École normale supérieure
Department of Computer Science

Causal analysis of rule-based models of signaling pathways

Jonathan Laurent
jonathan.laurent@ens.fr

Under the supervision of:

Walter Fontana
walter_fontana@hms.harvard.edu

Jérôme Feret
feret@di.ens.fr

August 2015

Acknowledgment

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the U.S. Army Research Office under grant number W911NF-14-1-0367. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

1. Introduction

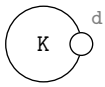
When a cell receives a signal from its environment through a membrane receptor, a cascade of internal reactions is triggered which leads ultimately to its response, usually a change in its metabolism, its ability to divide, or in the expression level of a gene. Many diseases including some forms of cancer are related to some defects in these signaling pathways. Such biological systems usually feature thousands of proteins interacting in a highly concurrent way and their complexity makes modelling necessary to the development of better biocuration techniques.

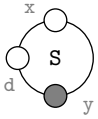
1.1. The kappa modelling language

The *kappa* language was designed to model interactions between proteins with rewriting rules over site-graphs. We illustrate this with the example of phosphorylation reactions.

Many proteins can be turned on and off by attaching or removing phosphate groups to some of their residues. The action of attaching a phosphate group to a protein is called *phosphorylation* and an enzyme catalyzing a phosphorylation reaction is called a *kinase*. A kinase may need to be phosphorylated itself in order to be effective and so signaling pathways often feature phosphorylation cascades. In kappa, proteins are modelled by agents. An agent features some sites through which it can bind other agents. Moreover, some sites can hold an internal state, usually u when unphosphorylated and p when phosphorylated. The number and the nature of the sites featured by an agent depend on its type, each type of agent being described in the *signature* of the kappa model. In most of the examples of this report, signatures have to be inferred directly from the rules.

Let's model a kinase by an agent of type K with one binding site d. It acts on a substrate which is an agent of type S with a binding site d and two phosphorylation sites x and y which can hold the states u or p. Below are the textual and graphical representations of a kinase with a free binding site and a substrate which is phosphorylated at site y but not at site x.

$K(d)$: 

$S(d, x_u, y_p)$: 

The interaction between the kinase and the substrate is captured by the rules displayed Figure 1. The first rule is bidirectional and states that a kinase and a substrate can bind (`b`) and unbind (`u`) to each other through their sites d. The second rule states that if a kinase is bound to a substrate with an unphosphorylated site x, the latter can get phosphorylated. The third does the same for y. In the textual notation, a shared exponent denotes the existence of a bond between two sites. Sites without exponents are considered to be free.

An important remark is that a rule can feature underspecified agents. When it does, it can be triggered whatever the binding or internal states of the missing sites are. This characteristic of kappa makes it different from most modelling techniques traditionally used by biologists like differential equations systems or Petri nets. Indeed, the latter require that one variable is introduced for each fully specified species, which leads quickly to a combinatorial explosion. For instance, if a protein has 10 distinct phosphorylation sites, it yields at least 1024 different

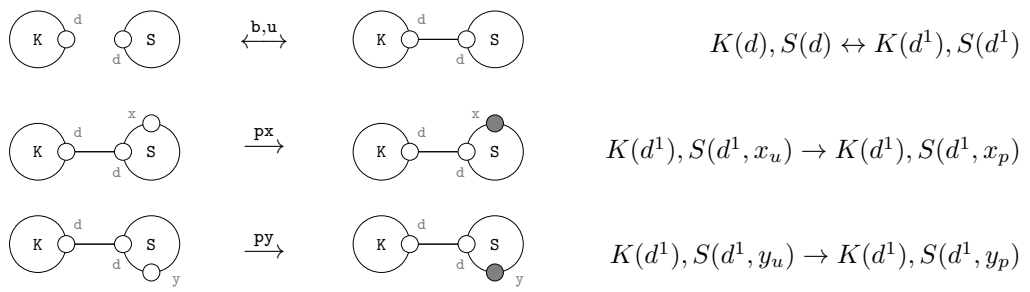


Figure 1: A simple kappa model for double-phosphorylation

species. The situation in real world signaling networks where many proteins can bind each other and form large complexes is even worst.

In kappa, a reaction mixture is modelled by a large site-graph. When a pattern matching the left hand side of a rule r is recognized in it, it can be updated locally according to r. The agents preserved by r are those of the longest prefix featuring agents of the same type and mentioning the same sites in both sides of r. The sites they mention are updated according to their state in the right hand side of r and the others are left unchanged. Agent featured in the left hand side of r and not in its right hand side are removed and agents featured in the the right hand side of r and not in its left hand side are created. For instance, the rule $K(d), S(x_p) \rightarrow K(d), K(d)$ deletes an instance of S and creates an instance of K. For a rigorous definition of kappa's semantics, see [3].

After having specified an initial mixture and a reaction rate for each rule, it is possible to run stochastic simulations of a model with the kappa simulator [2]. An example of source file for a small kappa model is provided Appendix A.

1.2. From rules to pathways

The aim of *causal analysis* is to understand better how pathways emerge from a multitude of potential low level protein-protein interactions. A great number of these interactions is known thanks to the progresses of experimental methods but the way they result in complex signaling behaviors remains to be understood. One difficulty in figuring this out comes from the density of the reaction network involved along with its apparent lack of structure. In particular, there are many crosstalks between signalling processes that seem completly unrelated. As a consequence, small signaling models that focus on a limited number of interactions in order to stay tractable for human reasoning usually bring a limited insight. Thus, the development of automated analysis techniques for large models of protein-protein interaction networks appears to be an important step towards a better understanding of signalling pathways.

In this report, we introduce a formal notion of *story* as an attempt to precise the fuzzy concept of pathway. Appendix B features an example of a story generated from the model of Appendix A. It describes how a set of rule applications can be chained together to make an

observable event happen along with the necessary temporal precedence relation between them. Here, the observable event is the formation of a *SoS-Grb2-Shc* complex, as mentioned in the last line of Appendix A. For clarity, the agents that are targeted by each rule application are not mentioned and only rule names are displayed.

In section 2, we introduce a generic framework for studying event systems and we define the notion of a story. An important class of such event systems for which efficient algorithms can be written is studied in section 3. Kappa models are described within this formalism and a compression algorithm is introduced to compute minimal stories from simulation traces. Then, we describe some techniques to produce stories statically in section 4.

2. A generic framework for event systems

2.1. Event systems

Let Q a set of states, intuitively the set of all the possible states of the world. A subset of Q is called a *context*. Moreover, an event e is defined by a label denoted $\mathsf{label}(e)$, a precondition $\mathsf{pre}^\flat(e) \subseteq Q$ and an effect $\mathsf{eff}^\flat(e) : \mathsf{pre}^\flat(e) \to Q$. The precondition of an event is the set of states from which it can be triggered and its effect maps each of these states to a new one. The flat symbol on $\mathsf{pre}^\flat$ distinguishes it from its dual operator pre which returns a set of constraints on states and which is defined in section 3.

An *event system* is a triple (Q, E, C) with E a set of events over Q and $C \subseteq \mathcal{P}(Q)$ a set of contexts such that for all event $e \in E$ we have $\mathsf{pre}^\flat(e) \in C$ and $\forall c \in C,\ \mathsf{eff}^\flat(e)(c) \in C$. All the concepts introduced in section 2 are defined relatively to an event system.

2.2. Traces

A *trace* is defined as a finite sequence of events. Its *precondition* is defined by induction on its size by the following equations:

$$\begin{aligned} \mathsf{pre}^\flat(\epsilon) &= Q \\ \mathsf{pre}^\flat(e,t) &= \left\{ q \in \mathsf{pre}^\flat(e) \,:\, \mathsf{eff}^\flat(e)(q) \in \mathsf{pre}^\flat(t) \right\} \end{aligned}$$

with e an event and t a trace. A trace t is said to be *valid* in the context c if $c \subseteq \mathsf{pre}^\flat(t)$, in which case we write $c \vdash t$. The effect and the postcondition of a trace in context c are defined by:

$$\mathsf{eff}^\flat{}_c(e_1, \ldots, e_n) = \mathsf{eff}^\flat(e_n) \circ \ldots \circ \mathsf{eff}^\flat(e_1)\Big|_c \qquad \mathsf{post}^\flat{}_c(t) = \left(\mathsf{eff}^\flat{}_c(t)\right)(c)$$

2.3. Concurrent events and equivalent traces

The order in which some events happen might not always matter. Let's take the example of a kappa agent which can be phosphorylated on its both sites in parallel by two different enzymes. Let's write p_1 the event of phosphorylating site 1 and p_2 the event of phosphorylating site 2. Then, we may want the traces (p_1, p_2) and (p_2, p_1) to be considered equivalent. Indeed, the definition of a trace forces us to decide on an order between p_1 and p_2 but this order is nor

relevant neither necessary. Worst, in the real world, p_1 and p_2 could have happened in the same time. This draws our motivation to formalize a concept of *concurrent* events.

Two events e_1 and e_2 are said to be concurrent in context c if the three following assertions are equivalent:

$$c \vdash e_1, e_2 \qquad c \vdash e_2, e_1 \qquad c \vdash e_1 \ \wedge \ c \vdash e_2$$

and $\mathsf{eff}_c(e_1, e_2) = \mathsf{eff}_c(e_2, e_1)$ when they hold. In this case, we write $e_1 \diamond_c e_2$. Moreover, e_1 and e_2 are said to be concurrent if they are in any context $c \in C$, which we write $e_1 \diamond e_2$. Finally, two events e_1 and e_2 are said to be *non-trivially* concurrent in context c if $e_1 \diamond_c e_2$ and $c \vdash e_1 \wedge c \vdash e_2$. They are *non-trivially* concurrent if they are concurrent in any context, this being non-trivially in at least one of them. This last notion admits a nice algebraic characterization for a class of systems kappa models belongs to, as we see in section 3.2.

As said before, we want to consider two traces differing only by the permutation of concurrent events to be equivalent. Let c a context. We define the relation $\simeq_c$ as the smallest equivalence relation on the set of valid traces in context c such that:

$$e_1 \diamond_{\mathsf{post}^\flat{}_c(t)} e_2 \implies t \cdot (e_1, e_2) \cdot t' \simeq_c t \cdot (e_2, e_1) \cdot t'$$

where $\cdot$ stands for the concatenation operator on traces. This relation is called the *Mazurkieviecz equivalence* in context c. We define $\sim$ as a stronger equivalence relation on traces that is context-independent. It is the smallest relation on traces such that:

$$e_1 \diamond e_2 \implies t \cdot (e_1, e_2) \cdot t' \sim t \cdot (e_2, e_1) \cdot t'$$

Two equivalent traces for $\sim$ are said to be *strongly similar*. Here, two elements can be permuted only if they are concurrent in any context and not only in the current one. Note that we have: $t \sim t' \implies t \simeq_{\mathsf{pre}^\flat(t)} t'$ but the converse is not true. However, we prove in section 3.2 that for an important subset of kappa models called *regular* models, the relations $\sim$ and $\simeq_c$ are identical in any context c.

