

Local Causal Reasoning in Multi-Agent Systems

Pinaki Chakraborty^{a,*}, Tristan Caulfield^{a,*}, David Pym^{a,b,*}

^a*University College London, United Kingdom*

^b*Institute of Philosophy, University of London, United Kingdom*

Abstract

Causal analysis of an asynchronous system becomes rather difficult when several agents control different parts of the system. A rule change made by one agent can propagate through other components, and its effect may depend on the initial configuration and the order in which components are updated. We study when the cause of an observed outcome can be established from a subsystem alone. In our model, an agent has authority to replace the update rules of specified components. In such a setting, an intervention installs replacement rules while leaving the current component values unchanged.

We consider causal diagnosis of recorded executions and verification of proposed repairs. To diagnose a recorded execution, we replay its update schedule from its initial configuration using replacement rules. We suppose the outcome refers only to components in an influence-closed subsystem, meaning that every input used by those components also lies in that subsystem. A rule replacement outside the subsystem cannot then affect the outcome, and every minimal execution-relative cause is contained within the subsystem. The result continues to hold when the subsystem contains feedback.

Repair verification has to consider every execution that may occur after a change. We suppose the system is split into two overlapping subsystems, with their shared components acting as an interface. If the interface remains fixed during the phase being analysed, the configurations reachable in the full system are exactly the combinations of configurations reachable in the two subsystems that agree on the interface. This means that properties of the unchanged subsystem can be verified once and then reused when checking repairs that affect only the other subsystem. If every component can be scheduled, the converse is true as well. Thus, recovering the global reachable set exactly, for every initial configuration, is possible only when the interface components remain unchanged.

For finite systems, we give algorithms for both forms of analysis and evaluate them on examples involving retry behaviour and retrieval-augmented generation.

*Corresponding authors.

Email addresses: `pinaki.chakraborty.22@ucl.ac.uk` (Pinaki Chakraborty),
`t.caulfield@ucl.ac.uk` (Tristan Caulfield), `david.pym@sas.ac.uk` (David Pym)

We also prove that finite replay can be represented exactly by an acyclic structural causal model. Finally, we establish bisimulation invariance for our logic and a Hennessy–Milner characterization over the relevant image-finite class.

Keywords: Logic, Transition systems, Distributed systems, Causality, Rule changes, Subsystem verification, Multi-agent systems, Retrieval-augmented generation

1. Introduction

Causal analysis of software and multi-agent systems often guides practitioners deciding which rule to change and which parts of the system to leave alone. Examples include retry policies in distributed services [1, 2], controller changes, and changes to rules in systems that combine machine-learning modules [3]. A structural causal model (SCM) uses *structural equations*, or functional dependencies, to represent causal dependencies. Such models support hard interventions, which fix variables to chosen values [4, 5], and soft interventions, which modify an equation and let the modified equation determine its output [6, 7]. We define these rule changes for systems whose components update one at a time and study when their effects can be analysed within one subsystem.

Each component has an observable behaviour, a specified set of inputs, and a rule that determines how its behaviour changes when the component is updated. In particular, each component c has a set $\mathbb{B}(c)$ of possible behaviours, and a configuration assigns one current behaviour to every component. The system evolves through a sequence of such updates, with a scheduler selecting one component at each step. We call the resulting sequence of configurations an *execution*, and the corresponding sequence of selected components its *schedule*. Throughout the paper, we use *configuration* for this assignment and reserve *state* mainly for established logical terminology, such as state formulas. An intervention replaces the rules of selected components without changing the current configuration, so a replacement rule has no effect until its component is subsequently scheduled.

Here, *multi-agent* means that different agents control different rules. A cause may span agents, as Proposition 6.5 shows, but a repair must be authorized by the agent controlling its target. We model neither utility nor strategic interaction.

We use rule replacement for two different purposes. In diagnosing a completed execution, we ask which of the rules that were in force caused a specified outcome at the end of that execution. The alternatives used for this diagnostic comparison need not be changes that an agent is actually authorized to make. In repair analysis, by contrast, an intervention is considered prospectively, and we ask whether the proposed rule change prevents the failure throughout the executions that can occur after it is installed.

When analysing a recorded execution, we replay it from the same initial configuration and under the same schedule, while allowing selected component rules

to be replaced. If the outcome depends only on components in an *influence-closed* region, meaning that every input read by a component in that region also comes from within it, then the replay can be confined to that region without changing the resulting local execution. Replacements outside the region cannot affect the final outcome, and every minimal cause of that outcome is contained within the region, even if the region itself contains feedback.

Repair verification is different because it concerns all executions that may occur after an intervention rather than the replay of one recorded execution, and requires stronger conditions on how the system is divided. We split the system into two overlapping subsystems whose shared components form an interface. If the interface remains fixed during the phase being analysed, the configurations reachable in the full system are exactly the combinations of configurations reachable in the two subsystems that agree on the interface. Consequently, when an intervention changes rules on only one side, a property established for the unchanged side can be reused rather than verified again. Moreover, when every component of the full system may be scheduled, this decomposition recovers the reachable configurations from every initial configuration only if scheduling the shared components leaves their values unchanged. Under these same conditions, any feedback loop must lie entirely within one non-interface side or entirely within the interface.

Our logic has operators for reachability, intervention, and restriction to a named subsystem. It includes a frame rule, which carries a proved property of an unchanged subsystem across several repairs to the other subsystem.

We call a cause *execution-relative* because it is tied to one finite recorded execution. A non-empty set of installed rules is a cause of a true-or-false claim about the final configuration if an allowed replacement makes that claim false under the same initial configuration and schedule. No non-empty proper subset may do so. Halpern–Pearl actual causation adds tests that hold other variables fixed. Our causal analysis is defined directly on replay in the transition system. For finite replays, an acyclic SCM with rule variables reproduces the same replacements.

The retry case study contains feedback between callers and the downstream service, so the recorded incident must be diagnosed in the full model. The rules used by callers during the incident explain that execution. The installation of a gateway cap prevents the failure on every schedule. In the RAG case study, retrieval has already committed the context used by generation. We check release policies on the generation side while reusing a property supplied at the retrieval boundary and preserved during the analysed phase.

The paper makes the following contributions.

1. We define components that update one at a time, specify who may replace their rules, and give a logic that checks whether each formula uses only the components available in its subsystem.
2. We prove that every minimal cause of an outcome stated using components in an influence-closed region lies in that region and contains only components

selected for update in the recorded sequence, which we call the schedule.

3. We prove that, across a stable interface, searches of the two subsystems recover every configuration the full system can reach. A converse shows when stability is necessary.
4. We show how a repair on one side of a stable interface preserves an unchanging property on the other. We also give counterexamples when the input requirements or stability fail.
5. We give two algorithms for finite systems, work through retry and RAG examples, and compare the numbers of configurations examined locally and globally.

For finite recorded executions, an acyclic structural causal model with rule variables represents the replay semantics exactly. The logic is invariant under the corresponding bisimulation, and the stated image-finite class satisfies a Hennessy–Milner converse. The locality theorems do not use either result.

Previous causal models examine how to abstract and combine smaller models [8, 9, 10]. Our transition system updates one component at a time. The conditions for local analysis refer to the inputs available to each rule and to whether shared components can change.

Section 2 relates the model to earlier work. Section 3 defines components, closed regions, cuts, and interventions. Sections 4 and 5 define authority and the logic. Section 6 studies causes of a recorded execution and proves the influence-closed locality result; it then validates the replay semantics by giving an equivalent SCM representation for finite executions. Section 7 proves the stable-interface factorization and applies it to repair checking. Section 8 gives two algorithms. Section 9 presents the RAG example, and Section 10 checks both finite examples. Section 11 proves bisimulation invariance and a Hennessy–Milner converse for the logical semantics.

2. Causal Models and Concurrent Systems

We use a transition system, which represents configurations and the possible updates between them. CSP, CCS, and the π -calculus describe how components interact, which executions are possible, and when systems have the same observable behaviour [11, 12, 13]. Models of distributed systems and ecosystems place similar emphasis on explicit components and boundaries [14, 15]. Resource-process calculi model the co-evolution of resources and processes and can specify that a system is a parallel composition of subsystems [16, 17]. Galmiche, Lang, and Pym give a general account of behaviours, components, the ways components realize their behaviour, and interfaces [18]. We retain that component view and add update rules that an intervention may replace. Here causation concerns the effect of changing an installed rule. In theories of concurrent systems, causality usually concerns dependency or precedence between events. Event structures, for example, represent which events depend on or exclude others [19].

These works provide points of comparison for determining how component-level information relates to the behaviour of an entire system. Rischel and Weichwald study abstractions between finite causal models and their behaviour under interventions [8]. Lorenz and Tull develop a diagrammatic treatment of causal models with exposed inputs and outputs [9]. Pruthi and Jensen examine causal-effect estimation in systems assembled from heterogeneous components [10]. We consider asynchronously evolving transition systems in which each transition updates exactly one component, and ask when subsystem executions determine global behaviour. In this setting, Theorems 6.9 and 7.4 characterize the conditions under which subsystem executions suffice to reconstruct a recorded global execution or recover its corresponding reachable set.

A separate line of work uses causal notions to explain failures of transition systems and other formal models. Beer et al. apply Halpern–Pearl-style causality to executions that violate a required property [20], and G"ossler and Le M"etayer identify blameworthy components from a single safety-violating execution [21]. Caltais, Leue, and Mousavi consider transition systems in which each step records the component responsible for the move; using Hennessy–Milner logic, they show that analyses of noncommunicating systems can be combined when the systems update in alternation [22]. G"ossler and Stefani extend fault attribution to concurrent executions and make explicit the choice of alternative executions against which causality is assessed [23]. For a broader account of causal explanation in formal verification, see the survey by Baier et al. [24].

These works provide different accounts of what may be varied or compared when evaluating causal alternatives. Parreaux, Piribauer, and Baier select alternative paths through a transition system according to similarity or distance criteria [25]. Dubslaff et al. consider features and their interactions as candidate causes in configurable software [26]. Tanhaei et al. embed SCMs into component-and-connector descriptions to diagnose causes and evaluate the effects of design changes [27]. The distinction from the latter approach concerns the role of the SCM: rather than taking an SCM as the starting point, we use it to represent a finite execution of the underlying transition system, with one variable per component recording the rule installed at that component. We construct alternative executions by replacing the installed rules, keeping the observed initial configuration and execution schedule fixed.

This construction also supports locality results. Corollary 6.10 shows that an influence-closed region contains every minimal cause of an outcome stated within that region. For future behaviour, Theorem 7.4 establishes that, when the interface between two subsystems remains unchanged, their separate analyses together recover the entire reachable set of the full system. Theorem 7.14 then allows repair conditions to be checked on one side of such a decomposition.

Decades of work on control and feedback has studied purposeful modification [28, 29]. In his influential books, Herbert A. Simon considered when a system can

be divided into parts whose internal interactions are stronger than their external ones [30, 31, 32, 33]. In our model, declared influence determines whether a region contains every input needed for its updates. Separate subsystem analyses recover every configuration of the full system when every interface rule leaves its component unchanged.

We use agents only to specify who may replace each rule. Agents do not otherwise change the system transitions. We do not address games derived from causal models [34] or information-security models of policy coordination [35, 36].

3. System Models

We model a system as a collection of components that update one at a time. Each component has a set of observable behaviours and a local rule for choosing its next behaviour. The rule may read the current behaviours of specified other components. We include internal state when it affects any of these observations or updates.

We extend the component model of Galmiche, Lang, and Pym [18] with rules that may be replaced. These rules allow a recorded execution to be replayed under alternatives without adopting a separate algebra for concurrent processes [16, 15, 14].

We represent time as a sequence of steps. A scheduler chooses one component at each step, and only that component is updated. Components that must update together may be treated as one component with a joint rule. We require every rule to state which other components it may read. In particular, a transaction manager that reads the entire system prevents a smaller analysis in Section 7. That analysis must include the complete system.

Section 4 defines agents and their authority to change rules. For now, let $\mathcal{C}$ be the component set and $\mathcal{B}$ a set of behaviours. A map $\mathbb{B} : \mathcal{C} \rightarrow 2^{\mathcal{B}}$ assigns a non-empty set $\mathbb{B}(c)$ of possible behaviours to each $c \in \mathcal{C}$. A configuration records one current behaviour for every component.

Definition 3.1 (Configuration). A *configuration* over $\mathcal{C}$ is a function defined for every component, $f : \mathcal{C} \rightarrow \mathcal{B}$ with $f(c) \in \mathbb{B}(c)$ for every $c \in \mathcal{C}$. The set of all configurations is $F_{\mathcal{C}}$. When $\mathcal{C}$ is clear, we write F . $\square$

For each component, we record which other components its rule may read.

Definition 3.2 (Inputs and update rules). For each $c \in \mathcal{C}$, its *input set* $\text{Inf}(c) \subseteq \mathcal{C} \setminus \{c\}$ is the set of other components whose current behaviours the rule for c may read. An *update rule* for c is a function $\mathcal{I}_c : \mathbb{B}(c) \times \prod_{d \in \text{Inf}(c)} \mathbb{B}(d) \rightarrow \mathbb{B}(c)$. The collection of rules is $\mathcal{I} = \{\mathcal{I}_c\}_{c \in \mathcal{C}}$. $\square$

In the full system, every component may be scheduled. An analysis of one subsystem may hold a shared interface at its current value and keep that value

available as input. We write $Z \subseteq \mathcal{C}$ for the components excluded from scheduling. The ordinary system has $Z = \emptyset$.

The possible one-step updates form a *transition relation*. A step is labelled by the component that is updated.

Definition 3.3 (Asynchronous transition relation). We fix a set $Z \subseteq \mathcal{C}$ of components held fixed. For each $c \in \mathcal{C} \setminus Z$, the c -labelled *transition relation* is given by $f \xrightarrow{c}_{\mathcal{I}, Z} f'$ exactly when $f'(c) = \mathcal{I}_c(f(c), (f(d))_{d \in \text{Inf}(c)})$ and $f'(d) = f(d)$ for every $d \neq c$. The unlabelled relation is their union: $f \Delta_{\mathcal{I}, Z} f'$ exactly when $f \xrightarrow{c}_{\mathcal{I}, Z} f'$ for some $c \in \mathcal{C} \setminus Z$. When $Z = \emptyset$, we abbreviate $\xrightarrow{\cdot}_{\mathcal{I}, \emptyset}$ and $\Delta_{\mathcal{I}, \emptyset}$ by $\xrightarrow{\cdot}_{\mathcal{I}}$ and $\Delta_{\mathcal{I}}$, respectively. This definition includes a step in which the rule returns the component's current behaviour. We call this a *stuttering update*, meaning an update that changes no value. $\square$

Because each rule is a function, the current configuration and the scheduled component determine one next configuration. The scheduler's choice is the only remaining choice. Rules that can return several results require the model to represent the source of that uncertainty.

Remark 3.4. Under Definition 3.3, one component updates at a time. A component with a joint rule can update several values together. The subsystem models defined below schedule only components outside Z . The diagnostic test in Section 6 follows the particular schedule recorded in the execution under study. $\square$

Definition 3.5 (Influence graph and feedback). The *influence graph* induced by Inf is $G_{\text{Inf}} = (\mathcal{C}, E_{\text{Inf}})$, where $(d, c) \in E_{\text{Inf}}$ exactly when $d \in \text{Inf}(c)$. An edge $d \rightarrow c$ means that the rule for c may read the behaviour of d . Whether the rule uses that behaviour depends on the installed rule. A directed cycle is a *feedback loop*. The first argument of $\mathcal{I}_c$ already represents dependence on a component's current behaviour, so the graph omits self-loops. $\square$

Remark 3.6 (Feedback under asynchronous updates). The influence graph may contain cycles. If c and d influence one another, for example, an update of c reads the current behaviour of d , and a later update of d may read the new behaviour of c . The scheduler determines their order. Section 6.4 assigns a separate variable to each time step. The resulting causal equations never depend on later variables. Causal models with circular dependencies require additional conditions to ensure that the equations have solutions and to define the effects of changes [37]. $\square$

Definition 3.7 (System model). A *system model* $\mathcal{M}^Z = (\mathcal{C}, \mathbb{B}, \text{Inf}, \mathcal{I}, F, Z, \Delta_{\mathcal{I}, Z}, \Gamma)$, where the entries have the meanings given above and $Z \subseteq \mathcal{C}$. Let $\mathcal{P}_{\mathcal{C}} = \{p_{c=b} \mid c \in \mathcal{C}, b \in \mathbb{B}(c)\}$ be the set of simple propositions about component behaviour. The map Γ assigns to each proposition the configurations in which it is true: $\Gamma : \mathcal{P}_{\mathcal{C}} \rightarrow 2^F$ is given by $\Gamma(p_{c=b}) = \{f \in F \mid f(c) = b\}$. This map is called a *state valuation*. We write $\mathcal{M}$ for the ordinary model $\mathcal{M}^\emptyset$, in which every component may be scheduled.

