in the system.
2.1.3 Executable Models with Molecular State

In this thesis, we study executable models which define state on the molecular level, because without such a fine level of granularity, it is difficult to ask questions about molecular mechanisms in complex protein networks. In Kappa and BioNetGen, this molecular state can be represented graphically, such that these modeling frameworks act as graphical modeling languages [12]. Indeed, these languages use reactions to transition between cellular states, where each state can be represented as a network of connected proteins or molecules with internal state (such as phosphorylation) at particular domains.

In this project, we study Kappa for its efficiency of simulating large systems, and for the availability of simulation analysis tools which build understanding based on Kappa simulations [6, 8, 9]. These analysis tools will serve as a building block for further tools that we develop in this thesis.

2.2 Kappa Background and Prior Work

Here we present a high level overview of Kappa simulations, which we will cover formally in Chapter 3.

In Kappa simulations, signaling systems are represented as a set of chemical rules, with each rule stipulating required local conditions for an interaction to occur, along with the impact of this rule on the local environment (Figure 2.1) [12]. For instance, as depicted in the top of three rules in Figure 2.1, a rule for the binding of two proteins $K$ and $S$ may require that the two proteins are initially unbound, and may cause the two proteins to be bound to each other at site $d$ on each protein. These rules avoid an explicit representation of the complete reaction network involving all possible molecular species; rather, the preconditions for a rule
application are only the portion of the state that the rule depends on.

At each step of a Kappa simulation, the model moves between states representing the current configuration of proteins and other molecules in the system, with each transition representing the application of one of the available chemical rules (Figure 2.2). A Kappa state can also be imagined as a graph of connected proteins or other agents, with every transition representing a rewrite of this state graph; for this reason, Kappa can be termed a graphical modeling language. KaSim is the currently available simulator that runs Kappa programs and analysis [KaSim].

![Fig. 2.1 Example of Rules in a Kappa System (Image reproduced from [12].)](image)

Capital letters are agents, lowercase letters are sites, and superscripts mark bonds between sites across agents. Bold numbers represent the rate of reaction. Each rule has a name (b - bind, u - unbind, px - phosphorylate site x, py - phosphorylate site y) and rules can be bidirectional (e.g. with b, u). Image reproduced from [12].

A single simulation represents one possible execution of a system for a period of time, with each time step and reaction choice determined probabilistically based on reaction rates for each rule. These choices are made based on the Gillespie algorithm [16], such that the simulations’ stochastic choices for transition rules at each step reflects the probabilistic progression of events in our cells. The resulting sequence of transitions can be analyzed to
Fig. 2.2 A Graphical Representation of Rule Applications in a Biological System (Image reproduced from [12].)

During a simulation, Kappa 1) finds all rules which apply to the system; 2) picks a rule to apply based on a distribution formed by rates; 3) applies the rule to particular agents (an instance); 4) repeat. Here, we see a chemical reaction rule (top A) applied to a particular instance (top B), and a protein interaction rule (bottom A) applied to a particular instance (bottom B). Image reproduced from [12].

Gain further understanding of the biological system. For instance, by analyzing a large set of simulations, one can learn about typical properties of the system.

When answering these questions, Kappa presents various benefits over other executable models for protein networks. First, Kappa simulations gain computational power by avoiding the need for enumerating all possible achievable components of the system [6]. Whereas some models require the enumeration of all such components before simulation can begin, which in some systems may even necessitate the enumeration of an infinite set, Kappa avoids this combinatorial explosion and thus enables building larger models.
Furthermore, Kappa is unique in the tools built to understand emergent properties of Kappa simulations. Kappa developers have created a framework for formally and precisely understanding the causal pathway that leads to a particular end event in a simulation [7]. Such a framework can enable researchers to ask powerful questions about the causal chain of events that are necessary to produce a biological event or phenotype of interest.

### 2.3 Prior Work on Pathways in Kappa Simulations

Causal analysis provides a formal description for the pathways that emerge from complex Kappa simulations. The framework of *stories* has been developed to specify the steps causally relevant to the production of an event of interest. For instance, in a simulation where the researcher is interested in the formation of a particular phosphorylated kinase, it is possible that many rules in the simulation system were unnecessary for the production of this kinase; all transitions corresponding to those rules would not be included in a story for this simulation. Focusing on the causal steps in a simulation can aid the process of biological inquiry based on Kappa simulations, as they distill irrelevant events from the sequence of transitions, and leave the user with a model resembling a pathway diagram intuitive to biologists (Figure 2.3).

Methods have been developed to generate stories from Kappa simulations, by equipping the sequence of steps in a simulation with causal ties, compressing these stories to remove intermediate steps that do not contribute to the event of interest, and abstracting the remaining steps to capture the types of reactions used rather than specific instantiations. Story-generation tools have been implemented in KaSim [KaSim]. Through these steps, a causal model of the reaction system is obtained which is specific enough to capture the particular pathways which lead to an event of interest, but abstract enough to represent many
Background and Related Work

Fig. 2.3 An Example of a Kappa Simulation and its Corresponding Pathway

Here, reactions of the form $X/$ introduce the protein $X$ into the system. Reactions of the form $X.Y$ bind the entity $X$ with the entity $Y$ to form the entity $XY$. Reactions of the form $X^R$ reverse the effects of $X$. In this system of rules, we are seeking to understand how the compound $ABC$ is created. **A:** We see a raw simulation, which contains an amalgam of events, some of which reverse each other or produce elements irrelevant to the final goal. **B:** We show the corresponding story for the formation of $ABC$, which eliminates irrelevant reactions and undoings, and causally links different reactions.

![Diagram A](image1.png)

![Diagram B](image2.png)

trivially similar event sequences.

Now, with the story model being used to understand real-scale systems, generating analysis tools using stories will be especially useful to gain biological understanding from Kappa models. Indeed, Kappa models have been generated for the cancer-relevant EGFR system with 70 rules and $10^{23}$ distinct achievable species, and using story generation for this system has led to insights into the role of particular proteins in this system [8]. For systems of this size, generating stories will provide greater insight if the causal explanations across simulations can be compared and analyzed programmatically, rather than generated and viewed individually.

Programmatic tools to analyze simulation systems using stories will be able to address
2.4 Overview of Contribution

various biologically interesting questions. For instance, we may hope to investigate the story prevalence question: whether for a particular cell type, we most often see a protein product formed via one pathway as opposed to another one. We may wonder whether this property holds across cell types, or whether for some initial conditions (representing cell type differences), the pathways produced vary. While tools have been developed to statically generate all possible stories from a set of chemical reactions defining a system [21], generating these stories will not yield information on the prevalence of each of these stories. Additional techniques will be necessary to achieve this goal.

Furthermore, we may want to ask which links in a causal pathway take the longest to achieve, and how this rate varies as a simulation runs its course. To answer this question and others merging system dynamics with stories, additional story analysis tools will be necessary.

Answering such questions can help us determine the links in pathways where drug interventions may be the most effective. Indeed, by developing methods to find dominant pathways in simulations of a cell type, we can pinpoint interactions present in the most dominant pathways towards disease products, thus finding interactions to interrupt with drugs to prevent the formation of the majority of these disease products. Alternatively, by identifying slow points in a story’s progression, we may learn how to activate a pathway needing upregulation to produce healthy phenotypes.

2.4 Overview of Contribution

In this thesis, we will develop a framework to define when a Kappa story matches a simulation, and we will develop algorithms to find these matches. Furthermore, we will use this tool to precisely and efficiently explore the story prevalence question and system dynamics for Kappa systems, illustrating the applicability of this approach.
Chapter 3

Kappa Formalism

Here, we present the Kappa model and story model as used in this work. We first discuss the structure of Kappa programs and their formal description ( Sections 3.1 - 3.2). We then describe the Kappa simulations that result from running Kappa programs in Section 3.3. Finally, we discuss the formalisms previously developed to understand the causal mechanism behind Kappa simulations in Section 3.4.

We restrict our focus to a particular type of simulation system called rectangular systems. In rectangular systems, an application of a chemical rule depends on checking the current state of individual proteins and other compounds in a system, as opposed to satisfying more complex formulas about the system as a whole. This reduces the complexity of our definitions, and restricts us to the local computations most common to biological systems.

Many of these definitions are described in [21]; however, we present distinct formulations for these definitions in Section 3.2 - 3.4 to formulate concepts in the context of rectangular systems.
Fig. 3.1 Simple Kappa Program and Corresponding Formalism

A: We see a simple Kappa program, following the outline described in Section 3.1. Here, two agents A and B can either be introduced or bind to one another. Lines 1-2 define the agents and sites in the system. Lines 4-5 specify the rules for the introduction of A and B, and Line 6 describes the binding of the two (as indicated by the shared label 1 linking the domain b for agent A to the domain a for agent B). These specifications include a rule name, a prior local pattern, a final local pattern, and a rate at which the rule should be applied. B: We show the corresponding Kappa program formalism for the program in A., using the definitions from Section 3.2.1 to 3.2.4.

### A. A simple Kappa program

1. `agent: A(b,c)`
2. `agent: B(a,c)`
3. `var: 'Kon' 1.0e-4`
4. `'A/' -> A(b,c) @ 1.0`
5. `'B/' -> B(a,c) @ 1.0`
6. `'A.B' A(b,c), B(a,c) -> A(b!1,c), B(a!1,c) @ 'Kon'`

### B. Kappa program formalism

Agent types: $T = \{A, B\}$

Agent sites: $\sigma_A = \{b, c\}, \sigma_B = \{a, c\}$

Rule `A/`:

* ag(A) = $\{(A, 1)\}$
* pre(A) = $\{(A, 1)_{\emptyset} = null\}$
* eff(A) = $\{(A, 1)_{\emptyset} = true\}$

Rule `B/`:

* ag(B) = $\{(B, 1)\}$
* pre(B) = $\{(B, 1)_{\emptyset} = null\}$
* eff(B) = $\{(B, 1)_{\emptyset} = true\}$

Rule `A.B`:

* ag(A.B) = $\{(A, 1), (B, 1)\}$
* pre(A.B) = $\{(A, 1)_{b} = free, (A, 1)_{c} = free,$
  $\langle (B, 1), a_{i} = free, (B, 1), c_{i} = free\}$
* eff(A.B) = $\{(A, 1)_{b} = bound((B, 1), a),$\n  $\langle (B, 1), a_{i} = bound((A, 1), b)\}$

### 3.1 Kappa Programs

We first briefly introduce the structure of Kappa programs, which we will discuss formally in Section 3.2. In a Kappa program, users must first specify agents, which often correspond to proteins that act in a cellular system. Agents contain sites, corresponding to different binding or modification domains on a protein. The user must then specify a set of rules dictating ways in which these agents interact. For instance, rules may change the internal state of an agent’s domain, or may cause two domains to bind together (specified by a shared binding
3.2 Kappa Program Formalism

In this section, we formally describe the building blocks of Kappa programs: agents and rules. Together, Kappa users employ these elements to represent the system they are trying to model.

3.2.1 Agents

Definition 1. An agent in a Kappa program is a molecule or protein that reacts in a chemical rule. Agents have types (for instance, EGFR); \( T \) is the set of all agent types used in a Kappa program, which is \( \{A, B\} \) in Figure 3.1. Each agent type defines a set of sites which can be modified in the application of rules, for instance a phosphorylation site. In particular, each agent type \( t \in T \) can be mapped to a set of sites \( \sigma_t \), where \( \sigma_t \subset S \) and \( S \) is the set of sites evinced by any agent in the program. In Figure 3.1, we use \( S = \{a, b, c\} \).

Agents resemble molecular entities in biological systems. For instance, they may act like proteins, which often will have multiple sites (or domains) at which chemical reactions can occur, with these sites specific to the type of protein. Together, agents and their sites serve
as a representation of the main entities of Kappa programs. In Section 3.2.2 - 3.2.5, we use these ideas to identify our simulation system’s state, and to define how a Kappa program influences that state.

### 3.2.2 Rules

**Definition 2.** A **rule** $r$ is Kappa’s representation of a chemical interaction, and is defined by the specification of three sets, $\text{ag}(r)$, $\text{pre}(r)$, and $\text{eff}(r)$, which will be defined below.

Through rules, Kappa’s agents interact to create new entities at each step of a Kappa simulation, moving the system from state to state. A rule is defined by the conditions it checks and the effect it has on the system, acting on labeled agents. We formalize this by identifying a rule $r$ with three sets: $\text{ag}(r)$, $\text{pre}(r)$, and $\text{eff}(r)$.

To define the entities on which Kappa rules act, for each rule $r$ we specify the set $\text{ag}(r)$ defined below.

**Definition 3.** Let $L$ be the **set of labels**, and $T$ be the set of agent types defined above. For rule $r$, we define $\text{ag}(r) \subset T \times L$ to be the set of labeled agents that are checked or modified by a rule.

Kappa rules’ agents must be labeled with some $l \in L$ because it is often necessary to distinguish the appearance of multiple agents of the same type in a Kappa rule. For instance, if a Kappa rule involves binding two distinct EGFR agents, these agents need to be given different labels in the rule. In Figure 3.1, we are using $L = \mathbb{N}$. When using Kappa rules in simulations, these labeled agents will be instantiated to allow the rule to work on particular agents in the simulation; this instantiation will be described in Section 3.3. From the example in Figure 3.1, for rule $A.B$ we would have $\text{ag}(A.B) = \{(A, 1), (B, 1)\}$. 
We use the sets pre\((r)\) and eff\((r)\) to specify the modifications encoded by a Kappa rule. To specify these sets, we develop a representation of state in the Kappa setting.

### 3.2.3 State

Here we build formalisms to discuss the state which Kappa rules act on.

**Definition 4.** We define a port as a tuple of a labeled agent and one of its sites, \(\langle (t, l), s \rangle\) where \((t, l) \in J\) and \(s \in \sigma_t\). The set of all ports used in a rule is denoted \(P(r)\).

**Definition 5.** We define the set of port types as \(PT = \prod_{t \in T} \{t\} \times \sigma_t\). The function \(proj\) will take a port to its type, such that \(proj(\langle (t, l), s \rangle) = (t, s)\) for all \(\langle (t, l), s \rangle \in P(r)\). A port’s type is thus the agent type with one of its sites, removing the particular agent label.

Below we will define a set of local state variables \(X_{ag(r)}\) which describe the state on which a rule in a Kappa program acts. Later in Section 3.3.1, we will discuss the global state used not in individual Kappa rules but rather during simulations of the full system of Kappa rules.

Local state variables are based on the set of agents used in a rule. For instance, a local state variable and the value it takes might represent whether the site \(a\) on agent \(B\) is bound to a site on another agent, or whether the site \(x\) of agent \(S\) is phosphorylated. These local state variables will be useful to define how rules act on labeled agents to achieve changes in Kappa systems.

First, we will define that the set of all possible internal state values.

**Definition 6.** Across all sites in the system, the set of possible internal state values is given by \(V_{int}\). For instance, \(V_{int}\) might include the internal state value \(p\) for phosphorylated.
**Definition 7.** Below we specify the local state variables $x \in X_{ag(r)}$ along with the set of values $V$, they can take. We have variables of the form $j_\emptyset$, $\langle j, s \rangle_\lambda$, and $\langle j, s \rangle_\iota$ where $j \in ag(r)$ and $\langle j, s \rangle \in P(r)$.

- **$j_\emptyset$:** For each $j \in ag(r)$, $j_\emptyset$ is true if $j$ is in the system, false if $j$ was previously in the mixture but currently not, and null if $j$ has never been in the system. Therefore, $V_{j_\emptyset} = \{true, false, null\}$.

- **$\langle j, s \rangle_\lambda$:** For each port $\langle j, s \rangle$, $\langle j, s \rangle_\lambda$ indicates the internal state of the port. Each $\langle j, s \rangle_\lambda$ can take on values in the set $V_{\langle j, s \rangle_\lambda}$. We have that $V_{\langle j, s \rangle_\lambda} \subset V_{int}$ is the subset of potential internal state values which this site can achieve. For instance, if $j = (t, l)$, we may have $V_{\langle (t, l), s \rangle_\lambda} = \{p, u, null\} \subset V_{int}$, representing that the site $s$ on labeled agent $j$ can take on phosphorylated and unphosphorylated states, or the value null if $j$ is not in the system. Furthermore, we have that for $proj(j) = proj(j')$, $V_{\langle j, s \rangle_\lambda} = V_{\langle j', s \rangle_\lambda}$. In words, the set of possible internal states is constant for all ports with the same port type, such that a particular agent type and site pair always takes on a value from one set of potential internal states.

- **$\langle j, s \rangle_\iota$:** For each port $\langle j, s \rangle$, $\langle j, s \rangle_\iota$ indicates the binding state of the port, which is free if the port is unbound, null if agent template $j$ is not in the system, bound if the port is bound to an unspecified binding partner, and bound($\langle j', s' \rangle$) if the port is bound to another port $\langle j', s' \rangle$. Thus, $V_{\langle j, s \rangle_\iota} = \{free, null, bound\} \cup \{bound(\langle j', s' \rangle) : \langle j', s' \rangle \in P(r), \langle j', s' \rangle \neq \langle j, s \rangle\}$.

### 3.2.4 Partial Valuations

We build a notion of partial valuations which will be useful for developing our description and analysis of Kappa simulations.
Definition 8. A **partial valuation** \(\rho\) is a set of elements of the form \((x, v)\) where \(x \in X_{ag(r)}\) and \(v \in V_x\). The empty partial valuation is given by \(\emptyset\), the empty set.

Definition 9. A **partial valuation is valid** if there are no two elements \((x, v_1)\) and \((x, v_2)\) in the partial valuation such that \(v_1 \neq v_2\).

Definition 10. \(\bar{\rho} \subset X\) is the **set of constrained variables** in the partial valuation; in other words, for every element \(x \in \bar{\rho}\) there must be some \(v \in V_x\) such that \((x, v) \in \rho\).

Definition 11. We will say that two partial valuations \(\rho_1\) and \(\rho_2\) are **compatible** (written \(\rho_1 \uparrow \rho_2\)) if their union is valid, and **incompatible** otherwise (written \(\rho_1 \downarrow \rho_2\)). We will also introduce notation for **removing constraints** involving particular variables from a partial valuation: \(\rho_1 \setminus \rho_2 = \{(x, v) : (x, v) \in \rho_1, x \not\in \rho_2\}\).

### 3.2.5 Rule Modifications

We can now finish defining Kappa rules, describing the checks and modifications these rules can carry out on their set of labeled agents. To specify these checks and modifications, we have the following definition.

Definition 12. We define \(\text{pre}(r)\) and \(\text{eff}(r)\) to be two valid partial valuations of variables in \(X_{ag(r)}\). \(\text{pre}(r)\) represents the preconditions for executing \(r\), and \(\text{eff}(r)\) are the modifications caused by \(r\).

For example, given a rule that checks if \(j \in ag(r)\) is phosphorylated at \(s\) and then unphosphorylates this site, we have \(ag(r) = \{j\}\), \(\text{pre}(r) = \{(j, s)_\lambda = p\}\), and \(\text{eff}(r) = \{(j, s)_\lambda = u\}\). We show another example in Figure 3.1. The task of writing a Kappa program mostly involves the specification of such rules through the Kappa language semantics.

Constraining the kinds of rules we allow in a simulation will help us reason more easily about Kappa’s simulations. Specifically, we focus our attention on regular systems.
**Definition 13.** Regular systems are those in which every site modified by a rule is first tested. In notation, $\text{eff}(r) \subseteq \text{pre}(r)$. For instance, if the effect contains some $\langle u, s \rangle_\iota = p$, we must have $\langle u, s \rangle_\iota = s$ in its precondition where $s \in \{p, u\}$.

### 3.2.6 Kappa Rates

Kappa rules are accompanied by rates when specified in Kappa programs. For instance, in Figure 3.1, we see that the rules $A/$ and $B/$ are annotated with the number 1.0, and the rule $A.B$ is annotated with the number 1.0e-4. These rate specifiers define how often a rule is chosen during a simulation. A rule with a relatively high associated number will be chosen more often than others when the context in which it can act is available. We do not provide a formal description of rates and their effect on simulations in this writeup, but instead refer the reader to the description of the Gillespie algorithm in [16], which details the method by which reaction rates are used in these simulations. For the purposes of this research, it is sufficient to note that the rates and thus the kinetics of a system are carefully specified within Kappa, and that it is of interest to build tools that examine the frequencies with which certain events and patterns occur in Kappa simulations, allowing researchers to explore relationships between these rates and the simulations that unfold.

### 3.3 Kappa Simulations

Running Kappa simulations involves applying general Kappa rules, as described in Section 3.2, to particular instances of agents in the solution. Indeed, Kappa rules can be seen as chemical reactions that transform one local pattern to another via manipulations of local state. Since Kappa simulations often involve multiple copies of each agent type, mirroring the multiplicity (or concentration) of protein copies in our cells, a Kappa rule often can apply to one of many instances of the local pattern it checks for. This is desirable, as specifying rules
instead in terms of global state would require many (in fact, an infinite) set of rules. To create simulations from Kappa rules, at each step we must apply a rule by instantiating its labeled agents to match specific agents in the simulation. We describe this process in Sections 3.3.1 - 3.3.3, ending this section with a description of the sequence of events produced by these simulations.

3.3.1 Kappa Simulation State

Definition 14. In a Kappa simulation, the molecular entities present are described as instantiated agents, each given an agent identifier in $T \times \mathbb{N}$. Let $I \subset T \times \mathbb{N}$ be the set of agent identifiers used by the simulation at any point.

This combination of type and integer allows Kappa to model the presence of multiple distinct copies of each agent in the simulation. Restricting to the set $I$ will allow us to consider a finite number of variables when describing the simulation state.

To develop a description of Kappa’s global state during a simulation, we use descriptions analogous to Section 3.2.3, now based on instantiated agents as opposed to labeled agents, which are the entities used in rules.

Definition 15. Analogous to the definition of ports in Section 3.2.3, we have that the set of instantiated ports in the Kappa simulation is given by $P = \{\langle (t, n), s \rangle : (t, n) \in T \times \mathbb{N}, s \in \sigma_t\}$.

The set of port types remains the same as in Section 3.2.3.

Definition 16. Now we can describe the set of global state variables $X$, which much like $X_{ag(r)}$ for a rule, can be used to describe the state of the Kappa simulation. For every $i \in I$, we define the same three kinds of variables as in Section 3.2.3: $i_\emptyset, \langle i, s \rangle_\lambda$, and $\langle i, s \rangle_1$. We
have that $i_\emptyset \in V_{i_\emptyset}$, $\langle i, s \rangle_\lambda \in V_{\langle i, s \rangle_\lambda}$, and $\langle i, s \rangle_t \in V_{\langle i, s \rangle_t}$. The definitions for these value sets are similar to Section 3.2.3, and are given below:

\[
V_{i_\emptyset} = \{true, false, null\}
\]
\[
V_{\langle i, s \rangle_\lambda} \subset V_{int}
\]
\[
V_{\langle i, s \rangle_t} = \{free, null, bound\} \cup \{bound(\langle i', s' \rangle) : \langle i', s' \rangle \in P, \langle i', s' \rangle \neq \langle i, s \rangle\}
\]

**Definition 17.** $Q$ is the set of valuations of the state variables in $X$. An element $c \in Q$ specifies the values taken by each element $x \in X$ by associating $x \in X$ with a value in $V_x$. Based on the above description, we have:

\[
Q = \prod_{i=(t,n) \in I, s \in \sigma_t} ((i_\emptyset) \times V_{i_\emptyset}) \times ((\{i, s\}_\lambda) \times V_{\{i, s\}_\lambda}) \times ((\{\langle i, s \rangle_t\}) \times V_{\{\langle i, s \rangle_t\}})
\]

### 3.3.2 Events

The application of a rule in a system is called an event, a notion which we define in this section.

**Definition 18.** To apply a rule, we define an instantiation map $\psi : ag(r) \rightarrow I$, which maps labeled agents in $ag(r)$ to instantiated agents in $I$ while preserving the types of the agents involved.

This instantiation map will specify how to apply the rule to the current global state. In Figure 3.2, we show a sample global state, along with a couple instantiation maps for a rule $r$. We then have the following definition for events.

**Definition 19.** An event is a tuple of a rule and an instantiation map, $\langle r, \psi \rangle$, which represents the application of a rule.

The precondition and effect of an event on the global state can be obtained by mapping the rule’s precondition and effect to the global state using this instantiation map.
We will map between these states using the map $m_\psi : X_{ag(r)} \rightarrow X$:

$$m_\psi(x) = \begin{cases} 
\psi(j) & \text{for } x = j_{\emptyset} \\
\langle \psi(j), s \rangle_\lambda & \text{for } x = \langle j, s \rangle_\lambda \\
\langle \psi(j), s \rangle_\iota & \text{for } x = \langle j, s \rangle_\iota 
\end{cases}$$

In essence, we have that $m_\psi(x)$ for $x \in X$ takes the variable defined based on $j$ (whether specifying the binding state of a particular site as in $x = \langle j, s \rangle_\iota$ or the presence of $j$ as in $x = j_{\emptyset}$) to the variable defined based on $\psi(j)$.

Now we can define the precondition and effect of the event $\langle r, \psi \rangle$ on a system’s global state as follows.

**Definition 20.** Say that $\text{pre}(r)$ is the precondition for rule $r$. As defined in Section 3.2.2, $\text{pre}(r)$ and $\text{eff}(r)$ are partial valuations of $X_{ag(r)}$. We can define $\text{pre}(\langle r, \psi \rangle)$ and $\text{eff}(\langle r, \psi \rangle)$ as:

$$\text{pre}(\langle r, \psi \rangle) = \{(m_\psi(x), v) : (x, v) \in \text{pre}(r)\}$$

$$\text{eff}(\langle r, \psi \rangle) = \{(m_\psi(x), v) : (x, v) \in \text{eff}(r)\}$$

Thus, we have defined the precondition and effect of an event as the precondition and effect of a rule after the instantiation map is applied.