2.4. Configurations

Let $\mathcal{T}$ the set of all traces on E. An element of the quotient $(\mathcal{T}/\sim)$ is called a *configuration* and its traces are said to be its *trajectories*. Let $t = e_1, \ldots, e_n$ a trace and $\mathcal{E}_t = \{e_i\}_i$ the set of its events, each one of them being distinguished so that $|\mathcal{E}_t| = n$. The *precedence* relation $\preceq_t$ is defined as the smallest order relation over $\mathcal{E}_t$ such that:

$$(i < j \ \wedge \ \neg (e_i \diamond e_j)) \implies e_i \preceq_t e_j$$

Besides, a permutation σ of $\{1, \ldots, n\}$ is said to preserve it when $e_{\sigma(i)} \preceq_t e_{\sigma(j)} \Rightarrow \sigma(i) \leq \sigma(j)$. Then, writing $\sigma(t) := e_{\sigma(1)}, \ldots, e_{\sigma(n)}$ the reordering of t by σ, the following theorem holds:

Theorem 1. *Let t and t' two traces. Then $t \sim t'$ if and only if $t' = \sigma(t)$ for σ a permutation preserving the precedence relation on t.*

As a consequence, a configuration is represented by a tuple consisting in a set of events along with an order relation on them. Then, the canonical projection $\mathcal{C} : \mathcal{T} \to (\mathcal{T}/\sim)$ maps t to $(\mathcal{E}_t, \preceq_t)$ as we have indeed $t \sim t' \iff (\mathcal{E}_t, \preceq_t) = (\mathcal{E}_{t'}, \preceq_{t'})$.

All trajectories of a configuration $(\mathcal{E}, \preceq)$ can be recovered by ordering the events of $\mathcal{E}$ preserving $\preceq$. Moreover, as the operators $\mathsf{pre}^\flat$, $\mathsf{eff}^\flat$ and $\mathsf{post}^\flat$ have the same values on strongly similar traces, they can also be defined on a configuration from any of its trajectory. As a consequence, a context being given, we can talk about *valid* configurations too. Finally, we say that $(\mathcal{E}, \preceq)$ is a sub-configuration of $(\mathcal{E}', \preceq')$ if $\mathcal{E} \subseteq \mathcal{E}'$ and $(\preceq) \subseteq (\preceq')$.

2.5. Stories

Let Ω a subset of interest of E called set of *observable events* and c a context. Then, a *story* reaching Ω from c is a configuration containing an event of Ω which is valid in c. We write $\mathcal{S}(\Omega, c)$ the set of all such stories. When it is non-empty, we say that Ω is *reachable* from c.

Therefore, we can see stories as proofs of reachability. However, this point of view is misleading as it takes us away from the spirit of biological pathways stories are aimed to formalize. Indeed, when biologists study pathways, the reachability of the observable is usually taken for granted and what really matters is the question of **how** this process is implemented. For instance, when designing drugs, it may be of particular interest to identify a step which is common to all the scenarios leading to the realization of a biological process. This example is in opposition with the principle of *proof-irrelevance* most logicians work with which stipulates that all the proofs of a fact should be considered equivalent. Clearly, we don't want to formalize pathways in this spirit. Then, if all elements of $\mathcal{S}(\Omega, c)$ should not be considered equivalent, should they be considered all different and equally relevant? It seems the answer here is still no. Indeed, some stories may contain some events which make no contribution in producing the observable. Moreover, two stories might feature the phosphorylation of the same agent by two different but isomorphic enzymes and one may argue they should not be considered different. This motivates us to introduce notions of *compression* along with some equivalence relations on $\mathcal{S}(\Omega, c)$.

2.5.1. Sub-stories and compression

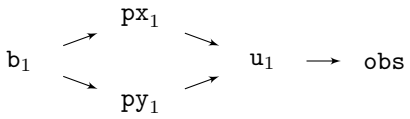
$\mathcal{S}(\Omega, c)$ can be equipped with a relation $\lhd_{\Omega,c}$ such that $s \lhd_{\Omega,c} s'$ if and only if s is a sub-configuration of s', in which case we say that s is a *sub-story* of s'. A story is said to be *minimal* if it has no strict sub-story. When discussing causality, people usually distinguish necessary causes from sufficient ones. In medicine or in biology, both notions are often too demanding to be met in practice and a notion of contributive causality is preferred, where an event e_1 is said to be a cause of an event e_2 if it is an insufficient but non-redundant part of a condition which is itself unnecessary but sufficient for the occurence of e_2. Minimal stories are defined in this exact spirit. Indeed, any event e_1 of a minimal story explaining e_2 is a cause of e_2 in this sense.

The kappa simulator implements a technique to extract minimal sub-stories of large stories called *weak-compression*. Some details are given in section 3.4. However, a stronger notion of compression is sometimes needed, as we illustrate now with an example.

2.5.2. The example of double phosphorylation

Let's consider the kappa model introduced in Figure 1 with the initial mixture $K(d), K(d), S(x_u, y_u)$ and let's take for an observable rule the rule `obs` which tests the pattern $S(x_p, y_p)$ and does

nothing. Then, here is an example of a story s_0 where the first enzyme binds the substrate, phosphorylates its two sites and then unbinds:



It is useful to see a story as a graph whose nodes are events and whose edges correspond to the precedence relation over them. For clarity, all the edges are not represented here: only a transitive reduction of the graph is displayed. Moreover, as explained in section 3.3, a kappa event consists in a rule along with an instantiation map giving the identifiers of the agents it targets. Here, an event is labelled $\mathtt{r}_i$ where $\mathtt{r}$ is the name of the rule applied, and $i \in \{1, 2\}$ gives the identifier of the enzyme involved. There is no need to specify which substrate is targeted as there is only one in the mixture. Finally s_0 illustrates the fact that $\mathtt{px}_1$ and $\mathtt{py}_1$ are concurrent. Here are two other stories about the same phenomenon:

$$s_1 \;:\; \mathtt{b}_1 \longrightarrow \mathtt{px}_1 \longrightarrow \mathtt{u}_1 \longrightarrow \mathtt{b}_1 \longrightarrow \mathtt{py}_1 \longrightarrow \mathtt{u}_1 \longrightarrow \mathtt{obs}$$

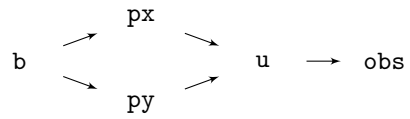
$$s_2 \;:\; \mathtt{b}_1 \longrightarrow \mathtt{px}_1 \longrightarrow \mathtt{u}_1 \longrightarrow \mathtt{b}_2 \longrightarrow \mathtt{py}_2 \longrightarrow \mathtt{u}_2 \longrightarrow \mathtt{obs}$$

In s_1, the first enzyme unbinds and rebinds the substrate after it has phosphorylated its first site and before it phosphorylates its second. Remark that if a story is seen as a graph whose edges correspond to the precedence relation and s and s' are two valid stories, we have $s \lhd s'$ if and only if the graph of s can be embedded in the graph of s'.[1] Here, it can be noticed that $s_0 \lhd s_1$. Indeed, we can get s_0 from s_1 by compressing the $(\mathtt{u}_1, \mathtt{b}_1)$ pattern in it.

In s_2, after the first enzyme has phosphorylated the site $\mathtt{x}$ of the substrate, it unbinds and the second enzyme binds to phosphorylate $\mathtt{y}$. Contrary to s_1, s_2 is minimal and cannot be compressed further. However, one may argue that whether kinase 1 or 2 is used to phosphorylate the substrate does not really matter. If we abstract away this information, stories s_1 and s_2 turn out to be the same and can be represented as follows, where nodes are not labelled by events anymore but by rules:

$$\mathtt{b} \longrightarrow \mathtt{px} \longrightarrow \mathtt{u} \longrightarrow \mathtt{b} \longrightarrow \mathtt{py} \longrightarrow \mathtt{u} \longrightarrow \mathtt{obs}$$

This graph can be considered as an *abstract story*. Similarly, s_0 can be abstracted into:



As the latter is embedded in the former, we say that s_0 is a strongly compressed form of both s_1 and s_2. In section 2.5.3, we formalize this notion of an abstract story and introduce *strong compression*, as implemented in the kappa simulator.

[1] An embedding of a graph $G = (V, E)$ in another graph $G' = (V', E')$ is a injective graph-morphism $\varphi : G \to G'$, that is a map from V to V' such that x and $\varphi(x)$ have the same label for all x and $\forall x, y,\ (x, y) \in E \implies (\varphi(x), \varphi(y)) \in E'$.

2.5.3. Abstract stories and compression

An abstraction over $\mathcal{S}(\Omega, c)$ consists in:

- An *abstract domain* A. In the last example, it would be the set of all directed graphs whose nodes are labelled by rules, which is roughly the choice made by the current version of the kappa simulator.
- An *abstraction map* $\alpha : \mathcal{S}(\Omega, c) \to A$. In the last example, α abstracts away the distinction between the two kinases present in the mixture.

An abstraction (A, α) yields an equivalence relation $\sim_\alpha$ such that $s \sim_\alpha s'$ if and only if $\alpha(s) = \alpha(s')$. In the last example, s_1 and s_2 are equivalent in this sense. Moreover, a common pattern is to take A as a set of labelled directed graphs. Then, it is possible to introduce a notion of compression modulo α as a relation $\to_\alpha$ over stories such that $s \to_\alpha s'$ if and only if $\alpha(s')$ is embedded into $\alpha(s)$. In this context, a story is said to be *minimal* if it can't be compressed into a shorter one. In the last example, we would have $s_1 \to_\alpha s_0$ and $s_2 \to_\alpha s_0$, s_0 being minimal.

The notion of *strong compression* as implemented in the kappa simulator corresponds to compression modulo α where α abstracts away the identity of the agents targeted by an event. More precisely, α maps a story to a graph with one node for each of its events whose edges correspond to the precedence relation on them. However, the node associated to an event e is not labelled with e itself, in which case we would get *weak compression*, but by the rule it instantiates.