When Z is non-empty, $\mathcal{M}^Z$ identifies the components held fixed, as in the subsystem models below. We omit the superscript when Z is clear from context. $\square$

Example 3.8. We consider a railroad-switch model with $\mathcal{C} = \{\text{Sw}, \text{LT}, \text{RT}, \text{Arr}, \text{Audit}\}$. Here **Sw** is the switch. The components **LT** and **RT** record whether the left and right routes have been activated. The component **Arr** records whether the train has arrived, and **Audit** stores the switch position that it most recently observed. Let $\mathbb{B}(\text{Sw}) = \{\text{left}, \text{right}\}$ and $\mathbb{B}(\text{LT}) = \mathbb{B}(\text{RT}) = \mathbb{B}(\text{Arr}) = \{\text{off}, \text{on}\}$ and $\mathbb{B}(\text{Audit}) = \{\text{left}, \text{right}\}$. The input sets are $\text{Inf}(\text{Sw}) = \emptyset$, $\text{Inf}(\text{LT}) = \{\text{Sw}\}$, $\text{Inf}(\text{RT}) = \{\text{Sw}\}$, $\text{Inf}(\text{Arr}) = \{\text{LT}, \text{RT}\}$, and $\text{Inf}(\text{Audit}) = \{\text{Sw}\}$. The switch keeps its current setting unless an intervention changes its rule: $\mathcal{I}_{\text{Sw}}(\text{left}) = \text{left}$ and $\mathcal{I}_{\text{Sw}}(\text{right}) = \text{right}$. Each route component keeps its current value until the switch points to its route. The left route then turns on, so $\mathcal{I}_{\text{LT}}(x, \text{left}) = \text{on}$. When the switch points right, the left route keeps x . The right route turns on when the switch points right, so $\mathcal{I}_{\text{RT}}(x, \text{right}) = \text{on}$, and it keeps x when the switch points left. **Arr** turns on once either route component is on and then stays on. We use $_$ as a wildcard for every permitted value in its argument position: $\mathcal{I}_{\text{Arr}}(\text{off}, \text{on}, _) = \text{on}$, $\mathcal{I}_{\text{Arr}}(\text{off}, _, \text{on}) = \text{on}$, $\mathcal{I}_{\text{Arr}}(\text{on}, _, _) = \text{on}$, and $\mathcal{I}_{\text{Arr}}(\text{off}, \text{off}, \text{off}) = \text{off}$. The audit component records the switch behaviour when scheduled. $\mathcal{I}_{\text{Audit}}(x, s) = s$. Let f_0 satisfy $f_0(\text{Sw}) = \text{right}$ and $f_0(\text{LT}) = f_0(\text{RT}) = f_0(\text{Arr}) = \text{off}$ and $f_0(\text{Audit}) = \text{right}$. We have f_1 with $f_1(\text{RT}) = \text{on}$ and all other component behaviours unchanged after updating **RT**. We then have f_2 with $f_2(\text{Arr}) = \text{on}$ after updating **Arr**. The path $f_0 \Delta_{\mathcal{I}} f_1 \Delta_{\mathcal{I}} f_2$ records the right-hand route being activated before the train arrives. $\square$

For any collection q with one entry for each component in $\mathcal{C}$, the notation $q \upharpoonright_D$ denotes its restriction to components in $D \subseteq \mathcal{C}$. The expression $f \upharpoonright_D$ gives the behaviours of the components in D .

Definition 3.9 (Restricted configuration). If $D \subseteq \mathcal{C}$, a *configuration over D* is a function $g : D \rightarrow \bigcup_{c \in D} \mathbb{B}(c)$ such that $g(c) \in \mathbb{B}(c)$ for every $c \in D$. The restriction of $f \in F$ to D is $f \upharpoonright_D$, and the set of configurations over D is F_D . $\square$

An analysis confined to a set of components needs every input that those components may read. We call a set with this property *influence-closed*.

Definition 3.10 (Influence-closed region). Let D be the component set of a system model or restriction. A set $E \subseteq D$ is *influence-closed* if $c \in E$ implies $\text{Inf}(c) \subseteq E$. $\square$

An influence-closed region contains every declared input to each of its components. Every declared input counts towards the condition, including inputs that the installed rule currently leaves unused.

To verify future behaviour, we split a component set into two overlapping subsystems. Their overlap is the interface. We call the split admissible when each non-interface component receives all inputs from its own side and each interface component receives all inputs from the interface.

Definition 3.11 (Admissible interface cut). Let D be the component set of a system model. Let $C_1, C_2 \subseteq D$ cover D , and let $I = C_1 \cap C_2$. We call I the *interface* between the two subsystems. The triple (C_1, I, C_2) is an *admissible interface cut* for D when $C_1 \setminus I \neq \emptyset$, $C_2 \setminus I \neq \emptyset$, and the following conditions hold.

1. If $c \in C_1 \setminus I$, then $\text{Inf}(c) \subseteq C_1$.
2. If $c \in C_2 \setminus I$, then $\text{Inf}(c) \subseteq C_2$.
3. If $c \in I$, then $\text{Inf}(c) \subseteq I$.

□

When an admissible cut is assigned a label λ , we write $\text{cut}(\lambda) = (C_1, I, C_2)$. The map cut assigns each label in its domain to its cut. Definition 3.22 includes this map in the augmented model.

Lemma 3.12 (Sides of a cut are closed). *If (C_1, I, C_2) is an admissible interface cut, then both C_1 and C_2 are influence-closed.*

Proof. For $c \in C_j \setminus I$, the claim follows from the condition for side j . For $c \in I$, interface closure implies $\text{Inf}(c) \subseteq I \subseteq C_j$. □

An admissible cut is *stable* when scheduling a shared component cannot change its value.

Definition 3.13 (Stable interface cut). An admissible interface cut (C_1, I, C_2) is *stable* for $\mathcal{I}$ if every rule for a shared component returns that component's current behaviour. In other words, $\mathcal{I}_i(b, u) = b$ for every $i \in I$, $b \in \mathbb{B}(i)$, and $u \in \prod_{d \in \text{Inf}(i)} \mathbb{B}(d)$. □

Interface closure limits what a rule for a shared component may read. Stability states that scheduling an interface component leaves it unchanged. A fixed-interface restriction places the interface in Z , making it unavailable to the scheduler. When the cut is stable, the unrestricted system also leaves the interface unchanged.

Remark 3.14 (Feedback across an admissible cut). An admissible cut may contain feedback within either side or within the interface. The three closure conditions place every directed cycle of G_{Inf} wholly in $C_1 \setminus I$, $C_2 \setminus I$, or I . All components in a feedback loop must lie on one side or in the interface. Otherwise, the cut is inadmissible. □

Example 3.15 (Interface). For the railroad-switch system of Example 3.8, let $C_1 = \{\text{Sw}, \text{Audit}\}$, $C_2 = \{\text{Sw}, \text{LT}, \text{RT}, \text{Arr}\}$, and $I = C_1 \cap C_2 = \{\text{Sw}\}$. The audit side and the routing side share only the switch. Since $\text{Inf}(\text{Sw}) = \emptyset$, the interface is closed. The input sets in the example satisfy the other locality conditions, so (C_1, I, C_2) is an admissible cut. □

The notation $q \upharpoonright_D$ applies to maps and indexed families. The up-harpoon is reserved for system models. To study future behaviour on one side, we hold the shared components fixed and analyse the remaining components on that side. We call the resulting subsystem a *fixed-interface restriction*. In $\mathcal{M}^Z \upharpoonright_{\lambda, C_j}$, the subscript identifies the cut λ and the selected side C_j .

For a recorded execution, we keep every component in the selected region available to the scheduler unless the original model already excludes it. We call this subsystem a *live restriction*, written $\mathcal{M}^Z \upharpoonright_E^{\text{live}}$.

Definition 3.16 (Fixed-interface restriction along a cut). Let $\mathcal{M}^Z$ have component set D , and let $\text{cut}(\lambda) = (C_1, I, C_2)$ be an admissible cut of D , and let $j \in \{1, 2\}$. We put $Z_j = (Z \cap C_j) \cup I$, $\text{Inf}_j = (\text{Inf}(c))_{c \in C_j}$, and $\mathcal{I}_j = (\mathcal{I}_c)_{c \in C_j}$. Let Γ_{C_j} be the state valuation on C_j , so $\Gamma_{C_j}(p_{c=b}) = \{g \in F_{C_j} \mid g(c) = b\}$ for $c \in C_j$. The C_j -restriction along λ , $\mathcal{M}^Z \upharpoonright_{\lambda, C_j}$, is the system model $(C_j, \mathbb{B} \upharpoonright_{C_j}, \text{Inf}_j, \mathcal{I}_j, F_{C_j}, Z_j, \Delta_{\mathcal{I}_j, Z_j}, \Gamma_{C_j})$. The set Z_j contains the interface and every member of Z that belongs to C_j . For $c \in C_j \setminus Z_j$, the step $g \xrightarrow{\mathcal{I}_j, Z_j} g'$ updates c to $\mathcal{I}_c(g(c), (g(d))_{d \in \text{Inf}(c)})$ and leaves every other $d \in C_j$ unchanged. The requirements above ensure that every input to $\mathcal{I}_c$ is available in C_j .

If an inner cut $\text{cut}(\mu) = (D_1, J, D_2)$ is used within the C_j -restriction, then its D_k -restriction $(\mathcal{M}^Z \upharpoonright_{\lambda, C_j}) \upharpoonright_{\mu, D_k}$ holds the components in $Z_{j,k} = (Z_j \cap D_k) \cup J$ fixed for $k \in \{1, 2\}$. Nested restrictions preserve components that were already held fixed. $\square$

Definition 3.17 (Live restriction to a closed region). Let $\mathcal{M}^Z$ have component set D , and let $E \subseteq D$ be influence-closed. Put $Z_E^{\text{live}} = Z \cap E$, $\text{Inf}_E = (\text{Inf}(c))_{c \in E}$, and $\mathcal{I}_E = (\mathcal{I}_c)_{c \in E}$. Let Γ_E be the state valuation on E , so $\Gamma_E(p_{c=b}) = \{g \in F_E \mid g(c) = b\}$ for $c \in E$. The *live restriction* $\mathcal{M}^Z \upharpoonright_E^{\text{live}}$ is

$$(E, \mathbb{B} \upharpoonright_E, \text{Inf}_E, \mathcal{I}_E, F_E, Z_E^{\text{live}}, \Delta_{\mathcal{I}_E, Z_E^{\text{live}}}, \Gamma_E).$$

The live restriction inherits the components that the original model holds fixed. Every component of $E \setminus Z$ remains schedulable, and influence closure makes its update well-defined from an E -configuration. $\square$

Remark 3.18 (Interpretation of a fixed-interface restriction). We use a fixed-interface restriction only after the interface has been established and its values no longer change. If the subsystems exchange new values, the interface must evolve. When the cut is stable, the unrestricted system also leaves these interface values unchanged. $\square$

An intervention replaces selected rules without changing the current configuration or the declared input sets.

Definition 3.19 (Intervention). For each component c , let $\text{Inp}(c) := \prod_{d \in \text{Inf}(c)} \mathbb{B}(d)$. An *intervention* is a pair $(T, \mathcal{J}_T)$, where $\emptyset \neq T \subseteq \mathcal{C}$ is the set of changed components

and $\mathcal{J}_T = (\mathcal{J}_c)_{c \in T}$ assigns each $c \in T$ a replacement rule $\mathcal{J}_c : \mathbb{B}(c) \times \text{Inp}(c) \rightarrow \mathbb{B}(c)$. The application of $(T, \mathcal{J}_T)$ to a system model $\mathcal{M}^Z$ yields $\mathcal{M}^Z[T \mapsto \mathcal{J}_T]$. For $c \in T$, the rule $\mathcal{I}_c$ is replaced by $\mathcal{J}_c$. All other rules and model entries remain unchanged. The new collection of rules and the same set Z determine the possible updates. If θ names the replacement $(T, \mathcal{J}_T)$, we write $P_\theta = (T, \mathcal{J}_T)$ for the replacement named by θ and $\mathcal{M}_\theta^Z = \mathcal{M}^Z[T \mapsto \mathcal{J}_T]$ for the resulting model. $\square$

Example 3.20. In the railroad model of Example 3.8, let θ_{left} name the replacement $P_{\theta_{\text{left}}} = (\{\text{Sw}\}, (\mathcal{J}_{\text{Sw}}^{\text{left}}))$. Here $\mathcal{J}_{\text{Sw}}^{\text{left}}(x) = \text{left}$ for both $x \in \{\text{left}, \text{right}\}$. The application of θ_{left} leaves the switch in its current position. When Sw is next scheduled, the replacement rule sets it to left. A later update of LT can then activate the left-hand route. The installation of a rule does not execute it. $\square$

Remark 3.21 (Preservation of a cut). Because an intervention preserves Inf, it preserves every admissible cut. It preserves the stability of a cut when each replacement on the interface also returns the component's current value. The simpler condition $T \cap I = \emptyset$ is sufficient and is satisfied by the interventions considered below. The claim does not cover an arbitrary replacement of a rule at an interface component. $\square$

To let formulas refer to particular cuts and rule changes, we add maps that assign names to them. A system model with these two maps is called an *augmented model*.

Definition 3.22 (Augmented model). An *augmented model* is a tuple $\widehat{\mathcal{M}}^Z = (\mathcal{C}, \mathbb{B}, \text{Inf}, \mathcal{I}, F, Z, \Delta_{\mathcal{I}, Z}, \Gamma, \text{cut}, \text{rep})$, where the first eight entries form a system model. The map cut , which may be undefined for some labels, names admissible cuts. If $\text{cut}(\lambda) = (D_1, I, D_2)$, its component set is $D_1 \cup D_2$, and the label λ may be used in a model precisely when this is the model's current component set. Restriction retains labels for cuts contained in the restricted component set. Such a label becomes available when a subsequent restriction reaches its declared component set.

The map rep assigns each intervention label θ a fixed replacement $\text{rep}(\theta) = P_\theta = (T_\theta, \mathcal{J}_{T_\theta}^\theta)$. The installation of this replacement yields $\widehat{\mathcal{M}}_\theta := \widehat{\mathcal{M}}[T_\theta \mapsto \mathcal{J}_{T_\theta}^\theta]$. The installation of a replacement does not change the map rep . A fixed-interface or live restriction retains exactly those intervention labels whose targets lie in the restricted component set. It also retains the replacement and every permitted cut label. If $T_\theta \subseteq C_j$, then $\widehat{\mathcal{M}}_\theta \upharpoonright_{\lambda, C_j} = (\widehat{\mathcal{M}} \upharpoonright_{\lambda, C_j})_\theta$. The application of an available intervention before or after restriction gives the same result. We suppress the superscript Z on $\widehat{\mathcal{M}}^Z$ when the components held fixed are clear from context. $\square$

4. Rule Replacement and Authorization

Agents control rule changes. We specify which agent controls each component and which rules that agent may install. We do not model utilities, strategic responses, or responsibility.

Logics for concurrent games represent what agents can achieve [38], and Gladyshev et al. derive such a game from a Halpern–Pearl model to study strategy and responsibility [34]. Causal games built from equations add variables for decisions and preferences, including dependencies between agents’ decision rules [39, 40]. Work on accountability has also treated program actions as actual causes [41]. Here an intervention replaces the update rule installed at a component.

Let $\widehat{\mathcal{M}}^Z = (\mathcal{C}, \mathbb{B}, \text{Inf}, \mathcal{I}, F, Z, \Delta_{\mathcal{I}, Z}, \Gamma, \text{cut}, \text{rep})$ be an augmented model, and let $\mathcal{A}$ be a finite agent set. For each component c , $A_c \subseteq \mathcal{A}$ is the set of agents allowed to change its rule. We assume $|A_c| \leq 1$, so at most one agent chooses the rule for each controlled component. The results about updates and subsystem analysis remain valid for shared control. The components controlled by a are $C_a := \{c \in \mathcal{C} \mid a \in A_c\}$.