We will sometimes refer to the set of agents acted upon by an event, which in essence will be all instantiated agents that are involved in a state variable which the event checks or modified. A precise definition is below.
**Definition 21.** We label the set of agents acted on by an event $e$ as $agents(e)$. This set is defined as follows for event $\langle r, \psi \rangle$. First consider the map $ag\_id : X \rightarrow I$ which extracts the agent identifier from the corresponding state variable:

$$
\begin{align*}
ag\_id(x) &= \begin{cases} 
    j & \text{for } x = j_{\varnothing} \\
    j & \text{for } x = \langle j, s \rangle_{\lambda} \\
    j & \text{for } x = \langle j, s \rangle_{\iota}
\end{cases}
\end{align*}
$$

Then if we have $S = \overline{\text{pre}}(\langle r, \psi \rangle) \cup \overline{\text{eff}}(\langle r, \psi \rangle)$, the set of agents acted on by the event $\langle r, \psi \rangle$ is $\{ag\_id(s) : s \in S\}$.

### 3.3.3 Trace

A Kappa simulation can be seen as a stochastic simulation that generates sequences of events according to rules and their rates. This leads us to define traces:

**Definition 22.** A trace is a finite sequence of events. Trace $t$ has length $|t|$ and events $t_i$.

We want to define the precondition and the effect of a trace. This specifies what a sequence of events requires of a system in order to run to completion, and also tells us what effect that sequence of events then has on the system.

**Definition 23.** The **precondition of a trace** is defined inductively below.

$$
\begin{align*}
\text{pre}(\epsilon) &= \emptyset \\
\text{pre}(et) &= \begin{cases} 
    \text{null} & \text{if } \text{pre}(t) = \text{null} \\
    \text{null} & \text{if } \text{eff}(e) \downarrow \text{pre}(t) \text{ or } \text{pre}(e) \downarrow (\text{pre}(t) \setminus \text{eff}(e)) \\
    \text{pre}(e) \cup (\text{pre}(t) \setminus \text{eff}(e)) & \text{otherwise}
\end{cases}
\end{align*}
$$
In this system, we have only one rule, the binding of agents A and B. Here, we demonstrate a potential global state of the Kappa simulation, and show two possible instantiations of this binding rule in the simulation. **A**: A Kappa program detailing the binding of A and B. **B**: Much like in Figure 3.1, we present the Kappa program formalism for the program in A as described in Section 3.2. **C**: We describe the global state of the Kappa simulation as one instantiation of all the global state variables based on the instantiated variables in I. This description aligns with the definitions provided in Section 3.3.1. **D**: We present two possible events obtained by applying the rule A.B in two different ways, with instantiation maps ψ₁ and ψ₂.

**A. A simple Kappa program**

1. %agent: A(b)
2. %agent: B(a)
3. %var: 'Kon' 1.0e-4
4. 'A.B' A(b), B(a) -> A(b!1), B(a!1) @ 'Kon'

**B. Kappa program formalism**

Agent types: \( T = \{A, B\} \)

Agent sites: \( \sigma_A = \{b\}, \sigma_B = \{a\} \)

Rule A.B:

\[
\begin{align*}
\text{ag}(A.B) & = \{(A,1), (B,1)\} \\
\text{pre}(A.B) & = \{\langle (A,1), b \rangle_1 = \text{free}, \langle (B,1), a \rangle_1 = \text{free}\} \\
\text{eff}(A.B) & = \{\langle (A,1), b \rangle_1 = \text{bound}(\langle (B,1), a \rangle), \\
& \quad \langle (B,1), a \rangle_1 = \text{bound}(\langle (A,1), b \rangle)\}
\end{align*}
\]

**C. Kappa simulation state**

Instantiated agents:

\( I = \{(A,182), (B,47), (A,25)\} \)

Global state \( c \in Q \):

\[
\begin{align*}
\langle (A,182) \rangle & = \text{true}, \langle (B,47) \rangle = \text{true}, \\
\langle (A,25) \rangle & = \text{false}, \langle (A,182), a \rangle_\lambda = \text{null}, \\
\langle (B,47), b \rangle_\lambda & = \text{null}, \langle (A,25), a \rangle_\lambda = \text{null}, \\
\langle (A,182), a \rangle_1 & = \text{free}, \langle (B,47), b \rangle_1 = \text{free}, \\
\langle (A,25), a \rangle_1 & = \text{free}
\end{align*}
\]

**D. Some possible rule instantiations**, defined by maps of the form \( \psi : \text{ag}(A.B) \to I \):

\( \psi_1 : \psi_1((A,1)) = (A,182), \quad \psi_1((B,1)) = (B,47) \)

\( \psi_2 : \psi_2((A,1)) = (A,25), \quad \psi_2((B,1)) = (B,47) \)

Events: \( \langle A.B, \psi_1 \rangle, \langle A.B, \psi_2 \rangle \)

The precondition of an empty trace is the empty partial valuation on \( X \), because there are no restrictions on the state of the system for the empty trace to run. We define the precondition of a non-empty trace inductively. Say the precondition of trace \( t' \) is \( \text{pre}(t') \). Now we try to
find the precondition of trace $t = et'$. If $\text{eff}(e) \downarrow \text{pre}(t')$ (say they assign the same variable to different values), we state that the $\text{pre}(t) = \text{null}$, representing that no system can produce this sequence of events. We then have $\text{pre}(t) = \text{pre}(e) \cup (\text{pre}(t') \backslash \text{eff}(e))$. This indicates that we take the requirements for the rest of the trace, $t'$, to run, remove the conditions created by the event $e$, and add the preconditions for $e$ to run. Finally, if $\text{pre}(t)$ is not valid, then we set $\text{pre}(t) = \text{null}$. To pass the null values, we have the rule that $\text{pre}(t) = \text{null}$ if $\text{pre}(t') = \text{null}$.

We define the effect of a trace as the series of modifications that carrying out a trace will lead to. As above, this is defined inductively; the definition is given below with description following.

**Definition 24.** The effect of a trace is defined as follows.

$$
\text{eff}(\varepsilon) = \emptyset, \quad \text{eff}(et) = \text{eff}(t) \cup (\text{eff}(e) \backslash \text{eff}(t))
$$

As a base case, the effect of the empty trace is an empty partial valuation. Say that the effect of trace $t'$ is $\text{eff}(t')$. Then consider the trace $t = et'$. The effect of this trace is $\text{eff}(t') \cup (\text{eff}(e) \backslash \text{eff}(t))$. Essentially, we successively add in results from previous steps that are not overridden by later steps, thus capturing the full set of effects of a trace.

**Definition 25.** If a trace $t$ has $\text{pre}(t) = \text{null}$, we say that this trace is **invalid**. It is a trace that cannot be executed in any context. We will also say that a trace is **valid** in a valuation or context $c \in Q$ if $\text{pre}(t) \subset c$.

### 3.4 Causal Analysis of Kappa Simulations

Now, we have formally described the result of running Kappa simulations: a sequence of events reflecting a plausible sequence of Kappa rule applications from an initial starting system. In this section, we will develop a description of the structures developed to capture
the causal mechanism underlying these simulations. We will build to a description of stories, our representation of biological pathways based on the Kappa formalism.

3.4.1 Concurrent Events

Concurrent events are those that can be rearranged with little impact, such that the same preconditions for the events to run hold whatever order they run in, and such that the events have the same net effect on the system, whatever their order of execution.

**Definition 26.** Two events \( e_1 \) and \( e_2 \) are **concurrent** in \( c \) where \( c \in Q \) is a complete valuation of a system, if the following two properties hold:

1. We have that the following conditions are equivalent:
   
   (a) \((\text{pre}(e_1) \subset c) \land (\text{pre}(e_2) \subset c)\)
   
   (b) \((\text{pre}(e_1 e_2) \subset c)\)
   
   (c) \((\text{pre}(e_2 e_1) \subset c)\)

2. If the above three conditions are true, we must have that \( \text{eff}(e_1 e_2) = \text{eff}(e_2 e_1) \).

If \( e_1 \) and \( e_2 \) are concurrent in \( c \), we will write \( e_1 \diamond_c e_2 \).

**Definition 27.** Consider two events \( e_1 \) and \( e_2 \) with \( e_1 \diamond_c e_2 \). If the conditions in the first property of Definition 26 are all false, then these two events cannot run in this context, and are **trivially concurrent** in \( c \). Otherwise, these events are **nontrivially concurrent** in \( c \).

**Definition 28.** If two events \( e_1 \) and \( e_2 \) are concurrent in \( c \) for all possible valuations of the system, then they are **concurrent**. The notation \( e_1 \diamond e_2 \) denotes two concurrent events.

It can be proven [21] that the following condition holds if and only if two events \( e_1 \) and \( e_2 \) are concurrent: We have that if \( e_1 \) and \( e_2 \) are two events, \( e_1 \) and \( e_2 \) are non-trivially concurrent if and only if \( \overline{\text{pre}(e_1)} \cap \overline{\text{eff}(e_2)} = \emptyset, \overline{\text{pre}(e_2)} \cap \overline{\text{eff}(e_1)} = \emptyset, \overline{\text{pre}(e_1)} \uparrow \overline{\text{pre}(e_2)} \), and
eff(\text{e}_1) \uparrow \text{eff}(\text{e}_2). \text{ It can also be demonstrated that these events are only trivially concurrent if and only if pre}(\text{e}_1) \downarrow \text{pre}(\text{e}_2), \text{ post}(\text{e}_1) \downarrow \text{pre}(\text{e}_2), \text{ and post}(\text{e}_2) \downarrow \text{pre}(\text{e}_1).

Furthermore, Laurent et al. [21] show that in regular systems (defined in Section 3.2.5), two events are concurrent if and only if they are concurrent in any valuation \(c\).

### 3.4.2 Precedence Relation over Traces

We can use our definition of concurrency to establish equivalence classes on traces. Traces \(t\) and \(t'\) will be equivalent when they only differ by the rearrangement of concurrent events. We define the precedence relation below to help describe these equivalence classes.

**Definition 29.** Consider a trace \(t = t_1t_2...t_{|t|}\). We define a trace precedence relation over these events such that if \(i < j\) and \(t_i\) and \(t_j\) are not concurrent, then \(t_i\) is before \(t_j\) in this precedence relation, denoted as \(t_i \leq t_j\).

This precedence relation allows us to obtain from a trace \(t\) two related constructions: a directed acyclic graph (DAG), and a set of precedence pairs. First, note that we can use this precedence relation to generate a DAG from a trace, define as follows.

**Definition 30.** We here define a DAG of a trace, denoted \(\text{DAG}(t)\) for trace \(t\). \(\text{DAG}(t)\) has nodes with unique labels \(n_i\) associated with events \(\text{event}(n_i)\). The edges of the DAG satisfy the following properties:

1. If there is an edge from node \(n_i\) to node \(n_j\), then \(\text{event}(n_i) \leq \text{event}(n_j)\).

2. If \(t_i \leq t_j\), there are some nodes \(n_p\) and \(n_q\) with an edge from \(n_p\) to \(n_q\), such that \(\text{event}(n_p) = t_i\) and \(\text{event}(n_q) = t_j\).

We will use this graph representation in later sections. When we draw graphs for traces based on precedence relations, we will often forgo drawing arrows that are the transitive closure
of other arrows in the DAG. For instance, if we have $a \leq_t b$, $b \leq_t c$, and $a \leq_t c$, we will eliminate the arrow $a \leq_t c$ from the graphical representation of this trace. This allows for a cleaner, more concise graphical representation of the trace, which we will use in figures in this report. Despite these concise visual representations, the DAG for a trace will formally include all pairs $(e_i, e_j)$ such that $e_i \leq e_j$.

This precedence relation also allows us to generate a natural set of ordered pairs of events for a trace.

**Definition 31.** For a trace $t$, we will consider as its set of precedence pairs all pairs $(t_i, t_j)$ such that $t_i \leq t_j$. We will denote this set of pairs as $pairs(t)$. Note that this corresponds to the set of edges in $DAG(t)$.

### 3.4.3 Trace Equivalence

Traces are equivalent if they vary only by the rearrangement of concurrent events. We define this equivalence relation below.

**Definition 32.** Consider permutations from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$ where there are $n$ events in the trace. A permutation $\sigma$ preserves a precedence relation if $t_i \leq_t t_j$ implies $\sigma(i) \leq \sigma(j)$. We write $t' = \sigma(t)$ if $t = t_1 t_2 \ldots t_{|t|}$ and $t' = t_{\sigma(1)} t_{\sigma(2)} \ldots t_{\sigma(|t|)}$. We have that $t'$ and $t$ are equivalent, denoted $t' \sim t$ if $t' = \sigma(t)$ for some permutation $\sigma$ preserving the precedence relation. This then defines an equivalence relation between traces, denoted $\sim$.

**Definition 33.** A trace class is an equivalence class of traces under the above equivalence relation. The elements in a trace class are traces that are equivalent up to rearrangements of concurrent events.

We now define the DAG and precedence pair set for a trace class, which we will then show is well-defined.
Definition 34. For a trace class $m$, we will define the DAG of the trace class $DAG(m)$ to be the DAG of any trace $t \in m$, and we will define the set of precedence pairs of the trace class $pairs(m)$ to be the set of precedence pairs for any $t \in m$.

To see that the definitions above are well-defined, note that the set of precedence pairs and DAG for every trace $t$ in a trace class is the same. As described in Section 3.4.2, these constructions are based on pairs $t_i$ and $t_j$ such that $t_i \leq t_j$. Since the ordering of $t_i$ and $t_j$ remains the same across all $t$ in a trace class, and the set of concurrent and nonconcurrent events remains the same across trace classes, the DAG and precedence pairs constructed from these traces is the same. Then, we can construct a DAG and set of precedence pairs for each trace class.

We similarly note that we can make the following well-defined definition.

Definition 35. A trace class $m$ is valid if any trace $t \in m$ is valid.

To see that the validity of trace classes is well-defined, note that we can show that for all $t, t'$ in a trace class, we have $pre(t) = pre(t')$, and $eff(t) = eff(t')$. Since traces within a trace class have the same precondition and effect, this implies that validity of traces extends to trace class.

3.4.4 Stories

Now we describe stories, which we will use to capture a biologist’s intuition for a pathway, or a causal mechanism. In a biological system, one might wonder which sequence of events can lead to the formation of a particular observable; in particular, it is interesting to determine the set of events which in some way, either directly or indirectly, contribute causally to the creation of this observable. Such a set of events along with the dependency between them can help researchers create strategies to intervene in protein networks. Stories are a formalism
meant to capture this notion of causal mechanism.

**Definition 36.** A story \( s_{(e,c)} \) is a trace class associated with an event \( e \) and a valuation \( c \in Q \), satisfying the following conditions:

- The trace class \( s_{(e,c)} \) is valid in the valuation \( c \in Q \), as defined in Section 3.4.3.
- For each \( t \in s_{(e,c)} \), there is \( t_i = e \) in the trace \( t \), such that there is no event \( t_k \) in \( t \) where \( t_i \leq t_k \) in the story. Note that if this property holds for one \( t \in s_{(e,c)} \), it holds for all other \( t' \in s_{(e,c)} \) as well by the definition of trace equivalence in Section 3.4.3.

We call \( e \) the **event of interest**, and the rule it applies the **rule of interest**.

By the second condition above, the event of interest is in some sense the last event of the story. We view this event as the final product whose causal mechanism or pathway we are trying to describe with the story. We will say that if trace \( t \) is in the trace class \( s_{(e,c)} \), that \( s_{(e,c)} \) is the story for trace \( t \)'s event of interest \( e \). Stories thus provide a collection of equivalent traces that explain the execution of an event \( e \).

**Definition 37.** We define \( S(r,c) \) as the set of stories \( s_{(e,c)} \) for all \( e = \langle r, \psi \rangle \) for some \( \psi : ag(r) \rightarrow I \). Then, \( S(r,c) \) is the **set of possible causal mechanisms for the rule of interest**.

Note that we have \( \text{DAG}(s) \) and \( \text{pairs}(s) \), as described in Section 3.4.2 - 3.4.3. We define some additional notation surrounding the DAG and set of precedence pairs of a story \( s \), which will be useful through this thesis.

**Definition 38.** We define the following **notation for stories**.

- For story \( s \), the DAG of this trace class is written \( \text{DAG}(s) \).
- The set of nodes in \( \text{DAG}(s) \) is denoted \( \text{nodes}(s) \).
• Each node in $\text{nodes}(s)$ is given a unique label $s_i$.

• The node $s_i$ will correspond to event $\text{event}(s_i)$.

• $\text{events}(s)$ will be $\{\text{event}(s_i) : s_i \in \text{nodes}(s)\}$.

• The final node in the topological order of $\text{DAG}(s)$ (a single node due to the second property in Definition 36) will be denoted $s_0$, and $\text{event}(s_0) = e_0$.

• The precedence pairs for the story $s$ will be $\text{pairs}(s)$, which is equivalent to the edges of $\text{DAG}(s)$.

A sample story is depicted using the DAG representation in Figure 3.3.

We can alternatively define stories as a set of precedence pairs of events. To link these definitions, a set of precedence pairs will be a story in $S(r, c)$ if it is a valid trace class - that is, there is some valid trace whose trace class is identified by this set of precedence pairs.

Since stories may grow unwieldy with many biologically redundant events (say, the repeated binding and unbinding of an enzyme to a substrate without any other permanent alteration made), it is necessary to compress stories before we can use them as a formalism to understand causal mechanism.

### 3.4.5 Motivating Further Compression for Stories

Even after marking traces with reordered concurrent events as equivalent, stories can remain overly specific, needing further processing. The collection of potential stories in $S(r, c)$ significantly overapproximates the set of truly interesting biological pathways for the triggering of the rule of interest $r$ for multiple reasons, three of which we outline below.
3.4 Causal Analysis of Kappa Simulations

1. Multiple stories in $S(r,c)$ may describe the same causal chain of events, only differing in the specific instantiated agents acted upon in the story. For instance, in a system in which the only rule available is the binding of agents $A$ and $B$, there should only be one pathway for the triggering of this binding rule from an empty context: the trivial story with the introduction of an $A$ and $B$, followed by a single binding event between $A$ and $B$. However, without further processing, given the description in Section 3.4.4, we have distinct stories for binding instantiated agent $(A,1)$ to $(B,1)$ versus binding instantiated agent $(A,2)$ and $(B,1)$, and so on. These distinctions are specific to particular Kappa simulation runs, and not biologically significant.

2. Multiple stories in $S(r,c)$ may essentially describe the same biological pathway, except that they differ in the presence of cycles of events. For instance, in a system in which a site on agent $A$ is modified upon the binding of an enzyme $E$, it is possible that the story includes repeated binding and unbinding to the enzyme $E$ before the modification event occurs. In fact, there would be an infinite number of stories for the modification of $A$, with varying numbers of binding and unbinding cycles between $A$ and $E$ before the modification event occurs. Panel B3 in Figure 3.3 shows an example of a trace in which one such unnecessary bind-unbind cycle is present. In some cases, the presence of such cycles in may be interesting when analyzing different stories from Kappa simulations, for instance if the researcher wishes to understand whether the biological system is stuck repeating a certain portion of a pathway before proceeding to an important event. However, when understanding the set of rules necessary for the activation of the rule $r$, these cycles are less important; instead, collapsing the set of stories that differ by cycles of events can allow researchers to more easily pinpoint rules necessary for the activation of a pathway.

3. Stories in $S(r,c)$ may differ from each other in the presence of events that are unnecessary for the final triggering of $r$. Consider as an example a Kappa system including the
rules necessary for the triggering of \( r \), but also a collection of completely irrelevant rules, perhaps the rules that act in an entirely different cellular system. By the story description in Section 3.4.4, these rules will remain in the story for the triggering of \( r \), though none will be related to \( r \) even indirectly through the precedence relation. This leads to a collection of stories in \( S(r, c) \) which only differ by particular irrelevant events that were triggered before \( r \).

Together, these three overapproximations of the set of interesting pathways lead to difficulties in using stories for the analysis of biological systems. With stories including event cycles and events not causally required for the triggering of the rule of interest, stories become large and unwieldy in ways that hide the real biological mechanism of interest. Furthermore, with these overapproximations leading stories to be labeled differently when they should not be, it becomes difficult to compare the stories for different trace’s occurrences for the event of interest \( e \); looking for stories that occur frequently becomes difficult, for instance, when nearly every Kappa simulation run produces a different story for the rule of interest \( r \). Finally, any algorithms that attempt to list and analyze all possible pathways for the triggering of \( r \) cannot simply list the stories in \( S(r, c) \) due to these overapproximations. Such a list would not usefully describe the set of biological pathways reachable in a simulation. Thus, with this formulation of stories, it becomes difficult to carry out analyses requiring either visual inspection of stories, comparisons of stories across simulations, or analysis on the set of possible stories. Further processing of stories is necessary.
3.4 Causal Analysis of Kappa Simulations

Fig. 3.3 Example Story and Weak-Compression for System Modeling Substrate Double-Phosphorylation by Enzyme

A: Here we demonstrate a Kappa program describing the phosphorylation of an agent A at two sites, x and y, by enzyme E. Note that the notation \(s \sim p\) indicates that site \(s\) is in the internal state \(p\), indicating phosphorylation. As used previously, the notation \(s!1\) specifies the external state of site \(s\), indicating that the site \(s\) is connected to another site which is also labeled with the label 1.

B1: The abbreviated global state through the simulation, detailing the production of two specific instances of A and E, along with the progress towards phosphorylating A at sites x and y, as indicated by the events in panel B2.

B2: Trace representing a potential sequence of events leading to the production of an unbound agent A with sites x and y phosphorylated. The instantiation maps \(\psi_i\) represent some instantiation map for the rule \(r\).

B3: A story, whose equivalence class includes the trace described by panel B2. Here we use the directed acyclic graph notation described in Section 3.4.2. This representation is abbreviated - note that arrow representing the transitive closure of depicted arrows (for instance, an arrow from \(\langle b, \psi_b1 \rangle\) to \(\langle u, \psi_{u1} \rangle\) is not depicted).

B4: A weakly compressed story for the story in panel B3. Note in this case, we are using alternative instantiation maps, for instance \(\psi_{A/2}\). For instance, this alternative map might work with instantiated agents \((A, 68)\) and \((E, 162)\). Weakly compressed stories provide more general story representations.

A. Kappa program for enzyme-substrate binding and substrate phosphorylation

\[
\begin{align*}
\%agent: & A(x^u, y^u, p, a) \\
\%agent: & E(e) \\
\%var: & 'HIGH' 1.0e6 \\
\hline
'A/' & -> A(x^u, y^u, a) @ 1.0 \\
'E/' & -> E(e) @ 1.0 \\
'b' & A(a), E(e) -> A(a!1), E(e!1) @ 1.0 \\
'u' & A(a!1), E(e!1) -> A(a), E(e) @ 1.0 \\
'px' & A(x^u, a!1), E(e!1) -> A(x^p, a!1), E(e!1) @ 1.0 \\
'py' & A(y^u, a!1), E(e!1) -> A(y^p, a!1), E(e!1) @ 1.0 \\
'obs' & A(x^p, y^p, a) -> A(x^p, y^p, a) @ 'HIGH'
\end{align*}
\]
3.4.6 Direct and Weak Compression for Stories

Background for Direct and Weak Compression

To address these challenges with the story formulation, weak compression has been developed. We will first develop theory to address the first challenge enumerated in Section 3.4.5, and then develop theory to address the last two challenges. Using these constructions, we will define weak compression in the following section.

To address the first overapproximation described in Section 3.4.5, weak compression will allow stories to be equivalent up to an isomorphism of agent identifiers. To do this, we use the map $I_\phi(s) : S(r,c) \rightarrow S(r,c)$, which we will define below such that if $s' = I_\phi(s)$, then $s'$ and $s$ only differ by the particular instantiated agents they are used on. Thus, we will be able to use this map to address the concern from the first overapproximation described in Section 3.4.5.

Note, the map $I_\phi$ is based on $\phi$, which we define below.

**Definition 39.** An agent isomorphism $\phi$ will be a map $\phi : I \rightarrow I$, where $I$ is the set of agent identifiers as defined in Section 3.3.1. We require that $\phi$ be an isomorphism from $I$ to $I$. Furthermore, we require that $\phi$ preserve the type of the agent, such that $\phi((t,n_1)) = (t,n_2)$ for all $(t,n_1) \in I$.

**Definition 40.** We now define the map of stories by agent isomorphism $I_\phi : S(r,c) \rightarrow S(r,c)$ by describing the construction of $s' = I_\phi(s)$ for any $s \in S(r,c)$. Take some trace $t = t_1...t_{|t|}$ with $t \in s$. Say each $t_i \in t$ has $t_i = \langle r_i, \psi_i \rangle$. Then consider the trace $t' = \langle r_1, \phi \circ \psi_1 \rangle...\langle r_{|t|}, \phi \circ \psi_{|t|} \rangle$. We define $s'$ to be the trace class $s' \in S(r,c)$ such that $t' \in s'$.

Thus, we’ve translated the story $s$ to the story $s'$ by applying the isomorphism $\phi$ to the instantiation maps used in $s$. We will use $I_\phi$ to design weak compression for stories such
3.4 Causal Analysis of Kappa Simulations

that stories cannot differ only by the specific agent identifiers assigned by the Kappa simulator.

Now we address the second two challenges described in Section 3.4.5, attempting to remove cycles and irrelevant rules in the path to the event of interest. Consider the following operation to transform a story into a simplified version via a graph embedding.

**Definition 41.** A graph $G$ embeds into a graph $G'$ if there is an injective mapping $f$ from nodes $v \in G$ to nodes $f(v) \in G'$ such that if $(v, w)$ is an edge in $G$ then $(f(v), f(w))$ is an edge in $G'$.

**Definition 42.** For stories, $s'$ embeds into $s$ if:

- $\text{DAG}(s')$ embeds into $\text{DAG}(s)$ with injective mapping $f$.
- The injective mapping $f$ constructed must preserve events. If $s_k = f(s'_k)$ for some $s'_k \in \text{nodes}(s')$ and $s_k \in \text{nodes}(s)$, then $\text{event}(s'_k) = \text{event}(s_k)$.