Note that many other relevant notions of compression can be defined with this formalism. For instance, any equivalence relation being given over the set E of possible events, we could label each node of an abstract story with the equivalence class of the event it corresponds to. In particular, we could define a notion of *filtered compression* where, compared to strong compression, only the identity of the agents of some type would be abstracted away. Finally, it is possible to abstract away the precedence order over the events of a story and to use another relation to define the edges of its abstract counterpart. This will be very useful in Section 4 when generating stories statically.

3. Rectangular systems

In this section, we study an important class of event systems we call *rectangular*. Most concepts defined previously admit a nice algebraic characterization in this setting and efficient algorithms can be provided that do not necessarily exist in general. An event system (Q, E, C) is said to be *rectangular* if:

- A state is a *valuation* of a set of *state variables*. That is, there exists a set of variables $\mathcal{X}$ and for each variable $x \in \mathcal{X}$ a set V_x of values it can take such that $Q = \prod_{x \in \mathcal{X}} V_x$.
- All contexts of C along with the sets $\mathsf{pre}^\flat(e)$ and $\mathsf{eff}^\flat(e)$ for $e \in E$ can be encoded as *partial valuations* of $\mathcal{X}$.

Before we provide a rigorous definition, we have to discuss this concept of a partial valuation.

3.1. Partial valuations

A partial valuation of $\mathcal{X}$ is a set of *traits* over $\mathcal{X}$, a trait being an element of $\bigcup_{x\in\mathcal{X}} \{x\} \times V_x$. A partial valuation φ can be interpreted both as a set of equality *tests*, a trait (x, v) standing for the constraint $x = v$, or as a set of *assignments*, (x, v) standing for $x := v$. In the first case, it denotes the set of states $[\,\varphi\,] = \{q \in Q : \forall (x, v) \in \texttt{val},\ q_x = v\}$. We provide the following operators for manipulating partial valuations:

- If φ is a partial valuation, we write $\overline{\varphi}$ its *support*, that is the set $\{x : (x, v) \in \varphi\}$ of the variables constrained in φ.
- A partial valuation is said to be *contradictory* if it contains two contradictory constraints, that is both (x, v) and (x, v') for $v \neq v'$.
- If $\varphi_1 \supseteq \varphi_2$, we say that φ_1 *implies* φ_2. Beware that $\varphi_1 \supseteq \varphi_2 \iff [\varphi_1] \subseteq [\varphi_2]$
- $\varphi_1 \cup \varphi_2$ is the *concatenation* of φ_1 and φ_2. Beware that $[\varphi_1 \cup \varphi_2] = [\varphi_1] \cap [\varphi_2]$
- φ_1 and φ_2 are said to be *compatible* if their concatenation is not contradictory. Then, we write $\varphi_1 \uparrow \varphi_2$.
- They are said to be *incompatible* otherwise, in which case we write $\varphi_1 \downarrow \varphi_2$.
- If ψ is a set of variables, then $\varphi \setminus \psi$ is defined as $\{(x, v) \in \varphi : x \notin \psi\}$. In particular, $\varphi_1 \setminus \overline{\varphi_2}$ is the set of constraints in φ_1 which involve variables that are unconstrained in φ_2.
- Finally, we define the *update* of φ_1 *by* φ_2 as: $\varphi_1 \,!\, \varphi_2 = \varphi_2 \cup (\varphi_1 \setminus \overline{\varphi_2})$

We write $\mathsf{pval}(\mathcal{X})$ the set of all non-contradictory partial valuations of $\mathcal{X}$. We're now able to define rectangular systems rigorously.

3.2. Definition and properties of rectangular systems

An event system (Q, E, C) is said to be *rectangular* if there exists a set of variables $\mathcal{X}$ such that Q is the set of valuations of $\mathcal{X}$ and C is the set of all $[\,c\,]$ for c a non-contradictory partial valuation of $\mathcal{X}$. Moreover, for each event $e \in E$, there has to exist $\theta, \mu \in \mathsf{pval}(\mathcal{X})$ such that $\mathsf{pre}^\flat\,(e) = [\,\theta\,]$ and $\mathsf{post}^\flat{}_{[c]}(e) = [\,c \,!\, \mu\,]$ for all $c \in \mathsf{pval}(\mathcal{X})$. We write θ and μ $\mathsf{pre}(e)$ and $\mathsf{eff}(e)$ respectively. Moreover, we write $\mathsf{post}(e) = \mathsf{pre}(e) \,!\, \mathsf{eff}(e)$.

Using the formalism of partial valuations, we can state and prove an algebraic characterization of concurrency.

Theorem 2. *Let e_1 and e_2 two events in a rectangular system. Then, e_1 and e_2 are non-trivially concurrent if and only if the following conditions hold:*

$$a.\quad \overline{\mathit{pre}\,(e_1)} \cap \overline{\mathit{eff}\,(e_2)} = \emptyset \qquad b.\quad \overline{\mathit{pre}\,(e_2)} \cap \overline{\mathit{eff}\,(e_1)} = \emptyset$$
$$c.\quad \mathit{pre}\,(e_1) \uparrow \mathit{pre}\,(e_2) \qquad d.\quad \mathit{eff}\,(e_1) \uparrow \mathit{eff}\,(e_2)$$

Moreover, they are trivially concurrent if and only if $\mathsf{pre}(e_1) \downarrow \mathsf{pre}(e_2)$, $\mathsf{post}(e_1) \downarrow \mathsf{pre}(e_2)$ and $\mathsf{post}(e_2) \downarrow \mathsf{pre}(e_1)$.

A rectangular event system (Q, E, C) is said to be *regular* if $\overline{\mathsf{eff}(e)} \subseteq \overline{\mathsf{pre}(e)}$ for any event $e \in E$. In other words, every variable which is modified by an event has to be tested first. Kappa models are not regular in general but it can be argued that biologically relevant ones are. One reason for this is that the kinetics of a chemical reaction modifying a protein residue should depend on the previous state of the latter. This is especially true in the particular case where the residue is already in the state the rule would set it to. By the way, the kappa simulator emits a warning when encountering a rule like $A(x) \to A(x_p)$ although it's perfectly valid kappa. A remarkable property of regular systems is the following:

Theorem 3. *In a regular system, the strong similarity relation $\sim$ is identical to the Mazurkieviecz equivalence relation $\simeq_c$ in any context c.*

3.3. The case of Kappa

In this section, we consider a kappa model and show how it is possible to build a rectangular event system (Q, E, C) from it.

3.3.1. States

Before we define events, we have to decide upon a set of states of the world Q. It is natural to think about Q as the set of all mixtures whose agents respect their type signature and are given unique identifiers. However, in order to fit the formalism of rectangular systems, such mixtures have to be encoded in a series of state variables.

We introduce a set I of agent identifiers. For practical reasons, we want to be able to extract the type of an agent from its identifier through a map: $\tau : I \to T$ where T stands for the set of agent types. Therefore, a natural choice for I is $T \times \mathbb{N}$ but other encodings are possible. We call *port* a tuple $\langle u, x \rangle$ where u is an agent identifier and x the name of a site which is present in the signature of $\tau(u)$. Finally, the state of the agents of a mixture is described by the following set of variables:

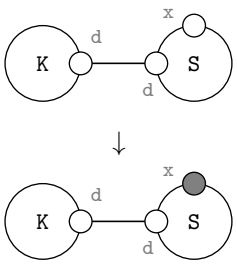
- For each agent identifier u, a variable $u_\exists$ is introduced which is `true` if u is in the mixture, `false` if it was once but was deleted, and `null` otherwise. It is important to distinguish between `false` and `null` so that an event can't give an identifier that had been used in the past to a newly created agent.

- For each port x, a variable x_ι is introduced, which describes its *internal state*. For instance, it might be `p` for phosphorylated or `u` for unphosphorylated. It is `null` when the agent x belongs to is not in the mixture.

- For each port x, a variable x_λ is introduced which describes its *linking state*. The latter can take three types of values: `free` if x is not bound, `bound`(y) where y is a port, or `null` if the agent x belongs to is not in the mixture.

For $J \subseteq I$, we write $\mathcal{X}_J$ the set of all these variables for the agent identifiers of J. Then, we define Q as the set of valuations over $\mathcal{X}_I$. Despite the fact that I and so $\mathcal{X}_I$ are typically infinite, any valuation describing a kappa mixture has only a finite number of non-null values.

$K(d^1), S(d^1, x_u) \;\rightarrow\; K(d^1), S(d^1, x_p)$



$\mathsf{ag}(\mathtt{px}) = \{k,\ s\}$

Precondition

$k_{\exists}$	=	`true`
$s_{\exists}$	=	`true`
$\langle k, d\rangle_{\lambda}$	=	`bound`$\langle s, d\rangle$
$\langle s, d\rangle_{\lambda}$	=	`bound`$\langle k, d\rangle$
$\langle s, x\rangle_{\lambda}$	=	`free`
$\langle s, x\rangle_{\iota}$	=	`u`

Effect

$\langle s, x\rangle_{\iota}$	:=	`p`

Figure 2: Example of encoding of the rule `px` of Figure 1.

3.3.2. Rules and events

A rule r is given by a set of agent identifiers $\mathsf{ag}(r) \subseteq I$ and two partial valuations of $\mathcal{X}_{\mathsf{ag}(r)}$ which are $\mathsf{pre}(r)$ and $\mathsf{eff}(r)$. An example is given Figure 2. An event is defined as a tuple $\langle r, \varphi\rangle$ where r is a rule and $\varphi : \mathsf{ag}\, r \to I$ an *instantiation map*. Note that φ has to be injective and to preserve the type of agents in the sense that $\forall a \in \mathsf{ag}(r),\ \tau(\varphi(a)) = \tau(a)$. If v is a partial valuation of $\mathsf{ag}(r)$, we write $\varphi(v)$ the partial valuation of I we get after renaming each agent identifier in it with φ. Then, it is possible to define the effect and the precondition of an event as:

$$\mathsf{pre}(\langle r, \varphi\rangle) = \varphi(\mathsf{pre}(r)) \qquad \mathsf{eff}(\langle r, \varphi\rangle) = \varphi(\mathsf{eff}(r))$$

For convenience, we take the initial conditions of a model into account by introducing an `init` rule which creates the initial mixture from scratch. Then, a trace or a configuration is said to be *valid* if it is in the context $\{\epsilon\}$ with ϵ the empty mixture where all variables are set to `null`. As a consequence, traces usually begin with an event instantiating `init`. Finally, a trait (x, v) is said to be *non-null* if and only if $v \neq \mathtt{null}$.