For each $c \in \mathcal{C}$, the set $\text{Adm}(c)$ lists the rules that an authorized agent may install: $\text{Adm}(c) \subseteq \{\mathcal{J}_c : \mathbb{B}(c) \times \prod_{d \in \text{Inf}(c)} \mathbb{B}(d) \rightarrow \mathbb{B}(c)\}$. We write $\text{Inp}(c) = \prod_{d \in \text{Inf}(c)} \mathbb{B}(d)$ for the possible tuples of inputs to c . For a group $A \subseteq \mathcal{A}$ of agents, a rule change by the group is an intervention $(T, \mathcal{J}_T)$. Its targets satisfy $\emptyset \neq T \subseteq \bigcup_{a \in A} C_a$, and each replacement rule $\mathcal{J}_c$ belongs to $\text{Adm}(c)$. The agents, their control sets, and the sets $\text{Adm}(c)$ remain fixed, so the model notation omits them.

We need a record of the authorized rule that each agent has installed at each component it controls. We call this record a *policy profile*.

Definition 4.1 (Policy installation). A *policy profile* is a family $\Pi = \{\Pi_a \mid a \in \mathcal{A}\}$. For every $c \in C_a$, the entry $\Pi_a(c) \in \text{Adm}(c)$ is the rule selected by agent a . The installation of Π replaces the initial rules at controlled components:

$$\text{inst}(\mathcal{I}, \Pi)_c = \begin{cases} \Pi_a(c) & \text{if there exists an agent } a \text{ with } c \in C_a, \\ \mathcal{I}_c & \text{otherwise.} \end{cases}$$

We install Π once, before evaluating the model at a configuration. After installation, $\mathcal{I}$ denotes the rules in force, and Π records which authorized rules are installed. The assignment of components to agents and the sets $\text{Adm}(c)$ determine which later interventions are authorized. System updates use the rules currently in force. $\square$

For $E \subseteq \mathcal{C}$, restriction to E retains the entries of a policy profile for components in E . The restricted control set of agent a is $C_a \cap E$, and we write $(\Pi \upharpoonright_E)_a = \Pi_a \upharpoonright_{C_a \cap E}$. The restriction retains $\text{Adm}(c)$ for every $c \in E$.

Definition 4.2 (Policy intervention). For a group $A \subseteq \mathcal{A}$ of agents and a finite non-empty target set $\emptyset \neq T \subseteq \bigcup_{a \in A} C_a$ with replacement rules $\mathcal{J}_T = (\mathcal{J}_c)_{c \in T}$, where $\mathcal{J}_c \in \text{Adm}(c)$ for every $c \in T$, the *policy intervention* label $\theta_{A, \mathcal{J}_T}$ names the replacement $\text{rep}(\theta_{A, \mathcal{J}_T}) = (T, \mathcal{J}_T)$.

After the intervention, the policy profile Π_θ is given, for $a \in A$ and $c \in C_a$, by

$$(\Pi_\theta)_a(c) = \begin{cases} \mathcal{J}_c & \text{if } c \in T, \\ \Pi_a(c) & \text{otherwise.} \end{cases}$$

For $a \notin A$, $(\Pi_\theta)_a = \Pi_a$. $\square$

For every intervention label θ , define its action on an augmented model and policy profile by $\llbracket \theta \rrbracket(\widehat{\mathcal{M}}, \Pi) := (\widehat{\mathcal{M}}_\theta, \Pi_\theta)$, where $\widehat{\mathcal{M}}_\theta$ installs the fixed replacement $\text{rep}(\theta)$ and Π_θ is the updated profile from Definition 4.2.

The application of an intervention before restricting the policy profile gives the same result as restricting first and then applying the available intervention. If $\text{cut}(\lambda) = (C_1, I, C_2)$ is available for the current component set, $j \in \{1, 2\}$, and $T_\theta \subseteq C_j$, then $\widehat{\mathcal{M}}_\theta \upharpoonright_{\lambda, C_j} = (\widehat{\mathcal{M}} \upharpoonright_{\lambda, C_j})_\theta$ and $\Pi_\theta \upharpoonright_{C_j} = (\Pi \upharpoonright_{C_j})_\theta$.

An agent can be a person, team, or automated controller. The signature in Section 5 includes only those policy interventions that satisfy the rules above about control and permitted choices.

Example 4.3 (Engineer policy). In the railroad-switch model of Example 3.8, let $\mathcal{A} = \{e\}$ with $A_{\text{Sw}} = \{e\}$ and $C_e = \{\text{Sw}\}$. We take $\text{Adm}(\text{Sw}) = \{\mathcal{J}_{\text{Sw}}^{\text{left}}, \mathcal{J}_{\text{Sw}}^{\text{right}}\}$, where $\mathcal{J}_{\text{Sw}}^{\text{left}}$ fixes the switch to left and $\mathcal{J}_{\text{Sw}}^{\text{right}}$ fixes it to right. A policy profile with $\Pi_e(\text{Sw}) = \mathcal{J}_{\text{Sw}}^{\text{right}}$ installs the right-setting rule. The policy intervention $\theta_{\{e\}, \{\mathcal{J}_{\text{Sw}}^{\text{left}}\}}$ replaces it with the left-setting rule. The switch changes only when it is next scheduled. A right-hand route that is already active retains its behaviour. $\square$

5. Syntax and Semantics

The language's *signature* lists the components and their possible behaviours, the intervention labels and their declared targets, and the cut labels and their declared cuts. Each name specifies the components on which it may be used, making the signature typed.

Definition 5.1. A *language signature* is a tuple $\Sigma = (\mathcal{C}, \mathbb{B}, \Theta, \text{tg}, \Lambda, \text{ct})$. Here $\mathcal{C}$ lists the components and $\mathbb{B}$ assigns their possible behaviours. The sets Θ and Λ contain the intervention and cut labels. Each intervention label has a declared non-empty target set $\emptyset \neq \text{tg}(\theta) \subseteq \mathcal{C}$. Each cut label specifies $\text{ct}(\lambda) = (C_1, I, C_2)$, where $C_1 \cup C_2 = D \subseteq \mathcal{C}$ and $I = C_1 \cap C_2$. We call D the *component set* of λ . $\square$

A model is *compatible with the signature* when its intervention labels have the declared targets, its retained cut labels have the declared cuts, and every replacement rule has the required inputs and outputs.

Definition 5.2 (Compatibility with the signature). Let $\widehat{\mathcal{M}}^Z$ be an augmented model with current component set $D \subseteq \mathcal{C}$. A cut label λ is *retained* when $\lambda \in \text{dom}(\text{cut})$. It is *available* at current component set D when it is retained, $\text{cut}(\lambda) = \text{ct}(\lambda) = (C_1, I, C_2)$, $C_1 \cup C_2 = D$, and this cut is admissible for D . The model is *compatible* with $\Sigma = (\mathcal{C}, \mathbb{B}, \Theta, \text{tg}, \Lambda, \text{ct})$ when the following conditions hold.

1. The available intervention labels are exactly those whose targets lie in the current component set:

$$\text{dom}(\text{rep}) = \{\theta \in \Theta \mid \text{tg}(\theta) \subseteq D\}.$$

For every such θ , if $\text{rep}(\theta) = (T_\theta, \mathcal{J}_{T_\theta}^\theta)$, then $T_\theta = \text{tg}(\theta)$ and, for every $c \in T_\theta$,

$$\mathcal{J}_c^\theta : \mathbb{B}(c) \times \prod_{d \in \text{Inf}(c)} \mathbb{B}(d) \longrightarrow \mathbb{B}(c).$$

2. For every retained cut label $\lambda \in \text{dom}(\text{cut})$,

$$\text{cut}(\lambda) = \text{ct}(\lambda).$$

An *augmented Σ -model on D* is an augmented model compatible with Σ that has component set D , behaviour map $\mathbb{B} \upharpoonright_D$, and labels from Θ and Λ . Every augmented Σ -model below meets these requirements. Interventions preserve compatibility because their replacement rules and both label maps remain fixed. Fixed-interface and live restrictions also preserve compatibility. They keep exactly those intervention labels whose targets lie in the restricted component set. They keep a cut label only when all components of its cut lie there. $\square$

The set Θ contains only authorized policy interventions. The group must control every target, and each proposed rule must belong to the relevant $\text{Adm}(c)$. Formulas use only labels in the signature. By Definition 5.2, an intervention label θ is available in a model on D exactly when $\text{tg}(\theta) \subseteq D$.

The full model uses $D = \mathcal{C}$. Restrictions use smaller component sets.

The simplest formula states that one component has one behaviour. We call this a *state atom*. For every $c \in \mathcal{C}$ and $b \in \mathbb{B}(c)$, the language contains the atom $p_{c=b}$. We recall that $\mathcal{P}_{\mathcal{C}}$ is the set of all such atoms. The typed language $\mathcal{L}_\Sigma$ is generated by

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Box\varphi \mid \Diamond\varphi \mid \Box^+\varphi \mid \Diamond^+\varphi \mid \langle\theta\rangle\varphi \mid \varphi *_\lambda \psi,$$

where $p \in \mathcal{P}_{\mathcal{C}}$, $\theta \in \Theta$, and $\lambda \in \Lambda$. As usual, $\varphi \vee \psi$, $\varphi \rightarrow \psi$, and $\top, \perp$ are defined classically. We read $\Diamond\varphi$ as “ φ holds after some next update” and $\Box\varphi$ as “ φ holds after every next update”. The operators $\Diamond^+$ and $\Box^+$ quantify over non-empty finite paths. The formula $\langle\theta\rangle\varphi$ evaluates φ after the intervention named by θ . The expression $\varphi *_\lambda \psi$ evaluates φ and ψ in the two fixed-interface restrictions named by the cut label λ .

Typing checks that every component and label is declared for the component set on which the formula is evaluated. Availability of a named cut is a semantic condition. We write $D \vdash_\Sigma \varphi$ when φ can be evaluated on $D \subseteq \mathcal{C}$. The judgement is

defined recursively as follows.

$$\begin{aligned}
D \vdash_{\Sigma} p_{c=b} & \text{ iff } c \in D, \\
D \vdash_{\Sigma} \neg \varphi & \text{ iff } D \vdash_{\Sigma} \varphi, \\
D \vdash_{\Sigma} \varphi \wedge \psi & \text{ iff } D \vdash_{\Sigma} \varphi \text{ and } D \vdash_{\Sigma} \psi, \\
D \vdash_{\Sigma} \Box \varphi, \Diamond \varphi, \Box^+ \varphi, \Diamond^+ \varphi & \text{ iff } D \vdash_{\Sigma} \varphi, \\
D \vdash_{\Sigma} \langle \theta \rangle \varphi & \text{ iff } \text{tg}(\theta) \subseteq D \text{ and } D \vdash_{\Sigma} \varphi, \\
D \vdash_{\Sigma} \varphi *_{\lambda} \psi & \text{ iff } \text{ct}(\lambda) = (D_1, I, D_2), D = D_1 \cup D_2, \\
& D_1 \vdash_{\Sigma} \varphi \text{ and } D_2 \vdash_{\Sigma} \psi.
\end{aligned}$$

The constants $\top$ and $\perp$ are well typed for every component set. We call φ *D-local* when $D \vdash_{\Sigma} \varphi$. The *support* of a formula is the set of components it mentions. It is defined by $\text{supp}(p_{c=b}) = \{c\}$. Formulas built with “not” or “and” inherit the support of their subformulas. The path operators do as well. For interventions, $\text{supp}(\langle \theta \rangle \varphi) = \text{tg}(\theta) \cup \text{supp}(\varphi)$. For separation, $\text{supp}(\varphi *_{\lambda} \psi) = \text{supp}(\varphi) \cup \text{supp}(\psi)$. Well-typedness implies $\text{supp}(\varphi) \subseteq D$. The determination of the component set of a subformula that uses the separation operator also requires its declared cut. Satisfaction in a model on D is defined only when $D \vdash_{\Sigma} \varphi$. Evaluation inside a restriction uses the component set of that restriction.

5.1. Satisfaction Relation

We interpret the language on augmented Σ -models with component set $D \subseteq \mathcal{C}$. We omit subscripts on F_D , Γ_D , and related notation when the component set is clear. The evaluation of a formula requires a model, its current configuration, and the policy profile recording the installed rules at controlled components. We call this triple a *pointed model*: $x = (\widehat{\mathcal{M}}, f, \Pi)$. Its entries are an augmented model, its current configuration, and its policy profile. It is *policy-consistent* when the profile records the rule that is actually in force, that is, $\mathcal{I}_c = \Pi_a(c)$ for every agent a and every controlled component $c \in C_a \cap D$. All pointed models below satisfy this condition. Policy interventions preserve it because they update the rule and its profile entry together. Fixed-interface and live restrictions preserve it because they restrict the rules and policy profile to the same component set. For an augmented Σ -model with policy profile $(\widehat{\mathcal{M}}, \Pi)$ and an intervention label θ whose target lies in D , the intervention action defined in Section 4 is $\llbracket \theta \rrbracket(\widehat{\mathcal{M}}, \Pi) = (\widehat{\mathcal{M}}_{\theta}, \Pi_{\theta})$. We also write the intervention as the single specified change $(\widehat{\mathcal{M}}, f, \Pi) \xrightarrow{\theta} (\widehat{\mathcal{M}}_{\theta}, f, \Pi_{\theta})$, which leaves the configuration unchanged.

If $\text{cut}(\lambda) = (C_1, I, C_2)$ and $j \in \{1, 2\}$, then $\widehat{\mathcal{M}}^Z \upharpoonright_{\lambda, C_j}$ holds the components in $Z_j = (Z \cap C_j) \cup I$ fixed and has transition relation $\Delta_{\mathcal{I}_j, Z_j}$ from Definition 3.16. Its state propositions obey $\Gamma_{C_j}(p_{c=b}) = \{g \in F_{C_j} \mid g(c) = b\}$ for $c \in C_j$. It retains intervention labels whose targets lie in C_j , with the same replacements, and cut labels declared for subsets of C_j . The restriction of a pointed model $(\widehat{\mathcal{M}}, f, \Pi)$ is $(\widehat{\mathcal{M}} \upharpoonright_{\lambda, C_j}, f \upharpoonright_{C_j}, \Pi \upharpoonright_{C_j})$.

A component held fixed is never scheduled, so its replacement rule remains unused. Nested restrictions retain components previously held fixed before adding their own interface. The locality results below use interventions whose targets lie outside the interface.

Definition 5.3. For a pointed model $x = (\widehat{\mathcal{M}}, f, \Pi)$, let $x[f']$ denote the same model and policy at configuration f' . Satisfaction is defined by the following clauses.

1. $x \models p$ iff $f \in \Gamma(p)$.
2. $x \models \neg\varphi$ iff $x \not\models \varphi$.
3. $x \models \varphi \wedge \psi$ iff both $x \models \varphi$ and $x \models \psi$.
4. $x \models \Diamond\varphi$ iff some $f' \in F$ satisfies $f\Delta_{\mathcal{I},Z}f'$ and $x[f'] \models \varphi$.
5. $x \models \Box\varphi$ iff every $f' \in F$ satisfying $f\Delta_{\mathcal{I},Z}f'$ also satisfies $x[f'] \models \varphi$.
6. $x \models \Diamond^+\varphi$ iff some $f' \in F$ satisfies $f(\Delta_{\mathcal{I},Z})^+f'$ and $x[f'] \models \varphi$.
7. $x \models \Box^+\varphi$ iff every $f' \in F$ satisfying $f(\Delta_{\mathcal{I},Z})^+f'$ also satisfies $x[f'] \models \varphi$.
8. Let x_θ be the model and policy after applying θ , at the unchanged configuration f . Then $x \models \langle\theta\rangle\varphi$ iff $x_\theta \models \varphi$.
9. $x \models \varphi *_\lambda \psi$ iff λ is available at the current component set D , with $\text{cut}(\lambda) = (C_1, I, C_2)$, the left restriction satisfies φ , and the right restriction satisfies ψ . These restrictions begin at $f \upharpoonright_{C_1}$ and $f \upharpoonright_{C_2}$ and use the policy profiles $\Pi \upharpoonright_{C_1}$ and $\Pi \upharpoonright_{C_2}$, respectively.