We call such a map $f$ story embedding mapping. We say that for stories $s$ and $s'$, $s \rightarrow s'$ if such an embedding can be constructed.

Note that for instance, a story with irrelevant rules removed will embed into a story with those rules present; in this way, the operator $s \rightarrow s'$ can be used to successfully remove the second two complications described in Section 3.4.5.

**Defining Direct and Weak Compression**

Having described the map $I_\phi$ and the operation of story graph embedding, we can define weak compression precisely.

**Definition 43.** Story $s'$ is a weak compression of story $s$ if $s, s' \in S(r, c)$ for some rule $r$ and context $c$, and we have that $I_\phi(s) \rightarrow s'$ for some agent isomorphism $\phi : I \rightarrow I$. If story $s'$ is a weak compression of story $s$, we will write $s \rightarrow_W s'$. 
Thus, weak compression for \( s \) allows us to construct more simplified stories via graph embeddings, and allows us to switch the particular agent identifiers acted on via the isomorphism \( \phi \). An example of weak compression is depicted in Figure 3.3.

For later use in this report, we will also define an even weaker form of compression which we call direct compression, which will essentially be weak compression without the isomorphism between agent identifiers.

**Definition 44.** Story \( s' \) is a **direct compression** of story \( s \) if \( s, s' \in S(r, c) \) for some rule \( r \) and context \( c \), if we have \( s \rightarrow s' \). We will write \( s \rightarrow_D s' \) if \( s' \) is a direct compression for \( s \). Thus, this compression only allows for graph embeddings and not agent identifier isomorphisms.

We can continually carry out compression on story \( s \), reaching further compressed stories. We define the notion of minimal compressions to describe when this process is complete.

**Definition 45.** Let \( s \) be a story in \( S(r, c) \). A story \( s^* \in S(r, c) \) is a **minimal weak compression** of \( s \) if \( s \rightarrow_W s^* \) and if there is no \( s' \in S(r, c) \) such that \( s^* \rightarrow_W s' \) and \( s^* \neq s' \). Similarly, a story \( s^* \in S(r, c) \) is a **minimal direct compression** of \( s \) if \( s \rightarrow_D s^* \) with no other \( s' \in S(r, c) \) such that \( s^* \rightarrow_D s' \). If \( s^* \) is either a minimal weak compression or minimal direct compression of \( s \), we call \( s^* \) a **minimal compression** of \( s \).

Note, we can also label \( s^* \in S(r, c) \) a minimal weak compression without having a starting story \( s \), by only requiring that there is no \( s' \in S(r, c) \) such that \( s^* \rightarrow_W s' \) and \( s^* \neq s' \). As shorthand, we will also state that \( s^* \) is a minimal compression of a trace \( t \) if \( s^* \) is a minimal compression of some story \( s \) with \( t \in s \).

The KaSim implementation of Kappa has implemented minimal weak compressed story construction. Indeed, in KaSim is a tool to construct minimally weak compressed stories for the trace \( t \) formed during a Kappa simulation of a regular system (as defined in Section
3.2.5). This construction allows us to eliminate some of the challenges in Section 3.4.5. A user can generate Kappa simulations and generate their minimal weak compressed stories, thus constructing more concise and manageable forms for stories for analysis.

3.4.7 Working with Stories

With compression, we have a method for expressing the pathways used in biological simulations. This formalism is general enough to ignore less meaningful distinctions between pathways, such as the specific agent identifiers used in a simulation or the cycles of events which do not contribute to the production of useful changes in a system. Stories in themselves can provide a useful formalism for elucidating the causal mechanisms underlying biological systems; for instance, by generating stories researchers can observe the graphical representation of the causal mechanism behind the event of interest. However, answering even basic biological questions, like for instance asking which pathway is most frequently executed in simulations of a Kappa system, requires building additional frameworks and tools on top of stories. Through this thesis, we will explore ways to work with stories to find answers to interesting biological questions.
Chapter 4

Story-Trace Matching

To answer interesting biological questions involving causal mechanisms, we explore the following challenge, which we will call the story-trace matching challenge:

**Story-Trace Matching Challenge:** Define precisely the criteria for a story $s \in S(r,c)$ to be a reasonable minimal weakly compressed story for the trace $t$; if this is the case, we say that story $s$ matches trace $t$. Then, develop an algorithm that, when given a minimal weakly compressed story $s$ and trace $t$, efficiently determines whether $s$ matches $t$.

In this chapter, we will first motivate the need to explore story-trace matching with a biological question in Section 4.1; later in Section 5, we will apply the story-trace matching algorithm to this question and others.

In Section 4.2 - 4.5, we then proceed to explore three story-trace matching criteria, each serving as successively more complex building blocks for story-trace matching, but also serving as successively more useful stand-alone story-trace matching algorithms. The three criteria are as follows:

1. Most stringent criteria: Determine if a story $s$ is a direct compression of trace $t$. 
2. Most lenient criteria: Determine if a story $s$ is a weak compression of trace $t$.

3. Final version: Determine if a story $s$ is a weak compression of trace $t$, refined to eliminate unreasonable matches. This approach will be defined in more detail in Section 4.4 - 4.5.

By the end of this chapter, we will have presented algorithms to determine if a story matches a trace according to each of the above three notions of story-trace matching, along with explicit conditions for these algorithms’ correctness. Moreover, the third notion for story-trace matching will provide a new matching criteria that will eliminate unreasonable matches. We will demonstrate that this third criteria ensures that in a subset of Kappa programs which we specify, only one story can match a trace. Furthermore, this criteria allows for the implementation of a more efficient story-trace matching algorithm.

We briefly discuss the implementation of these algorithms in the Kappa code development, KaSim, in Section 4.6.

### 4.1 Motivation for Story-Trace Matching

In this section, we motivate the need to precisely find story-trace matches by considering the challenge of determining the frequency with which various pathways occur in a simulation system. We consider other applications for a tool answering this question in Section 5.

#### 4.1.1 Introducing the Question of Story Prevalence in a Kappa System

Using Kappa, it would be useful to answer the question: what is the most prominent causal mechanism explaining the triggering of an rule of interest $r$? As discussed in Section 1, the question of which pathway is most prominent lies at the foundation of understanding systems as diverse as cell death and cancer, and answering this question accurately would help design
4.1 Motivation for Story-Trace Matching

Effective interventions in these systems. However, multiple stories may lead to the triggering of rule $r$ (as in, the set $S(r; c)$ contains more than one member). This leads us to ask: how can we determine which of these stories is important to act upon with an intervention or to study further in the pursuit of biological understanding?

Even in simple systems, multiple minimal weakly compressed stories may lead to the triggering of the same rule of interest, and these stories may be different in biologically interesting ways. For instance, in Figure 4.1, if Story 1 is far more prevalent than Story 2, and if $ABC$ is a disease target which we are seeking to inhibit, it would make more sense to disable the triggering of $A.C$ as opposed to $B.C$. To reach biological insight like this, it is desirable to build tools to determine which story is most prevalent in a simulation.

By exploring the story-trace matching challenge, we can build these tools. Indeed, to determine which stories of a set of stories is most often responsible for the production of an event of interest in a particular cell type, we can first run many simulations from initial conditions representing that cell type. We can then, for each of these simulations and each story in our set of stories, determine if this story matches that simulation. Thus, we can develop tallies for stories matching traces, and find which stories are most prevalent in simulation runs. We discuss this application further in Section 5.1, where we discuss using story-trace matching to determine story frequencies in a Kappa system representing cell apoptosis (cell death).
Fig. 4.1 Multiple Stories in Trimer Formation Kappa System

A. Here we demonstrate a program depicting the formation of a trimer ABC. The program consists of introduction rules which introduce agents A, B, and C into the system; dimer formation rules which detail the formation of AB, BC, and AC; and trimer formation rules which detail the binding of an individual agent to a dimer to produce ABC. Finally, the event ABC is included for ease of specifying stories. B. Two stories detailing the triggering of the event ABC. In Story 1, AC forms as the intermediate to production of ABC; in Story 2, BC is the intermediate. In this DAG, we represent the transitive closure arrows which are typically eliminated from story DAGs (Section 3.4.2). Also, we only represent the rule name, rather than the full event like \( \langle A/, \psi \rangle \), as the node of our graph.

A. Kappa program for ABC trimer formation

1 \%agent: A(b,c)
2 \%agent: B(a,c)
3 \%agent: C(a,b)
4 \%var: 'Kon' 1.0e-4
5 \%var: 'HIGH' 1.0e6
6 'A/' -> A(b,c) @ 1.0
7 'B/' -> B(a,c) @ 1.0
8 'C/' -> C(a,b) @ 1.0
9 'A.B' A(b,c), B(a,c) -> A(b!1,c), B(a!1,c) @ 'Kon'
10 'B.C' B(a,c), C(a,b) -> B(a,c!1), C(a,b!1) @ 'Kon'
11 'A.C' A(b,c), C(a,b) -> A(b,c!1), C(a!1,b) @ 'Kon'
12 'AB.C' A(b!1,c), B(a!1,c), C(a,b) -> A(b!1,c!3), B(a!1,c!2), C(a!3,b!2) @ 'Kon'
13 'AC.B' A(b,c!1), C(a!1,b), B(a,c) -> A(b!2,c!1), C(a!1,b!3), B(a!2,c!3) @ 'Kon'
14 'BC.A' B(a,c!1), C(a,b!1), A(b,c) -> B(a!2,c!1), C(a!3,b!1), A(b!2,c!3) @ 'Kon'
15 'ABC' A(b!1,c!3), B(a!1,c!2), C(a!3,b!2) -> A(b!1,c!3), B(a!1,c!2), C(a!3,b!2) @ 'HIGH'

B. Two stories for ABC trimer formation
4.1 Motivation for Story-Trace Matching

4.1.2 Why Use Story-Trace Matching for Finding Story Prevalence

Though it might seem feasible to determine story prevalence as described in Section 4.1.1 by generating stories from traces rather than matching stories to traces, story-trace matching is promising because it allows multiple stories to match a trace, and because matching stories to traces may be more efficient than generating stories from traces.

When trying to determine which story is most often the right causal explanation in Kappa simulations of a system from particular initial conditions, one approach might be to run many simulations and to use the tools built in KaSim to find the accompanying story for each simulation [KaSim]. We call this the story generation approach, as opposed to the story-trace matching approach in Section 4.1.1. This might provide counts for each story in a simulation system. However, this approach is not useful in practice for a couple reasons.

First, for a single event of interest in a trace, it is possible that multiple stories are valid weakly compressed stories for that event of interest; yet, the story generation approach can only produce one story per event of interest in a trace. This can be particularly problematic when trying to find dominant stories in a simulation system using the story generation approach, because this approach produces a single story from the trace, with the method arbitrarily deciding on one of the available stories in an unpredictable manner. This decision may lead an unreasonable answer for the story prevalence question described in Section 4.1.1. An example of this problem is shown in Figure 4.2. In this example, when two stories are available to be produced from the story generation approach, it is unclear which story will be chosen, leading to final tallies for story counts that are not based on any reality of the simulation system or underlying biological truth. For this reason, we turn to story-trace matching.
Here we describe a system in which in many cases, multiple stories can be considered the story for the generation of a phosphorylated protein \( A \). \( A \) is a Kappa program for the phosphorylation of \( A \). We do not provide introduction rules as in previous examples, and assume that the system is initialized with an \( A \), \( B \), and \( C \). The \( \text{obs} \) rule serves as the event of interest for forming stories. B. B1. In this panel, we show the global state of the system over time. \( A \) is initially bound to enzyme \( B \) and then unbound. It is then bound to a catalyst \( C \), rebound to enzyme \( B \), and phosphorylated at \( x \) by \( B \). B2. In this panel, we show the progression of events that make up the trace corresponding to the global state in Panel B1. The global state in Panel B1 above each event in Panel B2 represents the global state after the execution of the event in Panel B2. We write \( \psi \) to be the instantiation map for rule \( \psi \). B3. With this trace, there are two mechanisms for the phosphorylation of \( A \). In one mechanism, \( A \) binds to \( B \) via the rule \( b \) before getting phosphorylated by rule \( p \). In another mechanism, \( A \) may bind to a catalyst \( C \) first, then undergo an expedited binding to \( B \) via \( b^* \), and finally get phosphorylated by rule \( p \). Both these stories are valid weak compressions for the trace in Panel B2.

### A. Kappa program for phosphorylation of \( A \) with and without catalyzed binding

1. \%agent: \( A(a,x^u,p,c) \)
2. \%agent: \( B(b) \)
3. \%agent: \( C(c) \)
4. \%var: \'HIGH\' \( 1.0e6 \)

5. \'b\' \( A(a,c), B(b) \rightarrow A(a!1), B(b!1) @ 1.0 \)
6. \'bc\' \( A(a), C(c) \rightarrow A(a,c!2), C(c!2) @ 1.0 \)
7. \'b^*\' \( A(a,c!2), B(b), C(c!2) \rightarrow A(a!1, c!2), B(b!1), C(c!2) @ 2.0 \)
8. \'u\' \( A(a!1), B(b!1) \rightarrow A(a), B(b) @ 1.0 \)
9. \'p\' \( A(a!1, x^u), B(b!1) \rightarrow A(a!1, x^p), B(b!1) @ 0.1 \)
10. \'obs\' \( A(a, x^p) \rightarrow A(a, x^p) @ \'HIGH\' \)

### B. Two stories for phosphorylation of \( A \)

1. \( \psi_1 \)
2. \( \psi_2 \)
3. \( \psi_{\text{obs}} \)

<table>
<thead>
<tr>
<th>Story 1</th>
<th>( \langle b, \psi_b \rangle \rightarrow \langle p, \psi_p \rangle \rightarrow \langle u, \psi_u \rangle \rightarrow \langle \text{obs}, \psi_{\text{obs}} \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Story 2</td>
<td>( \langle bc, \psi_{bc} \rangle \rightarrow \langle b^<em>, \psi_{b^</em>} \rangle \rightarrow \langle p, \psi_p \rangle \rightarrow \langle u, \psi_u \rangle \rightarrow \langle \text{obs}, \psi_{\text{obs}} \rangle )</td>
</tr>
</tbody>
</table>
Beyond the benefit of allowing multiple stories to match a trace, it is possible that the story-trace matching approach can answer the story prevalence question discussed in Section 4.1.1 more efficiently than the story generation approach. Whereas the construction of minimal weak compressions per simulation example may be time-consuming, requiring the construction of graph embeddings, the story-trace matching approach has the promise of quickly matching previously generated stories to a trace.

### 4.2 Story-Trace Matching with Direct Compression

In this section, we describe a story-trace matching algorithm using the direct compression matching criteria, which we define below. Recall that direct compression was defined in Section 3.4.6.

**Matching Criteria 1.** If minimal compressed story $s \in S(r,c)$ is a direct compressed story for the trace $t$, we say that $s$ **direct matches** $t$.

In this section, we present an algorithm to determine if a story $s$ directly matches trace $t$.

Story-trace matching algorithms are dependent on the precise matching criteria for a story to match a trace. Here, the criteria which we impose for story $s$ to be reasonable minimal weakly compressed story is that $s$ is a direct compressed story for $t$. Note that all direct compressed stories are weakly compressed stories for $t$. However, direct compressed stories are required to use the identity isomorphism between agent identifiers. Thus, this matching criteria is quite stringent, and it is limiting to not allow stories to match traces when they differ by isomorphisms of agent identifiers (see Section 3.4.5).

In this section, we focus on the more limiting direct compression as a first step in developing a story-trace matching framework and algorithm for two reasons. First, the algorithm serves
as a useful building block for later matching algorithms, acting as a backbone structure onto which later algorithms add complexity. Moreover, the direct compression algorithm finds use in some applications on its own, as described in Section 5. Because the criteria for matching in this case is more stringent, direct compression matching can be more efficient than a more general story-trace matching; using this efficient approach when possible is useful.

4.2.1 Goal for Direct Matching Algorithm

To determine if story $s \rightarrow_D s(t)$, where $s(t)$ is the trace class which includes $t$, we must find an embedding from $DAG(s)$ to $DAG(s(t))$. In particular, we must find an embedding mapping preserving events $f$, such that for every $(s_1, s_2) \in pairs(s)$, we have $(f(s_1), f(s_2)) \in pairs(s(t))$.

To construct the embedding map $f$ above, it seems that we would have to construct $s(t)$ from $t$. This is undesirable, as finding the precedence relation for $t$ will take time. We will redefine the goal above to work directly with our inputs $s$ and $t$ rather than $s(t)$.

First, we can use the following claim:

Claim 1. For some embedding mapping $f$, the following are equivalent:

1. For all $(s_1, s_2) \in pairs(s)$, $(f(s_1), f(s_2)) \in pairs(s(t))$

2. For all $(s_1, s_2) \in pairs(s)$, event $(f(s_1))$ precedes event $(f(s_2))$ in $t$.

Proof:

1. $\implies$ 2. follows from the story definition.

2. $\implies$ 1.: If $(s_1, s_2) \in pairs(s)$, event $(s_1)$ and event $(s_2)$ are not concurrent. Then any nodes $s_1$ and $s_2$ such that event $(s_1) = e_1$ and event $(s_2) = e_2$ will remain as a pair in pairs$(s_i)$. Then, if event $(f(s_1))$ precedes event $(f(s_2))$ in $t$, we have $(f(s_1), f(s_2)) \in pairs(s(t))$. \qed
4.2 Story-Trace Matching with Direct Compression

Furthermore, note that every $s_i \in nodes(s(t))$ has a unique $event(s_i) \in t$, so instead of constructing $f$ above, we can construct map $M$ to take some $s_i \in nodes(s)$ to some $e \in t$.

Therefore, to determine if $s \rightarrow_D s_t$, it suffices to find a map $M$ from nodes in $s$ to events in $t$ defined as follows.

**Definition 46.** A direct story-trace embedding from story $s$ to trace $t$ is a mapping $M$ from $nodes(s)$ to events in $t$ which satisfies the following properties:

- $M$ is injective.
- If $M(s_i) = e$, we have $event(s_i) = e$.
- For every pair $(s_1, s_2) \in pairs(s)$, $M(s_1)$ precedes $M(s_2)$ in $t$.

**Goal:** Given story $s$ and trace $t$, find a direct story-trace embedding $M$ from $nodes(s)$ to events in $t$ if one exists. Otherwise, state that $s$ does not direct match $t$.

### 4.2.2 Direct Matching Algorithm

Now, we present the algorithm for finding $M$, which will be correct in a subset of Kappa programs as detailed in Section 4.2.3. This algorithm will simply traverse the trace backwards beginning with the last event, and greedily assign events in the story to the trace in topological order of the story.

As described in Section 3.3.3, we have trace $t$ with size $|t|$ and events $t_i$. As described in Section 3.4.4, we will use the $DAG(s)$ with node ids $s_i$, events $event(s_i)$, last node in topological order $s_0$, and $event(s_0) = e_0$.

Through the algorithm, we will build $M$ and $S$, defined as follows:
Data Structure 1. Embedding $M$: We will build $M$ as a set of pairs, with each pair $(s, e)$ representing a mapping from $s \in \text{nodes}(\text{DAG}(s))$ to $e \in t$. We will try to add to this set at every step as we move through the trace.

Data Structure 2. Enabled nodes $S$: We maintain this list of story nodes which we will try to map to events in $t$. A node is enabled once its successors have all been mapped in $M$.

The pseudocode is presented in Algorithm 1.

4.2.3 Correctness Condition for Direct Compression Matching Algorithm

In the above algorithm, we assign events in $s$ to the trace events in $t$ greedily. This correctly reports story embeddings when the story graph $\text{DAG}(s)$ satisfies the following property: there are no incoming arrows to events that instantiate concurrent rules. In this section, we present the precise definition of a concurrent rule, and in Section 4.2.4, we discuss the correctness of the greedy approach given that no application of concurrent rules has a predecessor in $\text{DAG}(s)$.

A concurrent rule is one in which no site state checked in its precondition is modified to contradict its original value. In terms of the formalism defined in Section 3, we have the following definition:

Definition 47. A rule $r$ is a concurrent rule if $\overline{\text{pre}(r)} \cap \overline{\text{eff}(r)} = \emptyset$.

For instance, consider the following example rules:

- \( \text{p'} A(x\sim u) \rightarrow A(x\sim p) \). A phosphorylation rule $p$ which tests if agent $A$ is unphosphorylated and then phosphorylates this agent is not a concurrent rule. For this
Algorithm 1 Direct Matching Algorithm

1: function GET_ALL_ENABLED_EVENTS(M, DAG(s), j)
2:    return all predecessors s' of s_j in DAG(s) where s'' ∈ M for all (s', s'') ∈ pairs(s)
3: end function

4: function DIRECT_STATE_UPDATE(S, M, DAG(s), i, j)
5:    M := M ∪ {i, j}
6:    S := S \ {s_j}
7:    S := S ∪ {GET_ALL_ENABLED_EVENTS(M, DAG(s), j)}
8:    return (S, M)
9: end function

10: function DIRECT_MATCHES(s_j, t_i)
11:    return event(s_j) = t_i
12: end function

13: function DIRECT_MATCHING_ALG(s, t)
14:    S := {s_0}, M := {}
15:    for i = |t| to 1 do
16:        if there exists s_j ∈ S such that DIRECT_MATCHES(s_j, t_i) then
17:            (S, M) := DIRECT_STATE_UPDATE(S, M, DAG(s), i, j)
18:            if (S = ∅) then
19:                return M
20:        end if
21:    end for
22:    if S ≠ ∅ then
23:        return Fail.
24:    else
25:        return M
26:    end if
27: end function
rule, the state variable describing the internal state of the phosphorylation site is in both pre(p) and eff(p).

- \texttt{\textbackslash 'A/' \rightarrow A().} An introduction rule \texttt{A/} which has no tests on the environment and then introduces an agent of type A into the environment is a concurrent rule, because pre(\texttt{A/}) = \emptyset.

- \texttt{\textbackslash 'r' A() \rightarrow B().} A rule \texttt{r} which tests if A is in the system and then creates B is a concurrent rule. It has a nonempty precondition, and produces an effect that does not change any aspect of this precondition.

Note that it is possible to apply a concurrent rule repeatedly without any intermediate rules required to reinstate the system’s state. Any two rule applications for a concurrent rule are concurrent events, in the sense described in Section 3.4.1.

As we will demonstrate in Section 4.2.4, the Direct Matching Algorithm will be correct whenever the following correctness condition holds:

\textbf{Theorem 1. Correctness Condition: The story graph DAG(s) includes no predecessor for an application of a concurrent rule. When this condition holds, the Direct Matching Algorithm is correct.}

For most Kappa systems, the correctness condition holds, allowing us to use the Direct Matching Algorithm. If the Kappa system has no concurrent rules, the story graph DAG(s) satisfies this correctness condition trivially. It is in fact unlikely for Kappa rules to be concurrent rules, since many biological patterns involve the checking of current state before modifying that state. One common appearance of a concurrent rule in Kappa systems is the introduction rule, used as the second example above. However, in the case where concurrent rules are present like \texttt{A/}, note that the correctness condition still holds as long as there is no
precedence pair \((s_i, s_j) \in \text{pairs}(s)\) such that \(\text{event}(s_j)\) is an application of \(A/\). Note that this is a reasonable expectation of Kappa systems’ introduction rules, as the only incoming arrow to an application of an introduction rule \(A/\) could be the application of a rule that depends on the lack of a particular instantiation of \(A\) (and thus is constrained to precede \(A/\)). This sort of a check would be unlikely in biological systems.

If this criteria on story graphs \(\text{DAG}(s)\) is indeed not met, we cannot use this greedy algorithm.

### 4.2.4 Correctness for Direct Compression Matching Algorithm

**Proof of Theorem 1.**

**Claim.** Every time a story-trace match is found by the algorithm, it is a true match according to Matching Criteria 1.

**Proof of Claim.** Note that because we are processing the story in topological order, if we have found an embedding map \(M\), for any pair \((s_i, s_j) \in \text{pairs}(s)\), we must have that if \(M(s_i) = t_m\) and \(M(s_j) = t_n\), then \(m \leq n\). Also, this embedding map must satisfy \(\text{event}(s_i) = M(s_i)\) for all \(s_i \in M\), because we only add to the partial mapping \(M\) when the trace event matches the story event to be matched. Finally, this map \(M\) is injective, because the trace events \(t_i\) can only be mapped to once during the course of the algorithm. Therefore, we have met the goal stated in Section 4.2.1, and \(s\) directly matches \(t\). □

**Claim.** Every time a story-trace match can be found, it is found.

**Proof of Claim.** First, we show the following: Consider any node \(s_k\) in \(\text{DAG}(s)\) such that \(s_k\) has some predecessor in \(\text{DAG}(s)\). At no point in the algorithm do we attempt to insert an event \(\text{event}(s_k)\) into the enabled node list \(S\) when \(\text{event}(s_k)\) is already mapped in \(S\).
To see this, first note that because $s_k$ has a predecessor, the rule corresponding to $s_k$ must not be a concurrent event by the correctness condition. For us to insert $\text{event}(s_k)$ into $S$ twice, there must not be an edge between $s_k$ and $s'_k$ for some $s_k$ with $\text{event}(s_k) = \text{event}(s'_k)$. If such an edge existed, the algorithm would not insert both $s_k$ and $s'_k$ into $S$. However, since any two applications of $s_k$ are not concurrent, there must be an edge between these two nodes, leading to a contradiction.