It is possible to generate stories from simulation traces. Indeed, for each instance of the observable in the simulator output, we can turn the trace leading to it into a story by computing its precedence relation thanks to Theorem 2 and then possibly use some compression algorithms. We show how the problem of weak-compression can be reduced to MINSAT in section 3.4. Then, in section 4, we introduce a completely different technique to generate stories statically, that is without using any stochastic simulation.

3.4. The weak-compression algorithm

In this section, we introduce a weak-compression algorithm for regular models. For convenience and efficiency, it is defined at the level of traces but we can show using the regularity hypothesis

that it gives the same result on two strongly similar traces in the sense it comes down to solving the same MINSAT instance. Thus, it can be defined on a story from any of its trajectories.

Let $t = e_0, \ldots, e_n$ a trace, e_0 being an instance of the `init` rule and e_n its unique observable event. We want to find a minimal valid subtrace of t which contains e_n. This problem is reducible to the MINSAT problem of finding a valuation satisfying a boolean formula with a minimal number of positive variables. Indeed, let's consider the variables $k_0, \ldots, k_n$, each k_i being interpreted as *"we keep e_i in the trace"*. For each boolean k_i and each variable x such that $(x, v) \in \mathsf{pre}(e_i)$ and $v \neq \texttt{null}$, we introduce the clause $C_{i,x,v}$ defined as follows:

$$k_i \implies \bigvee_{j \in A} (k_j \wedge \bigwedge_{l \in B_j} \neg k_l)$$

where $A = \{j \ : \ j < i \ \wedge \ (x, v) \in \mathsf{eff}(e_j)\}$ and $B_j = \{l \ : \ j < l < i \ \wedge \ (x, v') \in \mathsf{eff}(e_l), \ v' \neq v\}$ for each $j \in A$. Each valuation of the $\{k_i\}_i$ defines a subtrace of t. The latter is valid and contains e_n if and only if $(\bigwedge_{i,x,v} C_{i,x,v}) \wedge k_n$ is true. Therefore, compressing t comes down to finding a valuation of this formula with a minimal number of positive variables, which is an instance of MINSAT. Conversely, weak-compression is a NP-hard problem. An example of weak-compression is provided Appendix D.

4. Generating stories statically

We introduce in this section an algorithm to compute stories statically, that is without running any simulation. Like most static analysis techniques, it abstracts away the kinetics properties of a model and focuses on its mechanisms. As a consequence, many stories it produces are very unlikely to appear in a simulation trace. This behaviour is interesting in some situations where biocurators care about *sleeping* secondary pathways that become more active when a perturbation is introduced in the model. Producing unlikely stories might help spotting such perturbations. In other situations where only frequent stories are relevant, it is still possible to run simulations in order to annotate statically generated stories with frequency informations.

Besides, abstracting away kinetics gives kappa much smoother semantics properties that make reasoning over underspecified models easier. For instance, adding a rule to a model cannot discard a story although it can critically decrease its frequency by introducing competition effects. More generally, purely mechanistic properties are more robust regarding model composition. Therefore, static analysis techniques are especially relevant in the context of assembling models from nuggets of knowledge mined from litterature, which is a critical step towards making rule-based modelling widely used.

Finally, the techniques we introduce in this section can be easily adapted to the problem of finding stories respecting a user-defined property and they are best suited for interactive reasoning, as there are many ways a human agent can assist the computer in its exploration of the search space.

4.1. Problem statement

Most models admit an infinite set of stories. Although an algorithm might be able to generate a finite representation of such a set, we see no structure in it that could enable such a representation

to exist and we're not especially optimistic that there is one.

A natural way to circumvent this is to generate only the set of all minimal stories for some compression relation. With weak-compression, the set of minimal stories may still be infinite but it cannot in the case of strong-compression, as a consequence of Higman's theorem on well-quasi-orderings. Unfortunately, the problem of computing the set of all strongly compressed stories of a model is undecidable. Indeed, it is more difficult than the *reachability* problem which consists in deciding whether or not a given rule can belong to a valid trace. This last problem is undecidable itself as a consequence of the Turing-completeness of kappa.

In the following presentation of our algorithm, we get rid of termination problems by limiting the size of the generated stories.

4.2. Overview

Our algorithm doesn't generate stories directly but rather generates some abstract stories we call *local* stories. Then, it is easy to get concrete stories from them through a concretization operator. This abstract domain of local stories is especially designed to make the search process fast and non-redundant. Moreover, it admits a nice characterization of *valid* abstract stories, that is abstract stories s such that $\alpha^{-1}(\{s\}) \neq \emptyset$, and this enables us to cut early the search branches that contain none of them.

Here, we introduce a generic branch and cut exploration algorithm we use to find valid abstract stories. It is parameterized by some primitives whose implementation is given section 4.5. Let A a set of abstract stories and $A^* \subseteq A$ the set of valid ones. The algorithm features a set of disjoint sets $S = \{S_i \subseteq A\}_i$ sorted in a priority queue such that the following invariant holds:

$$A^* \subseteq \bigcup_i S_i \cup F$$

where $F \subseteq A^*$ is the set of the valid abstract stories that have already been found. It starts with $S = \{A\}$ and $F = \emptyset$ and terminates eventually when $S = \emptyset$ in which case $F = A^*$. At each step of the algorithm, an element of lowest cost is picked in S and removed from it, call it S_i. This element passes through a dual closure operator p with the following properties for $X, Y \subseteq A$:

$$p(X) \subseteq X \qquad p(X) \cap A^* = X \cap A^* \qquad (p \circ p)(X) = p(X) \qquad X \subseteq Y \Rightarrow p(X) \subseteq p(Y)$$

Then, three situations can occur:

1. We have $p(S_i) = \{a\}$ with $a \in A^*$ in which case a is added to F.
2. We can prove $p(S_i) \cap A^* = \emptyset$, in which case nothing is done.
3. Otherwise, $S := S \cup b\,(\,p\,(S_i)\,)$, with b a *branching* function partitioning $p(S_i)$ into a set of sets of strictly lower cardinal.

In the case A is finite, the algorithm is guaranteed to terminate. Indeed, let's assume A is of cardinal n. Then the vector $(v_1, \ldots, v_n)$ with $v_i = |\,\{k : |S_k| = i\}\,|$ decreases strictly for the lexicographic order at each iteration.

4.3. Local stories

A local story is a tuple (N, A) where:

- N is a set of *nodes*, each node i being labelled by a rule r_i.
- A is a set of *arrows*, each arrow connecting an assignment of a node to a test of another one. More formally, an arrow a is defined by:
  - Its source $\mathsf{src}(a) = \langle i, \mu \rangle$ with i a node and $\mu \in \mathsf{eff}(r_i)$.
  - Its destination $\mathsf{dst}(a) = \langle j, \theta \rangle$ with j a node and $\theta \in \mathsf{pre}(r_j)$.

  The source node i of a is written $\mathsf{srcn}(a)$ and its destination node j is written $\mathsf{dstn}(a)$. Finally, we write $\mathsf{srcv}(a)$ the variable modified by μ and $\mathsf{dstv}(a)$ the one tested by θ.

Contrary to a story whose nodes are labelled by events, the nodes of a local story are labelled by rules. However, for each test of each node, we keep track of the exact assignment that is responsible for its success.

4.3.1. Local stories as abstract stories

It is possible to map a valid trace $t = e_0, \ldots, e_n$ into the local story $l(t)$ defined as follows:

- For each event $e_i = \langle r_i, \varphi_i \rangle$ in t, we introduce a node i labelled by r_i.
- For each $i < j$ and each $(x, v) \in \mathsf{eff}(e_i) \cap \mathsf{pre}(e_j)$ such that x is not reassigned by any event between e_i and e_j, we introduce an arrow from $\langle i, \varphi_i^{-1}(x, v) \rangle$ to $\langle j, \varphi_j^{-1}(x, v) \rangle$, where $\varphi^{-1}(t)$ replaces any global agent identifier appearing in t by its local counterpart using φ^{-1}.

In the case of regular models, $t \sim t' \Rightarrow l(t) = l(t')$. Therefore, it is possible to define an abstraction (A, α) with A the abstract domain of local stories and, for all story s, $\alpha(s) = l(t)$ with t any one of its trajectories. In the next section, we introduce a necessary and sufficient condition for an abstract story a to be valid along with a characterization of $\alpha^{-1}(\{a\})$.

4.3.2. A characterization of validity

A first necessary condition for an abstract story to be valid is that there is exactly one incoming arrow towards each non-null test of each node. In this case, it is said to be *complete*. Moreover, the arrows have to be *well-typed* in the sense that they have to connect compatible variables. For instance, a binding assignment can't be connected to a phosphorylation test. More formally, for each arrow a with $\mathsf{src}(a) = \langle i, \mu \rangle$ and $\mathsf{dst}(a) = \langle j, \theta \rangle$, there has to exist two instantiation maps φ_1 and φ_2 such that $\varphi_1(\mu) = \varphi_2(\theta)$. This is true if and only if $\tau(\mu) = \tau(\theta)$ where $\tau(t)$ is defined as the *type* of a trait t by replacing each agent identifier i by $\tau(i)$ in t. Moreover, in the specific case of kappa, we have to ensure that for each arrow that explains a test of the form $\langle x \rangle_\lambda = \mathtt{bound} \langle y \rangle$, a symmetric arrow explains $\langle y \rangle_\lambda = \mathtt{bound} \langle x \rangle$ too.

In an abstract story s, agents are local to each node. Therefore, in order to build traces from s, it is necessary to map each local agent of each node into a global entity so each rule can be instantiated into an event. Thus, suppose we have a map $\pi : N \times I_L \to I_G$ with N the set of

nodes of s, I_L its set of local agent identifiers and I_G a set of global agent identifiers. Then, let's consider the set of events $E_\pi(s) = \{\langle r_i, \pi(i, \cdot)\rangle \,:\, i \in N\}$. Two necessary conditions on π for $E_\pi(s)$ to be the set of events of a trace t such that $(\alpha \circ \mathcal{C})(t) = s$ are the following:

1. Two different agents of a same node can't be mapped into the same global entity:

$$\forall i, x, y,\ \ \pi(\langle i, x\rangle) = \pi(\langle i, y\rangle) \implies x = y$$

2. For each arrow a, the local agent featured by the source of a and the one featured by its destination have to be mapped to the same global entity:

$$\pi(\langle \mathsf{srcn}(a),\, (\mathsf{ag} \circ \mathsf{srcv})(a)\rangle) = \pi(\langle \mathsf{dstn}(a),\, (\mathsf{ag} \circ \mathsf{dstv})(a)\rangle)$$

   where $\mathsf{ag}(v)$ maps a variable to the agent it describes for any variable v.