Here R^+ denotes a path of one or more R -steps, and R^* denotes a path of zero or more R -steps. The notation $\Box^*\varphi := \varphi \wedge \Box^+\varphi$ requires φ now and after every non-empty finite path.

□

Example 5.4 (Reading the formulas in the railroad example). Let f_0 be the initial right-switch configuration from Example 3.8. With the state atoms $p_{\text{RT}=\text{on}}$, $p_{\text{LT}=\text{on}}$, and $p_{\text{Arr}=\text{on}}$, we have $(\widehat{\mathcal{M}}, f_0, \Pi) \models \Diamond^+p_{\text{RT}=\text{on}}$ because the recorded execution through the right route reaches a configuration with $\text{RT} = \text{on}$. After the intervention that installs the constant-left switch rule, we have $(\widehat{\mathcal{M}}, f_0, \Pi) \models \langle\theta_{\text{left}}\rangle\Diamond^+p_{\text{LT}=\text{on}}$. For example, an execution may update **Sw** and then **LT**. The stronger formula $\langle\theta_{\text{left}}\rangle\Box^+\neg p_{\text{RT}=\text{on}}$ is false at f_0 , because a later sequence of updates may update **RT** before the new switch rule is executed.

□

6. Actual Causation in Recorded Executions

Halpern–Pearl models define causal variables through causal equations and represent an intervention by replacing selected equations [4, 5]. Their account of actual causation tests whether an effect occurred, whether changing the proposed cause removes it, and whether the proposed cause is minimal. Further conditions handle cases in which one potential cause prevents another from acting [42]. We study the

narrower question of which installed rules caused the final outcome of one completed execution. We call this execution-relative causation.

In a replay under alternative rules, only the installed rules vary. Component behaviours change when the recorded schedule applies those rules. Rule replacement covers changes to policies, controllers, and thresholds. Behaviours observed during the execution remain consequences of earlier updates. Because a replacement keeps the same input set, it introduces no new dependency. This choice fits accounts that explain causation through changes to rules or processes [43, 44, 6, 45, 46].

We use finite executions whose steps name the component selected at each time. Rule replacement has the same meaning here as in the intervention operator of the logic.

6.1. Schedule-Preserving Counterfactuals

A *Boolean state formula* is a true-or-false claim about one configuration. It is built from the state atoms of Section 5 using “not” and “and”. For $f \in F_D$, evaluate each Boolean state formula well typed on D directly at f : $f \models_B p_{c=b}$ exactly when $f(c) = b$, $f \models_B \neg\varphi$ exactly when $f \not\models_B \varphi$, and $f \models_B \varphi \wedge \psi$ exactly when $f \models_B \varphi$ and $f \models_B \psi$. For every such ψ , $(\widehat{\mathcal{M}}, f, \Pi) \models \psi$ exactly when $f \models_B \psi$. We evaluate the effect at the final configuration. An earlier event can be used as an effect only if the system records it, for example in a monitor that remains active after the event.

The result of a causal test depends on which replacement rules the analyst is willing to consider. We record those choices as the *allowed alternatives*.

Definition 6.1 (Allowed alternatives). We fix an augmented model and the record of its installed rules $(\widehat{\mathcal{M}}, \Pi)$. For each component c , let $\mathfrak{A}(c)$ be the replacement rules allowed in the analysis: $\mathfrak{A}(c) \subseteq \{\mathcal{I}_c : \mathbb{B}(c) \times \text{Inp}(c) \rightarrow \mathbb{B}(c)\} \setminus \{\mathcal{I}_c\}$. We write $\mathfrak{A} = \{\mathfrak{A}(c)\}_{c \in \mathcal{C}}$. One may choose $\mathfrak{A}(c) = \text{Adm}(c) \setminus \{\mathcal{I}_c\}$. Execution-relative replay may also use rules that no agent is authorized to install. Such rules have no label in Θ . $\square$

The set $\mathfrak{A}(c)$ may be empty, and every causal claim is relative to these choices. We write Z for the set of components held fixed in $\widehat{\mathcal{M}}$. A *finite scheduled execution* is a finite execution that records the component selected at every step:

$$\rho : f_0 \xrightarrow{s_1}_{\mathcal{I}, Z} f_1 \xrightarrow{s_2}_{\mathcal{I}, Z} \cdots \xrightarrow{s_n}_{\mathcal{I}, Z} f_n,$$

where each $s_i \in \mathcal{C} \setminus Z$. We write $\text{sched}(\rho) = (s_1, \dots, s_n)$ and $\text{last}(\rho) = f_n$.

The execution ρ uses one collection of rules and never schedules a component in Z . A replay replaces selected rules and keeps f_0 , the schedule $(s_1, \dots, s_n)$, and the execution length. An analysis that changes which component moves next, which branch is taken, or when execution stops requires a model of those choices.

Some causal analyses constrain alternative executions, derive causes from executions that violate a required property, or select nearby paths in a transition system

[20, 23, 25]. Our alternative execution retains the initial configuration, schedule, and execution length. Only selected mechanisms vary.

For finite $T \subseteq \mathcal{C}$ and alternatives $j_T \in \prod_{c \in T} \mathfrak{A}(c)$, define the collection of replacement rules by

$$\mathcal{I}_c^{T \leftarrow j_T} = \begin{cases} j_c & c \in T, \\ \mathcal{I}_c & c \notin T. \end{cases}$$

We call the result a *same-schedule replay* because it applies the new rules in the recorded order. It is the labelled sequence

$$\rho^{T \leftarrow j_T} : f_0^{T,j} \xrightarrow{s_1}_{\mathcal{I}^{T \leftarrow j_T, Z}} f_1^{T,j} \xrightarrow{s_2}_{\mathcal{I}^{T \leftarrow j_T, Z}} \cdots \xrightarrow{s_n}_{\mathcal{I}^{T \leftarrow j_T, Z}} f_n^{T,j},$$

where $f_0^{T,j} = f_0$ and

$$f_i^{T,j}(c) = \begin{cases} \mathcal{I}_c^{T \leftarrow j_T}(f_{i-1}^{T,j}(c), (f_{i-1}^{T,j}(d))_{d \in \text{Inf}(c)}) & c = s_i, \\ f_{i-1}^{T,j}(c) & c \neq s_i. \end{cases}$$

Section 6.4 uses K_c as a variable whose value is the rule selected at c .

Definition 6.2 (Execution-relative cause). We fix a model $\widehat{\mathcal{M}}$ at initial configuration f_0 with policy Π installed. Let Z contain the components held fixed, let $\mathfrak{A}$ give the allowed alternatives, and let ρ be a finite scheduled execution of the form defined above. Let ψ be a Boolean state formula and let $\emptyset \neq C^* \subseteq \mathcal{C}$ be finite. We use K_c as a name for the rule installed at c and K_{C^*} for the tuple over C^* . These names become variables that select rules in Section 6.4. We write $k_{C^*}^{\text{act}} = (\mathcal{I}_c)_{c \in C^*}$ for the rules installed during ρ . We write

$$\text{Cause}_{\widehat{\mathcal{M}}, \Pi, \mathfrak{A}}^{\text{er}}(\rho; K_{C^*} = k_{C^*}^{\text{act}}, \psi)$$

when the following conditions hold. The superscript *er* abbreviates “execution-relative”.

(RC1) The effect occurred. At the final configuration, $f_n \models_{\text{B}} \psi$.

(RC2) The effect is prevented by changing the rules. There is a choice $j_{C^*} \in \prod_{c \in C^*} \mathfrak{A}(c)$ such that, writing $\rho^{C^* \leftarrow j_{C^*}}$ for this replay,

$$\text{last}(\rho^{C^* \leftarrow j_{C^*}}) \not\models_{\text{B}} \psi.$$

(RC3) No smaller set works. No non-empty proper subset $C' \subsetneq C^*$ satisfies **RC2** for any $j_{C'} \in \prod_{c \in C'} \mathfrak{A}(c)$. $\square$

The cause depends on the model, initial configuration, schedule, final claim, and allowed alternatives. It is specific to the recorded execution and the available replacements.

Definition 6.3 (Labels for all alternatives). The intervention labels *cover* $\mathfrak{A}$ when, for every finite non-empty $T \subseteq \mathcal{C}$ and every $j_T \in \prod_{c \in T} \mathfrak{A}(c)$, there is a label θ_{T,j_T} that replaces exactly the rules in T by j_T . $\square$

Since every label in Θ is authorized, Θ can cover $\mathfrak{A}$ only if an authorized group of agents can install each allowed rule, although Definition 6.2 does not require this.

Proposition 6.4 (Labelled interventions reproduce direct replay). *If the intervention labels cover $\mathfrak{A}$, then the direct replay $\rho^{T \leftarrow j_T}$ agrees at every time with the replay produced by applying θ_{T,j_T} and then following $\text{sched}(\rho)$.*

Proof. Both replays start at f_0 , install the same rules, and apply those rules to the same scheduled component at each time. Induction over the schedule proves that the corresponding configurations are equal. $\square$

6.2. Retry Amplification in Distributed Services

Retry storms are a familiar source of cascading failure: a slow service causes callers to retry, the additional traffic raises the load, and the service slows further. Operational guidance recommends increasing the delay between retries, limiting their number, rejecting excess work, stopping calls to a failing service, and limiting traffic at the gateway [47, 48]. Causal methods have also been applied to finding causes of failures in services [49]. The example makes each dependency explicit and shows how retries amplify an initial slowdown.

Let $\mathcal{C}_{\text{svc}} = \{A, B, G, Q, S, D\}$. Here A and B are groups of callers, G is an API gateway, Q a request queue, S the workers that process requests, and D the database used by the service. The callers have $\mathbb{B}(A) = \mathbb{B}(B) = \{\text{idle}, \text{retry}\}$. The other behaviour sets are $\mathbb{B}(G) = \{\text{normal}, \text{surge}, \text{capped}\}$, $\mathbb{B}(Q) = \{\text{normal}, \text{high}\}$, $\mathbb{B}(S) = \{\text{healthy}, \text{slow}, \text{overloaded}\}$, and $\mathbb{B}(D) = \{\text{healthy}, \text{overloaded}\}$. The input sets are $\text{Inf}(A) = \text{Inf}(B) = \{S\}$, $\text{Inf}(G) = \{A, B\}$, $\text{Inf}(Q) = \{G\}$, $\text{Inf}(S) = \{Q\}$, and $\text{Inf}(D) = \{S\}$. Figure 1 shows these components and which components each rule may read.

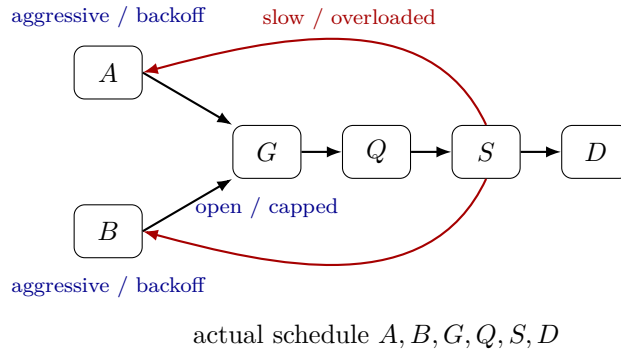


Figure 1: Retry amplification. The red influence edges close the feedback loop $S \rightarrow A, B \rightarrow G \rightarrow Q \rightarrow S$. The database D reads S and becomes overloaded after S does.

For $X \in \{A, B\}$, the installed rule retries whenever the service is slow or overloaded. Its alternative, called *backoff*, remains idle throughout the analysed interval:

$$\mathcal{I}_X^{\text{agg}}(x, s) = \begin{cases} \text{retry} & s \in \{\text{slow}, \text{overloaded}\}, \\ \text{idle} & s = \text{healthy}, \end{cases} \quad \mathcal{J}_X^{\text{backoff}}(x, s) = \text{idle}.$$

Here idle means that the population remains silent during the analysed interval. The installed gateway rule is open.

$$\mathcal{I}_G^{\text{open}}(g, a, b) = \begin{cases} \text{surge} & a = \text{retry} \text{ or } b = \text{retry}, \\ \text{normal} & \text{otherwise.} \end{cases}$$

The remaining rules are

$$\begin{aligned} \mathcal{I}_Q(q, g) &= \begin{cases} \text{high} & g = \text{surge}, \\ \text{normal} & \text{otherwise,} \end{cases} & \mathcal{I}_S(s, q) &= \begin{cases} \text{overloaded} & q = \text{high}, \\ \text{healthy} & q = \text{normal}, \end{cases} \\ \mathcal{I}_D(d, s) &= \begin{cases} \text{overloaded} & s = \text{overloaded}, \\ \text{healthy} & \text{otherwise.} \end{cases} \end{aligned}$$

We use four agents. The set $\mathcal{A}_{\text{svc}}$ contains AppA, AppB, Platform, and Service. The control sets are $C_{\text{AppA}} = \{A\}$, $C_{\text{AppB}} = \{B\}$, $C_{\text{Platform}} = \{G, Q\}$, and $C_{\text{Service}} = \{S, D\}$. Each $\text{Adm}(c)$ contains the displayed rule. In addition, $\text{Adm}(A)$ contains $\mathcal{J}_A^{\text{backoff}}$, $\text{Adm}(B)$ contains $\mathcal{J}_B^{\text{backoff}}$, and $\text{Adm}(G)$ contains the capped gateway rule introduced below. Let Π_{svc} select the displayed rules.

At the initial configuration f_0 , callers A and B are idle, the gateway and queue are normal, the service is slow, and the database is healthy. We treat the initial slowdown as part of the observed context. From this configuration, the installed rules $\mathcal{I}_A = \mathcal{I}_A^{\text{agg}}$ and $\mathcal{I}_B = \mathcal{I}_B^{\text{agg}}$ generate the recorded incident execution

$$\rho_{\text{inc}} : f_0 \xrightarrow{A} f_1 \xrightarrow{B} f_2 \xrightarrow{G} f_3 \xrightarrow{Q} f_4 \xrightarrow{S} f_5 \xrightarrow{D} f_6.$$

Table 1 records every configuration in the execution.

Table 1: The observed retry-amplification trace.						
configuration	A	B	G	Q	S	D
f_0	idle	idle	normal	normal	slow	healthy
f_1	retry	idle	normal	normal	slow	healthy
f_2	retry	retry	normal	normal	slow	healthy
f_3	retry	retry	surge	normal	slow	healthy
f_4	retry	retry	surge	high	slow	healthy
f_5	retry	retry	surge	high	overloaded	healthy
f_6	retry	retry	surge	high	overloaded	overloaded

We write $\psi_D := p_{D=\text{overloaded}}$ for the claim that the database is overloaded at the end. The allowed alternatives for the callers are $\mathfrak{A}_{\text{client}}(A) = \{\mathcal{J}_A^{\text{backoff}}\}$ and $\mathfrak{A}_{\text{client}}(B) = \{\mathcal{J}_B^{\text{backoff}}\}$. Every other $\mathfrak{A}_{\text{client}}(c)$ is empty. Table 2 lists the three relevant replays. In its final row, $\mathcal{J}^{\text{backoff}}$ denotes the backoff rule for the component in that row.