Now we move on to proving the claim. Say that a story-trace match can be found. There is then an embedding map preserving events $M$, such that if $(s_i, s_j) \in \text{pairs}(s)$ then $M(s_i)$ appears before $M(s_j)$ in $t$. Assume that we could not find a story-trace matching with the greedy algorithm. Then in our backwards traversal of the trace, for some story node $s_i$ in $\text{DAG}(s)$, we could not find $t_j$ such that $\text{event}(s_i) = t_j$. In other words, there must be some node $s_i$ for which we were able to assign a mapping for all its successors but not $s_i$ itself (this is any node that is left in enabled node list $S$). However, since at any point, there is only one possible match to a trace’s event in the $S$ (by the discussion above), and because the algorithm proceeds greedily, every mapping from node $s_k$ in $\text{DAG}(s)$ to the trace event $t_m$ chooses the highest possible $m$ such that a) $\text{event}(s_k) = t_m$ and b) the precedence pairs seen so far in $\text{DAG}(s)$ can be satisfied. Since all of $s_i$’s successors $s_k$ in $\text{DAG}(s)$ are mapped to this highest possible position in the trace, we cannot miss the trace event $M(s_i)$. If we did miss trace event $M(s_i)$, $s_i$ must map to a later position in the trace than its successors map to, violating the condition of the embedding map. Thus, we must be able to add the mapping $M(s_i)$, leading to a contradiction.

Thus, with Matching Criteria 1, we find a story-trace match when we can by the Direct Matching Algorithm, and when we find a match, it is correct.
4.3 Story-Trace Matching with Weak Compression

In this section, we discuss a story-trace matching algorithm using the weak compression matching criteria defined below. Weak compression was defined in Section 3.4.6.

**Matching Criteria 2.** If minimal compressed story \( s \in S(r, c) \) is a weakly compressed story for the trace \( t \) such that \( s \rightarrow_W t \), we say that \( s \) weak matches \( t \).

In this case, the criteria for a story \( s \) to be a reasonable weakly compressed story for the trace \( t \) is all-inclusive: any weakly compressed story \( s \) of trace \( t \) will match \( t \). Ultimately, this criteria will prove too lenient, with some weakly compressed stories matching the trace when they do not prove to be a reasonable match, as we discuss in Section 4.4. Nevertheless, we discuss this algorithm because it provides the most general form of weakly compressed story-trace matching and provides a building block for later refinement; we will later alter this algorithm slightly to handle the difficulties mentioned in Section 4.4.

In Section 4.3.2, we discuss the additional complexities that the weak compression matching criteria poses, setting up the changes that will need to be made to the Direct Matching Algorithm. In Section 4.3.3, we describe the Weak Matching Algorithm which will determine if \( s \) weak matches \( t \).

### 4.3.1 Goal for Weak Matching Algorithm

In the case of weak compression, our matching task is more challenging than the case of direct compression. Now, we must allow for the agent identifier isomorphism used in the definition of weak compression (Section 3.4.6) to be a non-trivial map. This means we must decide in our algorithm which of the specific rule instantiations in our trace to match a story’s DAG node to. With this choice, our algorithm cannot follow the greedy approach of Section 4.2.
To decide when a story node $s_j$ matches the trace event $t_i$ in the weak compression case, we can no longer directly check equality, as in the $\text{event}(s_j) = t_i$ check from Algorithm 1. Note that if the story’s event is $\text{event}(s_j) = \langle r, \psi_s \rangle$, we can have $t_i = \langle r, C \circ \psi_s \rangle$ for some map $C : I \rightarrow I$ consistent across the story-trace matching algorithm. This is the agent identifier isomorphism described in Section 3.4.6. We call this $C$ a concretization map, as it maps the story’s agent identifiers used in its events to the concrete examples in the trace. Our goal then is to find a concretization map $C$ along with the embedding map $M$ that we sought in Section 4.2. A concretization we make at a particular step of the algorithm must be consistent with concretizations from previous steps, since the agent identifier isomorphism as described in Section 3.4.6 is consistent across story nodes. To build $C$, we maintain the additional datastructure below.

**Data Structure 3. Concretization map $C$:** We use $C : I \rightarrow I$ to capture mappings from story agent identifiers to trace agent identifiers. For any $M(s_j) = t_i$, we will have that if $\text{event}(s_j) = \langle r, \psi_s \rangle$ and $t_i = \langle r, \psi_t \rangle$, then $\psi_t = C \circ \psi_s$.

**Definition 48.** A **weak story-trace embedding** from story $s$ to trace $t$ is a mapping $M$ from $\text{nodes}(s)$ to events in $t$ which satisfies the following properties:

- $M$ is injective.

- Say we have a concretization map $C : I \rightarrow I$. If $M(s_i) = e$, and if $\text{event}(s_i) = \langle r, \psi_s \rangle$, then $e = \langle r, C \circ \psi_s \rangle$.

- For every pair $(s_1, s_2) \in \text{pairs}(s)$, $M(s_1)$ precedes $M(s_2)$ in $t$.

We will refer to weak story-trace embeddings simply as story-trace embeddings in the future.

**Goal:** Given story $s$ and trace $t$, find a weak story-trace embedding $M$ from $\text{nodes}(s)$ to events in $t$ if one exists. Otherwise, state that $s$ does not direct match $t$. 


4.3.2 Additional Complexity of Weak Matching Algorithm: Parallel Exploration

Note, there can be multiple possible assignments available for a story node within the trace via the story-trace embedding map, and a greedy choice may not necessarily pick the correct match between a story node and a trace event. In fact, in Figure 4.3 - 4.4, we see two examples where the greedy approach to assigning story nodes to the first possible trace match will fail. In Figure 4.3, the greedy approach cannot consistently correctly concretize a story node to the current trace event, as multiple concretizations are available for this assignment. In Figure 4.4, it is incorrect to assign the story nodes to the first possible match in the backwards traversal of the trace, as doing so leads the algorithm to incorrectly report that the story does not match the trace in question.

Without changing the greedy algorithm in Section 4.2, we can thus incorrectly report that a story does not match a trace when it in reality does. For this reason, we need to assign story nodes to trace events via parallel exploration, allowing the algorithm to explore the both the possibility of extending C by assigning the story node to the trace event currently being processed, or skipping this story node assignment and choosing a later concretization. The state space exploration will be achieved in our algorithm via state duplications.

When we encounter a story node $s_k$ that could potentially be matched to the current trace event $t_i$, we will duplicate our current state, creating versions of our enabled nodes list and partial embedding map from Section 4.2.2 for two scenarios - one in which we can map $e_s$ to $e_t$ and the other in which we do not make this choice. For each step of the trace, we move each of our current states forward, further duplicating our states if necessary. To achieve this, we need to maintain a list of states which will grow through the algorithm’s execution. We precisely define this structure below.
A. Here we depict a Kappa program for the dimerization of two agents $A$; dimerization is the process through which an agent of type $t$ binds to another agent of type $t$. The rule $\text{DIMER}$ serves as the binding rule between two agents of type $A$. The rule $\text{obs}$ (the rule of interest in our Kappa system) occurs at a high rate once $\text{DIMER}$ occurs. Here we also have two routes for forming $A$: $\text{A/}$, which is the standard introduction rule, and $\text{A/c}$, which is a second introduction rule that occurs in the presence of $C$. While this set of rules is unlikely to occur in biological systems, we use it to simply illustrate the nondeterminism that can arise in larger Kappa systems.

B. Here we depict the story for the triggering of $\text{obs}$, which we will try to match to the trace in panel B3. In this depiction, as in previous figures, the events represented by each node are depicted in the story’s DAG. Above each event is the set of agents acted upon by the event, as defined in Section 3.3.2. In this story, one agent of type $A$, $(A, 5)$, is created via the rule $\text{A/c}$, and the other, $(A, 1)$, is created via the rule $\text{A/}$.

B2. The global state of the Kappa simulation system after each corresponding event in panel B3.

B3. The trace obtained by this Kappa simulation, which we are trying to match the story in panel B1 to.

B4. Two possible attempts at forming the concretization map $C$. Note that we must choose the second example, as $(A, 7)$ is the instantiated agent in the trace which is formed via $\text{A/c}$. For our algorithm in Section 4.3 to make this choice, it must explore both possible concretizations, as the backwards traversal algorithm has no way to a priori decide on the correct concretization.

A. Kappa program for AA dimer formation

```
1 %agent: A(a)
2 %agent: C()
3 %var: ’HIGH’ 1.0e6

5 ’A/’ -> A(a) @ 1.0
6 ’C/’ -> C() @ 1.0
7 ’A/c’ C() -> A(a), C() @ 2.0
8 ’DIMER’ A(a),A(a)->A(a!1), A(a!1) @ 1.0
9 ’obs’ A(a!1), A(a!1) -> A(a!1), A(a!1) @ ’HIGH’
```
4.3 Story-Trace Matching with Weak Compression

Fig. 4.4 Multiple Agent Identifier Isomorphisms in Weak Compression Story-Trace Matching: Greedy Assignment Failure

A. In this Kappa program, agent type \( A \) phosphorylates to agent type \( B \) by binding to \( B \). \( A \) can be produced via rule \( A/c \) or \( A \), as described in Figure 4.3. Here, note that phosphorylation occurs when \( B \) is bound to anything, with the bound agent left unspecified, as indicated by a \(_\). B. B1. Here we depict a story that we are attempting to match a Kappa simulation of this system to. As in Figure 4.3, above each event is the set of agents acted upon.

B2. Here we depict the global state of the Kappa system as it aligns with the trace in Panel B3. Agent identifiers are depicted, with superscripts and subscripts reflecting sites that can be modified. B3. The trace of a particular Kappa simulation run of the system described in A.

B4. Two attempts at building a concretization map \( C \), one of which fails to produce a story-trace match. Note here that because \( \text{obs} \) and \( p \) proceed without specifying the agent bound to \( B \), we cannot assign \((A, 9)\) in \( C \) until we reach a binding event \( b/ \), for which we have two choices. Here, the greedy approach would choose the incorrect instantiation of \( b/ \) to match the story in Panel B1.

A. Kappa program for \( B \) phosphorylation by \( A \), with two modes for producing \( A \)

```
%agent: A(b)
%agent: B(a,x^u^p)
%agent: C()
%var: 'HIGH' 1.0e6

'A/' -> A(b,x^u) @ 1.0
'B/' -> B(a) @ 1.0
'C/' -> C() @ 1.0
'A/c' C() -> A(b,x^u), C() @ 2.0
'b' A(b),B(a) -> A(b!1),B(a!1) @ 1.0
'u' A(b!1),B(a!1) -> A(b),B(a) @ 1.0
'p' B(a!_.,x^u) -> B(a!_.,x^p) @ 1.0
'obs' B(a!_.,x^p) -> B(a!_.,x^p) @ 'HIGH'
```

B. Multiple concretizations \( C \) in story-trace matching

```
1. \{(C,2)\} \rightarrow \{(A,9)\} \rightarrow \{(A,9),(B,5)\} \rightarrow \{(B,5)\}
 \{(B,5)\} \rightarrow \{(B,5)\}

2. \{(B,3)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a \rightarrow \{(A,2)\}_a \rightarrow \{(C,1)\}_a
\{(A,7)\}_a \rightarrow \{(A,2)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a
\{(A,2)\}_a \rightarrow \{(C,1)\}_a
\{(A,7)\}_a \rightarrow \{(C,1)\}_a
\{(A,2)\}_a \rightarrow \{(C,1)\}_a
\{(A,7)\}_a \rightarrow \{(C,1)\}_a

3. \{(B,3)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,2)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a \rightarrow \{(C,1)\}_a
\{(A,2)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a \rightarrow \{(C,1)\}_a
\{(A,2)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a \rightarrow \{(C,1)\}_a
\{(A,2)\}_a \rightarrow \{(C,1)\}_a \rightarrow \{(A,7)\}_a \rightarrow \{(C,1)\}_a

4. Failed \( C \) : \{(B,5) : (B,3), (A,9) : (A,2)\}
   Successful \( C \) : \{(B,5) : (B,3), (A,9) : (A,7), (C,2) : (C,1)\}
```
Data Structure 4. State list $\Omega$: The set of states which our algorithm is currently exploring is given by $\Omega = \{\Omega_1, ..., \Omega_{|\Omega|}\}$, where $\Omega_i = (S_i, M_i, C_i)$. In this way, each state $\Omega_i$ is captured by three quantities: the enabled story node list $S_i$, the embedding mapping from story nodes to trace events $M_i$, and the concretization map so far $C_i$.

We finally note that parallel state exploration provides additional motivation to explore the trace backwards, beginning at the event of interest. We hope that by starting our search for an embedding mapping from the story to a trace with the end event, we reduce the search space and resulting state blowup of our algorithm. This is possible because the event corresponding to the story DAG’s last node in topological order must map to the last event of the trace; this constraint initializes $C$ with some mappings between agent identifiers, reducing the number concretizations we must try later on in the trace traversal.

4.3.3 Weak Matching Algorithm

We present the pseudocode for the Weak Matching Algorithm in Algorithm 2, to achieve the goal stated in Section 4.3.1. The Weak Matching Algorithm can then be described as follows, using the line numbers from Algorithm 2.

1. Start iterating from last event in trace (Line 15), after initializing $\Omega$ described in Data Structure 4 to include a single state $\Omega_1$ (Line 14). $\Omega$ will start with the event of interest in the story in enabled story node set $S_1$, and with the concretization maps and embedding maps empty.

2. For each state $j$ in $\Omega$, try to extend the embedding map (Line 17). Recall that $\Omega_j = (S_j, M_j, C_j)$, with $S_j$ the enabled story nodes, $M_j$ the embedding mapping, and $C_j$ the concretization mapping. We will be assembling a new set of states, $\Omega'$, through this process.
Algorithm 2 Weak Matching Algorithm

1: function FIND_WEAK_MATCHES($s_k, t_i, C$)
2: \( \psi_s := \text{instantiation\_map}(\text{event}(s_k)) \)
3: \( \psi_t := \text{instantiation\_map}(t_i) \)
4: return all \( C' \) such that \( \psi_t = C' \circ \psi_s \), and \( C'[n] = C[n] \) for all \( n \in C \)
5: end function

6: function WEAK_STATE_UPDATE($S, M, C, t_i, s, k$)
7: new\_states := \{\}
8: C\_list := FIND_WEAK_MATCHES($s_k, t_i, C$)
9: \((S, M) := \text{DIRECT\_STATE\_UPDATE}(S, M, \text{DAG}(s), i, k) \quad \triangleright \text{See Algorithm 1} \)
10: new\_states := new\_states \cup \{(S, M, C') : C' \in C\_list\}
11: return new\_states
12: end function

13: function WEAK\_MATCHING\_ALG($s, t$)
14: \( \Omega = \{(\{s_0\}, \{\}, \{\})\} \)
15: for \( i = \|t\| \) to 1 do
16: \( \Omega' := \{\} \)
17: for \( \Omega_j = (S_j, M_j, C_j) \in \Omega \) do
18: \( \Omega' := \Omega' \cup \{\Omega_j\} \)
19: for \( s_k \in S_j \) such that \( \text{rule}(s_k) = \text{rule}(t_i) \) do
20: \( \Omega' := \Omega' \cup \text{WEAK\_STATE\_UPDATE}(S_j, M_j, C_j, t_i, s, k) \)
21: for \( (S_m, M_m, C_m) \in \Omega' \) do
22: if \( S_m = \emptyset \) then return \( M_m \)
23: end if
24: end for
25: end for
26: end for
27: \( \Omega := \Omega' \)
28: end for
29: return Fail.
30: end function
3. Iterate through each element \( s_k \) in \( S_j \) that corresponds to the same rule used in step \( i \) (Line 19).

4. Check if the next element of the trace \( t_i \) is a valid concretization of \( s_k \) (Line 1). We can attempt to find a valid concretization from \( s_k = \langle r, \psi_s \rangle \) to \( t_i = \langle r, \psi_t \rangle \) by finding a map \( C : I \rightarrow I \) such that \( \psi_t = C \circ \psi_s \). Furthermore, \( C \) must be consistent with \( C_j \); if \( C_j \) includes the pair of agent identifiers \((n_s, n_t)\), \( \phi \) must also take \( n_s \) to \( n_t \). This ensures that any mapping between agent identifiers used in the story to agent identifiers used in the trace is maintained across the steps of this algorithm. Note that even for a single choice of \( s_k \) and \( t_i \), multiple mappings \( C \) are possible; an example of this is shown in Figure 4.3.

5. Add \( \Omega_j \) to \( \Omega' \) (Line 18). In this state, we move to the next trace step without making an assignment for a story event. Whether or not it is possible to assign \( s_k \) to \( t_i \) via some concretization, we add \( \Omega_j \) to \( \Omega' \) to represent the choice of making no assignment.

6. If a valid concretization was found, add all possible valid concretizations as additional states to \( \Omega' \); these additions to \( \Omega' \) correspond to the choice of mapping the story event to the current trace event with a particular concretization, giving us the chance to explore these assignments as potentially building a valid story-to-trace embedding (Line 20). For each \( \Omega_i \) added to \( \Omega' \), we must have an updated enabled node list \( S \), embedding mapping \( M \), and concretization mapping \( C \). We update the enabled node list \( S \) and current embedding map \( M \) in a method analogous to that outlined in Section 4.2 (Line 9).

7. Replace \( \Omega \) with \( \Omega' \), updating the list of states which our algorithm will explore in future steps of the trace’s backwards traversal (Line 27).

8. Report that the story is a valid weak compression of this trace if the list of enabled nodes for any of these state machines becomes empty (Line 22). This indicates that we
4.4 Describing Undone Story-Trace Embeddings

The Weak Matching Algorithm accurately determines if a weakly compressed story matches a trace for an event of interest according to the definition of weak compression in Section 3.4.6. However, the algorithm is quite expensive due to the parallel exploration of multiple possible concretizations, and it produces story matches that do not reasonably represent the causal mechanism underlying a trace. In this section, we describe these unreasonable matches precisely. Later, in Section 4.5, we will describe the impact of generating more selective story-trace matches on the efficiency of the Weak Matching Algorithm.

4.4.1 Undesirable Story-Trace Matches: Undone Story-Trace Embeddings

The Weak Matching Algorithm produces some undesirable matches between a story and a trace. In particular, a story-trace match may be unreasonable because the story nodes embed into early events of the trace, whose effects on the system may be undone by later events; in this case, an entirely different causal explanation is more appropriate for this trace’s event of interest.

We can see examples of story-trace embeddings of this sort which we hope to avoid in Figure 4.2 and Figure 4.4. In Figure 4.2, Story 1 seems like an unreasonable match for this trace. Its initial application of binding rule b is completely undone by the unbinding rule u, and thus is irrelevant to any later product produced. In fact, the production of the phosphorylated agent A can be better explained by the more recent application of the catalyzed
binding rule, b*. Similarly, in Figure 4.4, the story match reported by the Successful C should in reality not be reported, because the binding event between (A, 7) and (B, 3) is undone before the production of the phosphorylated product; in reality, the binding between (A, 2) and (B, 3) is responsible for producing this product.

More than reporting unreasonable story-trace matches, these failures can create unrepresentative statistics when exploring questions like the story prevalence question outlined in Section 4.1.1. Recall that the strategy for using story-trace matching to answer that question is the following: first run many simulations, and for each of those simulations attempt to match stories to tally counts for each story. To see the challenge posed by unreasonable story matches, consider the case of the Kappa system described in Figure 4.2, and imagine that the rules b and u occur at a much higher rate than b*. In this case, a b and u may appear in nearly every simulation, such that the story for the phosphorylation event occurring via b accounts for almost 100% of simulations. Say we are researchers attempting to understand the causal mechanisms used in this system, and that we are only aware of the causal mechanism involving b, which is reasonable if it occurs more frequently in biological systems and thus in experimental data. We may see this story prevalence data marking that all simulations match the b story, and we may thus assume no other causal mechanism is possible. However, this assumption misses the fact that another story, involving b*, accounts for some percentage of the triggering of the phosphorylation event. In this way, unreasonable story-trace matches can hide the true underlying causal mechanisms of a Kappa system. It is desirable to define these failure cases precisely, and to design story-trace matching algorithms to avoid reporting them.
4.4.2 Defining Undone Story-Trace Embeddings

Here, we precisely define the criteria for a story embedding to be an undone story embedding. We will define story undoneness as a property of a particular story-trace embedding map $M$, as defined in Section 4.2. A story embedding is undone if any precedence pair in the story is invalid, as defined below.

Consider some precedence pair $(s_1, s_2) \in \text{pairs}(s)$. Intuitively, if the effect of $\text{event}(s_1)$ contributing to the precondition of $\text{event}(s_2)$ is undone before $\text{event}(s_2)$ occurs, it seems incorrect to indicate that $\text{event}(s_1)$ causes $\text{event}(s_2)$. Causal pair invalidity will capture this intuition precisely.

**Definition 49.** A precedence pair $(s_1, s_2) \in \text{pairs}(s)$ is a causal pair if $\{(x, v) : (x, v) \in \text{eff}(\text{event}(s_1)) \cap \text{pre}(\text{event}(s_2))\} = S \neq \emptyset$. In words, there is some state assignment that $\text{event}(s_1)$ produces which $\text{event}(s_2)$ depends on.

**Definition 50.** We call $S = \text{eff}(\text{event}(s_1)) \cap \text{pre}(\text{event}(s_2))$ the assignment list of the causal pair $(s_1, s_2)$, and we write $\text{assignment_list}((s_1, s_2)) = S$.

**Definition 51.** An assignment $(x, v)$ is an undone between events $t_i$ and $t_j$ in the trace if there is some element of the trace $t_k$ between $t_i$ and $t_j$ such that $(x, w) \in \text{eff}(t_k)$, $v \neq w$.

**Definition 52.** A precedence pair $(s_1, s_2)$ is an invalid causal pair if:

1. $(s_1, s_2) \in \text{pairs}(s)$ is a causal pair.

2. For every $(x, v) \in \text{assignment_list}((s_1, s_2))$, $(x, v)$ is undone between $M(s_1)$ and $M(s_2)$.

Note that by Section 3.4.1, since precedence pairs of events are nonconcurrent, they must have either $\overline{\text{pre}}(e_1) \cap \overline{\text{eff}}(e_2) \neq \emptyset$, $\overline{\text{pre}}(e_2) \cap \overline{\text{eff}}(e_1) \neq \emptyset$, $\overline{\text{pre}}(e_1) \downarrow \overline{\text{pre}}(e_2)$, or $\overline{\text{eff}}(e_1) \uparrow \overline{\text{eff}}(e_2)$. 


Therefore, while some precedence pairs in the story will not satisfy criteria 1. in the above definition of causal pair invalidity, any pair of story nodes \((s_1, s_2)\) satisfying criteria 1. above must have an edge between them in the story’s DAG, as this would imply \(\text{pre}(\text{event}(s_2)) \cap \text{eff}(\text{event}(s_1)) \neq \emptyset\).

**Definition 53.** The story-trace embedding \(M\) from story nodes in \(s\) to events in \(t\) is an **undone embedding** if with embedding \(M\), any precedence pair \((s_1, s_2)\) \(\in\) \(\text{pairs}(s)\) is an invalid causal pair.

We wish to only report a story-trace match if we can find a story-trace embedding \(M\) that is not undone.

### 4.4.3 Properties of Undone Story Embeddings

Story embedding undoneness allows us to state some nice properties about story-trace matching. Proofs and examples for these properties can be found in Appendix A. In particular, we have the following theorems.

**Theorem.** Consider a trace \(t\). There exists at least one story \(s\) such that the story-trace embedding \(M\) from \(s\) to \(t\) is not undone.

**Theorem.** Say we have two minimally compressed stories \(s\) and \(s'\) that are weak matches for trace \(t\), such that the embeddings \(M\) and \(M'\) from \(s\) and \(s'\) respectively to \(t\) are not undone. Furthermore, say Property 1 holds of \(M\) and \(M'\), or Property 2 holds of \(s\) and \(s'\). We must then have that \(s' = I_\phi(s)\) for some agent isomorphism \(\phi\) (defined in Section 3.4.6). Thus, aside from agent isomorphisms, \(s\) becomes the unique story that matches the trace \(t\).

**Property 1.** Say we have a story-trace embedding \(M\) from story \(s\) to trace \(t\). For every assignment \((x, v) \in \text{pre}(M(s_j))\) for any \(s_j \in \text{nodes}(s)\), there is some node \(s_k\) such that \((x, v) \in \text{eff}(M(s_k))\) and \((x, v)\) is not undone between \(M(s_k)\) and \(M(s_j)\), as defined in Section 4.4.2. In
Appendix A, we will describe cases in which a story-trace match is not undone, and yet this property does not hold.

**Property 2.** For every $s_k \in \text{nodes}(s)$, $\text{event}(s_k)$ has a single effect, such that $\text{eff}(\text{event}(s_k))$ has one element.

With the two claims above, we have that story embedding undoneness allows for nice properties surrounding story-trace matching. In particular, when either Property 1 or Property 2 holds, a story-trace matching algorithm which eliminates undone embeddings will only report a match for the single best story for the simulation, except for stories that differ by an agent isomorphism. Furthermore, this property allows for more efficient story-trace matching, as described below in Section 4.5.

### 4.5 Refined Story-Trace Matching with Weak Compression

Here, we discuss the impact of story embedding undoneness on the story-trace matching algorithm. In Section 4.5.2, we discuss the potential for story embedding undoneness to eliminate some parallel exploration involved in the story-trace matching algorithm described in Section 4.3. In Section 4.5.3, we discuss changes to the story-trace matching algorithm, such that no undone story-trace match embeddings are reported as true matches.

#### 4.5.1 Goal for Refined Matching Algorithm

Given the discussion on undone story-trace embeddings above, we present a new story-trace matching criteria defined below.

**Matching Criteria 3.** If $s$ weak matches $t$ and there is a story-trace embedding $M$ from $s$ to $t$ such that $M$ is not an undone embedding, we say that $s$ **refined matches** $t$.

Then we have the following goal for the Refined Matching Algorithm:
Goal: Given story $s$ and trace $t$, find a story-trace embedding $M$ from $\text{nodes}(s)$ to events in $t$ such that $M$ is not an undone embedding, if such an embedding exists. Otherwise, state that $s$ does not direct match $t$.