Now, suppose that these two conditions are met. By hypothesis, $\pi(\mathsf{srcn}(a), \cdot)(\mathsf{srcv}(a))$ and $\pi(\mathsf{dstn}(a), \cdot)(\mathsf{dstv}(a))$ are equal for each arrow a and we write them $\mathsf{var}_\pi(a)$. We call $\mathsf{var}_\pi(a)$ the *global variable* featured by a. Let $\preceq$ a total order on $E_\pi(s)$ and t the unique trace whose events are in $E_\pi(s)$ and are ordered by $\preceq$. Two necessary conditions on $\preceq$ so that $(\alpha \circ \mathcal{C})(t) = s$ are the following:

1. For each arrow a, $\mathsf{srcn}(a) \prec \mathsf{dstn}(a)$.

2. For each triple of distinct nodes (x, y, z) such that there exists an arrow a such that:

$$x = \mathsf{srcn}(a) \ \wedge\ y = \mathsf{dstn}(a) \ \wedge\ \mathsf{var}_\pi(a) \in \overline{\pi(z, \cdot)(\mathsf{eff}(r_z))}$$

   we have $z \prec x \ \vee\ z \succ y$.

When these two conditions hold, we say that $\preceq$ is a *valid π-scheduling* of s. An example of an abstract story which admits no valid π-scheduling is provided Figure 3. This leads us to our main theorem.

Theorem 4. *Let s an abstract story with well-typed arrows, N its set of nodes and I_L its set of local agent identifiers. Let $\equiv$ the smallest equivalence relation on $N \times I_L$ such that for every arrow a of s:*

$$\langle \mathsf{srcn}(a),\, (\mathsf{ag} \circ \mathsf{srcv})(a)\rangle \equiv \langle \mathsf{dstn}(a),\, (\mathsf{ag} \circ \mathsf{dstv})(a)\rangle$$

Let $I_G = (N \times I_L \,/\, \equiv)$ and π the canonical projection from $N \times I_L$ into I_G. Then s is valid if and only if:

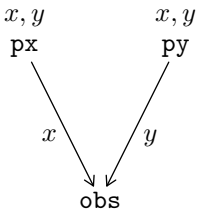
1. *s is complete*
2. *π maps different agents of a same node of s into different global entities*
3. *s admits a valid π-scheduling $\preceq$*

Moreover, if s is valid, then $\alpha^{-1}(\{s\})$ contains only one story (modulo renaming of agents) that is equal to $\mathcal{C}(t)$ for t any trace whose events are in $E_\pi(s)$ and are oredered by $\preceq$.

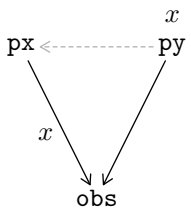
Note that the last part of the theorem only holds for regular models, in which case it suggests a straightforward concretization operator.

Let's consider three rules px, py and obs and an abstract story s featuring a node for each one of them – among other things that are not shown here:

$$\begin{array}{lcll} \texttt{px} & : & A(x_u, y_p) & \to \quad A(x_p, y_u) \\ \texttt{py} & : & A(x_p, y_u) & \to \quad A(x_u, y_p) \\ \texttt{obs} & : & A(x_p, y_p) & \to \quad A(x_p, y_p) \end{array}$$



Let's suppose that the phosphorylation of the site x of the unique agent featured by obs is due to px and that the phosphorylation of the site y of the same agent is due to py, which gives us two arrows. These arrows make all the three agents featured by each node equivalent for $\equiv$, and so they are all mapped into the same global instance by π, let's write it a.
On the diagram above, we write x the global variable $\langle a, x\rangle_\iota$ and y the global variable $\langle a, y\rangle_\iota$. We write next to each node the set of global variables it modifies and each arrow a is labelled by $\mathsf{var}_\pi(a)$. We show below that s does not admit a valid scheduling, which implies that s is not valid.



There is an arrow from py to obs:

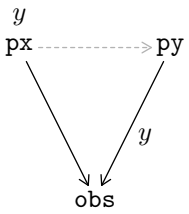
$$\texttt{py} \prec \texttt{obs}$$

The node py modifies a variable that is featured by an arrow going from px to py:

$$(\texttt{py} \prec \texttt{px}) \ \vee \ (\texttt{py} \succ \texttt{obs})$$

Combining these two clauses:

$$\texttt{py} \prec \texttt{px}$$



With a symmetric reasoning, we can show that:

$$\texttt{px} \prec \texttt{py}$$

which is a contradiction. As a consequence, s is not valid.

Figure 3: An example of impossible scheduling

4.4. Partial abstract stories

In the high-level description of the search algorithm of section 4.2, we manipulated sets of abstract stories. These sets are encoded by *partial abstract stories*.

4.4.1. Definition

Let s and s' two abstract stories whose sets of nodes are N and N'. An injective map $f : N \to N'$ is said to be an *embedding* of s into s' if it preserves both the nodes labels and the arrows of s.

A rooted abstract story is a tuple (s, ω) where s is a connex abstract story and ω one of its nodes, usually the observable. We say that (s, ω) is embedded into (s', ω') if there exists an embedding f of s into s' such that $f(\omega) = f(\omega')$. Note that as s and s' are connex, such an embedding is unique if it exists.

A *partial abstract story* is a triple (s, ω, c) such that (s, ω) is a rooted abstract story and c is a set of *alternatives*. An alternative a is given by:

- A *destination* $\mathsf{dst}(a) = \langle i, \theta \rangle$ where i is a node of s and $\theta \in \mathsf{pre}(r_n)$, like in the definition of the destination of an arrow.
- A set of *possible sources* $\mathsf{src}^\star(a)$, a possible source being a tuple $\langle \hat{n}, \mu \rangle$ where $\hat{n}$ is either:
  - an existing node of s labelled by a rule r
  - the special value $\texttt{new}\,(r)$ with r a rule

  and $\mu \in \mathsf{eff}(r)$.

Intuitively, an alternative describes a set of possible arrows with the same destination but different sources. These possible sources can feature nodes that do not exist yet in s but whose associated rule is known.

An alternative a is said to be *empty* if $\mathsf{src}^\star(a) = \emptyset$ and it is said to be resolved if $\mathsf{src}^\star(a)$ is a singleton. Finally, if $p = (s, \omega, c)$ is a partial abstract story, we write $\underline{p} = s$ its underlying abstract story and $\mathsf{alt}(p) = c$ its set of alternatives.

4.4.2. The extension relation

We say that a rooted abstract story (s', ω') extends a partial abstract story (s, ω, c) when the following conditions hold:

- There exists an embedding f from (s, ω) into (s', ω').
- For each $a \in c$ with $\mathsf{dst}(a) = \langle m, \theta \rangle$, there exists $\langle \hat{n}, \mu \rangle \in \mathsf{src}^\star(a)$ such that there is an arrow in s' whose destination is $\langle f(m), \theta \rangle$ and whose source is:
  - $\langle n, \mu \rangle$ for n a node of s' labelled with r which is not in $f(N)$ if $\hat{n} = \texttt{new}\,(r)$, with N the set of nodes of s
  - $\langle f(\hat{n}), \mu \rangle$ otherwise

This can be used to represent sets of rooted abstract stories with partial abstract stories. Indeed, for s_p a partial abstract story, we define $[s_p]$ as the set of complete rooted abstract stories with well-typed arrows extending s_p.

4.5. The search algorithm

Now that we have introduced the notion of a partial abstract story, we can complete our description of the search algorithm introduced in section 4.2. All abstract stories are implicitly rooted in their unique observable node and we use partial abstract stories to represent sets of them. In particular, the set S of section 4.2 is now regarded as a set of partial abstract stories.

4.5.1. Manipulating partial abstract stories

The following invariant is preserved during the search process:

Invariant 1. *For all $s \in S$, each non-null test of each node of s is the destination of an arrow or of an alternative (exclusively).*

As we see later, it guarantees that only complete abstract stories are generated. Moreover, no valid abstract story is forgotten as guaranteed by the invariant:

Invariant 2. $A^* \subseteq \bigcup_{s \in S}[s] \cup F$.

Suppose that we have a partial abstract story s with a resolved alternative, that is an alternative a such that $\mathsf{src}^\star(a)$ is a singleton. Then, it is natural to remove this alternative and add the unique arrow it suggests in s. This is indeed a part of what the propagation operator actually does. In the case where $\mathsf{src}^\star(a) = \{\langle \mathtt{new}(r), \mu\rangle\}$, adding such an arrow requires adding a new node n labelled by r. Then, a new alternative has to be added to s for each non-null test of r in order to preserve Invariant 1. Moreover, this whole operation should leave $[s] \cap A^*$ invariant so that Invariant 2 is preserved. For this to hold, these alternatives have to be kind of *exhaustive* but that's not sufficient. Indeed, if $\langle \mathtt{new}(r), \mu'\rangle \in \mathsf{src}^\star(a')$ for another alternative a', then $\langle n, \mu'\rangle$ has to be added to $\mathsf{src}^\star(a')$ too.

In order to make node creation easier to handle, we don't encode $\mathsf{alt}(s)$ as a set of alternatives. Instead, we define for each abstract story x a set $\mathcal{A}(x)$ such that:

$$\forall s,\ [s] \cap A^* \neq \emptyset \implies \mathsf{alt}(s) \subseteq \mathcal{A}(\underline{s})$$

and we encode instead its complementary set of *disqualified* possibilities $\mathsf{disq}(s)$ such that $\mathsf{alt}(a)$ is obtained from $\mathcal{A}(\underline{s})$ by:

- Removing in it all the alternatives a such that $\mathsf{dst}(a)$ is the destination of an arrow in $\underline{s}$.
- For all remaining alternative a, removing in $\mathsf{src}^\star(a)$ the elements t such that $(a, t) \in \mathsf{disq}(s)$.