Table 2: Replays under the allowed client-policy alternatives.

replacement	gateway at its update	final D	ψ_D
$A \mapsto \mathcal{J}_A^{\text{backoff}}$	surge (B retries)	overloaded	true
$B \mapsto \mathcal{J}_B^{\text{backoff}}$	surge (A retries)	overloaded	true
$\{A, B\} \mapsto \mathcal{J}^{\text{backoff}}$	normal	healthy	false

Proposition 6.5 (Retry amplification). *Let $\widehat{\mathcal{M}}_{\text{svc}}$ be the ordinary model just defined, with no components held fixed, and let Π_{svc} be the policy profile defined above. Relative to f_0 , ρ_{inc} , and $\mathfrak{A}_{\text{client}}$, the installed pair $k_{\{A,B\}}^{\text{act}} = (\mathcal{I}_A^{\text{agg}}, \mathcal{I}_B^{\text{agg}})$ is a minimal execution-relative cause of database overload.*

$$\text{Cause}_{\widehat{\mathcal{M}}_{\text{svc}}, \Pi_{\text{svc}}, \mathfrak{A}_{\text{client}}}^{\text{er}}(\rho_{\text{inc}}; K_{\{A,B\}} = k_{\{A,B\}}^{\text{act}}, p_{D=\text{overloaded}}).$$

Proof. For **RC1**, $D = \text{overloaded}$ at f_6 . For **RC2**, replacing both aggressive-retry rules by backoff makes G normal, then Q normal, S healthy, and D healthy under the same schedule. For **RC3**, replacing only one rule leaves the other caller population retrying, so G still surges and the final database overload remains. The pair satisfies the dependence test; each singleton fails it. $\square$

Remark 6.6 (The answer depends on the allowed alternatives). We suppose the allowed alternatives also include the capped gateway rule

$$\mathcal{J}_G^{\text{cap}}(g, a, b) = \begin{cases} \text{capped} & a = \text{retry} \text{ or } b = \text{retry}, \\ \text{normal} & \text{otherwise.} \end{cases}$$

Because $\mathcal{I}_Q$ returns **normal** when the gateway is either **normal** or **capped**, $K_G = \mathcal{I}_G^{\text{open}}$ is an additional singleton cause for the same final outcome. It is incomparable by set inclusion with the client pair $\{K_A, K_B\}$. The client pair remains a cause because neither client rule alone prevents the overload. The same incident has minimal explanations containing different numbers of components, depending on the allowed alternatives. $\square$

An influence-closed region containing A and B must also contain S , Q , and G . Because the effect concerns D , the region must contain D as well. The smallest closed region containing the proposed cause and the effect is the full model. Equivalently, the feedback cycle $S \rightarrow A, B \rightarrow G \rightarrow Q \rightarrow S$ crosses the caller–service boundary, so that boundary is not an admissible cut.

The gateway can also prevent the overload. Platform is authorized to apply θ_{cap} , which replaces $\mathcal{I}_G^{\text{open}}$ by $\mathcal{J}_G^{\text{cap}}$. We define

$$\begin{aligned}\chi_{\text{safe}} := & (p_{G=\text{normal}} \vee p_{G=\text{capped}}) \wedge p_{Q=\text{normal}} \\ & \wedge (p_{S=\text{healthy}} \vee p_{S=\text{slow}}) \wedge p_{D=\text{healthy}}.\end{aligned}$$

Proposition 6.7 (Limiting requests at the gateway). *Let $x_{\text{svc}}(f)$ denote the service model at configuration f with policy Π_{svc} . At every f satisfying χ_{safe} , $x_{\text{svc}}(f) \models \langle \theta_{\text{cap}} \rangle \Box^* \chi_{\text{safe}}$. It follows that $x_{\text{svc}}(f) \models \langle \theta_{\text{cap}} \rangle \Box^* \neg p_{D=\text{overloaded}}$.*

Proof. We consider each possible scheduled component. An update of A or B may produce retries but leaves every part of χ_{safe} unchanged. The capped gateway can produce only **normal** or **capped**, so Q remains normal. An update of S or D produces **healthy**. Every update, including a stuttering update, preserves χ_{safe} . Induction over the schedule proves the result. $\square$

Proposition 6.5 identifies the two caller rules that caused the recorded failure. Proposition 6.7 proves that changing the gateway prevents the failure on every continuation.

6.3. Causal Analysis in Influence-Closed Subsystems

To analyse a recorded execution within a subsystem, delete steps that update components outside an influence-closed region and keep only the component values inside it. We call the resulting execution a *projection*. Let $E \subseteq \mathcal{C}$ be influence-closed. For a scheduled execution $\rho : f_0 \xrightarrow{s_1} \dots \xrightarrow{s_n} f_n$, let $i_1 < \dots < i_m$ be exactly the indices for which $s_{i_j} \in E$. We define its *E-projection* by retaining those labelled steps and restricting the retained configurations to E .

$$\rho \upharpoonright_E : f_0 \upharpoonright_E \xrightarrow{s_{i_1}} f_{i_1} \upharpoonright_E \xrightarrow{s_{i_2}} \dots \xrightarrow{s_{i_m}} f_{i_m} \upharpoonright_E.$$

If no scheduled component belongs to E , the projection is the execution with no steps at $f_0 \upharpoonright_E$. Steps scheduled outside E leave components in E unchanged, so the last configuration of the projected execution is $f_n \upharpoonright_E$.

Lemma 6.8 (Projection of an execution). *If E is influence-closed, then $\rho \upharpoonright_E$ is an execution of the live restriction $\widehat{\mathcal{M}} \upharpoonright_E^{\text{live}}$.*

Proof. We consider a retained step labelled by $c \in E$. Influence closure implies $\text{Inf}(c) \subseteq E$, so the global and restricted executions apply the same rule to the same current behaviour of c and the same input tuple. They produce the same update. A deleted step schedules a component outside E and leaves the restricted configuration on E unchanged. The removal of the deleted steps yields exactly the displayed execution. $\square$

The argument also applies after selected rules have been replaced.

Theorem 6.9 (Projection preserves replay). *Let E be influence-closed, let $T \subseteq \mathcal{C}$ be finite, and let $j_T \in \prod_{c \in T} \mathfrak{A}(c)$. We write $j_{T \cap E}$ for the restriction of this replacement tuple to $T \cap E$. Then*

$$\rho^{T \leftarrow j_T} \upharpoonright_E = (\rho \upharpoonright_E)^{T \cap E \leftarrow j_{T \cap E}}$$

as labelled executions of the live restriction. In particular, $f_n^{T,j} \upharpoonright_E = \text{last}((\rho \upharpoonright_E)^{T \cap E \leftarrow j_{T \cap E}})$.

Proof. We use induction over the global schedule. A step outside E leaves the restricted configuration on E unchanged for every installed rule. For a step $s_i \in E$, the replay uses a replacement exactly when $s_i \in T \cap E$. Both the global replay and the replay within the live restriction use that same rule, and influence closure gives the same input tuple in the two executions. The two replays agree on E after each retained step. The deletion of the other steps yields the claimed equality of executions and final configurations. $\square$

Corollary 6.10 (Minimal causes lie in the region and the schedule). *Let $E \subseteq \mathcal{C}$ be influence-closed, let ρ be a finite scheduled execution, and let ψ_E be a Boolean state formula well typed on E . Every minimal execution-relative cause C^* of ψ_E satisfies*

$$C^* \subseteq E \cap \text{occ}(\text{sched}(\rho)),$$

where occ returns the components occurring in the schedule.

Proof. Let j_{C^*} demonstrate **RC2**. By Theorem 6.9, replacing C^* has the same effect on the final outcome as replacing only $C^* \cap E$. If this intersection were empty, the projected replay would agree with the recorded final outcome, contradicting **RC2**. If it were non-empty and proper, its restricted replacement tuple would satisfy **RC2**, contradicting **RC3**. It follows that $C^* \subseteq E$.

A replacement rule is never used at a component absent from the recorded schedule. The removal of all such components from the replacement tuple leaves the replay unchanged. If the resulting tuple were empty, **RC2** would fail. If it were a proper subset, **RC3** would fail. No unscheduled component belongs to C^* . $\square$

Theorem 6.11 (Causes can be found within a closed region). *Let $E \subseteq \mathcal{C}$ be influence-closed, let ρ be a finite scheduled execution, let $\emptyset \neq C^* \subseteq E$ be finite, and let ψ_E be a Boolean state formula well typed on E . Let $\mathfrak{A} \upharpoonright_E$ retain only the allowed alternatives on E . Then*

$$\text{Cause}_{\widehat{\mathcal{M}}, \Pi, \mathfrak{A}}^{\text{er}}(\rho; K_{C^*} = k_{C^*}^{\text{act}}, \psi_E)$$

holds if and only if

$$\text{Cause}_{\widehat{\mathcal{M}}_E^{\text{live}}, \Pi_E, \mathfrak{A}_E}^{\text{er}}(\rho \upharpoonright_E; K_{C^*} = k_{C^*}^{\text{act}}, \psi_E)$$

holds in the live restriction.

Proof. Because ψ_E is Boolean and uses only components in E , it has the same truth value at a global configuration and at its E -restriction. Lemma 6.8 makes **RC1** identical globally and locally. For every replacement of C^* , Theorem 6.9 makes **RC2** identical as well. The same reasoning for every non-empty proper subset of C^* makes the **RC3** tests identical. $\square$

The test needs no component outside an influence-closed region that contains the cause and effect. Feedback may occur within E because replay follows the recorded schedule.

Corollary 6.12 (Finding causes on either side of a cut). *If $\text{cut}(\lambda) = (C_L, I, C_R)$ is an admissible cut, then Theorem 6.11 applies with $E = C_R$ and, symmetrically, with $E = C_L$.*

Proof. Both sides are influence-closed by Lemma 3.12. $\square$

For a stable cut, we can use a fixed-interface restriction. Let ρ_R^{proj} be obtained from $\rho|_{C_R}$ by deleting interface-labelled steps.

Corollary 6.13 (Local causation at a stable cut). *Let $\text{cut}(\lambda) = (C_L, I, C_R)$ be stable, let $\emptyset \neq C^* \subseteq C_R \setminus I$, and let ψ_R be a Boolean state formula well typed on C_R . Then the cause statement for the full model and ψ_R in Theorem 6.11 is equivalent to the same judgement in the fixed-interface restriction $\widehat{\mathcal{M}}|_{\lambda, C_R}$ on the projected execution ρ_R^{proj} .*

Proof. By stability, every retained interface step in the live C_R -restriction changes no value. All replacement targets lie in $C_R \setminus I$, so those steps stutter in the actual execution and in every replay used by **RC2** or **RC3**. Their deletion yields exactly the fixed-interface execution and its replays, with every claim about the final configuration preserved. $\square$

Example 6.14 (A cause found in one railroad subsystem). We return to the railroad model and the initial configuration f_0 from Example 3.8. We take an augmented version of that model with the same rules and a policy profile Π consistent with them. Its recorded execution is

$$\rho_{\text{rail}} : f_0 \xrightarrow{\text{RT}}_{\mathcal{I}} f_1 \xrightarrow{\text{Arr}}_{\mathcal{I}} f_2.$$

It ends with $\text{Arr} = \text{on}$. We allow one alternative rule for RT:

$$\mathcal{J}_{\text{RT}}^{\text{stay}}(x, s) = x, \quad \mathfrak{A}(\text{RT}) = \{\mathcal{J}_{\text{RT}}^{\text{stay}}\},$$

and let $\mathfrak{A}(c) = \emptyset$ for every other component. In the replay using this alternative, the update of RT leaves it off. The following update of Arr leaves Arr off. The installed rule at RT satisfies **RC1** and **RC2** for the final claim $p_{\text{Arr}=\text{on}}$. The proposed cause has no non-empty proper subset, so **RC3** also holds.

We name the cut in Example 3.15 λ_{rail} . Its right side is $C_2 = \{\text{Sw}, \text{LT}, \text{RT}, \text{Arr}\}$ and its interface is $I = \{\text{Sw}\}$. The rule for **Sw** returns its current setting, so the cut is stable, and $\text{RT} \in C_2 \setminus I$. By Corollary 6.13, the installed rule at **RT** is an execution-relative cause of $\text{Arr} = \text{on}$ in the full model if and only if it is a cause in the fixed-interface restriction on C_2 . The local calculation uses four of the five components. It omits **Audit** and gives the same causal judgement. $\square$

6.4. Correspondence with Structural Causal Models

The locality results above use the transition-system replay semantics directly. A finite recorded execution can also be unfolded into an acyclic SCM in which interventions on rule variables reproduce exactly the rule replacements of Definition 6.2. Theorem 6.11 does not use this SCM representation.

Throughout this subsection, $\mathcal{C}$ is finite, so that the causal model has finitely many variables. The definitions of updates and execution-relative causes also apply to infinite component sets.

For a Boolean state formula ψ , let $\tau_i(\psi)$ be the corresponding claim about the causal variables at time i . It replaces each $p_{c=b}$ by $(V_c^i = b)$ and leaves the “not” and “and” structure unchanged.

We fix an augmented model $\widehat{\mathcal{M}}^Z$ with its current rules and allowed alternatives $\mathfrak{A}$. Let ρ be a finite scheduled execution of length n . We create one behaviour variable for each component and time, and one rule variable for each component. The resulting SCM is $M_{\text{unf}}^K(n, \widehat{\mathcal{M}}, \mathfrak{A}) = \langle U, V^K, F^K \rangle$. Variables fixed by the recorded execution are called *exogenous*. The equations do not determine their values. Variables determined by the equations are called *endogenous*. The sets U and V^K contain the exogenous and endogenous variables, respectively, and F^K contains the equations. The notation $\text{dom}(X)$ denotes the possible values of X .

1. The endogenous variables are $V^K = \{K_c \mid c \in \mathcal{C}\} \cup \{V_c^i \mid c \in \mathcal{C}, i = 0, \dots, n\}$, with $\text{dom}(K_c) = \{\mathcal{I}_c\} \cup \mathfrak{A}(c)$ and $\text{dom}(V_c^i) = \mathbb{B}(c)$.
2. The exogenous variables are $U = \{U_0, S^1, \dots, S^n\}$, where U_0 fixes the initial configuration and each S^i selects the component updated at step i . Their domains are $\text{dom}(U_0) = F_{\mathcal{C}}$ and $\text{dom}(S^i) = \mathcal{C} \setminus Z$ for $i = 1, \dots, n$. The recorded execution supplies the settings $\vec{u}_\rho = (U_0 = f_0, S^1 = s_1, \dots, S^n = s_n)$, which keep the schedule variables fixed throughout the causal analysis.
3. The update equations are

$$\begin{aligned} K_c &= \mathcal{I}_c, \\ V_c^0 &= U_0(c), \\ V_c^i &= \begin{cases} K_c(V_c^{i-1}, (V_d^{i-1})_{d \in \text{Inf}(c)}) & \text{if } S^i = c, \\ V_c^{i-1} & \text{if } S^i \neq c, \end{cases} \quad i = 1, \dots, n. \end{aligned}$$

Each K_c ranges over rules with the required inputs and output, and its equation uses no other variable determined by the model. Each equation for V_c^i refers only to behaviour variables from time $i-1$. The model $M_{\text{unf}}^K(n, \widehat{\mathcal{M}}, \mathfrak{A})$ is acyclic: no equation depends, directly or indirectly, on its own result. This remains true when G_{Inf} contains feedback. A causal model with circular equations would require additional conditions to guarantee a solution [37]. Under the context $\vec{u}_\rho$, its unique solution has $K_c = \mathcal{I}_c$ and $V_c^i = f_i(c)$.

For a finite replacement (T, j_T) , setting the rule variables as $[K_T \leftarrow j_T]$ produces the direct replay in this SCM. No equation for component behaviour changes. The same variable K_c is used each time c occurs in the schedule, so replacing it changes every scheduled use of c .

Lemma 6.15 (Rule variables reproduce replay). *Let $T \subseteq \mathcal{C}$ be finite and $j_T \in \prod_{c \in T} \mathfrak{A}(c)$. In $M_{\text{unf}}^K(n, \widehat{\mathcal{M}}, \mathfrak{A})$ under $\vec{u}_\rho$, the unique solution after $[K_T \leftarrow j_T]$ satisfies $V_c^i = f_i^{T,j}(c)$ for every $c \in \mathcal{C}$ and $i = 0, \dots, n$. Let m_ρ denote the causal model with its recorded initial configuration and schedule. Then, for every Boolean state formula ψ , $f_n^{T,j} \models_{\text{B}} \psi$ exactly when $m_\rho \models [K_T \leftarrow j_T] \tau_n(\psi)$.*

Proof. At time 0, the replay and the causal model both retain f_0 . For step i , assume $V_c^{i-1} = f_{i-1}^{T,j}(c)$ for every c . The recorded settings fix $S^i = s_i$. The changed rule variable K_{s_i} uses exactly the rule in $\mathcal{I}^{T \leftarrow j_T}$. Every other behaviour variable keeps its value from time $i-1$. The equation for $V_{s_i}^i$ gives the same value as the replay rule for $f_i^{T,j}(s_i)$, and all other components retain their previous behaviours. Induction proves the displayed equality for all times. The final equivalence follows from the definition of τ_n . $\square$

Theorem 6.16 (Replay and the causal model agree). *We assume $\mathcal{C}$ is finite. Let $\rho : f_0 \xrightarrow{s_1}_{\mathcal{I}, Z} \dots \xrightarrow{s_n}_{\mathcal{I}, Z} f_n$ be a finite scheduled execution, let ψ be a Boolean state formula, and let $C^* \subseteq \mathcal{C}$ be finite and non-empty. Then $\text{Cause}_{\widehat{\mathcal{M}}, \Pi, \mathfrak{A}}^{\text{et}}(\rho; K_{C^*} = k_{C^*}^{\text{act}}, \psi)$ holds if and only if the following conditions hold in $(M_{\text{unf}}^K(n, \widehat{\mathcal{M}}, \mathfrak{A}), \vec{u}_\rho)$.*

1. $\tau_n(\psi)$ is true in the solution under the recorded settings.
2. For some $j_{C^*} \in \prod_{c \in C^*} \mathfrak{A}(c)$, $[K_{C^*} \leftarrow j_{C^*}] \neg \tau_n(\psi)$.
3. No non-empty proper subset $C' \subsetneq C^*$ satisfies the preceding dependence condition for any assignment $j_{C'} \in \prod_{c \in C'} \mathfrak{A}(c)$.