4.5.2 Impact of Story Undoneness on Story-Trace Matching Algorithm

In this section, we demonstrate that for a story-trace matching algorithm that does not report undone story-trace embeddings, we do not need some of the parallel exploration and state duplications described in Section 4.3. In Theorem 2, we first show that greedy assignments from story nodes to trace events are valid if the story node is causally relevant, as defined below, and in Theorem 3, we then show that all story nodes in minimal weakly compressed stories are causally relevant.

Definition 54. For story $s$, we define a **causally relevant** node to be an $s_k \in \text{nodes}(s)$ such that there is some $s'_k \in \text{nodes}(s)$ with $(s_k, s'_k)$ a causal pair (defined in Section 4.4.2).

Theorem 2. Say we are attempting to construct a story-trace embedding $M$ from story $s$ to trace $t$ according to the goal in Section 4.5.1, such that $s$ refined matches $t$. If the story node $s_k$ is causally relevant in $s$, it must be assigned in $M$ greedily.

Before proving this theorem, we first make the notion of greedy assignment in Theorem 2 more precise. Say the successors of $s_i \in \text{nodes}(s)$ are $s_{i1}, \ldots, s_{ik}$, so for each $j \in [1, k]$ we have $(s_i, s_{ij}) \in s$. Also, say that our algorithm has created a partial embedding mapping $M$ that maps each $s_{ij}$ to some trace event. Note that the algorithm in Section 4.3 would indeed have assigned all $s_{ij} : (s_i, s_{ij}) \in s$ to a trace event via the map $M$ by the time that $s_i$ is processed, because this algorithm traverses the story in backwards topological order. Now say that $r$ is $\min\{n : t_n = M(s_{ij}), \text{ for some } (s_i, s_{ij}) \in s\}$; thus, $r$ is the earliest trace position to which a successor of $s_i$ in the story is assigned. The assignment $M(s_i)$ is greedy if the choice $M(s_i) = t_m$ maximizes $m$ such that $m < r$, $\text{rule}(t_m) = \text{rule}(\text{event}(s_i))$, and the concretization
4.5 Refined Story-Trace Matching with Weak Compression

from \textit{event}(s_i) to \textit{tm} agrees with the concretization map \textit{C} assembled so far. We will argue below that if \(s_i\) is causally relevant, \(M(s_i)\) must be assigned greedily.

\textit{Proof of Theorem 2.}

We will show that if \(M(s_i)\) is not assigned greedily for a causally relevant node \(s_i\), this story embedding \(M\) is undone.

Say \textit{event}(\(s_i\)) = \(\langle r, \psi \rangle\). If \(M(s_i) = \textit{tm} = \langle r, \psi' \rangle\) is not assigned greedily, there is another event \(\textit{tp} = \langle r, \psi'' \rangle\) between \(\textit{tm}\) and \(\textit{tr}\) (where \(r\) is the earliest trace position to which \(s_i\)'s successors are assigned, as described above.) Furthermore, if \(\psi' = C' \circ \psi\) and \(\psi'' = C'' \circ \psi\), we must have that \(C'\), the concretization from \(s_k\) to \(\textit{tm}\), and \(C''\), the concretization from \(e\) to \(\textit{tp}\), both agree with the concretization map \(C\) so far in the algorithm, at the point when all of \(s_k\)’s successors have been assigned to a trace event by the embedding. Thus, for all \(n \in C\), we must have that \(C'[n] = C[n]\), and \(C''[n] = C[n]\). This condition on \(C'\) and \(C''\) holds because the algorithm in Section 4.3 tries to preserve the concretization map through the algorithm.

We will now show that if \(M(s_i) = \textit{tm}\) as above, \(M\) generates some invalid causal pair \((s_i, s'_i) \in \textit{pairs}(s)\), as defined in Section 4.4.2. To show this, we must demonstrate that 1. there is a pair \((s_1, s_2) \in \textit{pairs}(s)\) which is a causal pair, and 2. for every \((x, v) \in \textit{assignment_list}((s_1, s_2))\), \((x, v)\) is undone between \(M(s_1)\) and \(M(s_2)\).

To show condition 1., note that because \(s_i\) is causally relevant, for some \(s'_i \in \textit{nodes}(s)\), \((s_i, s'_i)\) is a causal pair. Then \((s_i, s'_i)\) satisfies this condition.

Now to show condition 2., consider any \((x, v) \in \textit{assignment_list}((s_i, s'_i))\), a state assignment which \textit{event}(\(s_i\)) creates and \textit{event}(\(s'_i\)) depends on. Note that \textit{ag_id}(x) must have been
assigned in $C$ to some agent identifier in the global state when $s'_i$ was handled by the algorithm (recall, $ag\_id$ is defined in Section 3.3.2, taking a state variable to the agent identifier in the system it describes). Since both $t_m$ and $t_p$ must agree with $C$, we know that $t_m$ and $t_p$ have in their effect $(x', v)$ where $ag\_id(x') = C(ag\_id(x))$. Of note, $t_m$ and $t_p$ are acting on the same instantiated agent for this environment change. Then the assignment $(x', v)$ must be undone between events $t_m$ and $t_p$ in the trace. However, as discussed above, $t_p$ is before $t_r$ in the trace, which is at most at the position of $M(s'_i)$ in the trace. Then, the assignment $(x', v)$ is undone between $t_m$ and $M(s'_i)$. Since this is true of all $(x, v) \in assignment\_list((s_i, s'_i))$, we have that all the effects on the environment that $e$ creates and $e'$ depends on must be undone between $M(s_i)$ and $M(s'_i)$, satisfying condition 2.

Therefore, $(s_i, s'_i)$ is an invalid causal pair, and the story embedding $M$ is undone. Then, for a causally relevant event $s_i$ in the story, the story-trace matching algorithm must assign $M(s_i)$ greedily.

\begin{flushright}
$\Box$
\end{flushright}

**Theorem 3.** In a minimal compressed story $s$, every node $s_i \in nodes(s)$ is causally relevant.

**Proof.** Proof of Theorem 3

If there is no $s'_i$ with $(s_i, s'_i) \in pairs(s)$, we could construct a story $s'$ with $s \rightarrow s'$ without the node $s_i$; this should not be possible because $s$ is a minimal compression. Furthermore, say that there is no causal pair $(s_i, s'_i) \in pairs(s)$, with nonempty assignment\_list($(s_i, s'_i)$).

Then there is no aspect of the environment which $event(s_i)$ is required to set up in order to progress towards the triggering of the event of interest. In fact, $s_i$ need not be included in the story. However, we should not be able to remove $s_i$ from the story, since our story is minimal compressed. Therefore, $s_i$ must be causally relevant. 

\begin{flushright}
$\Box$
\end{flushright}
**Result.** From Theorem 2 and Theorem 3, to determine if a minimal weakly compressed story refined matches a trace, we should assign all story events to the trace greedily.

### 4.5.3 Refined Matching Algorithm

Here, we present the Refined Matching Algorithm, which will match a story \( s \) to \( t \) according to the matching criteria in Section 4.5.1. This algorithm will make use of the story-trace embedding undoneness properties discussed in Section 4.4.2 - Section 4.5.2, modifying the Weak Matching Algorithm. First we discuss three modifications we will make to the Weak Matching Algorithm to achieve this matching criteria, and we then present pseudocode for the algorithm.

**Modification 1: Greedy Embedding Choices**

By the discussion in Section 4.5.2, we can eliminate some of the parallel exploration in the story-trace matching algorithm. For instance, we can eliminate the non-determinism demonstrated in Figure 4.4, matching the binding event \( b/ \) from the story to the first available match and thus assigning \((A, 9)\) to \((A, 2)\) in the concretization map \(C\). To achieve this, we will change Line 18 of the Weak Matching Algorithm to Line 24 - 28 in the Refined Matching Algorithm, such that we only add \(\Omega_j\) to \(\Omega'\) if no valid concretization from \(s_k\) to \(t_i\) found. Thus, if it is possible to concretize \(s_k\) to \(t_i\) in a way that agrees with the concretization map \(C\) assembled so far, we make the decision to do so in the algorithm.

Despite this improvement, some parallel exploration still remains, as in Figure 4.3. Indeed, this is the case because we cannot know ahead of time which concretization to use from story agent identifiers to trace agent identifiers, if multiple are available for the greedy
story-trace embedding choice. However, the elimination of the parallel exploration in Figure 4.4 should substantially reduce state space explosion in this modified algorithm.

**Modification 2: Eliminating All Undone Story Embeddings**

The only story-trace embeddings eliminated by the above modification to a more greedy approach are undone story-trace embeddings, as explained in Section 4.5.2. Thus, the modification above allows us to eliminate some false positive story-trace embeddings. However, story-trace embeddings reported by this modified algorithm may still be undone; we are not yet explicitly removing all possible false positives.

To go further and eliminate any undone story-trace embeddings, we make the following additional modification. While assigning story nodes to trace events we keep track of whether the preconditions of story events are undone before the event’s predecessor is assigned. We will use the structure $P$ defined below to help us determine if a story-trace embedding we are building is undone.

**Data Structure 5. Causal pair assignments list** $P$: $P$ will be a set of tuples of causal pairs and assignments; for instance, we might have $((s_k, s'_k), (x, v)) \in P$ for some causal pair $(s_k, s'_k)$ and assignment $(x, v)$. We will define $\text{fst}(P) = \{ p : (p, a) \in P \}$, and $\text{snd}(P) = \{ a : (p, a) \in P \}$. We will have the property that $(x, v) \in \text{assignment_list}((s_k, s'_k))$ for each tuple $((s_k, s'_k), (x, v)) \in P$.

Note, since there is still some parallel exploration involved in the algorithm, we will still have $\Omega$, a list of states representing the duplicated states of the algorithm. Now, we extend $\Omega_i$ to include its own causal pair assignments list $P_i$, such that $\Omega_i = (S_i, M_i, C_i, P_i)$.

We now briefly discuss how we will use $P_i$ in the Refined Matching Algorithm. Say the algorithm is assembling an embedding map $M_i$ from story events to trace events. For each
4.5 Refined Story-Trace Matching with Weak Compression

Algorithm 3 Refined Matching Algorithm

1: function REFINED_UPDATE_STATES(S, M, C, P, t_i, s, k)
2:   states := WEAK_STATE_UPDATE(S, M, C, t_i, s, k)  > See Algorithm 2
3:   new_states := {}
4:   for (S_j, M_j, C_j) ∈ states do
5:     new_nodes := keys(M_j) \ keys(M)
6:     remove_pairs := all causal pairs (s_k, s'_k) ∈ pairs(s) with s_k ∈ new_nodes
7:     P_j := P \ { (p, a) ∈ P : p ∈ remove_pairs }
8:     new_states := new_states ∪ {(S_j, M_j, C_j, P_j)}
9:   end for
10:  return new_states
11: end function

12: function REFINED_WEAK_MATCHING_ALG(s, t)
13:   Ω := [( {s_0}, {} , {} , {} )]
14:   for i = |t| to 1 do
15:     Ω' := {}
16:     for Ω_j = (S_j, M_j, C_j, P_j) ∈ Ω do
17:       P'_j := P_j \ { (p, (x, w)) : (x, v) ∈ eff(t_i) , v ≠ w }
18:       if fst(P_j) ≠ fst(P'_j), continue to next j in Line 16
19:       for s_k ∈ S_j such that rule(s_k) = rule(t_i) do
20:         for all causal pairs (s'_k, s_k) ∈ pairs(s) do
21:           P'_j := P'_j ∪ { (s'_k, s_k), (x, v) : (x, v) ∈ assignment_list(s'_k, s_k) }
22:         end for
23:       new_states := REFINED_UPDATE_STATES(S_j, M_j, C_j, P'_j, t_i, s, k)
24:       if new_states = ∅ then
25:         Ω' := Ω' ∪ Ω_j
26:       else
27:         Ω' := Ω' ∪ new_states
28:       end if
29:     end for
30:   end for
31:   Ω := Ω'
32: end function

33: return Fail.
precedence pair \((s_k, s'_k)\) in the story such that \(M_i(s'_k)\) has been assigned, we add all pairs \(((s_k, s'_k), (x, v))\) such that \((x, v) \in \text{assignment_list}((s_k, s'_k))\) to \(P_i\) (Line 21). Furthermore, we track whether \((x, v)\) is undone before \(M_i(s_k)\) is assigned; if prior to assigning \(M_i(s_k)\), we have some trace event \(t_k\) with \((x, w) \in \text{eff}(t_k), w \neq v\), we remove \(((s_k, s'_k), (x, v))\) from \(P_i\) (Line 17). If all \((x, v) \in \text{assignment_list}((s_k, s'_k))\) are undone before \(M_i(s_k)\) is assigned, we quit this branch of parallel exploration, since the story cannot match the trace with this state (Line 18).

Indeed, if this scenario is reached, any assignment of \(M_i(s_k)\) must lead the trace-embedding to be undone. However, if we do find a mapping \(M_i(s_k)\) before these relevant assignments are undone, we stop tracking assignments involved in \((s_k, s'_k)\) in \(P_i\) (Line 7).

**Modification 3: Using Single Story Match Properties from Section 4.4.3**

Say we are trying to see which stories within a set of stories matches a given trace, perhaps because we are answering the story-prevalence question described in Section 4.1.1 for a collection of stories. By the discussion in Section 4.4.3, if we find a story-trace match and if Property 1 or 2 holds of the story-trace embedding we constructed, we know that this is the only story-trace match we will be able to construct. Thus, we can stop our search for story-trace matches early at this point, avoiding checks for story-trace matches for the rest of set of stories.

### 4.6 KaSim Implementation

At this point, we have discussed three story-trace matching algorithms in Section 4.2, Section 4.3, and Section 4.5. These algorithms have been implemented in KaSim, the Kappa implementation [KaSim]; the implementation can be found at the GitHub development branch accessible at https://github.com/ramyarangan/Which-Pathway. The implementation is in OCaml, to integrate with the rest of KaSim.
We provide for two commandline options which can be used when running Kappa simulations: \texttt{-save-story [filename]} and \texttt{-match-story [filename]}. The \texttt{-save-story} option allows the user to save the story from their simulation to a file, and the \texttt{-match-story} option allows the user to match a story from file to the current trace.
Chapter 5

Applications

Having described algorithms to carry out story-trace matching, we are poised to derive additional biological insight from Kappa simulations. In this section, we start to explore these possibilities.

We begin by presenting an analysis of a Kappa system detailing the mechanisms of cellular death. In this case, the story-trace matching algorithm confirmed experimental findings detailing the prevalence of particular pathways for cellular death across various cell types. Additionally, we present an exploration of the dynamics of ring formation in biological systems. Through this example, we demonstrate that story-trace matching can help uncover properties of the dynamics of a system, for instance helping us pinpoint events that take long delays to execute. Finally, we discuss why story-trace matching is particularly useful for these applications, comparing this approach to existing tools.

5.1 Determining Prevalence of Apoptosis Pathways

To understand the usefulness of the story-trace matching algorithms of Section 4, we first investigate a system modeling the mechanisms of cellular death, or apoptosis. We will
discuss the decision to investigate this system, state the particular aim for this investigation, and finally describe the methodology and results of this analysis.

5.1.1 Apoptosis Pathway Modeling

Cellular death, known as apoptosis, is a carefully regulated biological process that can be altered in disease states, for instance to lead to delayed death in some cancer cells [24]. The biological pathways underlying cellular death are intertwined and complex, with different causal mechanisms leading to apoptosis triggered in varying cell types or initial cell conditions. The reactions underlying these pathways lend themselves to analysis via tools like Kappa, which can help carefully model the interactions occurring within this protein network.

Various models based on ODEs and Kappa have been developed to study the interactions involved in apoptosis, and insights from these models have been supported with biological experiments [3, 2, 23]; with prior biological validation, this system serves as an interesting example for analysis via Kappa, potentially leading to biologically relevant insight. Albeck et al. have built mathematical models to study the regulation of apoptosis, and these models’ predictions have been supported through biological experiments [2]. For instance, they have developed hypotheses to understand the mechanisms allowing for misregulated partial cell death in cells with abnormal protein levels, and have supported these hypotheses with wetlab techniques [2]. Furthermore, Lopez et al. have used these mathematical models to formulate apoptosis systems using PySB, a framework in which systems are described in Python and compiled to Kappa files for simulation [23]. As a model with some verified biological validity, the Kappa system compiled from this PySB model serves as a useful starting point for testing the story-trace matching algorithm.

Furthermore, the model generated by Lopez et al. presents an opportunity to study a Kappa
system of real scale. The initial counts for proteins involved in the system were determined by fitting model parameters to experimental data [2]. With up to a million copies per protein involved in the system, these counts mirror true counts for proteins present in our cells [22]. While handling this system will not imply that the story-trace matching algorithm can process all biological systems, it will indicate that for some real scale systems, this approach can be used to generate useful insights.

### 5.1.2 Type I and Type II Apoptosis Pathways

This apoptosis Kappa system is particularly interesting to investigate from the lens of story-trace matching, as apoptosis exhibits multiple pathways whose frequencies vary between initial conditions representing different cell types. As discussed in Section 4.1.1, these pathway frequencies can be understood by generating simulations and using the story-trace matching algorithm to determine which story (each representing one possible pathway) matches each simulation.

Two apoptosis pathways, Type I and Type II apoptosis, have been characterized in the biology literature [26]. As depicted in Figure 5.1a, Type I apoptosis proceeds through the production of caspase 3 via caspase 8, denoted by the arrow from C8 to C3. On the other hand, also as depicted in Figure 5.1a, Type II apoptosis proceeds via the mitochondria cell structure, dependent on the release from the mitochondria of cytochrome C (labeled CyC or in later figures, Cyto C); cytochrome C then joins other elements to activate caspase 3 and trigger apoptosis, denoted by an arrow from the Apoptosome complex containing CyC to C3. In this figure, the production of cPARP can be viewed as a proxy for apoptosis levels, as cPARP levels have been used in wetlab settings to measure the extent to which cells have progressed into cellular death [4].
Fig. 5.1 Apoptosis Pathway Diagram

Here we depict some of the key interactions involved in the apoptosis pathway in (a), and the relative activation of these interactions upon modulation of the levels of proteins XIAP and Bcl2 in (b). In both diagrams, labeled elements (either in ovals or not) are proteins or groups of proteins. An arrow from X to Y indicates that the presence of X in the system increases the presence of Y, and a blocked arrow from X to Y indicates that the presence of X decreases the presence of Y. As cPARP production occurs, cells move towards apoptosis. The Type I apoptosis pathway proceeds directly from DISC to C8 to C3 to cPARP production, whereas the Type II apoptosis pathway proceeds on the right branch, through the inactivation of XIAP by Smac. In (b), we see a subset of the interactions depicted in (a), and view how Type I vs Type II apoptosis proceeded in different experimental conditions, as described in Section 5.1.2. Thicker colored shading represents increased activity of a portion of the pathway.

(a) Apoptosis Pathway Including Type I and Type II Mechanism. Figure from [2].

(b) Apoptosis Modulation via XIAP Knockout and Bcl2 Overexpression. Figure from [4].
The differences between these pathways have implications for interventions targeting cell instability and disease. As an example of such instability, XIAP disregulation has been shown to lead to partial cell death instead of complete cell death, dangerously leaving cells alive with partially degraded genetic content [2]. Understanding whether a cell is Type I or Type II can inform how to intervene to prevent this partial cell death. Thus, to better understand when and how to intervene in apoptosis, it is of interest to characterize the conditions in which Type I vs Type II pathways are triggered.

Past work by Aldridge et al. has investigated Type I and Type II pathway prevalence as levels of XIAP and Bcl2 are modulated via wetlab experiments [4]; attempting to recreate these findings using the apoptosis Kappa system will highlight the usefulness of story-trace matching. Aldridge et al. measured Type I and Type II cell apoptosis levels in various cell types by staining products of these pathways in these cells. A schematic depicting the result of this experiment is shown in Figure 5.1b. In the wildtype (WT) condition Type II apoptosis was activated, in the XIAP knockout (ΔXIAP) condition Type II apoptosis was reduced, and in the simultaneous XIAP knockout and Bcl2 overexpression condition (OE-Bcl2/ΔXIAP) Type II apoptosis was eliminated. These conditions can be simulated in the corresponding Kappa apoptosis system by modulating initial conditions for the agents involved in the system.

5.1.3 Apoptosis Pathway Analysis Approach

Apoptosis Kappa System

As described in Section 5.1.1, the Kappa system we used was generated from the PySB system described by Lopez et al. [23]. The full system used can be found in Appendix B.
Story Formation

We first produced instances of Kappa stories capturing Type I and Type II apoptosis. We then used story-trace matching with these stories to understand the prevalence of Type I and Type II apoptosis in varying cell types.

To generate stories for Type I and Type II apoptosis, the formation of cPARP was used as the event of interest. As depicted in Figure 5.1, cPARP is a downstream product of both Type I and Type II apoptosis, and stories for the creation of cPARP will capture differences between these two pathways. Furthermore, the level of cPARP production is a useful indicator of progress in apoptosis, as discussed in Section 5.1.2.

We ran Kappa simulations with the apoptosis system above, generating the Type I and Type II stories depicted in Figure 5.2 using the story generation implementation in KaSim [KaSim]. These stories match the pathway diagrams depicted in Figure 5.1. Type I apoptosis proceeds via the production of C8 from DISC, and Type II apoptosis proceeds through the Apoptosome which includes Cytochrome C released from the mitochondria.

Of note, to generate Type II stories efficiently, we altered the initial conditions to over-express AMito, more quickly triggering the formation of a Type II pathway which otherwise took large simulation time to generate. This illustrates that in some cases, it may be desirable to change the Kappa system to produce desired stories, or even obtain stories without simulation at all. The story-trace matching approach does not require that stories be produced from the simulation system being analyzed.
Fig. 5.2 Stories Representing Type I and Type II Apoptosis

(a) Type I Apoptosis

- bind_L_pR
  - produce_DISC
    - bind_pC8_DISC
      - produce_C8_via_DISC
        - bind_pC3_C8
          - produce_C3_via_C8
            - bind_PARP_C3
              - produce_CPARP_via_C3
                - CPARP_total

(b) Type II Apoptosis

- bind_mCytC_AMito
  - produceACYtC_via_AMito
    - transloc_cCytoCACYtC
      - bind_Apaf_cCytoC
        - produce_aApaf_via_cCytoC
          - bind_aApaf_pC9_as_Apop
            - bind_pC3_Apop
              - produce_C3_via_Apop
                - bind_PARP_C3
                  - produce_CPARP_via_C3
                    - CPARP_total
Story-Trace Matching Setup

We aimed to understand the prevalence of the Type I and Type II stories generated above across the cell conditions explored by Aldridge et al. [4]. Our approach would be to run long simulations for each of these conditions, generating many instances of cPARP per simulation to represent the progression in apoptosis. We would then use the Refined Matching Algorithm from Section 4.5 to determine, for each instance of cPARP produced in the simulation trace, whether it matched Type I or Type II stories.

Using the Refined Matching Algorithm was complicated by the feature that multiple minimal weakly compressed stories can represent Type I or Type II pathways. For instance, some stories for Type I apoptosis differed from the example in Figure 5.2 because the agent C8 used was not present in the initial condition of the system, but rather was added to the system via the rule pC8_syn. Thus, some Type I apoptosis stories involved the rule pC8_syn. These sorts of differences complicate this story-trace matching approach, as matching stringently to the examples in Figure 5.2 would lead various real examples of Type I and Type II mechanism to not match the appropriate story, altering the data we collect.

To account for this, we extended the Refined Matching Algorithm to match a story suffix, and only matched the Type I story from the rule produce_C3_via_C8 onwards, and the Type II story from the rule produce_C3_via_Apop onwards. Matching this suffix ensured that we could distinguish Type I and Type II stories while allowing for all possible Type I and Type II stories to pass the story-trace matching query. Note that if the story suffix matches the trace, by an argument that follows the proof of Theorem 4 in Appendix A, we must have a story-trace embedding from a complete story $s$ to the trace such that the embedding is not undone, and such that the suffix of $s$ is the story suffix we matched.
5.1 Determining Prevalence of Apoptosis Pathways

Measuring Type I vs Type II Apoptosis Across Cell Conditions

We ran Kappa simulations for three cell conditions to mirror those explored by Aldridge et al. [4]. In the wildtype (WT) conditions, the system began with high levels of XIAP and Bid. To mirror the XIAP knockout (XIAP KO) condition, we set XIAP’s initial value in the system to 0. To mirror the XIAP knockout and Bcl2 overexpression condition, we set XIAP’s initial value to 0 and Bid’s initial value to 0; we call this condition the XIAP/Bid KO condition. Without Bid, production of Bax should be downregulated as if increasing Bcl-2 production, as depicted in Figure 5.1.

For each condition, we ran simulations until 80% of Smac in the system was released, such that cSmac_total in Appendix B was at least 80000. Ending the simulation based on sufficient Smac levels helped us pause the simulation when we would expect to see Type II apoptosis if it were triggered in the system (note that Smac is involved in Type II apoptosis as depicted in Figure 5.1). We observed that in the WT condition, apoptosis occurred slowly as measured by levels of cPARP production, with only 3.25% of cPARP produced by the time Smac was 80% activated; on the other hand, 97.2% and 99.6% of cPARP was produced in the XIAP KO and XIAP/Bid KO conditions respectively. These abundance levels of cPARP are to be expected. In Type II apoptosis, which should be preffered in the wildtype condition, much of the cPARP should be produced upon the entrance of Smac [4]. The XIAP KO condition reduces Type II apoptosis levels, and the XIAP/Bid KO condition further reduces Type II apoptosis, aligning with the level of cPARP in these two settings.

We would hypothesize that the first 3.25% of cPARP produced in the wildtype condition arise from a mix of Type I and Type II apoptosis, whereas the first 3.25% of cPARP produced in the knockout conditions should primarily arise from Type I apoptosis. To test this hypothesis, we carry out the Refined Matching Algorithm for matching Type I and Type II stories to
traces from simulations of these conditions. In particular, for each condition, we generate a trace including 32500 cPARP creation events, representing the production of 3.25% of cPARP in the system. For each of these cPARP creation events, we consider the trace up to that point. If in the trace $t = t_1 \ldots t_n$ a cPARP creation event occurs at event $t_k$, we define the trace $t^{(k)} = t_1 \ldots t_k$. We match both the Type I and Type II story to trace $t^{(k)}$ for all such $k$ using the Refined Matching Algorithm. Traces formed had between 2 million and 8 million events, and story-trace matching was executed for 32500 events of interest per condition.