It remains to define $\mathcal{A}$. For this, let's consider an abstract story s. If n is a node of s and t a trait, the *kind* of the tuple $\langle n, t\rangle$ is defined by $\kappa_s(\langle n, t\rangle) = \langle r_n, t\rangle$ with r_n the rule labelling n in s. For r a rule and $\theta \in \mathsf{pre}(r)$, we define $\mathsf{src\text{-}kinds}\,(\langle r, \theta\rangle)$ as the set of all tuples $\langle r', \mu\rangle$ for r' a rule and $\mu \in \mathsf{eff}(r')$ such that $\tau(\mu) = \tau(\theta)$. Then, for any valid abstract story $s \in A^*$ and a an arrow of s, we have:

$$(\kappa_s \circ \mathsf{src})(a) \in (\mathsf{src\text{-}kinds} \circ \kappa_s \circ \mathsf{dst})(a)$$

As a consequence, $\mathsf{src\text{-}kinds}(k)$ can be regarded as a superset of all the possible kinds of sources for an arrow whose destination has kind k in a valid abstract story. Then, for s an abstract story and N its set of nodes, $\mathcal{A}(s)$ is defined as the set of alternatives a with $\mathsf{dst}(a) = \langle j, \theta \rangle$ and:

$$\begin{array}{rcl} \mathsf{src}^\star(a) & = & \{ \quad \langle \mathtt{new}(r), \mu \rangle \quad : \qquad\qquad \langle r, \mu \rangle \in (\mathsf{src\text{-}kinds} \circ \mathsf{dst})(a) \quad \} \\ & \cup & \{ \qquad \langle i, \mu \rangle \qquad : \quad i \in N, \quad \langle r_i, \mu \rangle \in (\mathsf{src\text{-}kinds} \circ \mathsf{dst})(a) \quad \} \end{array}$$

for $j \in N$ and $\theta \in \mathsf{pre}(r_j)$.

4.5.2. Initialization of the algorithm

The algorithm is initialized with $S = \{s_0\}$ where s_0 is a partial abstract story with one node labelled by `obs` and no arrow and such that $\mathsf{disq}(s_0) = \emptyset$. Indeed, $A^* \subseteq [s_0]$ and Invariant 1 is respected by definition of disq.

4.5.3. Consistency checks

The search algorithm has to estimate at each iteration whether or not a partial story s is inconsistent, that is $[s] \cap A^* = \emptyset$. Of course, such a test is undecidable in general as the reachability problem can be reduced to it. Hopefully, a wide range of inconsistent partial stories can be recognized using Theorem 4. Indeed, if s is a partial abstract story such that either condition 2 or 3 of Theorem 4 does not hold for $\underline{s}$, then it is inconsistent.

In order to check whether or not we are in this case, we first compute the equivalence relation $\equiv$ for $\underline{s}$ with a union-find datastructure, whose `find` operation corresponds to the π map. Then, it is possible to check in quasi-linear time that no two agents of a same node are equivalent for $\equiv$. After this, we check that s admits a valid scheduling for π, which is a problem of satisfiability modulo the theory of partial orders. More specifically, there is one variable for each node of $\underline{s}$ and clauses are of the form $a \prec b$ or $(x \prec a) \vee (x \succ b)$. The number of clauses is cubic in the number of nodes of $\underline{s}$ in the worse case, but it is closer to be linear in pratice. Moreover, although the satisfiability test is exponential in the worse case, it is often linear as the high proportion of clauses with only one litteral makes constraint propagation very efficient. Finally, when a size limit for generated stories is given, partial abstract stories whose number of nodes exceeds this limit are considered inconsistent. Moreover, we may want that the number of nodes labelled by `init` does not exceed one.

4.5.4. Validity checks

An other test the search algorithm has to perform is whether or not $[s] = \{a\}$, $a \in A^*$ for s a partial abstract story. This one is especially easy as it holds if and only if s passes the consistency check described above and has an empty set of alternatives. Indeed, if s has an empty set of alternative, it is complete thanks to Invariant 1. We conclude using Theorem 4.

4.5.5. The p propagation operator

Let s a partial abstract story, $a \in \mathsf{alt}(s)$ and $t \in \mathsf{src}^\star(a)$. We write $\mathsf{choose}_{(a,t)}(s)$ the partial abstract story built from s by adding the arrow whose source is given by t and whose destination

is given by $\mathsf{dst}(a)$ to it, creating a new node if needed. Moreover, each time an arrow explaining a test of the form $\langle x \rangle_\lambda = \mathtt{bound}\langle y \rangle$ is added, the appropriate symmetric arrow explaining $\langle y \rangle_\lambda = \mathtt{bound}\langle x \rangle$ has to be added too. The value of disq does not need to be updated in $\mathsf{choose}_{(a,t)}(s)$, which is one reason this encoding of the set of alternatives is relevant.

The p operator takes a partial abstract story s and does two things to it:

1. For every alternative a of s, if a is empty, we mark s as inconsistent so it is further eliminated. If a is resolved, that is $\mathsf{src}^\star(a)$ is a singleton $\{t\}$, then s is updated into $\mathsf{choose}_{(a,t)}(s)$.

2. For each alternative a of s and $t \in \mathsf{src}^\star(a)$, if $\mathsf{choose}_{(a,t)}(s)$ is provably inconsistent, then we add (a, t) to $\mathsf{disq}(s)$. Here, we don't use the expensive consistency check introduced in section 4.5.3 but a constant-time version of it where scheduling constraints are generated only for the neighborhood of the new arrow.

These two operations are repeated until they leave s unchanged.

4.5.6. The b branching operator

Let s a partial abstract story, $a \in \mathsf{alt}(s)$ and $t \in \mathsf{src}^\star(a)$ two *well-chosen* values. Then, $b(s) = \{s_1, s_2\}$ where $s_1 = \mathsf{choose}_{(a,t)}(s)$ and s_2 is built from s by adding (a, t) to $\mathsf{disq}(s)$. Indeed, $\{[s_1], [s_2]\}$ is a partition of $[s]$. The choice of (a, t) is governed by a heuristic we discuss in section 4.5.7.

4.5.7. Cost of a partial story

On many real world models, the search space becomes so large that this algorithm is unlikely to generate the whole set of valid stories in a reasonable amount of time. However, when guided in its exploration by a good heuristic function, it may find the most interesting ones very quickly.

Heuristics play an important role in two parts of the algorithm: when picking an element of S and when branching. Indeed, each story of S is associated a cost and an element of lowest cost is picked at each iteration. Moreover, branching is usually done in a way that minimizes the cost of the first branch s_1. Many strategies are possible to assign costs to partial abstract stories but here are some basic principles:

- Penalizing node creation is usually a good greedy strategy to generate minimal stories first.
- Two arrows a and a' such that $\mathsf{var}_\pi(a) = \mathsf{var}_\pi(a')$ should share the same source as often as possible and partial abstract stories that are parcimonious in this sense should be assigned lower costs.
- If the user is confident about the branching strategy and he wants to get to the first story very quickly, a depth-first search is a good bet.
- Partial abstract stories that are *similar* to stories that have already been found can be penalized so that the first stories to be generated are more representative of the whole set of them.

4.6. Experimental results and prospects

A generic OCaml implementation of this algorithm is available at:

`https://github.com/jonathan-laurent/kappa-stories`

It was tested on the Kasim test suite provided at:

`https://github.com/Kappa-Dev/KaSim/tree/master/models/test_suite/cflows`

and asked to generate a single compressed story for each test case: each run took less than one second on average. However, in the current stage of its development, it is not well-suited for generating exhaustive sets of strongly compressed stories, even for small models and with a bound on the size of relevant stories.

Indeed, although the number of strongly compressed stories might be reasonably small for most models of interest, generating all of them would require to cut very early huge parts of the search space containing only uncompressed stories. No solution to this problem is currently implemented but we expect some progresses in a near future. Indeed, we are working on defining a notion of *compressible pattern* on partial abstract stories such that the following property holds:

Proposition. *If s is a partial abstract story such that $\mathsf{choose}_{(a,t)}(s)$ features a compressible pattern for any choice of (a,t), then $\alpha^{-1}([s])$ does not contain any compressed stories.*

Finally, the heuristic which is currently implemented is very naive – a depth-first search strategy penalizing the creation of nodes – but we're confident that some minor improvements in it will make our implementation scalable to much larger models.

5. Conclusion and future work

In this report, we introduced *stories* as formal counterparts to biological pathways and provided some generic techniques to generate them from rule-based models of protein-protein interaction networks. A lot of work remains to be done on causal analysis, in addition to what was mentioned in section 4.6. For instance, it would be interesting to break stories into logical blocks that would be larger than atomic events. Such blocks may consist in recurrent biological patterns corresponding to specific computational or assembling primitives. They could be defined *a priori* from well-known motifs or generated *a posteriori* by learning algorithms. Moreover, although stories may be considered useful *per se* for the intuitive insight they provide into the structure of a model when displayed as graphs, more subtle analysis techniques may be developped to extract useful informations from them. In particular, we could use causal analysis to highlight the behavioural differences between two variations of a same model.

References

[1] V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling, invited paper. In L. Caires and V.T. Vasconcelos, editors, *Proceedings of the Eighteenth International Conference on Concurrency Theory, CONCUR '2007, Lisbon, Portugal*, volume 4703 of *Lecture Notes in Computer Science*, pages 17–41, Lisbon, Portugal, 3–8 September 2007. Springer, Berlin, Germany.

[2] V. Danos, J. Feret, W. Fontana, and J. Krivine. Scalable simulation of cellular signaling networks, invited paper. In Z. Shao, editor, *Proceedings of the Fifth Asian Symposium on Programming Systems, APLAS '2007, Singapore*, volume 4807 of *Lecture Notes in Computer Science*, pages 139–157, Singapore, 29 November – 1 December 2007. Springer, Berlin, Germany.

[3] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, Jonathan Hayman, Jean Krivine, Christopher D. Thompson-Walsh, and Glynn Winskel. Graphs, rewriting and pathway reconstruction for rule-based models. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012*, volume 18 of *LIPIcs*, pages 276–288. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[4] Vincent Danos, Jérôme Feret, Walter Fontana, Russell Harmer, and Jean Krivine. Rule-based modelling, symmetries, refinements. In Jasmin Fisher, editor, *Proceedings of the First International Workshop, Formal Methods in Systems Biology, FMSB '2008*, volume 5054 of *Lecture Notes in BioInformatics*, pages 103–122, Cambridge, UK, 4–5 June 2008. Springer, Berlin, Germany.