Definition 6.2 states a minimal dependence of the final effect on the rule variables: changing them prevents the effect, and no smaller non-empty set does. The initial configuration, schedule, final claim, and allowed alternatives remain fixed.

Proof. For **RC1**, the equations for the rule variables give $K_c = \mathcal{I}_c$ for every c , and the model without a rule change has $V_c^n = f_n(c)$. The statement $f_n \models_{\text{B}} \psi$ holds exactly when its translation $\tau_n(\psi)$ holds under the recorded settings.

For **RC2**, Lemma 6.15 matches each direct replacement tuple j_{C^*} with the change to the rule variables $[K_{C^*} \leftarrow j_{C^*}]$. The final effect is false in the same-schedule replay if and only if its corresponding claim at time n is false in the solution after that change.

Proper subsets of the rule variables K_{C^*} correspond one-to-one with proper subsets of C^* . The equivalence for **RC2** makes **RC3** identical in both formulations. $\square$

The SCM represents the replay semantics; it adds no causal criterion. Its rule-variable interventions reproduce exactly the counterfactual replays used by Definition 6.2. The locality results above remain statements about the transition-system semantics itself. Halpern–Pearl actual causation adds conditions that allow other variables to be held fixed and test whether the conclusion survives specified changes. These conditions handle cases in which one potential cause prevents another from acting. When a system evolves over time, time indices record successive uses of the selected rules. They can be omitted from a model of one update whose equations contain no circular dependency.

7. Compositional Verification of Repairs

To verify a repair, we consider every possible order of later updates after changing one subsystem. Under a stable cut (Definition 3.13), a proof about the unchanged subsystem remains valid when the intervention changes rules only on the other side. Definition 7.1 calls such a change *cut-local*.

Definition 7.1 (Cut-local intervention). Let $\text{cut}(\lambda) = (C_L, I, C_R)$ be a named admissible cut. An intervention label θ is λ -right-local when $\text{tg}(\theta) \subseteq C_R \setminus I$, and λ -left-local when $\text{tg}(\theta) \subseteq C_L \setminus I$. $\square$

Lemma 7.2 (Cut preservation). *A cut-local intervention preserves admissibility. If the cut is stable before the intervention, it remains stable afterwards.*

Proof. An intervention preserves every input set, so the three requirements for the cut are unchanged. A cut-local target is disjoint from the interface, so every rule for a shared component is also unchanged. $\square$

For any admissible cut, a change on one side leaves the fixed-interface restriction on the other side unchanged.

Lemma 7.3 (A change on one side preserves the other). *Let $x = (\widehat{\mathcal{M}}, f, \Pi)$ have the named admissible cut $\text{cut}(\lambda) = (C_L, I, C_R)$. Let θ be λ -right-local, and let α be any formula well typed on C_L . We write $x_L = (\widehat{\mathcal{M}} \upharpoonright_{\lambda, C_L}, f \upharpoonright_{C_L}, \Pi \upharpoonright_{C_L})$. Then the fixed-interface restriction of x_θ on the left is x_L , and $x \models \alpha *_\lambda \top$ iff $x \models \langle \theta \rangle (\alpha *_\lambda \top)$.*

Proof. On C_L , the intervention changes neither the current configuration nor the rules or policy entries. It also preserves which components are excluded from scheduling and which cut and intervention labels are available. The two left restrictions are identical. Admissibility is preserved by Lemma 7.2, and the satisfaction clause for $*_\lambda$ yields the displayed equivalence. $\square$

To reason about all continuations, we first characterize the configurations that the full system can reach. For a system model $\mathcal{N}$ at configuration g , write

$$\text{Reach}_{\mathcal{N}}(g) = \{h \mid g(\Delta_{\mathcal{N}})^*h\},$$

where $\Delta_{\mathcal{N}}$ denotes its current relation of possible updates. For sets $R_L \subseteq F_{C_L}$ and $R_R \subseteq F_{C_R}$, their *compatible join* contains each full configuration whose left and right parts belong to the two sets:

$$R_L \bowtie_I R_R = \{h \in F_{C_L \cup C_R} \mid h \upharpoonright_{C_L} \in R_L \text{ and } h \upharpoonright_{C_R} \in R_R\}.$$

Every configuration reachable in a fixed-interface restriction agrees with the initial configuration on I , so compatibility is automatic.

Theorem 7.4 (Reachability across a stable cut). *Let $\text{cut}(\lambda) = (C_L, I, C_R)$ be stable in $\mathcal{M}^Z$, and let f be a configuration. We put $\mathcal{M}_L = \mathcal{M}^Z \upharpoonright_{\lambda, C_L}$ and $\mathcal{M}_R = \mathcal{M}^Z \upharpoonright_{\lambda, C_R}$. Then $\text{Reach}_{\mathcal{M}^Z}(f) = \text{Reach}_{\mathcal{M}_L}(f \upharpoonright_{C_L}) \bowtie_I \text{Reach}_{\mathcal{M}_R}(f \upharpoonright_{C_R})$.*

If θ is λ -right-local, then with $\mathcal{M}_{\theta}^Z$ the intervened model and $\mathcal{M}_{R, \theta} = \mathcal{M}_{\theta}^Z \upharpoonright_{\lambda, C_R}$, $\text{Reach}_{\mathcal{M}_{\theta}^Z}(f) = \text{Reach}_{\mathcal{M}_L}(f \upharpoonright_{C_L}) \bowtie_I \text{Reach}_{\mathcal{M}_{R, \theta}}(f \upharpoonright_{C_R})$.

The intervention leaves the reachable set on the left unchanged.

Proof. We prove the second equality. The omission of θ gives the first. The cut remains stable by Lemma 7.2.

For the inclusion from left to right, take a finite global path from f to h . We delete every step scheduled in $C_R \setminus I$ and every interface step. Interface steps stutter by stability, and a step in $C_L \setminus I$ reads only C_L , so the remaining sequence is a path of $\mathcal{M}_L$ from $f \upharpoonright_{C_L}$ to $h \upharpoonright_{C_L}$. Symmetrically, deleting steps in $C_L \setminus I$ and interface steps yields a path of $\mathcal{M}_{R, \theta}$ to $h \upharpoonright_{C_R}$. The configuration h belongs to the compatible join.

For the reverse inclusion, choose any $g_L \in \text{Reach}_{\mathcal{M}_L}(f \upharpoonright_{C_L})$ and $g_R \in \text{Reach}_{\mathcal{M}_{R, \theta}}(f \upharpoonright_{C_R})$. By the definition of $\text{Reach}(\cdot)$, finite local paths reach g_L and g_R . We follow the left path first in the full model after the intervention. Each step uses the same rule and the same inputs as in the left restriction. The behaviours of the components in $C_R \setminus I$ and the interface remain fixed. Then follow the right path. The same argument applies there. The left restriction remains g_L . The two local configurations have a unique union whose restrictions are g_L and g_R . Every compatible pair is globally reachable. $\square$

Corollary 7.5 (Number of reachable configurations). *Under the hypotheses of Theorem 7.4, let θ be a λ -right-local intervention. If $\text{Reach}_{\mathcal{M}_L}(f \upharpoonright_{C_L})$ and $\text{Reach}_{\mathcal{M}_{R,\theta}}(f \upharpoonright_{C_R})$ are finite, then*

$$|\text{Reach}_{\mathcal{M}_\theta^Z}(f)| = |\text{Reach}_{\mathcal{M}_L}(f \upharpoonright_{C_L})| |\text{Reach}_{\mathcal{M}_{R,\theta}}(f \upharpoonright_{C_R})|.$$

Proof. Every configuration reachable in either fixed-interface restriction agrees with $f \upharpoonright_I$ on the interface. Every pair of locally reachable configurations is compatible and has a unique union on $C_L \cup C_R$. We apply Theorem 7.4. $\square$

For an ordinary model, stability is also necessary if this equality is to hold from every initial configuration.

Theorem 7.6. *Let $\mathcal{M}$ be an ordinary model with $Z = \emptyset$, and let $\text{cut}(\lambda) = (C_L, I, C_R)$ be admissible. The cut is stable iff for every initial configuration f ,*

$$\text{Reach}_{\mathcal{M}}(f) = \text{Reach}_{\mathcal{M} \upharpoonright_{\lambda, C_L}}(f \upharpoonright_{C_L}) \bowtie_I \text{Reach}_{\mathcal{M} \upharpoonright_{\lambda, C_R}}(f \upharpoonright_{C_R}).$$

Proof. Stability implies the equality by Theorem 7.4. Conversely, suppose the cut is not stable. Then for some $i \in I$, behaviour $b \in \mathbb{B}(i)$, and values u for its interface inputs, $\mathcal{I}_i(b, u) \neq b$. Because F contains every well-typed configuration, choose f realizing $f(i) = b$ and the tuple u on $\text{Inf}(i)$. Since $Z = \emptyset$, scheduling i produces a next full configuration f' with $f'(i) \neq f(i)$. Both fixed-interface restrictions hold every component of I fixed, so every configuration in their compatible join still restricts to $f \upharpoonright_I$. The globally reachable configuration f' is absent from the join, contradicting the stated equality. $\square$

The two local analyses recover reachability from every initial configuration only if the unrestricted system also leaves the interface unchanged.

An *invariant* is a property that holds now and after every later update. The following theorem gives the condition under which two subsystem invariants hold in the full system.

Theorem 7.7 (Combining invariants across a stable cut). *Let $x = (\widehat{\mathcal{M}}^Z, f, \Pi)$ have a named cut $\text{cut}(\lambda) = (C_L, I, C_R)$ that is stable, let θ be λ -right-local, let α_L be well typed on C_L , and let α_R be well typed on C_R . Let x_L be the fixed-interface restriction of x on the left, and let $x_{R,\theta}$ be the restriction on the right after applying θ . Then $x \models \langle \theta \rangle \Box^*(\alpha_L *_{\lambda} \alpha_R)$ iff $x_L \models \Box^* \alpha_L$ and $x_{R,\theta} \models \Box^* \alpha_R$.*

Proof. At every globally reachable configuration g , the fixed-interface restriction on the left uses the model and policy of x_L at configuration $g \upharpoonright_{C_L}$. After θ , the right restriction uses the model and policy of $x_{R,\theta}$ at configuration $g \upharpoonright_{C_R}$. Only the current configuration changes. Each local formula can be evaluated in its own restriction.

For the left-to-right direction, take any configuration reachable in one local restriction and pair it with the initial configuration of the other. The right-to-left

direction of Theorem 7.4 yields a global configuration with this pair of restrictions. The assumed global formula then gives the required local formula. For the right-to-left direction, the restriction of every globally reachable configuration to either side is locally reachable, so both restrictions satisfy their respective formulas. The current configuration is included by $\Box^*$. $\square$

A frame rule carries a property of an unchanged subsystem across an intervention on the other subsystem.

Corollary 7.8 (Frame rule). *Let $x = (\widehat{\mathcal{M}}^Z, f, \Pi)$ have stable cut $\text{cut}(\lambda) = (C_L, I, C_R)$, let θ be λ -right-local, and let ι_L be well typed on C_L . If the fixed-interface restriction x_L on the left satisfies $\Box^* \iota_L$, then $x \models \langle \theta \rangle \Box^* (\iota_L *_{\lambda} \top)$. The left-local version is symmetric.*

Proof. We apply Theorem 7.7 with $\alpha_R = \top$. $\square$

7.1. Necessity of the Frame Conditions

The input assumptions apply to every rule with the specified inputs and output. The following examples give countermodels when input closure or stability fails. Proposition 7.10 proves necessity of influence closure, and Theorem 7.6 proves necessity of stability for the universal reachability equality.

Example 7.9. We take components l, r with $E = \{r\}$ and $l \in \text{Inf}(r)$. Let both behaviour sets be $\{0, 1\}$ and choose a rule for r that copies l . Two global configurations can agree on r and assign distinct values to l . A scheduled update of r yields one next behaviour in each configuration. No transition rule based only on the restricted configuration over E can reproduce both updates, so Theorem 6.9 cannot be guaranteed without influence closure. $\square$

Proposition 7.10 (Why influence closure is necessary). *Let $E \subseteq D$. We suppose some $c \in E$ has a declared input $d \in \text{Inf}(c) \setminus E$, and suppose both $\mathbb{B}(c)$ and $\mathbb{B}(d)$ contain at least two behaviours. Then there exists a collection of rules respecting the declared input sets for which scheduling c produces a restricted configuration on E that is not determined by the current restricted configuration. Except when a component has only one possible behaviour, any projection theorem that covers every rule with the stated inputs and outputs requires influence closure.*

Proof. We choose distinct $d_0, d_1 \in \mathbb{B}(d)$ and distinct $c_0, c_1 \in \mathbb{B}(c)$. We define a permitted rule for c that returns c_0 when the d -input is d_0 and c_1 when it is d_1 . Its outputs on all other inputs may be arbitrary. We choose two global configurations that agree on all of E and on every other input of c . We assign d_0 and d_1 to d in the respective configurations. The scheduling of c produces c_0 and c_1 , respectively, from configurations that initially agree on E . No transition function of the restricted configuration on E alone can reproduce both global updates. $\square$

Example 7.11. We take $C_L = \{l, i\}$, $C_R = \{i, r\}$, and $I = \{i\}$. Let r have no inputs, let the rule for the shared component copy r , and let l copy i . The left and right first two input requirements hold. The requirement for shared components fails because $r \in \text{Inf}(i)$. A right-local replacement of the rule at r , followed by updates of r, i, l , can change l . A frame statement for the left side is invalid for this cut. $\square$

Example 7.12. We take an admissible cut with one interface bit i whose input set is empty and whose rule toggles the bit. An update of i in the full model changes both the C_L - and C_R -restrictions. Each fixed-interface restriction holds i fixed. The next full configuration cannot be obtained by joining two configurations reachable in the fixed-interface restrictions, as in the converse direction of Theorem 7.6. $\square$

7.2. Local Verification of Repairs

Theorem 7.4 allows us to check a repair in the two subsystem models. We accept a change when the failure is possible before the repair, the unchanged subsystem preserves its required property, and every configuration reachable after the change is safe.

Definition 7.13 (Local repair test). We fix $x = (\widehat{\mathcal{M}}^Z, f, \Pi)$ with stable cut $\text{cut}(\lambda) = (C_L, I, C_R)$, a formula ι_L well typed on C_L , and Boolean state formulas β and σ_R well typed on C_R . Let x_L and x_R be the two fixed-interface restrictions at f . A λ -right-local intervention θ passes the local repair test when

1. $x_R \models \neg\beta \wedge \Diamond^+\beta$.
2. $x_L \models \Box^*\iota_L$.
3. $(x_R)_\theta \models \Box^*(\sigma_R \wedge \neg\beta)$.

The first clause requires the failure to be initially absent and later reachable. The second gives the property preserved by the unchanged subsystem. The third requires every continuation of the modified right subsystem to satisfy σ_R and exclude the failure. The definition requires safety from the instant of installation. An intervention changes the rule but not the current configuration, and $\Box^*$ includes that initial configuration. A $\Box^+$ condition would omit only the configuration immediately after the rule is installed. A guarantee of eventual recovery would also need an assumption that the scheduler eventually chooses the required components. $\square$

Theorem 7.14 (The local and global repair checks agree). *Under Definition 7.13, a right-local intervention θ passes the local repair test if and only if the following three formulas hold: $x \models \neg\beta \wedge \Diamond^+\beta$, $x_L \models \Box^*\iota_L$, and $x \models \langle\theta\rangle\Box^*((\iota_L *_\lambda \sigma_R) \wedge \neg\beta)$.*

Proof. Because β is a Boolean state formula well typed on C_R , a configuration satisfying β is reachable globally exactly when one is reachable in the fixed-interface restriction on the right. The initial configuration has the same truth value for β in both models. Because β is initially false, any path that first reaches a configuration satisfying β must contain an update in $C_R \setminus I$. An update in $C_L \setminus I$ cannot change any

component in C_R . An update of a shared component changes nothing by stability. Since β is a Boolean state formula well typed on C_R , both kinds of step preserve its truth value. A non-empty global path reaches β exactly when a non-empty path in the right fixed-interface restriction does.