5.1.4 Apoptosis Pathway Analysis Results

In Figure 5.3, we depict the result of the Refined Matching Algorithm on the WT and XIAP/Bid KO conditions. Note that in the WT case, some cPARP is produced via the Type II pathway. Later occurrences of cPARP in the WT trace matched with the Type II story. On the other hand, in the XIAP/Bid KO condition, no traces matched with the Type I story. Thus, as expected, in WT conditions, some early cPARP production (and thus early indicators of apoptosis) are formed via the Type II pathway; on the other hand, when XIAP and Bid are knocked out, Type I apoptosis completely controls cell death. The results from the XIAP KO condition were identical to those of the XIAP/Bid KO condition for the first 3.25% of cPARP produced, and thus are not shown. Indeed, knocking out XIAP was sufficient to remove Type II pathway occurrences from the first 3.25% of cPARP produced.

Thus, in this case, using story-trace matching helped identify the relative prevalence of two types of apoptosis over the course of a long simulation. These results correspond closely to those found experimentally by Aldridge, et al [4]. When used to analyze systems where various biologically distinct pathways are present, story-trace matching can thus help understand their prevalence, elucidating changes as different initial conditions are modeled.
5.1 Determining Prevalence of Apoptosis Pathways

Fig. 5.3 cPARP Production via Type I and Type II Apoptosis

In the top two graphs, for the WT and XIAP/Bid KO condition, we depict the total number of cPARP released via Type I vs Type II apoptosis as a function of the percent of total PARP cleaved. In the third graph, we show the rise of Smac level as a function of percent of PARP cleaved.

(a) cPARP in Wildtype Condition

(b) cPARP in XIAP/Bid Knockout Condition

(c) Smac Release
5.2 Characterizing System Dynamics in Ring Formation

Story-trace matching can be used to answer interesting biological questions beyond the story prevalence question explored in Section 5.1. In particular, story-trace matching may help uncover interesting properties of the dynamics of a system, for instance illustrating the amount of time spent in a particular state during the execution of a pathway. Understanding the time spent at steps in a pathway can help biologists find ways to better intervene in a pathway. For instance, if the goal is to increase the production of a final product, altering the reaction that is the bottleneck step in the pathway leading to that product may be the most effective modification.

In Section 5.2.1, we discuss the applicability of story-trace matching to investigating questions of system dynamics. In Section 5.2.2, we discuss systems which describe the formation of rings of agents; these systems have interesting dynamical properties which will highlight the capabilities of the story-trace matching approach. We discuss our method of analyzing this system in Section 5.2.3, and present results in Section 5.2.4.

5.2.1 Investigating Dynamics with Story-Trace Matching

By constructing an embedding from a story $s$ to a trace $t$ in the story-trace matching algorithm, we have an opportunity to understand the dynamics of a system. The embedding $M$ constructed pinpoints events in the trace which are causally relevant to the event of interest. Then, to understand the time that elapses between two story nodes $s_k$ and $s_k'$, we can look at the number of trace steps between $M(s_k)$ and $M(s_k')$, or we can look at the Kappa system time that this difference corresponds to. This leads us to define the following metric.
5.2 Characterizing System Dynamics in Ring Formation

**Definition 55.** If $M(s_k) = t_i$ and $M(s'_k) = t_j$, we will call $|j - i|$ the **trace delay between** $s_k$ and $s'_k$.

Understanding the trace delays of a story-trace embedding can help reach biological insight about a system. If for instance we find that the embedding $M(s'_k)$ occurs soon after the embedding $M(s_k)$ for all $(s_k, s'_k) \in \text{pairs}(s)$, such that the trace delay between $s'_k$ and all of its predecessors is low, the triggering of $s'_k$ is likely not limiting the progression of the story. As described above, these sorts of insights can provide essential understanding for biological pathways beyond stories themselves, as stories do not capture information on the dynamics of a system.

Importantly, the Refined Matching Algorithm finds embeddings that are not undone, allowing the trace delays reported by this approach to be accurate. To see this, consider a story-trace matching algorithm which allows for undone embeddings to be reported. In such an algorithm, for some $(s_k, s'_k) \in \text{pairs}(s)$, it is possible that $M(s_k)$ is a trace event that is far earlier than $M(s'_k)$, such that the effects of $M(s_k)$ are undone and irrelevant to the triggering of $M(s'_k)$. Computing trace delays in such a setting would lead to arbitrary results on the time spent between an event’s occurrence and the event that it contributes to causally. Thus, the Refined Matching Algorithm in particular provides a useful setting to understand system dynamics by computing trace delays.

### 5.2.2 Test System: Ring Formation

To test the applicability of story-trace matching to answering questions about system dynamics, we explored a system detailing the formation of three-member rings. In this system, agents $A$, $B$, and $C$ successively bind to each other to form trimers $ABC$. As depicted in Figure 5.4b, any monomer ($A$, $B$, or $C$), can bind to any other monomer to form a dimer ($AB$, $AC$, $BC$).
These dimers can undergo dissociation to return to the constituent monomers, for instance with $AB$ unbinding to form $A$ and $B$ again. A dimer can bind to a free monomer of the remaining type to form the three-member ring, $ABC$. Note that two dimers $AB$ and $BC$ cannot interact to form a trimer, as the two $B$ proteins will clash (Figure 5.4b).

We chose to analyze this system because it exhibits interesting dynamical properties. In fact, Deeds et al. found that different rates for dimer dissociation influenced the efficiency of forming trimers $ABC$, and hypothesized two distinct and competing explanations for this finding [10]. First, at high dissociation rates, they hypothesized that the intermediate dimers required to produce $ABC$ do not last until the formation of the trimer, hindering trimer production. Second, at low dissociation rates, they hypothesized that while the dimers required to form $ABC$ do last for long stretches during the simulation, it is difficult for dimers to find a free monomer of the third type to complete trimer formation; most such monomers will be bound in a long-lasting dimer of their own. These competing effects on trimer production efficiency create an intermediate optimal dimer dissociation rate for efficiently producing $ABC$ [10].

In this system, we see an opportunity to use story-trace matching to illustrate these competing factors by analyzing trace delays. We used the Kappa program included in Figure 5.4a to explore this system. While this system is a toy system in that it does not directly model a particular biological pathway, analyzing this system can provide insights into the basic biological pattern of ring formation.

### 5.2.3 Ring Formation Dynamics Analysis Approach

To apply the Refined Matching Algorithm to the ring system illustrated in Figure 5.4, we constructed a story for formation of the trimer $ABC$ using the story generation tools implemented in KaSim [KaSim]. Three symmetric stories are possible; for analysis, without loss
Fig. 5.4 Ring Formation Kappa System

(a) Kappa Program for Ring Formation with Three Agents

(b) Schematic Depicting Three Rule Types in Kappa System (B-D). Figure from [10].

(c) Kappa Story for Ring Formation

In this program, we have three agents, A, B, and C. In lines 8-10, we have their introduction rules. In Lines 11-13, we have the formation of all possible dimers, and in Lines 17-19, we have the dissociation of these dimers; these dimer rules correspond to panel B in the schematic below. In Lines 14-16, we have the formation of trimers, as in the schematic. The choice of parameters in this system will be explained in Section 5.2.4.
of generality, we choose the story depicted in Figure 5.4c, in which $B$ and $C$ first bind to produce dimer $BC$, and $A$ then binds to form $ABC$.

We then ran simulations of the system in various conditions, and attempted to match the story of Figure 5.4c to the resulting traces using the Refined Matching Algorithm. To study the effect of changes to the dissociation rate in the system, we explored various rate assignments in the Kappa system of Figure 5.4a through different simulation conditions. For each condition, we ran a simulation for 1 million trace steps. For every trace event in these simulations that applied the rule $A+BC@B$, we considered the resulting trace up to that point, and attempted to match the story in Figure 5.4c to this portion of the trace. While all such story-trace matching attempts would necessarily yield that the story did indeed match the trace, each would produce a distinct embedding map. We recorded the story-trace embedding for each event of interest, and we would use these embeddings to calculate trace delays to answer questions about the dynamics of this system.

When considering trace delays from these story-trace embeddings, we normalized in the following way. Say we computed the trace delay between nodes $s_1$ and $s_2$. For each story-trace embedding, we considered the position of the embedding of the final story node $A+BC@B$ as the length of the full story. We then computed the percentage of that length spent between $s_1$ and $s_2$ by computing the trace delay of $s_1$ and $s_2$, and then computing the percent of the total story length that this value corresponded to. Normalizing to the story length allowed for easier comparison between traces of highly varying lengths through the 1 million step length simulation. We will call this percentage the normalized trace delay between $s_1$ and $s_2$. 
5.2 Characterizing System Dynamics in Ring Formation

5.2.4 Ring Formation Dynamics Analysis Results

To determine the impact of dissociation rates on the dynamics of the three-member ring formation, we recorded embedding maps as discussed above across three conditions, with parameterizations for the Kappa program in Figure 5.4a detailed in Table 5.1. We calculated two normalized trace delays across the resulting 1 million step simulations: those between A/ and A+BC@B, and those between B+C and A+BC@B. Figure 5.5 depicts histograms of normalized trace delays across the three conditions in Table 5.1. In these histograms, we randomly sample 100 story-trace embeddings constructed per condition, allowing us to more easily compare histograms across conditions.

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Low A/, Low koff</th>
<th>Low A/, High koff</th>
<th>Low A/, Low B+C koff</th>
</tr>
</thead>
<tbody>
<tr>
<td>kon</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Akon</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>BCkoff</td>
<td>0.1</td>
<td>100</td>
<td>0.1</td>
</tr>
<tr>
<td>koff</td>
<td>0.1</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 5.1 Rate Parameters across Conditions for Kappa Ring Formation System

Note that in condition Low A/, Low koff, when we have low equal dissociation rates for all dimers and a low introduction rate for A, the trace delay between A/ and A+BC@B (Figure 5.5a) is consistently low compared to the trace delay between B+C and A+BC@B (Figure 5.5b). This implies that when the dimer BC is formed, the story must pause before encountering a free monomer A.

We note that if we then increase the dimer dissociation rate of AB and AC, leaving the dissociation rate of BC constant, the trace delay between A and A+BC@B (Figure 5.5c) is no longer consistently lower than the trace delay B+C and A+BC@B (Figure 5.5d). This supports the hypothesis from Deeds et al. [10] that in condition Low A/, Low koff, the stability of AB
Fig. 5.5 Ring Formation Dynamics with Various Intermediate Binding Affinities

We depict the time spent in the story between either A/ or B+C and the final story event, as it varies with conditions which are defined in Section 5.2.3.

(a) Rule A/, Condition Low A/, Low koff
(b) Rule B+C, Condition Low A/, Low koff
(c) Rule A/, Condition Low A/, Low BC koff
(d) Rule B+C, Condition Low A/, Low BC koff
(e) Rule A/, Condition Low A/, High koff
(f) Rule B+C, Condition Low A/, High koff
and AC takes free monomers A from the system, limiting the formation of the trimer ABC; when AB and AC dissociate more frequently, this hurdle is not posed to trimer formation.

In Figure 5.5f, we see that a high dissociation rate for the dimer BC causes the trace delay from B+C to A+BC@B to be low; in fact, this trace delay becomes as low as the trace delay from A to A+BC@B. In this case, we see that there must then be a long delay from the initial monomers B and C to the formation of the BC before the trimer is created. This again supports the hypothesis from Deeds et al. [10] that when dissociation rates are high, the dimer intermediates are not stable enough to lead to efficient trimer production.

Thus, in the case of three-member ring formation, we see the use of story-trace matching for analyzing system dynamics. The story-trace matching algorithm leaves room for the development of additional metrics on system dynamics. Indeed, the ability to find the causal embedding of a story into a trace provides an opportunity to understand bottleneck points and other dynamical properties of biological systems.

5.3 Discussion of Story-Trace Matching Applicability

The story-trace matching approach has been explored above in two separate settings: when trying to determine which of a set of stories is most prevalent through the course of a trace, and when trying to answer questions about the dynamics of a system. A few properties of story-trace matching outlined below allowed this approach to be uniquely useful in tackling these challenges, compared to currently available story analysis tools which are able to generate stories from traces [KaSim].

- Story-trace matching involves identifying an embedding of a story to a trace. In the case of the Refined Matching Algorithm, this embedding is especially useful. In this
case, the trace events chosen in the embedding are causally relevant to the story’s progression, producing effects that are not undone or irrelevant to future events. This adds to the accuracy of statistics collected in the applications above. When a story matches a trace with this algorithm, it is a reasonable (not undone) match for the trace, leading to more accurate statistics when understanding the prevalence of one story over others in a simulation system. Furthermore, as discussed above, the embedding map picks out trace events which are most causally relevant to future elements of the story, lending more consistency to the measurements of trace delays and resulting insights on system dynamics.

- Especially in the case of long simulation traces, it is possible that story-trace matching is the most efficient available method for determining whether a particular story is relevant to a simulation, as compared to producing stories from traces. In the case of the apoptosis system in Section 5.1 for instance, this difference became clear when understanding the production of cPARP. The first instance of cPARP in the system consistently occurred after 2400000 trace steps. Over five trials, generating a story for the trace ending at this instance of cPARP took an average of 13.81 s, whereas matching the complete story described in Section 5.1.3 to the trace took on average 1.87 s. When we know the story we are looking for, it is thus likely more efficient to match a story to traces rather than produce these stories from scratch. The performance difference between these approaches likely varies greatly between systems depending on the complexity of the story involved; further analysis of these differences would be useful.

- In Section 5.1, story-trace matching allowed for understanding the prevalence of not a fully specified story, but rather a subset of a story that we were interested in characterizing. In this case, using a story generation approach would be additionally time-consuming; without additions to available story-generation tools, we would need
5.3 Discussion of Story-Trace Matching Applicability

to first produce full-length stories from traces and then post-process these stories to determine if they match the desired pattern. With story-trace matching, we can match the desired pattern directly.

• As discussed in Section 4.1.1, in the case when multiple stories can match a trace, the story generation approach will produce one such story from the trace; on the other hand, story-trace matching has the potential to match all available stories to the trace. This can then lead to more reasonable statistics when pursuing applications such as those in Section 5.1.

In this chapter, we have discussed some applications for the story-trace matching algorithms described in Section 4. While story-trace matching has been applied to known biological phenomena in this thesis with the hopes of confirming its usefulness, it promises to be useful for similar analyses on less characterized systems, potentially generating novel biological insight. Next, we will pose additional potential applications and extensions for story-trace matching.
Chapter 6

Future Work

The story-trace matching algorithms of Section 4 can be useful in various applications for understanding properties of biological systems, beyond those explored in Section 5. In this chapter, we outline some of these possible extensions, and we describe how it might be useful to further extend story-trace matching.

6.1 Future Applications of Story-Trace Matching

6.1.1 Assessing Stories Generated without Simulation

Techniques have been developed to generate the set of all possible stories that might arise from a simulation system for an event of interest [21]. These techniques proceed by analyzing the rule set of a Kappa program directly, rather than by running simulations from this program and analyzing the resulting simulations. In producing stories, these techniques do not take into account the rates associated with rules in the system. Thus, some of the stories produced may be very unlikely to appear in simulations.

The story-trace matching approach can be useful for understanding the prevalence of these
stories. However, these techniques produce stories using compression techniques that more drastically reduce story size compared to weak compression. Thus, while the algorithms discussed in this thesis may provide a starting point for this application, they will need to be extended before assessing the prevalence of these stories.

6.1.2 Efficient Story Generation

It is possible that story-trace matching can be used to more efficiently generate stories from Kappa simulations. In particular, consider the scenario in which a long Kappa simulation is run with many instances of the event of interest. For initial instances of this event, the trace may be relatively short, such that story generation can proceed efficiently. For later instances of this event, however, story generation via the tools implemented in KaSim may prove too inefficient. In this case, we can use stories generated from initial events of interest in the trace as a guess for the story for future events of interest; instead of generating stories for these future events of interest, we can simply try matching from our assembled set of guesses. In this way, story-trace matching can be useful when the trace grows long for finding the story corresponding to a trace.

It is also possible that in this setting, the Direct Matching Algorithm can be useful; by matching stories generated from earlier portions of the trace to the current trace with particular agent identifiers preserved, the story-trace matching approach to story generation may become even more efficient. More work on understanding the relative efficiency of these techniques will be needed before understanding if story-trace matching is useful for generating stories efficiently, and whether the Direct Matching Algorithm or Refined Matching Algorithm is more useful.
6.1.3 Parameter Sensitivity

Since rates associated with chemical reactions are often difficult to determine precisely, it is of interest to understand how the story prevalence changes when rates are modified slightly. This can help modelers understand the sensitivity of their system to small modifications, which likely reflect differences between Kappa models and the real biological systems being simulated. To understand this sensitivity, one could run a simulation at a particular choice of parameters and generate stories from the resulting traces. Then after changing parameters slightly, story-trace matching could be used to determine if the original stories continue to match new simulations.

6.2 Further Improving Story-Trace Matching

6.2.1 Exploring Multiple Story to Trace Matching

In applications of story-trace matching such as the story prevalence application, or in the applications discussed in Section 6.1.1 - 6.1.2, it can become necessary to match a large set of stories to a trace. To improve story-trace matching, it would be interesting to explore algorithms which simultaneously attempt to match multiple stories to a trace, rather than making these matches successively. Indeed, if these stories overlap, it may be possible to make use of these overlaps to generate more efficient story-trace matching algorithms. As one example, if many stories have the same suffix, a shared embedding for this suffix to the trace can be used to more efficiently find a story-trace embedding for all of these stories.

6.2.2 Story Pattern to Trace Matching

As seen in Section 5.1, it is useful to match a portion of a story to a trace, rather than the full story. To phrase this question precisely, consider the case in which we have a set of pairs $S$
of story nodes of the form \((s_i, s'_i)\). We might which to determine if a story \(s\) exists such that \(S \subseteq \text{pairs}(s)\) and \(s\) refined matches the trace \(t\).

We have implemented a solution for one particular version of this problem in our implementation of the Refined Matching Algorithm. In particular, our implementation handles the case in which \(s\) is constrained to have \(S\) as a suffix, such that we are guaranteed that there is no \((s_1, s_2) \in \text{pairs}(s)\) such that \((s_1, s_2) \notin S\) but \((s_1, k) \in S\) or \((k, s_1) \in S\) for some \(k\). It is far more challenging to solve this problem in general, when \(s\) is not constrained.

### 6.2.3 Additional Forms of Compression

An important future extension to story-trace matching will be to extend this approach to other forms of story compression beyond weak compression. While weak compression does eliminate some redundancy from stories, even the stories produced from weak compression can grow numerous and unwieldy, as suggested by the discussion of the apoptosis stories found in Section 5.1. Other forms of compression have been proposed to eliminate some of this complexity, including strong compression [21, 7]. Once the questions that these forms of compression can address have been identified precisely, it will be useful to understand whether story-trace matching can apply in these settings.
Chapter 7

Conclusion

The development of Kappa provides biologists with the unique opportunity to better understand the networks of interactions underlying cellular phenomena. When taken alone, Kappa simulations cannot answer some of the interesting biological questions we might want to answer, as the traces produced can be unwieldy and difficult to analyze. With the development of the story formalism, we can better understand the underlying mechanism behind Kappa simulations; indeed, Kappa users can generate pathway diagrams for the simulations they run to view possible causal explanations for those simulations. Even with this story formalism, however, tools to programmatically evaluate these stories to glean biological insight from them were limited.

In this thesis, we explored techniques to determine whether a story matches a trace, thus developing a programmatic method to process stories and understand their relevance to Kappa systems. We explored applications of story-trace matching in the context of two biological systems: cellular death and ring formation. In these settings, we found we were able to support findings from the systems biology literature. Moreover, we found our algorithms to be particularly useful for their efficiency and accuracy in tackling these challenges. While this analysis is only the first step of applying story-trace matching, it suggests that this approach
can be promising for analyzing additional biological systems.

Though matching stories to traces is a useful primitive for programmatically processing stories, it is certainly not the only way to gain insight from stories. For instance, recent work has already explored generating stories from the Kappa rule set rather than from simulations [21]. As another example, it might be interesting to develop methods to divide stories into hierarchical classes based on shared blocks of nodes; strategies like this might help create more manageable classes of stories from a currently often unmanageable number of weakly compressed stories. Like story-trace matching, these potential approaches to analyzing stories would strive to make the analysis of biological systems more feasible and powerful using the story formalism. While viewing the story for a particular simulation can be useful on a smaller scale, these analytical frameworks on top of stories likely have greater power to analyze systems when a large number of simulations would be useful.

Ultimately, investigating the story-trace matching algorithm unveiled challenges with the story formalism which leave room for future exploration. In particular, in finding undone story-trace embeddings, we revealed that not all stories produced from traces seem to be reasonable representations of the underlying causal mechanism of the trace. As we move forwards to additional tools and analyses that make use of stories, it will be important to also step back and understand the precise failures and successes of the story model. Defining the current story formulation has already led to the development of useful analytical tools; even more precisely capturing a biologist’s notion of a pathway will pave the way for future contributions.
References


Appendix A

Story-Trace Embedding Undoneness Properties

Here we present proofs of the theorems on story-trace embedding undoneness presented in Section 4.4.3. Together, these properties will demonstrate that when some basic properties hold of the story in consideration, there will be effectively one unique story that matches a trace. However, when these properties do not hold, multiple distinct stories may match a trace.

In Section A.1, we will demonstrate that there must be at least one story that matches a trace such that the embedding map from the story to the trace is not undone, as defined in Section 4.4. Next, in Section A.2, we will state conditions in which only one minimal compressed story can match a trace, and we will prove that distinct stories cannot match a trace in this setting. Finally, in Section A.3, we will demonstrate that when these conditions do not hold, more than one story may match a trace.

For simplicity, in this appendix we will assume that the initial starting state of the system before the trace’s first event is empty. Thus, any agents used in the system are assumed to enter the system via the application of introduction rules, as depicted in figures in Section
3 - 4, rather than as initial state in the system. We will refer to this assumption as the empty starting state assumption in the proofs below. This assumption will allow us to state the theorems and proofs below more cleanly; slightly modified versions of the proofs discussed will still hold when a more general starting condition is considered.

### A.1 At Least One Reasonable Story-Trace Match

In this section, we prove the following theorem to demonstrate that there must be at least one reasonable story for a trace $t$, using the definitions of undoneness from Section 4.4.

**Theorem 4.** Consider a trace $t$. There exists at least one story $s$ such that the story-trace embedding $M$ from $s$ to $t$ is not undone.

**Proof of Theorem 4.** We will provide an algorithm for finding such a story $s$ given a trace $t$, constructing the set of precedence pairs $\text{pairs}(s)$. Pseudocode for the algorithm can be found in Algorithm 4, and a brief text overview is below which references lines from this pseudocode.

**Algorithm Description**

The algorithm will begin with the final event in $t$, traversing the trace backwards (Line 5) as it assembles the $\text{pairs}(s)$ in reverse topological order. The story we create will have as its event of interest the final event of the trace (Line 2). As we traverse the trace, we will choose some trace events as causally relevant to those chosen for the story so far; when we find such an event that we need to represent in our story, we will create new story precedence pairs, which we will add to the set $B$ defined below.

**Data Structure 6. Precedence Pairs $B$:** In the algorithm, $B$ will be a set of pairs of story nodes, which will become $\text{pairs}(s)$ for the desired story $s$ by the end of the algorithm.
Algorithm 4 Finding Story that is Not Undone

1: function FIND_STORY(t)
2:     create new \( s_0 \) with \( \text{event}(s_0) = t_{[t]} \)
3:     \( B := \{ \} \)
4:     \( A := \{ (s_0,(x,v)) : (x,v) \in \text{pre}(t_{[t]}) \} \)
5:     for \( i \) in \( |t| \) to 1 do
6:         if \( A = \{ \} \) then
7:             return \( B \)
8:         end if
9:         \( A' := A \)
10:        added := false
11:        create new \( s_j \) with \( \text{event}(s_j) = t_i \)
12:        for \( (s_k,(x,v)) \) in \( A \) do
13:            if \( (x,v) \in \text{eff}(t_i) \) then
14:                \( A' := A' \setminus (s_k,(x,v)) \)
15:                if \( (s_j,s_k) \notin B \) then
16:                    \( B := B \cup (s_j,s_k) \)
17:                    added := true
18:                end if
19:            end if
20:        end for
21:        if added then
22:            \( A' := A' \cup \{ (s_j,a) \text{ for all } a \in \text{pre}(t_{[t]}) \} \)
23:        end if
24:        \( A := A' \)
25:    end for
26:    return \( B \)
27: end function
To decide on which events of the trace to include in the story we are building, we will maintain a list of assignments; these assignments are ones that the story nodes chosen so far require of the system as preconditions. We will call this set of assignments $A$.

**Data Structure 7. Outstanding Assignments** $A$: *A will describe the set of assignments which story nodes chosen so far depend on. Each element of $A$ will be a tuple of a story node and an assignment, such that $(s_k, (x,v)) \in A$ will have $(x,v) \in \text{pre}(\text{event}(s_k))$.*

The algorithm will manipulate $A$ as follows. When we add a new story node to a pair in $B$, we will add all of the assignments in its precondition to the set $A$ (Line 4, Line 22). These are elements of the system’s state that must be set up by other events in order for this story node to run. Whenever we find a new trace event that creates an assignment in $A$ (Line 13), we will try to include it in our list of precedence pairs, as it creates an effect that is useful to the story assembled so far (Line 16). We also remove the assignments this trace event handles from $A$, as we no longer have to take care of these assignments with a different trace event (Line 14). Once we have no more elements of $A$ to assign, we are finished (Line 7). Since the trace $t$ is valid, we know that all assignments must either have been created by some event, or must be present in the initial state of the system. With the empty starting state assumption, we then know that upon traversing the trace we must have $A$ empty. Thus, we are guaranteed to return $B$ at the end of the algorithm, knowing that it is a complete story for the trace $t$.