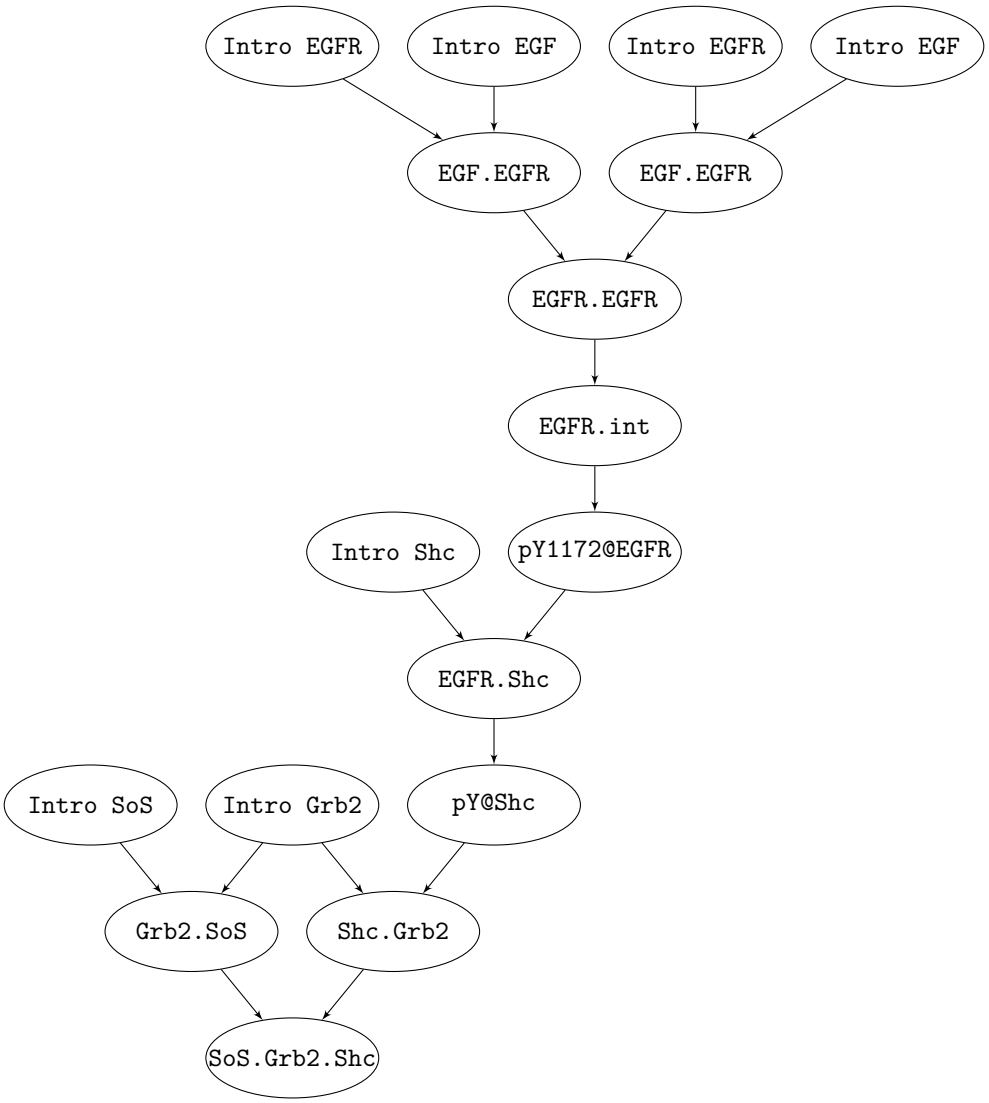
[5] John L. Mackie. *The Cement of the Universe: A study in Causation.* 1988.

Appendices

A. A kappa model of *SoS* recruitment

```
# Agent signatures
%agent: EGF(r)
%agent: EGFR(CR,C,N,L,Y1016~u~p,Y1092~u~p,Y1172~u~p)
%agent: SoS(PR,S~u~p)
%agent: Shc(Y~u~p,PTB~u~p)
%agent: Grb2(SH3c,SH3n,SH2~u~p)

# Constants
%var: 'Avogadro' 6.0221413E+23
%var: 'V'        4.2E-14
%var: 'k_on'     2.5E08 / ('Avogadro' * 'V') # molecule^-1 s^-1
%var: 'k_off'    2.5 # s^-1
%var: 'k_cat'    1 # s^-1

# Initial mixture
%init: 5000   EGF(r)
%init: 5000   EGFR(CR,C,N,L,Y1016~u,Y1092~u,Y1172~u)
%init: 10000  SoS(PR,S~u)
%init: 10000  Shc(Y~u,PTB~u)
%init: 5000   Grb2(SH3c,SH3n,SH2~u)

# Rules
'EGFR.EGFR'   EGF(r!2), EGFR(L!2,CR,N,C), EGF(r!3), EGFR(L!3,CR,N,C) <->
              EGF(r!2), EGFR(L!2,CR!1,N,C), EGF(r!3), EGFR(L!3,CR!1,N,C) @ 'k_on'/2, 'k_off'/2
'EGF.EGFR'    EGF(r), EGFR(L,CR) <-> EGF(r!1), EGFR(L!1,CR) @ 'k_on', 'k_off'
'Shc.Grb2'    Shc(Y~p), Grb2(SH2) -> Shc(Y~p!1), Grb2(SH2!1) @ 5*'k_on'
'Shc/Grb2'    Shc(Y~p!1), Grb2(SH2!1) -> Shc(Y~p), Grb2(SH2) @ 'k_off'
'EGFR.Grb2'   EGFR(Y1092~p), Grb2(SH2) <-> EGFR(Y1092~p!1), Grb2(SH2!1) @ 'k_on', 'k_off'
'EGFR.Shc'    EGFR(Y1172~p), Shc(PTB) -> EGFR(Y1172~p!1), Shc(PTB!1) @ 'k_on'
'EGFR/Shc'    EGFR(Y1172~p!1), Shc(PTB!1) -> EGFR(Y1172~p), Shc(PTB) @ 'k_off'
'Grb2.SoS'    Grb2(SH3n), SoS(PR,S~u) -> Grb2(SH3n!1), SoS(PR!1,S~u) @ 'k_on'
'Grb2/SoS'    Grb2(SH3n!1), SoS(PR!1) -> Grb2(SH3n), SoS(PR) @ 'k_off'
'EGFR.int'    EGFR(CR!1,N,C), EGFR(CR!1,N,C) -> EGFR(CR!1,N!2,C), EGFR(CR!1,N,C!2) @ 'k_on'
'EGFR/int'    EGFR(CR!1,N!2,C), EGFR(CR!1,N,C!2) -> EGFR(CR!1,N,C), EGFR(CR!1,N,C) @ 'k_off'
'pY1092@EGFR' EGFR(N!1), EGFR(C!1,Y1092~u) -> EGFR(N!1), EGFR(C!1,Y1092~p) @ 'k_cat'
'pY1172@EGFR' EGFR(N!1), EGFR(C!1,Y1172~u) -> EGFR(N!1), EGFR(C!1,Y1172~p) @ 'k_cat'
'uY1092@EGFR' EGFR(Y1092~p) -> EGFR(Y1092~u) @ 'k_cat'
'uY1172@EGFR' EGFR(Y1172~p) -> EGFR(Y1172~u) @ 'k_cat'
'pY@Shc'      EGFR(Y1172~p!1), Shc(PTB!1,Y~u) -> EGFR(Y1172~p!1), Shc(PTB!1,Y~p) @ 'k_cat'
'uY@Shc'      Shc(Y~p) -> Shc(Y~u) @ 'k_cat'

# The pattern we're interested in
%obs: 'SoS.Grb2.Shc' Grb2(SH2!1,SH3n!2),SoS(PR!2),Shc(Y~p!1)
```

B. A possible scenario for *SoS* recruitment



C. Demonstrations of all theorems

Theorem 1

Proof. It is easy to check that two traces differing only in the permutation of two concurrent consecutive events have the same precedence relation. By induction and using this fact, we can prove the first direction.

Conversely, let t a trace, $\preceq_t$ its precedence relation and σ a reordering preserving it. We will prove that $\sigma(t) \sim t$ by induction on the size of t. For this, we show that $e_{\sigma(1)}$ is concurrent with all the events of t from e_1 to $e_{\sigma(1)-1}$. Indeed, let's suppose $\neg\left(e_i \diamond e_{\sigma(1)}\right)$ for $1 \leq i < \sigma(1)$. Then :

$$\begin{array}{lll} & e_i \preceq_t e_{\sigma(1)} & \text{(by definition of the precedence relation)} \\ \therefore & e_{\sigma(\sigma^{-1}(i))} \preceq_t e_{\sigma(1)} & \text{(because } \sigma(\sigma^{-1}(i)) = i\text{)} \\ \therefore & \sigma^{-1}(i) \leq 1 & \text{(because } \sigma \text{ preserves } \preceq_t\text{)} \end{array}$$

Therefore, $\sigma^{-1}(i) \leq 1$ and then $i = \sigma(1)$, which is a contradiction. As a consequence, it is possible to set $e_{\sigma(1)}$ in first position in t by making it commute with all the events from e_1 to $e_{\sigma(1)-1}$. The trace obtained – call it t' – is strongly similar to t. We conclude by applying the induction hypothesis on t and t' without their first element. □

Theorem 2

The proof relies on the convention that, for any event e, $\mathsf{pre}(e) \cap \mathsf{eff}(e) = \emptyset$. This means that if an event has the test $x = v$ in its precondition, the assignment $x := v$ can't belong to $\mathsf{eff}(e)$. It is quite natural as such an assignment, if present, would be useless. The two parts of Theorem 2 are proved separately. Moreover, contexts are identified with their underlying partial valuation here.

Characterization of non-trivial concurrency:

Proof. Let's suppose the conditions *a.* to *d.* hold and take a context c.

- Let's prove that: $(c \vdash e_1 \ \wedge \ c \vdash e_2) \Rightarrow (c \vdash e_1, e_2 \ \wedge \ c \vdash e_2, e_1 \ \wedge \ \mathsf{eff}_c(e_1) = \mathsf{eff}_c(e_2))$.
  Suppose that $c \vdash e_1$ and $c \vdash e_2$. In particular, we have $c \vdash e_2$ and then $c \supseteq \mathsf{pre}(e_2)$. Combining this with *b.* it yields $c \,!\, \mathsf{eff}(e_1) \supseteq \mathsf{pre}(e_2)$ which implies $c \vdash e_1, e_2$. We can show $c \vdash e_2, e_1$ the same way using *a.* Finally, the fact that $\mathsf{eff}_c(e_1) = \mathsf{eff}_c(e_2)$ is a direct consequence of *d.*

- Then, let's prove that: $(c \vdash e_1, e_2) \Rightarrow (c \vdash e_1 \ \wedge \ c \vdash e_2)$.
  Suppose $c \vdash e_1, e_2$. Then, $c \vdash e_1$ holds obviously. Moreover, as $c \,!\, \mathsf{eff}(e_1) \supseteq \mathsf{pre}(e_2)$, using *b*, we get $c \supseteq \mathsf{pre}(e_2)$ and so $c \vdash e_2$.