If the three local conditions hold, Theorem 7.7 combines the property preserved on the left with the property on the right $\Box^*(\sigma_R \wedge \neg\beta)$. Since β is right-local and Boolean, at each global configuration $\iota_L *_{\lambda} (\sigma_R \wedge \neg\beta)$ is equivalent to $(\iota_L *_{\lambda} \sigma_R) \wedge \neg\beta$. Conversely, every continuation of the right restriction can be paired with the initial left configuration to obtain a global continuation. The global formula implies the property on the right. The condition for the left side is the same in both formulations. $\square$

For finite systems, Theorem 7.14 supports checking each candidate in the subsystem it changes. Algorithm 2 states the procedure. The control relation and the sets $\text{Adm}(c)$ determine which candidates are authorized. The profile Π records the installed rules.

8. Finite-State Verification Algorithms

Theorems 6.11 and 7.14 yield algorithms when component behaviour sets, allowed alternatives, and repair candidates are finite. We write $\text{supp}(\psi)$ for the components mentioned by state atoms in a Boolean formula ψ , and $\text{Cl}_{\text{inf}}(S)$ for the least influence-closed set containing S . The operation **Replay** starts from the recorded initial configuration and applies the replacement rules under the recorded schedule. The operation **Reach** searches a graph whose nodes are configurations and whose arrows are possible updates. It includes the initial configuration.

Algorithm 1 Finding causes of a recorded execution

Require: finite $\widehat{\mathcal{M}}$, execution ρ , final claim ψ , alternatives $\mathfrak{A}$

Ensure: all minimal cause sets, each with one replacement that demonstrates it

```

1: if  $\text{last}(\rho) \not\models_{\text{B}} \psi$  then
2:   return  $\emptyset$  ▷ RC1 fails
3: end if
4:  $E \leftarrow \text{Cl}_{\text{Inf}}(\text{supp}(\psi))$ 
5:  $\mathcal{M}_E \leftarrow \widehat{\mathcal{M}}|_E^{\text{live}}$ ,  $\rho_E \leftarrow \rho|_E$ , and  $\mathfrak{A}_E \leftarrow \mathfrak{A}|_E$ 
6:  $Q \leftarrow \{c \in E \mid c \text{ occurs in } \text{sched}(\rho_E) \text{ and } \mathfrak{A}_E(c) \neq \emptyset\}$ 
7:  $\text{Causes} \leftarrow \emptyset$ 
8: for  $k = 1, \dots, |Q|$  do
9:   for all  $T \subseteq Q$  with  $|T| = k$  do
10:    if  $\exists (C, j_C) \in \text{Causes}$  with  $C \subsetneq T$  then
11:      continue ▷ RC3 fails
12:    end if
13:    for all  $j_T \in \prod_{c \in T} \mathfrak{A}_E(c)$  do
14:       $\rho' \leftarrow \text{REPLAY}(\mathcal{M}_E, \rho_E, T, j_T)$ 
15:      if  $\text{last}(\rho') \not\models_{\text{B}} \psi$  then ▷ RC2 holds
16:         $\text{Causes} \leftarrow \text{Causes} \cup \{(T, j_T)\}$ 
17:        break
18:      end if
19:    end for
20:  end for
21: end for
22: return  $\text{Causes}$ 

```

The closure in line 4 contains the components in the final claim and recursively adds all of their declared inputs. A component outside this closure cannot change the final outcome under replay. A component absent from the recorded schedule never uses its replacement rule. Theorem 6.9 and Corollary 6.10 justify lines 4–6. The algorithm examines smaller candidate sets first. Once it finds a cause, RC3 excludes every proper superset of that cause. The algorithm returns every minimal cause set and one set of replacement rules that demonstrates RC2 for each set.

Algorithm 2 Checking repairs locally

Require: finite x with stable cut λ , bad condition β , left invariant ι_L , right target σ_R , candidates Θ_R^{cand} that are authorized and right-local

Ensure: candidates that pass Definition 7.13

```
1: construct the fixed-interface restrictions  $x_L$  and  $x_R$  at  $x$ 
2:  $L \leftarrow \text{REACH}(x_L)$ 
3: if  $\exists \ell \in L$  with  $x_L[\ell] \not\models \iota_L$  then
4:   return  $\emptyset$  ▷ the left invariant fails
5: end if
6:  $R_0 \leftarrow \text{REACH}(x_R)$ 
7: if  $x_R \not\models \neg\beta$  or  $\nexists r \in R_0$  with  $r \models_{\text{B}} \beta$  then
8:   return  $\emptyset$  ▷ the check before repair fails
9: end if
10:  $\text{Repairs} \leftarrow \emptyset$ 
11: for all  $\theta \in \Theta_R^{\text{cand}}$  do
12:    $R_\theta \leftarrow \text{REACH}((x_R)_\theta)$ 
13:   if  $\forall r \in R_\theta : r \models_{\text{B}} \sigma_R \wedge \neg\beta$  then
14:      $\text{Repairs} \leftarrow \text{Repairs} \cup \{\theta\}$ 
15:   end if
16: end for
17: return  $\text{Repairs}$ , optionally ordered by a supplied ranking
```

When ι_L is a Boolean state formula, as in the finite examples below, the first test is a direct scan of the configurations in L . For a general C_L -local formula, it invokes the model-checking semantics at each pointed model $x_L[\ell]$.

The check before repair uses non-empty reachability. Since it first checks $\neg\beta$ at the initial configuration, searching the reachable set for a β -configuration is equivalent to the $\Diamond^+\beta$ clause. The algorithm searches the left restriction and the right restriction before repair once, then searches only the right restriction for each candidate. By Theorem 7.14, Algorithm 2 returns exactly the supplied candidates that pass the local repair test. The optional ranking does not affect which candidates pass.

9. Safety Verification in Retrieval-Augmented Generation

We apply Theorem 7.14 to a RAG system after retrieval has committed the context used by generation. We study safety in the generation subsystem. Analysing retrieval and generation together would also require a model of the earlier retrieval phase.

9.1. System Architecture

Retrieval-augmented generation (RAG) couples a retrieval pipeline with a large language model by selecting and ranking evidence that is then supplied to the

model as context [50, 51]. The components implementing retrieval, ranking, and subsequent processing may be neural networks, hand-written rules, search procedures, or compositions thereof. More generally, systems such as DSPy structure language-model applications as compositions of modules, including modules that invoke language models themselves [52].

Our model begins after retrieval and ranking have committed the context supplied to generation. A *committed* context remains fixed for the phase being analysed. The component **PortCtx** stores this context, and its rule returns the stored value when scheduled. We test whether an invariant established at the commit boundary remains valid when a release rule in the generation subsystem is analysed and replaced. Tracing an output back to earlier retrieval decisions would require a model of the preceding phase, including the transfer of context between the two phases. The example verifies release control after commitment but does not verify retrieval, and the invariant concerns only the committed context and its validation record.

We represent the system by the augmented model $\widehat{\mathcal{M}}_{\text{rag}}$ of Definition 3.22. In the full model for the phase after commit, $Z = \emptyset$, so every component may be scheduled. Its components are Tok, Retr, Rank, CtxOut, Prov, PortCtx, PortDecReq, Tool, Dec, Guard, Release, and Mem. At any configuration f , write $x_{\text{rag}}(f) = (\widehat{\mathcal{M}}_{\text{rag}}, f, \Pi)$. The retrieval side tokenizes the query, retrieves and ranks evidence, constructs a context in CtxOut, and uses Prov to record whether checks on the source of the context passed. The analysed phase begins after the context and any tool request have been committed to PortCtx and PortDecReq. The decoder reads those values. The guard accepts or blocks the draft, and Release decides whether to publish it.

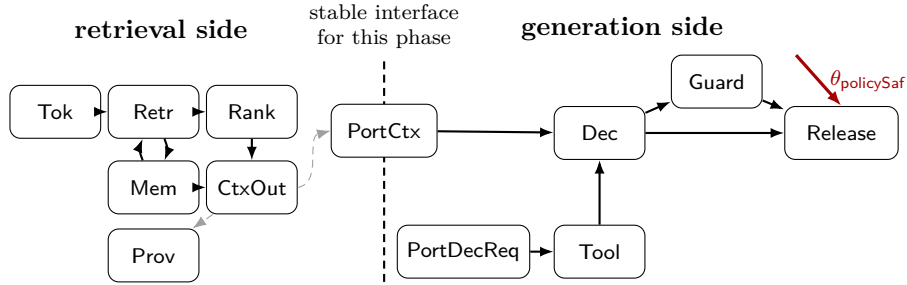


Figure 2: The RAG system after context is committed. Solid arrows show which components may read which others during the analysed phase, including both directions between Retr and Mem. Dashed grey arrows show work completed earlier. The shared component **PortCtx** is held fixed in each subsystem model, and the intervention changes only **Release**.

The behaviour sets are $\mathbb{B}(\text{Tok}) = \{\text{ok}, \text{err}\}$, $\mathbb{B}(\text{Retr}) = \{\text{none}, \text{rel}, \text{spur}\}$, $\mathbb{B}(\text{Rank}) = \{\text{good}, \text{bad}\}$, and $\mathbb{B}(\text{CtxOut}) = \{\text{short}, \text{long}, \text{noctx}\}$. $\mathbb{B}(\text{Prov}) = \{\text{valid}, \text{invalid}\}$, $\mathbb{B}(\text{PortCtx}) = \{\text{short}, \text{long}, \text{noctx}\}$, $\mathbb{B}(\text{PortDecReq}) = \{\text{off}, \text{on}\}$, and $\mathbb{B}(\text{Tool}) = \{\text{off}, \text{on}, \text{fail}\}$. $\mathbb{B}(\text{Dec}) = \{\text{safe}, \text{unsafe}, \text{abort}\}$, $\mathbb{B}(\text{Guard}) = \{\text{pass}, \text{block}\}$,

$\mathbb{B}(\text{Release}) = \{\text{hold}, \text{safe-out}, \text{unsafe-out}\}$, and $\mathbb{B}(\text{Mem}) = \{\text{hit}, \text{miss}\}$.

The declared input sets for the analysed phase are $\text{Inf}(\text{Tok}) = \emptyset$, $\text{Inf}(\text{Retr}) = \{\text{Tok}, \text{Mem}\}$, $\text{Inf}(\text{Rank}) = \{\text{Retr}\}$, and $\text{Inf}(\text{CtxOut}) = \{\text{Rank}, \text{Mem}\}$. Prov , PortCtx , and PortDecReq have no declared inputs: $\text{Inf}(\text{Prov}) = \text{Inf}(\text{PortCtx}) = \text{Inf}(\text{PortDecReq}) = \emptyset$. The remaining input sets are $\text{Inf}(\text{Tool}) = \{\text{PortDecReq}\}$, $\text{Inf}(\text{Dec}) = \{\text{PortCtx}, \text{Tool}\}$, $\text{Inf}(\text{Guard}) = \{\text{Dec}\}$, $\text{Inf}(\text{Release}) = \{\text{Dec}, \text{Guard}\}$, and $\text{Inf}(\text{Mem}) = \{\text{Retr}\}$. The components Tok , Retr , Rank , CtxOut , Tool , Dec , Guard , and Mem may use any rule defined for every permitted combination of inputs. The results below hold for every such choice.

The rule for **Release** is currently permissive.

$$\mathcal{I}_{\text{Release}}^{\text{perm}}(r, d, g) = \begin{cases} \text{unsafe-out} & d = \text{unsafe} \text{ and } g = \text{pass}, \\ \text{safe-out} & d = \text{safe} \text{ and } g = \text{pass}, \\ \text{hold} & \text{otherwise.} \end{cases}$$

During this phase, the rule $x \mapsto x$ applies to each of PortCtx , PortDecReq , and Prov . It returns the current value. All three remain schedulable in the full model, and scheduling any of them changes nothing.

9.2. Component Ownership and Interface Invariants

The agent set is $\mathcal{A}_{\text{rag}} = \{\text{Orch}, \text{Safety}\}$. The orchestrator's control set C_{Orch} contains Retr , Rank , CtxOut , Dec , and Tool . The safety owner's set C_{Safety} contains Guard and Release . For each controlled component c , the profile Π gives its installed rule from $\text{Adm}(c)$. For **Release**, the authorized choices are the permissive rule currently in force, $\mathcal{I}_{\text{Release}}^{\text{perm}}$, a rule that always holds the output, and a rule defined below that releases only safe drafts. Initially, $\Pi_{\text{Safety}}(\text{Release}) = \mathcal{I}_{\text{Release}}^{\text{perm}}$.

Let $\lambda = \lambda_{\text{retr-gen}}$ name the cut with interface $I^{\text{retr}} = \{\text{PortCtx}\}$. Its left component set C_1^{retr} contains Tok , Retr , Rank , Mem , CtxOut , Prov , and PortCtx . Its right component set C_2^{gen} contains PortCtx , PortDecReq , Tool , Dec , Guard , and Release . The input sets satisfy the requirements for analysing each side separately, and $\text{Inf}(\text{PortCtx}) = \emptyset \subseteq I^{\text{retr}}$. The rule for PortCtx returns its current value, which makes the cut stable. The commit shown by the dashed arrow in Figure 2 belongs to the preceding phase. In both fixed-interface restrictions, PortCtx is held fixed and cannot be scheduled.

The invariant supplied at the boundary of the analysed phase is

$$\iota_{\text{retr}} := p_{\text{Prov}=\text{valid}} \wedge p_{\text{PortCtx}=\text{short}}.$$

$\text{Prov} = \text{valid}$ records that the source of the committed context passed the stated checks. The model does not represent whether the context is factually correct. We assume that the initial configuration f of this phase satisfies ι_{retr} . The model proves preservation during this phase, not establishment during the preceding retrieval phase. Write $x_{\text{rag}}^L(f)$ for the left restriction. Since PortCtx is held fixed and the

rule for **Prov** returns its current value, $x_{\text{rag}}^L(f) \models \Box^* \iota_{\text{retr}}$. Other components on the retrieval side may continue to update, but their behaviour is not needed to prove this invariant.

The model leaves the private retrieval-side components schedulable during this phase but fixes the context exported through **PortCtx**. Those components may evolve internally without changing the value seen by the generation subsystem. Independent post-commit updates on the two sides produce the state-space product counted below. The counts apply to this finite model and do not estimate the reduction in a deployed RAG system.

9.3. Verification of Release Policies

A rule that always holds the output prevents unsafe publication but also blocks safe drafts. A guarded rule releases safe drafts and holds all others. Define $\mathcal{J}_{\text{Release}}^{\text{hold}}(r, d, g) = \text{hold}$ for every input, and define the guarded release rule

$$\mathcal{J}_{\text{Release}}^{\text{safe}}(r, d, g) = \begin{cases} \text{safe-out} & d = \text{safe} \text{ and } g = \text{pass}, \\ \text{hold} & \text{otherwise.} \end{cases}$$

Take

$$\text{Adm}(\text{Release}) = \{\mathcal{I}_{\text{Release}}^{\text{perm}}, \mathcal{J}_{\text{Release}}^{\text{hold}}, \mathcal{J}_{\text{Release}}^{\text{safe}}\}.$$

Let θ_{hold} install the hold rule and let $\theta_{\text{policySaf}}$ install the guarded rule. The safety owner may authorize either intervention. Both are right-local for λ . We compare two release rules by the inputs on which they release a safe output. For rules J_1, J_2 , write $J_1 \preceq_{\text{rel}} J_2$ when

$$\{(r, d, g) \mid J_1(r, d, g) = \text{safe-out}\} \subseteq \{(r, d, g) \mid J_2(r, d, g) = \text{safe-out}\}.$$

Under this ordering, J_2 ranks at least as high as J_1 when it releases safely on every input where J_1 does. We write $J_1 \prec_{\text{rel}} J_2$ when the first inclusion holds and the reverse inclusion fails.