**Algorithm Correctness**

Note that in this algorithm, it is not possible for a pair in $B$ to be an invalid causal pair. This is the case because for every causal pair $(s_k, s'_k)$, for some assignment $(x,v)$ in $\text{eff}(\text{event}(s'_k))$, $\text{event}(s_k)$ is the first trace event that that creates this assignment, with $(x,v) \in \text{pre}(\text{event}(s_k))$. Thus, it is not possible for an intervening trace event to undo this assignment. With no invalid
A.2 At Most One Reasonable Story-Trace Match

Causal pairs in $B$, the story-trace embedding constructed is not undone.

Note, the story $s$ constructed by Algorithm 4 is not necessarily minimally compressed. If we desire a minimal compressed story $s^*$ that satisfies the property that the embedding from $s^*$ to $t$ is not undone, we can simply compress $s$ found via the algorithm above to find $s^*$.

Thus, it is always possible to construct a minimally compressed story $s$ for trace $t$ such that there is an embedding from $s$ to $t$ that is not undone, completing the proof of the theorem.

\[\square\]

A.2 At Most One Reasonable Story-Trace Match

In this section, we prove the Theorem 5, demonstrating cases in which using the ideas of story-trace embedding undoneness, there is effectively a single story that matches a trace.

**Theorem 5.** Say we have two minimally compressed stories $s$ and $s'$ that are weak matches for trace $t$, such that the embeddings $M$ and $M'$ from $s$ and $s'$ respectively to $t$ are not undone. Furthermore, using the properties below, say that $M$ and $M'$ are complete, or that $s$ and $s'$ are single effect stories. We must then have that $s' = I_\phi(s)$ for some agent isomorphism $\phi$ (defined in Section 3.4.6). Thus, aside from agent isomorphisms, $s$ becomes the unique story that matches the trace $t$.

**Property.** Say we have a story-trace embedding $M$ from story $s$ to trace $t$. We will say that the embedding $M$ is complete if the following property holds. For every assignment $(x, v) \in \text{pre}(M(s_j))$ for any $s_j \in \text{nodes}(s)$, there is some node $s_k$ such that $(x, v) \in \text{eff}(M(s_k))$ and $(x, v)$ is not undone between $M(s_k)$ and $M(s_j)$, as defined in Section 4.4.2. Intuitively, for $M$ to be complete, every assignment in a precondition of a story node’s event must be
handled properly; the embedding must choose some other event from the trace that produces this assignment without it being undone.

**Property.** For every \( s_k \in \text{nodes}(s) \), \( \text{event}(s_k) \) has a single effect, such that \( \text{eff}(\text{event}(s_k)) \) has one element. If this property holds, we will say that \( s \) is a **single effect** story.

**Proof of Theorem 5.**

We first show that the second property above implies the first; we will show that \( s \) being a single effect story implies that any story-trace embedding \( M \) that is not undone is complete. This will allow us to focus on using the property that \( M \) is complete through the rest of the proof.

**Claim 2.** If \( s \) is a single effect story and the story-trace embedding \( M \) from \( s \) to \( t \) is not undone, then \( M \) is complete.

**Proof of Claim 2.** To show that \( M \) is complete, we must show that for every \( s_j \in \text{nodes}(s) \), each of its assignments \( (x,v) \in \text{pre}(M(s_j)) \) has the property that there is some node \( s_k \) such that \( (x,v) \in \text{eff}(M(s_k)) \), and \( (x,v) \) is not an undone assignment between \( M(s_k) \) and \( M(s_j) \) (defined in Definition 51).

Consider some \( s_j \in \text{nodes}(s) \). Note that due to the empty starting state assumption, every assignment \( (x,v) \in \text{pre}(M(s_j)) \) must have the property that \( (x,v) \in \text{eff}(M(s_k)) \) for some \( s_k \in \text{nodes}(s) \). If this were not the case, then the precondition for \( M(s_j) \) to run would not be satisfied.

However, because the story \( s \) is a single effect story, we have that \( (x,v) \) is the only assignment in \( \text{eff}(M(s_k)) \). Say that \( (x,v) \) were to be undone between \( s_k \) and \( s_j \). Then, all assignments in \( \text{assignment_list}(s_k, s_j) \) are undone between \( M(s_k) \) and \( M(s_j) \). As defined in Definition 52, we then have that \( (s_k, s_j) \) is an invalid causal pair, making \( M \) an
undone story embedding. Since this is not the case, \((x,v)\) cannot be undone between \(s_k\) and \(s_j\), and we have found the node \(s_k\) necessary to demonstrate that \(M\) is complete. 

With the claim above, we can prove the theorem by making the weaker assumption that \(M\) is complete, rather than the assumption that \(s\) is a single-effect story.

Now we continue with our proof of Theorem 5. We will first show the lemma below, which will demonstrate that the embedding maps between each of the stories and the trace must be the same for two stories satisfying the conditions of Theorem 5. The theorem will follow as a corollary of this lemma, as described after the lemma is proven.

**Lemma 1.** We will show that if \(s\) and \(s'\) with embeddings \(M\) and \(M'\) respectively into the trace \(t\) satisfy the conditions of Theorem 5, for every \((s_i,s_j) \in \text{pairs}(s)\) we must have some \((s'_i,s'_j) \in \text{pairs}(s')\) with \(M(s_i) = M'(s'_i)\) and \(M(s_j) = M'(s'_j)\).

**Proof of Lemma 1.** We proceed inductively in the backwards topological order of \(DAG(s)\), which will also become the topological order of \(DAG(s')\) as we prove the lemma.

**Base case:** First, we note that \(M(s_0) = M'(s'_0)\) for \(s_0\) and \(s'_0\) the final nodes in the topological order of \(DAG(s)\) and \(DAG(s')\); this is the case because these final nodes are both mapped to the last event of the trace.

**Inductive hypothesis:** Now, say that by the inductive hypothesis, we have found that for some \(s_j \in \text{nodes}(s)\), for every pair \((s_j,s_k) \in \text{pairs}(s)\) we have a \((s'_j,s'_k) \in \text{pairs}(s')\) with \(M(s_j) = M'(s'_j)\) and \(M(s_k) = M'(s'_k)\).

**Inductive step:** We now hope to show that for every \((s_i,s_j) \in \text{pairs}(s)\), there is a
Let us label the predecessors of $s_j$ as $p_1, \ldots, p_n$, and the predecessors of $s'_j$ as $q_1, \ldots, q_m$. We call $p_i$ a predecessor of $s_j$ if there is some $(p_i, s_j) \in \text{pairs}(s)$. Furthermore, without loss of generality, we assign these labels such that if $i < j$, $M(p_i)$ occurs before $M(p_j)$ in the trace, and $M'(q_i)$ occurs before $M'(q_j)$. We will show that we must have $M(p_i) = M'(q_i)$ for $i \in \{1, \ldots, n\}$. Note that this will imply that for each $(p_i, s_j) \in \text{pairs}(s)$, we have that $(q_i, s'_j) \in \text{pairs}(s')$ satisfies $M(p_i) = M'(q_i)$, as desired. Thus, we try to show the following claim.

**Claim.** Consider some story nodes $s_j \in \text{nodes}(s)$ and $s'_j \in \text{nodes}(s')$ with predecessor sets $\{p_1, \ldots, p_n\}$ and $\{q_1, \ldots, q_m\}$ respectively, such that $M(s_j) = M'(s'_j)$. Say that $M(p_i)$ is before $M(p_j)$ for $i < j$ and $M'(q_i)$ is before $M'(q_j)$ for $i < j$. Then we will have that $M(p_i) = M'(q_i)$ for $i \in \{1, \ldots, n\}$.

**Proof.** We prove this claim via induction on $i$. We will use this induction to both show that $m > n$ and to show the claim above.

**Base case:** Consider $p_1$. Note that $\text{eff}(M(p_1))$ must include some assignment $(x, v)$ in $\text{pre}(M(s_j)) = \text{pre}(M'(s'_j))$. Then, $\text{pre}(M'(s'_j))$ is nonempty, and there must be some $q_1$ in the set of predecessors of $s'_j$.

Now, we hope to show that $M(p_1) = M'(q_1)$. Assume this is not true, and without loss of generality, that $M(p_1)$ appears before $M'(q_1)$.

- First, note that $M(p_1)$ must have some $(x, v) \in \text{eff}(M(p_1))$ such that $(x, v) \in \text{pre}(M(s_j))$; otherwise, $p_1$ should not be included in this minimally compressed
story, as removing it would allow for more compression. Thus, \((p_1, s_j)\) is a causal pair by Definition 49.

- Note that for every assignment \((x, v) \in \text{pre}(M(s_j))\), we must have some predecessor \(s_k\) of \(s_j\) such that \((x, v) \in \text{eff}(M(s_k))\) (by the completeness of \(M\)). Then, every assignment \((x, v) \in \text{eff}(M(p_1)) \cap \text{pre}(M(s_j)) = \text{assignment}_\text{list}((p_1, s_j))\) must be in \(\text{eff}(M'(q_k))\) for some \(q_k\). Since \(M(p_1)\) appears before every \(M'(q_k)\), we must then have that every such assignment \((x, v)\) is undone between \(M(p_1)\) and \(M(s_j)\).

With the above two properties, we have that \((p_1, s_j)\) is an invalid causal pair by Definition 52. Then, the story embedding \(M\) is undone, violating the hypothesis of the theorem. This gives us a contradiction, implying that \(M(p_1) = M'(q_1)\).

**Inductive hypothesis:** \(M(p_r) = q_r\) for all \(r < i\).

**Inductive step:** Consider \(p_i\). We first show that there must be an \(i\)th predecessor of \(s'_j\), \(q_i\). To see this, note that \(M(p_r) = M'(q_r)\) for \(r < i\). Thus, \(\bigcup_{r=1}^{i-1} \text{eff}(M(p_r)) = \bigcup_{r=1}^{i-1} \text{eff}(M'(q_r))\). We know that \(\text{eff}(M(p_i))\) must contain some assignment present in \(\text{pre}(M(s_j)) \setminus \bigcup_{r=1}^{i-1} \text{eff}(M'(q_r))\). If it did not, it would be unnecessary to include \(p_i\) in the minimally compressed story \(s\). Then, we know that the set of uncovered assignments \(\text{pre}(M'(s'_j)) \setminus \bigcup_{r=1}^{i-1} \text{eff}(M'(q_r))\) is non-empty. Then, for \(s'\) to be valid, there must be a predecessor \(q_i\) for \(s'_j\) to cover these remaining assignments.

We now hope to show that \(M(p_i) = M'(q_i)\). Suppose this is not the case. Again without loss of generality, assume that \(M(p_i)\) appears before \(M'(q_i)\) in the trace.
Note that as described above in the base case, \((p_i, s_j)\) must be a causal pair for this story \(s\) to be minimally compressed. Also, note that if every assignment \((x, v) \in \text{assignment} \_\text{list}((p_i, s_j))\) were in \(\text{eff}(M'(q_i))\) for some \(r \in \{i, ..., m\}\), we would have that \((p_i, s_j)\) is an invalid causal pair, again by the reasoning in the base case. Therefore, for \(M(p_i) \neq M'(q_i)\), we must have that there is some assignment \((x, v) \in \text{assignment} \_\text{list}((p_i, s_j))\) not in \(\text{eff}(M'(q_i))\) for any \(r \in \{i, ..., m\}\). Also, as described in the base case, every \((x, v) \in \text{assignment} \_\text{list}((p_i, s_j))\) must be in \(\text{eff}(M'(q_s))\) for some \(s \in \{1, ..., m\}\).

Therefore, we must have that this \((x, v)\) satisfies \((x, v) \in \text{eff}(M'(q_s))\) for some \(s \in \{1, ..., i - 1\}\). This is where we use the property that \(M\) is complete, noting that in particular, for some \(s \in \{1, ..., i - 1\}\) we must have \((x, v) \in \text{eff}(M'(q_s))\) such that \((x, v)\) is not undone between \(M'(q_s)\) and \(M'(s'_j)\).

Now we use the inductive hypothesis. For all \(s \in \{1, ..., i - 1\}\), by induction we have that \(M'(q_s) = M(p_s)\). Therefore, \(M'(q_s) = M(p_s)\) is a trace event that occurs before \(M(p_i)\) (recall that we had ordered the predecessors such that if \(i < j\) then \(M(p_i) < M(p_j)\)). Furthermore, \((x, v)\) is in \(\text{eff}(M(p_s))\) and in \(\text{eff}(M(p_i))\). Therefore, \((x, v)\) is undone between \(M(p_s)\) and \(M(p_i)\), and thus is undone between \(M'(q_s) = M(p_s)\) and \(M'(s'_j)\). This then contradicts the property that \(M\) is complete. Thus, we cannot have \(M(p_i) \neq M'(q_i)\), completing the induction. 

Thus, we have shown that once we are at \(s_j\) in our induction on the topological order of \(\text{DAG}(s)\), with \(M(s_j) = M'(s'_j)\), we can demonstrate that for any \((p_i, s_j) \in \text{pairs}(s)\), we have \(M(p_i) = M'(q_i)\) for some pair \((q_i, s'_j) \in \text{pairs}(s')\). This completes the inductive step, and proves the lemma.
A.3 Settings with More than One Story-Trace Match

Thus, we have proven that for every \((s_i, s_j) \in \text{pairs}(s)\), we have some \((s'_i, s'_j) \in \text{pairs}(s')\) with \(M(s_i) = M'(s'_i)\) and \(M(s_j) = M'(s'_j)\). Note that we can apply this lemma symmetrically from \(s'\) to \(s\) as well, such that we know the following property: a pair \((s_i, s_j)\) is in \(\text{pairs}(s)\) if and only if there is a pair \((s'_i, s'_j)\) with \(M(s_i) = M'(s'_i)\) and \(M(s_j) = M'(s'_j)\).

Now we recall that by the definition of weak compression, we have some map \(\phi\) for \(s\) and \(\phi'\) for \(s'\) such that if \(\text{event}(s_i) = \langle r, \psi \rangle\), then \(M(s_i) = \langle r, \phi \circ \psi \rangle\), and if \(\text{event}(s'_i) = \langle r', \psi' \rangle\), then \(M'(s'_i) = \langle r', \phi' \circ \psi' \rangle\). Note that by the above, when \(M(s_i) = M'(s'_i)\), we will have that \(r' = r\).

We can then see that \(s = I_{\phi^{-1} \circ \phi'}(s')\); this transformation first takes \(s'\) to the domain of the trace’s events, and then takes the resulting story to the domain of the story \(s\). Thus, we have demonstrated that \(s\) and \(s'\) only differ by an isomorphism of agent identifiers, completing the proof of Theorem 5.

Together with the proof in Section A.1, we have thus shown that it is possible to find a story \(s\) matching the trace such that the resulting embedding map \(M\) is not undone, and that when a story \(s\) has a single effect or when the \(M\) is complete, any other story \(s'\) satisfying this property will differ from \(s\) only by an agent isomorphism. As described in Section 4.5.3, this property has implications for the efficiency of story-trace matching when matching a set of stories to the trace.

A.3 Settings with More than One Story-Trace Match

One might wonder if the property that the story \(s\) has a single effect or that \(M\) is complete is necessary for proving Theorem 5. In Figure A.1, we provide an example of a trace which two stories match to, such that neither story embeds into the trace with an undone embedding.

This system violates the property that \(s\) is single effect and that \(M\) is complete. Note that
Here we depict a toy Kappa system which demonstrates the possibility of having two stories matching a trace, with neither embedding undone. In (a), we have the Kappa program for this system. In Panel 1 of (b), we show the changes in global state, with each column corresponding to the state after the event below is applied. In Panel 2 of (b), we show the trace, eliding the introduction events. In Panel 3 of (b), we show two stories for this trace; neither embedding into the trace is undone, and it is unclear which story is preferable, if any.

(a) Kappa program including rules with multiple effects

%agent: A(x~u~p,y~u~p,z~u~p,b,c)
%agent: B(a)
%agent: C(a)

'A/' -> A(x~u,y~u,z~u,b,c) @ 1.0
'B/' -> B(a) @ 1.0
'C/' -> C(a) @ 1.0
'r1' A(x~u) -> A(x~p) @ 1.0
'r2' A(y~u,z~u) -> A(y~p,z~p) @ 1.0
'r2*' A(y~p) -> A(y~u) @ 1.0
'r3' A(y~u,b),B(a) -> A(y~p,b!1),B(a!1) @ 1.0
'r4' A(b),B(a) -> A(b!1),B(a!1) @ 1.0
'r4*' A(b!1),B(a!1) -> A(b),B(a) @ 1.0
'r5' A(c),C(a) -> A(c!1),C(a!1) @ 1.0

(b) Sample Trace, Global State, and Two Stories

1. \begin{align*}
(C,1)_a & \quad (C,1)_a & \quad (C,1)_a & \quad (C,1)_a & \quad (C,1)_a & \quad (C,1)_a & \quad (C,1)_a \\
\cdot_{(B,1)_a} & \cdot_{(B,1)_a} & \cdot_{(B,1)_a} & \cdot_{(B,1)_a} & \cdot_{(B,1)_a} & \cdot_{(B,1)_a} & \cdot_{(B,1)_a}
\end{align*}

2. \begin{align*}
\langle r1, \psi_{r1} \rangle & \quad \langle r2, \psi_{r2} \rangle & \quad \langle r2*, \psi_{r2*} \rangle & \quad \langle r3, \psi_{r3} \rangle & \quad \langle r4*, \psi_{r4*} \rangle & \quad \langle r4, \psi_{r4} \rangle & \quad \langle r5, \psi_{r5} \rangle
\end{align*}

3. 

Story 1: \langle r1, \psi_{r1} \rangle \rightarrow \langle r2, \psi_{r2} \rangle \rightarrow \langle r4, \psi_{r4} \rangle \rightarrow \langle r5, \psi_{r5} \rangle

Story 2: \langle r1, \psi_{r1} \rangle \rightarrow \langle r2, \psi_{r2} \rangle \rightarrow \langle r2*, \psi_{r2*} \rangle \rightarrow \langle r3, \psi_{r3} \rangle \rightarrow \langle r5, \psi_{r5} \rangle
rules like \( r3 \) in this Kappasystem produce multiple effects, and are unlikely to appear as such in Kappa programs. We also note that the embeddings \( M \) of these stories into the trace are not complete. For instance, when we embed Story 1 into the trace, we see that the phosphorylation of \( y \) is undone by rule \( r2* \); yet, the rule \( r2 \) is the only one in this story that produces this phosphorylation. Therefore, this assignment is not captured by some earlier story event such that it is not undone, and \( M \) is not complete in this scenario.

With these properties violated, it is possible for multiple stories to match this trace, and that indeed is the case in this example. Neither Story 1 or Story 2 have embeddings that are undone.

We present this system to serve as an illustrative example which might inspire further progress on story-trace matching. It would be interesting in the future to determine if either of the stories present in this system are preferable. If one of these stories is preferable, it may be useful to develop additional conditions like the story undoneness condition that allow us to further filter unreasonable story-trace matches.
Appendix B

Apoptosis System

B.1 Kappa System for Apoptosis

Below is the Kappa system we use for simulations of apoptosis, compiled from the PySB system described in [23].

```
%agent: L(b)
%agent: pR(b)
%agent: DISC(b)
%agent: flip(b)
%agent: pC8(b)
%agent: C8(b)
%agent: BAR(b)
%agent: pC3(b)
%agent: C3(b)
%agent: pC6(b)
%agent: C6(b)
%agent: XIAP(b)
%agent: C3_U(b)
%agent: PARP(b)
%agent: CPARP(b)
%agent: Bid(b)
%agent: tBid(b)
%agent: Mcl1(b)
%agent: Bax(b)
%agent: aBax(b)
%agent: MBax(b)
```
Apoptosis System

%agent: Bcl2(b)
%agent: Bax2(b)
%agent: Bax4(b)
%agent: Mito(b)
%agent: AMito(b)
%agent: mCytoC(b)
%agent: ACytoC(b)
%agent: mSmac(b)
%agent: ASmac(b)
%agent: cCytoC(b)
%agent: Apaf(b)
%agent: aApaf(b)
%agent: pC9(b)
%agent: Apop(b)
%agent: cSmac(b)

%var: 'L_0' 3.000000e+03
%var: 'pR_0' 1.000000e+03
%var: 'flip_0' 2.000000e+03
%var: 'pC8_0' 1.000000e+04
%var: 'BAR_0' 1.000000e+03
%var: 'pC3_0' 1.000000e+04
%var: 'pC6_0' 1.000000e+04
%var: 'XIAP_0' 1.000000e+05
%var: 'PARP_0' 1.000000e+06
%var: 'Bid_0' 6.000000e+04
%var: 'Mcl1_0' 2.000000e+04
%var: 'Bax_0' 8.000000e+04
%var: 'Bcl12_0' 3.000000e+04
%var: 'Mito_0' 5.000000e+05
%var: 'mCytoC_0' 5.000000e+05
%var: 'mSmac_0' 1.000000e+05
%var: 'pC9_0' 1.000000e+05
%var: 'Apaf_0' 1.000000e+05
%var: 'kf1' 4.000000e-07
%var: 'kr1' 1.000000e+06
%var: 'kc1' 1.000000e+02
%var: 'kf2' 1.000000e+06
%var: 'kr2' 1.000000e+03
%var: 'kf3' 1.000000e+07
%var: 'kr3' 1.000000e+03
%var: 'kc3' 1.000000e+00
%var: 'kf4' 1.000000e-06
%var: 'kr4' 1.000000e-03
%var: 'kf5' 1.000000e-07
%var: 'kr5' 1.000000e-03
%var: 'kc5' 1.000000e+00
%var: 'kf6' 1.000000e-07
%var: 'kr6' 1.000000e-03
%var: 'kc6' 1.000000e+00
%var: 'kf7' 1.000000e-07
%var: 'kr7' 1.000000e-03
%var: 'kc7' 1.000000e+00
%var: 'kf8' 2.000000e-06
%var: 'kr8' 1.000000e-03
%var: 'kc8' 1.000000e-01
%var: 'kf9' 1.000000e-06
%var: 'kr9' 1.000000e-03
%var: 'kc9' 2.000000e+01
%var: 'kf10' 1.000000e-07
%var: 'kr10' 1.000000e-03
%var: 'kc10' 1.000000e+00
%var: 'kf11' 1.000000e-06
%var: 'kr11' 1.000000e-03
%var: 'kf12' 1.000000e-07
%var: 'kr12' 1.000000e-03
%var: 'kc12' 1.000000e+00
%var: 'kf13' 1.000000e-02
%var: 'kr13' 1.000000e+00
%var: 'kf14' 1.000000e-04
%var: 'kr14' 1.000000e-03
%var: 'kf15' 2.000000e-04
%var: 'kr15' 1.000000e-03
%var: 'kf16' 1.000000e-04
%var: 'kr16' 1.000000e-03
%var: 'kf17' 2.000000e-04
%var: 'kr17' 1.000000e-03
%var: 'kf18' 1.000000e-04
%var: 'kr18' 1.000000e-03
%var: 'kf19' 1.000000e-04
%var: 'kr19' 1.000000e-03
%var: 'kc19' 1.000000e+00
%var: 'kf20' 2.000000e-04
%var: 'kr20' 1.000000e-03
%var: 'kc20' 1.000000e+01
%var: 'kf21' 2.000000e-04
%var: 'kr21' 1.000000e-03
%var: 'kc21' 1.000000e+01
%var: 'kf22' 1.000000e+00
%var: 'kr22' 1.000000e-02
%var: 'kf23' 5.000000e-07
%var: 'kr23' 1.000000e-03
%var: 'kc23' 1.000000e+00
%var: 'kf24' 5.000000e-08
%var: 'kr24' 1.000000e-03
%var: 'kf25' 5.000000e-09
%var: 'kr25' 1.000000e-03
%var: 'kc25' 1.000000e+00
%var: 'kf26' 1.000000e+00
%var: 'kr26' 1.000000e-02
%var: 'kf27' 2.000000e-06
%var: 'kr27' 1.000000e-03
%var: 'kf28' 7.000000e-06
%var: 'kr28' 1.000000e-03
%var: 'kf31' 1.000000e-03
%var: 'kdeg_Mcl1' 1.000000e-04
%var: 'kdeg_AMito' 1.000000e-04
%var: 'kdeg_C3_U' 0.000000e+00
%var: 'ks_L' 0.000000e+00
%var: 'kdeg_L' 2.900000e-06
%var: 'kdeg_pR' 2.900000e-06
%var: 'ks_pR' 4.350000e-04
%var: 'kdeg_flip' 2.900000e-06
%var: 'ks_flip' 8.700000e-04
%var: 'kdeg_pC8' 2.900000e-06
%var: 'ks_pC8' 4.350000e-03
%var: 'kdeg_BAR' 2.900000e-06
%var: 'ks_BAR' 4.350000e-04
%var: 'kdeg_pC3' 2.900000e-06
%var: 'ks_pC3' 4.350000e-03
%var: 'kdeg_pC6' 2.900000e-06
%var: 'ks_pC6' 4.350000e-03
%var: 'kdeg_XIAP' 2.900000e-06
%var: 'ks_XIAP' 4.350000e-02
%var: 'kdeg_PARP' 2.900000e-06
%var: 'ks_PARP' 4.350000e-01
%var: 'kdeg_Bid' 2.900000e-06
%var: 'ks_Bid' 2.610000e-02
%var: 'ks_Mcl1' 3.000000e-01
B.1 Kappa System for Apoptosis