- Symmetrically, we can show that $(c \vdash e_2, e_1) \Rightarrow (c \vdash e_1 \ \wedge \ c \vdash e_2)$.

Therefore, e_1 and e_2 are concurrent. They are non-trivially concurrent in the context $\mathsf{pre}(e_1) \cup \mathsf{pre}(e_2)$, which is coherent thanks to c.

Conversely, let's suppose that e_1 and e_2 are non-trivially concurrent. Then:

- Property $c.$ is true thanks to the *non-triviality* hypothesis.
- Property $d.$ is true because in a context c where $c \vdash e_1$ and $c \vdash e_2$, whose existence is guaranteed by the previous point, $\mathsf{eff}_c(e_1) = \mathsf{eff}_c(e_2)$.
- Property $a.$ is true. Indeed, Let's suppose $\overline{\mathsf{pre}\,(e_1)} \cap \overline{\mathsf{eff}\,(e_2)} \neq \emptyset$. Then, there are two possible cases:
  - $(x, v) \in \mathsf{pre}\,(e_1)$ and $(x, v) \in \mathsf{eff}\,(e_2)$. In this case, let's take a context c such that $c \vdash e_2, e_1$. Such a context exists thanks to the *non-triviality* hypothesis. Then, let's consider $c' = c \setminus \{x\} \cup \{(x, v')\}$ where $v' = \hat{v}$ if $(x, \hat{v}) \in \mathsf{pre}(e_2)$ or any value different from v otherwise. Then, $c' \vdash e_2, e_1$ but $c' \nvdash e_1$, which is a contradiction. Here, we use the facts that $(x, v) \notin \mathsf{pre}(e_2)$ as $\mathsf{pre}(e_2) \cap \mathsf{eff}(e_2) = \emptyset$ and that each variable can take at least two different values, so we can choose $v' \neq v$.
  - $(x, v) \in \mathsf{pre}\,(e_1)$ and $(x, v') \in \mathsf{eff}\,(e_2)$. This contradicts the *non-triviality* hypothesis which assures the existence of a context c such that $c \vdash e_2, e_1$.

This terminates the proof. □

Characterization of trivial concurrency:

Proof. The most difficult thing to prove here is that $\exists c,\ c \vdash e_1, e_2 \iff \mathsf{post}(e_1) \uparrow \mathsf{pre}(e_2)$. Suppose $\mathsf{post}(e_1) \uparrow \mathsf{pre}(e_2)$. Then:

$$\begin{aligned} & (\mathsf{eff}(e_1) \ \cup \ \mathsf{pre}(e_1) \setminus \overline{\mathsf{eff}(e_1)}) \uparrow \mathsf{pre}(e_2) \\ \therefore \quad & (\mathsf{pre}(e_1) \setminus \overline{\mathsf{eff}(e_1)}) \uparrow \mathsf{pre}(e_2) \\ \therefore \quad & \mathsf{pre}(e_1) \uparrow (\mathsf{pre}(e_2) \setminus \overline{\mathsf{eff}(e_1)}) \end{aligned}$$

thus the context $c = \mathsf{pre}(e_1) \ \cup \ (\mathsf{pre}(e_2) \setminus \overline{\mathsf{eff}(e_1)})$ is coherent. In this context, $c \vdash e_1, e_2$. The converse is easy. □

Theorem 3
See the paragraph before the proof of Theorem 2.

Proof. All comes down to the fact that for all context c, if e_1 and e_2 are non-trivially concurrent in context c, then they are non-trivially concurrent. Indeed, let c such a context.

- $\mathsf{pre}\,(e_1) \uparrow \mathsf{pre}\,(e_2)$ as $c \vdash e_1$ and $c \vdash e_2$
- $\mathsf{eff}\,(e_1) \uparrow \mathsf{eff}\,(e_2)$ as $\mathsf{eff}_c(e_1, e_2) = \mathsf{eff}_c(e_2, e_1)$
- $\overline{\mathsf{pre}\,(e_1)} \cap \overline{\mathsf{eff}\,(e_2)} = \emptyset$. Indeed, if this is wrong, two cases are possible:
  - $(x, v) \in \mathsf{pre}\,(e_1)$ and $(x, v) \in \mathsf{eff}\,(e_2)$. As we are working in a regular system, $x \in \overline{\mathsf{pre}(e_2)}$. Moreover, $(x, v) \notin \mathsf{pre}(e_2)$ by the convention introduced before the proof of Theorem 2. Therefore, $(x, v') \in \mathsf{pre}\,(e_2)$ for $v \neq v'$, which contradicts the fact that $c \vdash e_1 \ \wedge \ c \vdash e_2$.

- $(x, v) \in \mathsf{pre}\,(e_1)$ and $(x, v') \in \mathsf{eff}\,(e_2)$ for $v \neq v'$. This contradicts the fact that $c \vdash e_2, e_1$.

- $\overline{\mathsf{pre}\,(e_1)} \cap \overline{\mathsf{eff}\,(e_2)} = \emptyset$ can be shown symmetrically.

□

Theorem 4

Proof. We have already proved that conditions 1 to 3 are necessary for s to be valid. It remains to be proven that if they hold then $\mathcal{C}(t) \in \alpha^{-1}(\{s\})$ for any trace t whose events are in $E_\pi(s)$ and are ordered by $\preceq$ and that all elements of $\alpha^{-1}(\{s\})$ are equivalent modulo the renaming of their agents.

Suppose conditions 1 to 3 hold and let t a trace whose events are in $E_\pi(s)$ and are ordered by $\preceq$. The elements of $E_\pi(s)$ are well-defined thanks to condition 2 which guarantees the injectivity of each instantiation map $\pi(i, \cdot)$. Moreover, t is valid. Indeed, for each non-null test (x, v) of each event e of t, an other event e' sets x to the right value by condition 1. By condition 2, e' occurs before e and x is not modified between e' and e. Therefore, $\mathcal{C}(t)$ is a story and it is easy to show that it is abstracted into s.

Let t and t' two traces such that both $(\alpha \circ \mathcal{C})(t)$ and $(\alpha \circ \mathcal{C})(t')$ belong to $\alpha^{-1}(\{s\})$. We rename their agents so they have identifiers in $(N \times I_L \,/\, \equiv)$ as demonstrated in 4.3.2. Then, we show that $t \sim t'$ using Theorem 1 by proving that for all pair of events (e, e') such that e comes before e' in t and $\neg(e \diamond e')$, e comes before e' in t'. Indeed, let e and e' two events of t such that e comes before e' in t and $\neg(e \diamond e')$. We prove that e comes before e' in t' by induction on the distance between e and e' in t. By Theorem 2, we are in one of the following situations:

1. There exists $x \in \overline{\mathsf{eff}(e)} \cap \overline{\mathsf{pre}(e')}$. If an event e_m modifies x between e and e' in t, we conclude using the induction hypothesis on (e, e_m) and (e_m, e'). Otherwise, there is an arrow between the nodes corresponding to e and e' in s and then e comes before e' in t' by condition 3.

2. There exists $x \in \overline{\mathsf{pre}(e)} \cap \overline{\mathsf{eff}(e')}$. Similarly, we can suppose that x is not modified by any event between e and e'. Let e_c the last event modifying x before e. Then e' comes before e_c or after e in t' by condition 3. Moreover, by regularity, e' is testing x ans there is an arrow in s between e_c and e'. Therefore, e' has to come after e_c in t' and we can conclude.

3. We have $\mathsf{pre}\,(e) \downarrow \mathsf{pre}\,(e')$, in which case any variable x tested by e and e' with different values is modified by an event e_m between e and e' and we can use the induction hypothesis on (e, e_m) and (e_m, e').

4. We have $\mathsf{eff}\,(e) \downarrow \mathsf{eff}\,(e')$ and case 1 applies by regularity.

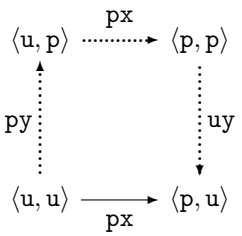
Thus, t and t' are strongly similar and then $\mathcal{C}(t) = \mathcal{C}(t')$. This terminates the proof.

□

D. An example of weak compression

Let's consider a model made of a single agent S with two phosphorylation sites `x` and `y` and three rules acting on it. It is possible to represent the four possible states of the system along with the transition corresponding to each rule on a square diagram:

$$
\begin{array}{lcll}
\texttt{px} & : & S(x_u) & \to \quad S(x_p) \\
\texttt{py} & : & S(x_u, y_u) & \to \quad S(x_u, y_p) \\
\texttt{uy} & : & S(x_p, y_p) & \to \quad S(x_p, y_u)
\end{array}
$$



Suppose the initial state is $S(x_u, y_u)$ and the observable state is $S(x_p, y_u)$. A trace leading to the observable is given by the dotted path (`init`, `py`, `px`, `uy`, `obs`). It can be compressed into the solid path (`init`, `px`, `obs`).

		$\langle s, x \rangle_\iota$		$\langle s, y \rangle_\iota$	
		?	**!**	?	**!**
0	`init`	×	**u**	×	**u**
1	`py`	u		u	**p**
2	`px`	u	**p**		
3	`uy`	p		p	**u**
4	`obs`	p		u	

$$
\begin{array}{lcl}
C_{4,x,\mathtt{p}} & : & (x_4 \Rightarrow x_2) \\
C_{3,x,\mathtt{p}} & : & (x_3 \Rightarrow x_2) \\
C_{2,x,\mathtt{u}} & : & (x_2 \Rightarrow x_0) \\
C_{1,x,\mathtt{u}} & : & (x_1 \Rightarrow x_0) \\
C_{4,y,\mathtt{u}} & : & x_4 \Rightarrow (x_3 \ \vee \ \neg x_1 \wedge x_0) \\
C_{3,y,\mathtt{p}} & : & (x_3 \Rightarrow x_1) \\
C_{1,y,\mathtt{u}} & : & (x_1 \Rightarrow x_0)
\end{array}
$$

In the table above, each column corresponds to a variable ($s_\exists$ is omitted) and is split into two parts: on the left are its tested values and on the right are its assigned values. The character × corresponds to the `null` value. Next to this table are the clauses corresponding to the two variables $\langle s, x \rangle_\iota$ and $\langle s, y \rangle_\iota$, abbreviated in x and y.