%var: 'kdeg_Bax' 2.900000e-06
%var: 'ks_Bax' 3.480000e-02
%var: 'kdeg_Bcl2' 2.900000e-06
%var: 'ks_Bcl2' 1.305000e-02
%var: 'kdeg_Mito' 2.900000e-06
%var: 'ks_Mito' 2.175000e-01
%var: 'kdeg_mCytoC' 2.900000e-06
%var: 'ks_mCytoC' 2.175000e-01
%var: 'kdeg_mSmac' 2.900000e-06
%var: 'ks_mSmac' 4.350000e-02
%var: 'kdeg_Apaf' 2.900000e-06
%var: 'ks_Apaf' 4.350000e-02
%var: 'kdeg_pC9' 2.900000e-06
%var: 'ks_pC9' 4.350000e-02
%var: 'kdeg_L_pR' 2.900000e-06
%var: 'ks_L_pR' 0.000000e+00
%var: 'kdeg_DISC' 2.900000e-06
%var: 'ks_DISC' 0.000000e+00
%var: 'kdeg_DISC_flip' 2.900000e-06
%var: 'ks_DISC_flip' 0.000000e+00
%var: 'kdeg_DISC_pC8' 2.900000e-06
%var: 'ks_DISC_pC8' 0.000000e+00
%var: 'kdeg_C8' 2.900000e-06
%var: 'ks_C8' 0.000000e+00
%var: 'kdeg_BAR_C8' 2.900000e-06
%var: 'ks_BAR_C8' 0.000000e+00
%var: 'kdeg_C8_pC3' 2.900000e-06
%var: 'ks_C8_pC3' 0.000000e+00
%var: 'kdeg_Bid_C8' 2.900000e-06
%var: 'ks_Bid_C8' 0.000000e+00
%var: 'kdeg_C3' 2.900000e-06
%var: 'ks_C3' 0.000000e+00
%var: 'kdeg_tBid' 2.900000e-06
%var: 'ks_tBid' 0.000000e+00
%var: 'kdeg_C3_pC6' 2.900000e-06
%var: 'ks_C3_pC6' 0.000000e+00
%var: 'kdeg_C3_XIAP' 2.900000e-06
%var: 'ks_C3_XIAP' 0.000000e+00
%var: 'kdeg_C3_PARP' 2.900000e-06
%var: 'ks_C3_PARP' 0.000000e+00
%var: 'kdeg_Mcl1_tBid' 2.900000e-06
%var: 'ks_Mcl1_tBid' 0.000000e+00
%var: 'kdeg_Bax_tBid' 2.900000e-06
%var: 'ks_Bax_tBid' 0.000000e+00
%var: 'kdeg_C6' 2.900000e-06
%var: 'ks_C6' 0.000000e+00
%var: 'ks_C3_U' 0.000000e+00
%var: 'kdeg_CPARP' 2.900000e-06
%var: 'ks_CPARP' 0.000000e+00
%var: 'kdeg_aBax' 2.900000e-06
%var: 'ks_aBax' 0.000000e+00
%var: 'kdeg_C6_pC8' 2.900000e-06
%var: 'ks_C6_pC8' 0.000000e+00
%var: 'kdeg_MBax' 2.900000e-06
%var: 'ks_MBax' 0.000000e+00
%var: 'kdeg_Bcl2_MBax' 2.900000e-06
%var: 'ks_Bcl2_MBax' 0.000000e+00
%var: 'kdeg_Bax2' 2.900000e-06
%var: 'ks_Bax2' 0.000000e+00
%var: 'kdeg_Bax2_Bcl12' 2.900000e-06
%var: 'ks_Bax2_Bcl12' 0.000000e+00
%var: 'kdeg_Bax4' 2.900000e-06
%var: 'ks_Bax4' 0.000000e+00
%var: 'kdeg_Bax4_Bcl12' 2.900000e-06
%var: 'ks_Bax4_Bcl12' 0.000000e+00
%var: 'kdeg_Bax4_Mito' 2.900000e-06
%var: 'ks_Bax4_Mito' 0.000000e+00
%var: 'ks_AMito' 0.000000e+00
%var: 'kdeg_AMito_mCytoc' 2.900000e-06
%var: 'ks_AMito_mCytoc' 0.000000e+00
%var: 'kdeg_AMito_mSmac' 2.900000e-06
%var: 'ks_AMito_mSmac' 0.000000e+00
%var: 'kdeg_ACytoc' 2.900000e-06
%var: 'ks_ACytoc' 0.000000e+00
%var: 'kdeg_ASMac' 2.900000e-06
%var: 'ks_ASMac' 0.000000e+00
%var: 'kdeg_cCytoc' 2.900000e-06
%var: 'ks_cCytoc' 0.000000e+00
%var: 'kdeg_cSmac' 2.900000e-06
%var: 'ks_cSmac' 0.000000e+00
%var: 'kdeg_Apaf_cCytoc' 2.900000e-06
%var: 'ks_Apaf_cCytoc' 0.000000e+00
%var: 'kdeg_XIAP_cSmac' 2.900000e-06
%var: 'ks_XIAP_cSmac' 0.000000e+00
%var: 'kdeg_aApaf' 2.900000e-06
%var: 'ks_aApaf' 0.000000e+00
\%var: 'kdeg_Apop' 2.900000e-06
\%var: 'ks_Apop' 0.000000e+00
\%var: 'kdeg_Apop_pC3' 2.900000e-06
\%var: 'ks_Apop_pC3' 0.000000e+00
\%var: 'kdeg_Apop_XIAP' 2.900000e-06
\%var: 'ks_Apop_XIAP' 0.000000e+00

'bind_L_pR' L(b),pR(b) -> L(b!1),pR(b!1) @ 'kf1'
'bind_L_pR_rev' L(b!1),pR(b!1) -> L(b),pR(b) @ 'kr1'
'produce_DISC' L(b!1),pR(b!1) -> DISC(b) @ 'kc1'
'inhibit_DISC_by_flip' DISC(b),flip(b) -> DISC(b!1),flip(b!1) @ 'kf2'
'inhibit_DISC_by_flip_rev' DISC(b!1),flip(b!1) -> DISC(b),flip(b) @ 'kr2'
'produce_DISC' DISC(b),pC8(b) -> DISC(b!1),pC8(b!1) @ 'kf3'
'produce_DISC' DISC(b!1),pC8(b!1) -> DISC(b),pC8(b) @ 'kr3'
'inhibit_DISC_by_flip' DISC(b),C8(b) -> DISC(b!1),C8(b!1) @ 'kf4'
'inhibit_DISC_by_flip_rev' DISC(b!1),C8(b!1) -> DISC(b),C8(b) @ 'kr4'
'produce_DISC' C8(b),pC8(b) -> C8(b!1),pC8(b!1) @ 'kf5'
'produce_DISC' C8(b!1),pC8(b!1) -> C8(b),pC8(b) @ 'kr5'
'inhibit_DISC_by_flip' C8(b),C3(b) -> C8(b!1),C3(b!1) @ 'kf6'
'inhibit_DISC_by_flip_rev' C8(b!1),C3(b!1) -> C8(b),C3(b) @ 'kr6'
'produce_DISC' C3(b),pC3(b) -> C3(b!1),pC3(b!1) @ 'kf7'
'produce_DISC' C3(b!1),pC3(b!1) -> C3(b),pC3(b) @ 'kr7'
'inhibit_DISC_by_flip' C3(b),PARP(b) -> C3(b!1),PARP(b!1) @ 'kf8'
'inhibit_DISC_by_flip_rev' C3(b!1),PARP(b!1) -> C3(b),PARP(b) @ 'kr8'
'produce_DISC' C3(b),C3_U(b) -> C3(b!1),C3_U(b!1) @ 'kc8'
'bind_PARP_C3' C3(b),PARP(b) -> C3(b!1),PARP(b!1) @ 'kf9'
'bind_PARP_C3_rev' C3(b!1),PARP(b!1) -> C3(b),PARP(b) @ 'kr9'
'produce_CPARP_via_C3' C3(b!1),PARP(b!1) -> C3(b),CPARP(b) @ 'kc9'
Apoptosis System

269 'bind_Bid_C8' C8(b),Bid(b) -> C8(b!1),Bid(b!1) @ 'kf10'
270 'bind_Bid_C8_rev' C8(b!1),Bid(b!1) -> C8(b),Bid(b) @ 'kr10'
271 'produce_tBid_via_C8' C8(b!1),Bid(b!1) -> C8(b),tBid(b) @ 'kc10'
272 'inhibit_tBid_by_Mcl1' tBid(b),Mcl1(b) -> tBid(b!1),Mcl1(b!1) @ 'kf11'
273 'inhibit_tBid_by_Mcl1_rev' tBid(b!1),Mcl1(b!1) -> tBid(b),Mcl1(b) @ 'kr11'
274 'bind_Bax_tBid' tBid(b),Bax(b) -> tBid(b!1),Bax(b!1) @ 'kf12'
275 'bind_Bax_tBid_rev' tBid(b!1),Bax(b!1) -> tBid(b),Bax(b) @ 'kr12'
276 'produce_aBax_via_tBid' tBid(b!1),Bax(b!1) -> tBid(b),aBax(b) @ 'kc12'
277 'transloc_MBax_aBax' aBax(b) -> MBax(b) @ 'kf13'
278 'transloc_MBax_aBax_rev' MBax(b) -> aBax(b) @ 'kr13'
279 'inhibit_MBax_by_Bcl2' MBax(b),Bcl2(b) -> MBax(b!1),Bcl2(b!1) @ 'kf14'
280 'inhibit_MBax_by_Bcl2_rev' MBax(b!1),Bcl2(b!1) -> MBax(b),Bcl2(b) @ 'kr14'
281 'dimerize_MBax_to_Bax2' MBax(b),MBax(b) -> Bax2(b) @ 'kf15'
282 'dimerize_MBax_to_Bax2_rev' Bax2(b) -> MBax(b),MBax(b) @ 'kr15'
283 'inhibit_Bax2_by_Bcl2' Bax2(b),Bcl2(b) -> Bax2(b!1),Bcl2(b!1) @ 'kf16'
284 'inhibit_Bax2_by_Bcl2_rev' Bax2(b!1),Bcl2(b!1) -> Bax2(b),Bcl2(b) @ 'kr16'
285 'dimerize_Bax2_to_Bax4' Bax2(b),Bax2(b) -> Bax4(b) @ 'kf17'
286 'dimerize_Bax2_to_Bax4_rev' Bax4(b) -> Bax2(b),Bax2(b) @ 'kr17'
287 'inhibit_Bax4_by_Bcl2' Bax4(b),Bcl2(b) -> Bax4(b!1),Bcl2(b!1) @ 'kf18'
288 'inhibit_Bax4_by_Bcl2_rev' Bax4(b!1),Bcl2(b!1) -> Bax4(b),Bcl2(b) @ 'kr18'
289 'bind_Bax4_Mito' Bax4(b),Mito(b) -> Bax4(b!1),Mito(b!1) @ 'kf19'
290 'bind_Bax4_Mito_rev' Bax4(b!1),Mito(b!1) -> Bax4(b),Mito(b) @ 'kr19'
291 'produce_AMito' Bax4(b!1),Mito(b!1) -> AMito(b) @ 'kc19'
292 'bind_mCytC_AMito' AMito(b),mCytoC(b) -> AMito(b!1),mCytoC(b!1) @ 'kf20'
293 'bind_mCytC_AMito_rev' AMito(b!1),mCytoC(b!1) -> AMito(b),mCytoC(b) @ 'kr20'
B.1 Kappa System for Apoptosis

\[
\begin{align*}
&\text{'produce_ACytoC_via_AMito'} \quad \text{AMito}(b!1),\text{mCytoC}(b!1) \rightarrow \text{AMito}(b),\text{ACytoC}(b) @ 'kc20' \\
&\text{'bind_mSmac_AMito'} \quad \text{AMito}(b),\text{mSmac}(b) \rightarrow \text{AMito}(b!1),\text{mSmac}(b!1) @ 'kf21'
&\text{'bind_mSmac_AMito_rev'} \quad \text{AMito}(b!1),\text{mSmac}(b!1) \rightarrow \text{AMito}(b),\text{mSmac}(b) @ 'kr21'
&\text{'produce_ASmac_via_AMito'} \quad \text{AMito}(b!1),\text{mSmac}(b!1) \rightarrow \text{AMito}(b),\text{ASmac}(b) @ 'kc21'
&\text{'transloc_cCytoC_ACytoC'} \quad \text{ACytoC}(b) \rightarrow \text{cCytoC}(b) @ 'kf22'
&\text{'transloc_cCytoC_ACytoC_rev'} \quad \text{cCytoC}(b) \rightarrow \text{ACytoC}(b) @ 'kr22'
&\text{'bind_Apaf_cCytoC'} \quad \text{cCytoC}(b),\text{Apaf}(b) \rightarrow \text{cCytoC}(b!1),\text{Apaf}(b!1) @ 'kf23'
&\text{'bind_Apaf_cCytoC_rev'} \quad \text{cCytoC}(b!1),\text{Apaf}(b!1) \rightarrow \text{cCytoC}(b),\text{Apaf}(b) @ 'kr23'
&\text{'produce_aApaf_via_cCytoC'} \quad \text{cCytoC}(b!1),\text{Apaf}(b!1) \rightarrow \text{cCytoC}(b),\text{aApaf}(b) @ 'kc23'
&\text{'bind_pC3_Apop'} \quad \text{Apop}(b),\text{pC3}(b) \rightarrow \text{Apop}(b!1),\text{pC3}(b!1) @ 'kf25'
&\text{'bind_pC3_Apop_rev'} \quad \text{Apop}(b!1),\text{pC3}(b!1) \rightarrow \text{Apop}(b),\text{pC3}(b) @ 'kr25'
&\text{'produce_C3_via_Apop'} \quad \text{Apop}(b!1),\text{pC3}(b!1) \rightarrow \text{Apop}(b),\text{C3}(b) @ 'kc25'
&\text{'DISC_deg'} \quad \text{DISC}(b) \rightarrow \text{L}(b),\text{pR}(b) @ 'kf31'
&\text{'AMito_deg'} \quad \text{AMito}(b) \rightarrow \text{Mito}(b) @ 'kdeg_AMito'
&\text{'L_syn'} \quad \rightarrow \text{L}(b) @ 'ks_L'
&\text{'L_deg'} \quad \text{L}(b) \rightarrow @ 'kdeg_L'
&\text{'pR_syn'} \quad \rightarrow \text{pR}(b) @ 'ks_pR'
&\text{'pR_deg'} \quad \text{pR}(b) \rightarrow @ 'kdeg_pR'
\end{align*}
\]
Apoptosis System

'flip_syn' -> flip(b) @ 'ks_flip'
'flip_deg' flip(b) -> @ 'kdeg_flip'
'pC8_syn' -> pC8(b) @ 'ks_pC8'
'pC8_deg' pC8(b) -> @ 'kdeg_pC8'
'BAR_syn' -> BAR(b) @ 'ks_BAR'
'BAR_deg' BAR(b) -> @ 'kdeg_BAR'
'pC3_syn' -> pC3(b) @ 'ks_pC3'
'pC3_deg' pC3(b) -> @ 'kdeg_pC3'
'pC6_syn' -> pC6(b) @ 'ks_pC6'
'pC6_deg' pC6(b) -> @ 'kdeg_pC6'
'XIAP_syn' -> XIAP(b) @ 'ks_XIAP'
'XIAP_deg' XIAP(b) -> @ 'kdeg_XIAP'
'PARP_syn' -> PARP(b) @ 'ks_PARP'
'PARP_deg' PARP(b) -> @ 'kdeg_PARP'
'Bid_syn' -> Bid(b) @ 'ks_Bid'
'Bid_deg' Bid(b) -> @ 'kdeg_Bid'
'Mcl1_syn' -> Mcl1(b) @ 'ks_Mcl1'
'Mcl1_deg' Mcl1(b) -> @ 'kdeg_Mcl1'
'Bax_syn' -> Bax(b) @ 'ks_Bax'
'Bax_deg' Bax(b) -> @ 'kdeg_Bax'
'Bcl2_syn' -> Bcl2(b) @ 'ks_Bcl2'
'Bcl2_deg' Bcl2(b) -> @ 'kdeg_Bcl2'
'Mito_syn' -> Mito(b) @ 'ks_Mito'
'Mito_deg' Mito(b) -> @ 'kdeg_Mito'
'mCytoC_syn' -> mCytoC(b) @ 'ks_mCytoC'
'mCytoC_deg' mCytoC(b) -> @ 'kdeg_mCytoC'
'mSmac_syn' -> mSmac(b) @ 'ks_mSmac'
'mSmac_deg' mSmac(b) -> @ 'kdeg_mSmac'
'Apaf_syn' -> Apaf(b) @ 'ks_Apaf'
'Apaf_deg' Apaf(b) -> @ 'kdeg_Apaf'
'pC9_syn' -> pC9(b) @ 'ks_pC9'
'pC9_deg' pC9(b) -> @ 'kdeg_pC9'
'L_pr_syn' -> L(b!1),pR(b!1) @ 'ks_L_pr'
'L_pr_deg' L(b!1),pR(b!1) -> @ 'kdeg_L_pr'
'DISC_syn' -> DISC(b) @ 'ks_DISC'
'DISC_flip_syn' -> DISC(b!1),flip(b!1) @ 'ks_DISC_flip'
'DISC_flip_deg' DISC(b!1),flip(b!1) -> @ 'kdeg_DISC_flip'
'DISC_pC8_syn' -> DISC(b!1),pC8(b!1) @ 'ks_DISC_pC8'
'DISC_pC8_deg' DISC(b!1),pC8(b!1) -> @ 'kdeg_DISC_pC8'
'C8_syn' -> C8(b) @ 'ks_C8'
'C8_deg' C8(b) -> @ 'kdeg_C8'
'BAR_C8_syn' -> BAR(b!1),C8(b!1) @ 'ks_BAR_C8'
'BAR_C8_deg' BAR(b!1),C8(b!1) -> @ 'kdeg_BAR_C8'
B.1 Kappa System for Apoptosis

363 'C8_pC3_syn' -> C8(b!1),pC3(b!1) @ 'ks_C8_pC3'
364 'C8_pC3_deg' C8(b!1),pC3(b!1) -> 0 'kdeg_C8_pC3'
365 'Bid_C8_syn' -> Bid(b!1),C8(b!1) @ 'ks_Bid_C8'
366 'Bid_C8_deg' Bid(b!1),C8(b!1) -> 0 'kdeg_Bid_C8'
367 'C3_syn' -> C3(b) @ 'ks_C3'
368 'C3_deg' C3(b) -> 0 'kdeg_C3'
369 'tBid_syn' -> tBid(b) @ 'ks_tBid'
370 'tBid_deg' tBid(b) -> 0 'kdeg_tBid'
371 'C3_pC6_syn' -> C3(b!1),pC6(b!1) @ 'ks_C3_pC6'
372 'C3_pC6_deg' C3(b!1),pC6(b!1) -> 0 'kdeg_C3_pC6'
373 'C3_XIAP_syn' -> C3(b!1),XIAP(b!1) @ 'ks_C3_XIAP'
374 'C3_XIAP_deg' C3(b!1),XIAP(b!1) -> 0 'kdeg_C3_XIAP'
375 'C3_PARP_syn' -> C3(b!1),PARP(b!1) @ 'ks_C3_PARP'
376 'C3_PARP_deg' C3(b!1),PARP(b!1) -> 0 'kdeg_C3_PARP'
377 'Mcl1_tBid_syn' -> Mcl1(b!1),tBid(b!1) @ 'ks_Mcl1_tBid'
378 'Mcl1_tBid_deg' Mcl1(b!1),tBid(b!1) -> 0 'kdeg_Mcl1_tBid'
379 'Bax_tBid_syn' -> Bax(b!1),tBid(b!1) @ 'ks_Bax_tBid'
380 'Bax_tBid_deg' Bax(b!1),tBid(b!1) -> 0 'kdeg_Bax_tBid'
381 'C6_syn' -> C6(b) @ 'ks_C6'
382 'C6_deg' C6(b) -> 0 'kdeg_C6'
383 'C3_U_syn' -> C3_U(b) @ 'ks_C3_U'
384 'C3_U_deg' C3_U(b) -> 0 'kdeg_C3_U'
385 'CPARP_syn' -> CPARP(b) @ 'ks_CPARP'
386 'CPARP_deg' CPARP(b) -> 0 'kdeg_CPARP'
387 'aBax_syn' -> aBax(b) @ 'ks_aBax'
388 'aBax_deg' aBax(b) -> 0 'kdeg_aBax'
389 'C6_pC8_syn' -> C6(b!1),pC8(b!1) @ 'ks_C6_pC8'
390 'C6_pC8_deg' C6(b!1),pC8(b!1) -> 0 'kdeg_C6_pC8'
391 'MBax_syn' -> MBax(b) @ 'ks_MBax'
392 'MBax_deg' MBax(b) -> 0 'kdeg_MBax'
393 'Bcl2_MBax_syn' -> Bcl2(b!1),MBax(b!1) @ 'ks_Bcl2_MBax'
394 'Bcl2_MBax_deg' Bcl2(b!1),MBax(b!1) -> 0 'kdeg_Bcl2_MBax'
395 'Bax2_syn' -> Bax2(b) @ 'ks_Bax2'
396 'Bax2_deg' Bax2(b) -> 0 'kdeg_Bax2'
397 'Bax2_Bcl2_syn' -> Bax2(b!1),Bcl2(b!1) @ 'ks_Bax2_Bcl2'
398 'Bax2_Bcl2_deg' Bax2(b!1),Bcl2(b!1) -> 0 'kdeg_Bax2_Bcl2'
399 'Bax4_syn' -> Bax4(b) @ 'ks_Bax4'
400 'Bax4_deg' Bax4(b) -> 0 'kdeg_Bax4'
401 'Bax4_Bcl2_syn' -> Bax4(b!1),Bcl2(b!1) @ 'ks_Bax4_Bcl2'
402 'Bax4_Bcl2_deg' Bax4(b!1),Bcl2(b!1) -> 0 'kdeg_Bax4_Bcl2'
403 'Bax4_Mito_syn' -> Bax4(b!1),Mito(b!1) @ 'ks_Bax4_Mito'
404 'Bax4_Mito_deg' Bax4(b!1),Mito(b!1) -> 0 'kdeg_Bax4_Mito'
405 'AMito_syn' -> AMito(b) @ 'ks_AMito'
'AMito_mCytoC_syn' -> AMito(b!1),mCytoC(b!1) @ 'ks_AMito_mCytoC'
'AMito_mCytoC_deg' AMito(b!1),mCytoC(b!1) -> @ 'kdeg_AMito_mCytoC'
'AMito_mSmac_syn' -> AMito(b!1),mSmac(b!1) @ 'ks_AMito_mSmac'
'AMito_mSmac_deg' AMito(b!1),mSmac(b!1) -> @ 'kdeg_AMito_mSmac'
'ACytoC_syn' -> ACytoC(b) @ 'ks_ACytoC'
'ACytoC_deg' ACytoC(b) -> @ 'kdeg_ACytoC'
'ASmac_syn' -> ASmac(b) @ 'ks_ASmac'
'ASmac_deg' ASmac(b) -> @ 'kdeg_ASmac'
'cCytoC_syn' -> cCytoC(b) @ 'ks_cCytoC'
'cCytoC_deg' cCytoC(b) -> @ 'kdeg_cCytoC'
'cSmac_syn' -> cSmac(b) @ 'ks_cSmac'
'cSmac_deg' cSmac(b) -> @ 'kdeg_cSmac'
'Apaf_cCytoC_syn' -> Apaf(b!1),cCytoC(b!1) @ 'ks_Apaf_cCytoC'
'Apaf_cCytoC_deg' Apaf(b!1),cCytoC(b!1) -> @ 'kdeg_Apaf_cCytoC'
'XIAP_cSmac_syn' -> XIAP(b!1),cSmac(b!1) @ 'ksXIAP_cSmac'
'XIAP_cSmac_deg' XIAP(b!1),cSmac(b!1) -> @ 'kdegXIAP_cSmac'
'aApaf_syn' -> aApaf(b) @ 'ks_aApaf'
'aApaf_deg' aApaf(b) -> @ 'kdeg_aApaf'
'Apop_syn' -> Apop(b) @ 'ks_Apop'
'Apop_deg' Apop(b) -> @ 'kdeg_Apop'
'Apop_pC3_syn' -> Apop(b!1),pC3(b!1) @ 'ks_Apop_pC3'
'Apop_pC3_deg' Apop(b!1),pC3(b!1) -> @ 'kdeg_Apop_pC3'
'Apop_XIAP_syn' -> Apop(b!1),XIAP(b!1) @ 'ks_Apop_XIAP'
'Apop_XIAP_deg' Apop(b!1),XIAP(b!1) -> @ 'kdeg_Apop_XIAP'

%obs: 'Bid_unbound' |Bid(b)|
%obs: 'PARP_unbound' |PARP(b)|
%obs: 'mSmac_unbound' |mSmac(b)|
%obs: 'tBid_total' |tBid()|
%obs: 'CPARP_total' |CPARP()|
%obs: 'cSmac_total' |cSmac()|
%def: "displayCompression" "weak"
%mod: [T]>0 do $TRACK 'CPARP_total'[true]
%mod: 'cSmac_total' > 80000 do ($STOP)
%init: 'L_0' L(b)
%init: 'pR_0' pR(b)
%init: 'flip_0' flip(b)
%init: 'pC8_0' pC8(b)
%init: 'BAR_0' BAR(b)
%init: 'pC3_0' pC3(b)
%init: 'pC6_0' pC6(b)
%init: 'XIAP_0' XIAP(b)
%init: 'PARP_0' PARP(b)
%init: 'Bid_0' Bid(b)
%init: 'Mcl1_0' Mcl1(b)
%init: 'Bax_0' Bax(b)
%init: 'Bcl2_0' Bcl2(b)
%init: 'Mito_0' Mito(b)
%init: 'mCytoC_0' mCytoC(b)
%init: 'mSmac_0' mSmac(b)
%init: 'Apaf_0' Apaf(b)
%init: 'pC9_0' pC9(b)