Take a configuration f at the start of the phase that satisfies $f(\text{Dec}) = \text{unsafe}$, $f(\text{Guard}) = \text{pass}$, and $f(\text{Release}) = \text{hold}$. Let x_R be the fixed-interface restriction of $x_{\text{rag}}(f)$ on the right. Before intervention, scheduling **Release** under the permissive rule currently in force immediately gives **unsafe-out**. The right restriction satisfies $x_R \models \neg p_{\text{Release}=\text{unsafe-out}} \wedge \Diamond^+ p_{\text{Release}=\text{unsafe-out}}$.

Proposition 9.1. *At the configuration f above:*

1. *The permissive rule currently in force fails the safety condition.*
2. *Both θ_{hold} and $\theta_{\text{policySaf}}$ establish $\Box^* \neg p_{\text{Release}=\text{unsafe-out}}$ in x_R .*
3. *Among the rules in $\text{Adm}(\text{Release})$ whose installation makes unsafe release unreachable from x_R , $\mathcal{J}_{\text{Release}}^{\text{safe}}$ is the only rule that releases safely on every input where any other such rule does.*

Proof. For item 1, a **Release** step from the stated configuration returns **unsafe-out**. For item 2, the **hold** rule returns only **hold**; the guarded rule returns only **hold** or **safe-out**. Updates of other components leave the behaviour of **Release** unchanged. No schedule reaches **unsafe-out** from the stated configuration after either intervention.

For item 3, $\mathcal{J}_{\text{Release}}^{\text{hold}}$ never releases a safe output. The rule $\mathcal{J}_{\text{Release}}^{\text{safe}}$ releases a safe output exactly when $d = \text{safe}$ and $g = \text{pass}$, for every current behaviour r of **Release**. We have $\mathcal{J}_{\text{Release}}^{\text{hold}} \prec_{\text{rel}} \mathcal{J}_{\text{Release}}^{\text{safe}}$. Item 1 excludes the permissive rule. Together with the strict inclusion above, this proves item 3. $\square$

Both replacement rules satisfy the release-safety condition in x_R . When the retrieval-side invariant also holds, the local repair test combines it with the independently established release-safety condition. The guarded rule ranks higher because it releases safe output.

9.4. Preservation of the Retrieval Invariant

For an initial configuration f satisfying the boundary invariant, write $x = x_{\text{rag}}(f)$. Let x_L be its fixed-interface restriction on the left. Let x_R be the right restriction defined in Section 9.3. By the argument in Section 9.2, $x_L \models \Box^* \iota_{\text{retr}}$. By Proposition 9.1, $(x_R)_{\theta_{\text{policySaf}}} \models \Box^* \neg p_{\text{Release=unsafe-out}}$, and unsafe release is reachable on the right before the repair. Taking $\beta = p_{\text{Release=unsafe-out}}$ and $\sigma_R = \top$, all clauses of the local repair test in Definition 7.13 hold for both replacement rules. The formulas below use $\theta_{\text{policySaf}}$, the higher-ranked rule.

Theorem 7.14 yields these formulas for the full system:

$$x \models \neg p_{\text{Release=unsafe-out}} \wedge \Diamond^+ p_{\text{Release=unsafe-out}}$$

and

$$x \models \langle \theta_{\text{policySaf}} \rangle \Box^* ((\iota_{\text{retr}} *_{\lambda} \top) \wedge \neg p_{\text{Release=unsafe-out}}).$$

Theorem 7.7 also combines the retrieval invariant on the left with the release-safety invariant on the right. The retrieval-side preservation argument is reused for every candidate release rule.

For this finite example, Corollary 7.5 determines the number of configurations that the full system can reach. If N_L configurations of the retrieval side and N_R^θ configurations of the generation side are reachable under a candidate θ , then the full system can reach $N_L N_R^\theta$ configurations. After checking the N_L retrieval configurations once, each candidate requires only the reachable configurations on the right.

9.5. Verification Assumptions and Limitations

The model begins after **Retr** and **Rank** have produced the committed context. The component **PortCtx** retains that value throughout the phase. The guarantee covers a change to the release rule for that context. An analysis of retrieval and generation together would also require a model of the earlier retrieval phase.

10. Finite Model Verification

A Python program evaluates Algorithms 1 and 2 on the finite examples. It enumerates the configurations and transitions, replays the recorded schedules, determines which components must be included with the stated inputs, and finds the reachable configurations. It reproduces the reported causes, the three RAG traces, and the equality between the configurations reachable by the full model and all compatible pairs from the two subsystem sets. These checks concern only the finite models. They do not measure a service in operation. To compare the RAG release rules, the program examines every input (r, d, g) and records those for which each rule returns **safe-out**. It then orders these sets by inclusion.

For the retry model, the program reproduces every row of Table 1. Table 3 reports the causal search results. Here, starting from D , the calculation repeatedly adds the declared inputs of each retained component. It retains all six components, as predicted by the feedback analysis in Section 6.2. With only the two client alternatives, the program performs two singleton replays and one pair replay, returning exactly $\{A, B\}$, as Proposition 6.5 predicts. The addition of the gateway cap produces the additional singleton cause $\{G\}$ from Remark 6.6. Once $\{G\}$ is found, RC3 excludes its proper supersets, so only one additional replay is needed.

Table 3: Cause search for the retry model in Section 6.2.

components with alternatives	components in E	replays	minimal causes
A, B	6	3	$\{A, B\}$
A, B, G	6	4	$\{G\}, \{A, B\}$

The safety proof does not constrain most rules outside **Release**. To enumerate all reachable configurations, the finite model assigns each component one rule defined for every permitted input. In this model, a valid token produces relevant retrieval, a good rank, and a cache hit. The context is short exactly when the rank is good and the cache records a hit. The tool follows the committed request. The decoder returns a safe draft exactly when the context is short and the tool is on, and the guard passes exactly those safe drafts. In the global model, **Prov**, **PortCtx**, and **PortDecReq** use the identity rules specified in Section 9.1. The two local restrictions also hold **PortCtx** fixed. The complete initial configuration and these rules are given in `prototype/experiments.py`, in `build_rag_models`. The conditions in Section 9.3 list the facts about this initial configuration that the proof uses. Here, scheduling **Release** immediately under the permissive rule yields **unsafe-out**. Under the guarded rule, scheduling **Release** once leaves the release on hold. The schedule (**Tool**, **Dec**, **Guard**, **Release**) under the guarded rule reaches **safe-out**, demonstrating that the selected repair permits safe publication.

Table 4 reports the reachability counts. The program checks the left invariant over six configurations. It checks whether failure is reachable before repair over 13 configurations of the right restriction. The two candidates require six and seven

configurations of that restriction. The global calculations confirm the products $6 \cdot 13 = 78$, $6 \cdot 6 = 36$, and $6 \cdot 7 = 42$. Here, checking the permissive rule and both candidates examines 32 local configurations. Three global checks examine 156, a factor of 4.875 in this example.

Table 4: Configurations examined for the RAG model in Section 9.

release rule	right count	global count	result
permissive before repair	13	78	unsafe reachable
complete hold	6	36	accepted
guarded release	7	42	accepted, ranked first

On a 13th-generation Intel Core i7-13620H with Python 3.14.4, 100 repetitions gave median times of 0.45 ms for both retry cases and 12.10 ms for all RAG checks, including the separate calculations on the full model. These timings are only illustrative. The numbers of configurations and replays are fixed. The number of combinations considered while finding causes grows exponentially with the number of candidate components and alternatives. The work needed to find reachable configurations grows in direct proportion to the number of configurations and possible updates. Those numbers may themselves grow exponentially with the number of components. In these finite examples, subsystem analysis avoids enumerating compatible combinations with configurations of the unchanged subsystem. The reported ratios describe these particular finite models and are not general complexity improvement factors. Subsystem analysis does not change these upper bounds.

11. Validation of the Logical Semantics

The locality theorems do not depend on the results in this section. Formulas in the logic of Section 5 should not distinguish systems that agree on current facts and match one another’s updates, interventions, and named subsystem restrictions.

We define this behavioural equivalence as a bisimulation (cf. [53, 54]). Bisimilar pointed models satisfy the same well-typed formulas. On the stated image-finite class closed under the relevant constructions, agreement on all well-typed formulas also yields a bisimulation.

The compared models use the same signature Σ . Throughout this section, they also use the same agent set, control relation, and authorization sets $\text{Adm}(c)$. Every intervention label θ is interpreted in each model by an authorized policy intervention with declared target $\text{tg}(\theta)$. The particular replacement associated with a label may be model-dependent; the intervention clause of bisimulation compares the resulting intervened models. By Definition 5.2, each model’s intervention targets and retained cuts match those declared in the signature. These results concern the formulas of Section 5. The causal test also depends on transition labels and a recorded execution.

Definition 11.1 (Image-finiteness). A class of pointed models based on augmented Σ -models is *image-finite*, meaning each model, including any model obtained by intervention or restriction, has only finitely many immediate successors. The label sets Θ and Λ may still be infinite. $\square$

Definition 11.2 (Bisimulation). For $x_i = (\widehat{\mathcal{M}}_i, f_i, \Pi_i)$, let Z_i be the components held fixed in $\widehat{\mathcal{M}}_i$. We write $x_i[y]$ for model i with the same policy at state y , and write x_i^θ for the result of applying θ to x_i . A relation $\mathcal{R}$ between pointed Σ -models with the same current component set is a *bisimulation for the language* if the following conditions hold whenever $x_1 \mathcal{R} x_2$.

1. **State facts.** The two states agree on every state atom that uses the current component set.
2. **Steps.** If $f_1 \Delta_{\mathcal{I}_1, Z_1} y_1$, some y_2 satisfies $f_2 \Delta_{\mathcal{I}_2, Z_2} y_2$ and $x_1[y_1] \mathcal{R} x_2[y_2]$. The same condition holds with the roles of f_1 and f_2 reversed.
3. **Interventions.** For every θ whose target set lies in the current component set, $x_1^\theta \mathcal{R} x_2^\theta$.
4. **Named cuts.** For every $\lambda \in \Lambda$ whose declared component set is the current component set, the two models agree on the availability of that cut. If

$$\text{cut}_1(\lambda) = \text{ct}(\lambda) = (C_1, I, C_2),$$

then $\text{cut}_2(\lambda) = \text{ct}(\lambda)$ is also defined and admissible relative to the current component set. For $j = 1, 2$, write $x_{i,\lambda,j} = (\widehat{\mathcal{M}}_i \upharpoonright_{\lambda, C_j}, f_i \upharpoonright_{C_j}, \Pi_i \upharpoonright_{C_j})$. We require $x_{1,\lambda,j} \mathcal{R} x_{2,\lambda,j}$. The same condition holds with the models exchanged. $\square$

Theorem 11.3 (Bisimulation invariance). *If $x_1 \mathcal{R} x_2$ for a bisimulation, then $x_1 \models \varphi$ if and only if $x_2 \models \varphi$ for every formula $\varphi \in \mathcal{L}_\Sigma$ well typed at their current component set.*

Proof. We proceed by induction on how the formula is built. Simple propositions and the cases for “not” and “and” are immediate. The forward and backward step clauses, together with the induction hypothesis, prove the $\Diamond$ - and $\Box$ -cases. Repeated use of those clauses matches every finite non-empty path and proves the $\Diamond^+$ - and $\Box^+$ -cases. For $\langle \theta \rangle \psi$, the intervention clause relates the two intervened models, whose configurations are unchanged. We apply the induction hypothesis to ψ . For $\varphi *_\lambda \psi$, the definition uses the same named cut in both models and relates their left and right restrictions. The two induction hypotheses give the two required parts of the satisfaction rule. The reverse implications use the same matching conditions in the opposite direction. $\square$

Theorem 11.4 (Hennessy–Milner converse). *Let x_1 and x_2 be pointed models based on augmented Σ -models with the same current component set. We suppose they*

belong to an image-finite class. We assume that the class is closed under taking pointed successors, under every intervention label available at the current component set, and under every available named fixed-interface restriction. If $x_1 \models \varphi$ if and only if $x_2 \models \varphi$ for every well-typed Σ -formula on that component set, then a bisimulation relates x_1 and x_2 .

Proof. For a pointed model x , let $\text{Th}_\Sigma(x)$ be the set of well-typed Σ -formulas true at x . This set is called the model's *theory*. We relate two pointed models when these sets agree. Note that, such models agree on every state atom.

For the forward step condition, suppose $f_1 \Delta_{\mathcal{I}_1, Z_1} g_1$ and write $y_1 = x_1[g_1]$. If no successor of x_2 had the theory of y_1 , enumerate the finitely many successors of x_2 as $y_2^1, \dots, y_2^m$. For each j , choose a formula φ_j distinguishing y_1 from y_2^j , negating it if necessary so that $y_1 \models \varphi_j$ and $y_2^j \not\models \varphi_j$. Then $x_1 \models \Diamond \bigwedge_{j=1}^m \varphi_j$. Every successor of x_2 falsifies at least one part, so x_2 falsifies this formula. When $m = 0$, the formula is $\top$. This contradicts equality of theories. The reverse step condition is symmetric.

For an intervention θ , suppose the two models after the change had unequal theories. Some ψ would distinguish them, so $\langle \theta \rangle \psi$ would distinguish x_1 and x_2 . The changed models must be related. The argument applies to an arbitrary set of intervention labels.

For a cut label λ whose declared component set is the current component set, suppose first that λ is available in x_1 . Then $x_1 \models \top *_\lambda \top$. Equality of theories gives $x_2 \models \top *_\lambda \top$, whose satisfaction clause makes λ available in x_2 . The converse is symmetric, so the two models agree on the availability of λ . Its declared cut in both models is $\text{ct}(\lambda) = (C_1, I, C_2)$. Whenever the label is available, the left restriction of x_i satisfies a C_1 -local formula α exactly when $x_i \models \alpha *_\lambda \top$, for $i = 1, 2$. Since x_1 and x_2 satisfy the same formulas, their left restrictions have the same local theory and are related. Formulas of the form $\top *_\lambda \beta$ give the same result for the right restrictions. Equality of theories defines the required bisimulation. $\square$

12. Conclusion

The two questions studied in this paper require different subsystem boundaries. For a recorded execution, we keep the initial configuration and schedule fixed while replacing selected rules. The projection of this replay to an influence-closed region preserves its final configuration. Every minimal cause of an outcome stated within that region lies there, including when the region contains feedback.

To verify future behaviour, we use a stable cut. The configurations reachable by the full system are exactly the compatible combinations of the configurations reachable on its two sides. A property of the unchanged side needs to be proved only once. Each repair is checked on the side it changes. In an ordinary model, this equality holds from every initial configuration only when every interface rule leaves its component unchanged.

The retry system in Section 6.2 requires all six components because its feedback loop crosses the proposed boundary. Proposition 6.5 identifies the two rules that make callers retry aggressively as a minimal cause of the recorded overload. Proposition 6.7 proves that a gateway cap prevents overload on every continuation. In the RAG system of Section 9, retrieval has already committed the context through the stable shared component `PortCtx`. We assume the retrieval-side property at the phase boundary, prove that it is preserved during the analysed phase, and compare release rules only on the generation side. In the finite instance, Table 4 reports that the three checks examine 32 local configurations in place of 156 global configurations. Proposition 9.1 proves that both authorized repairs are safe and that the guarded rule releases more safe outputs than the rule that always holds them.

Algorithms 1 and 2 carry out the two analyses for finite systems.

For finite recorded executions, the replay construction has an equivalent acyclic SCM representation in which rule-variable interventions reproduce rule replacement. Independently, the logic is invariant under the corresponding bisimulation and, on the stated image-finite class, logical equivalence characterizes that bisimulation. Neither validation result is required for the two locality theorems.

The results have three main limits. Influence closure follows the inputs that have been declared, so it may miss an independence specific to the installed rule. The causal test holds the observed schedule and execution length fixed. Changes to scheduling, termination, or the branch that execution follows must be included in the model. The reachability result fixes the interface throughout the phase. A phase in which the subsystems continue to exchange information needs a model in which the interface can change.

Future work could model this exchange through communication channels that carry information in one direction and assumptions about what they send. Updates with stated probabilities and costs for changing rules would support numerical comparisons among repairs [55]. Interventions that change the influence graph would require new conditions on the inputs that each region must contain. They could be related to methods that infer causal dependencies from data [56, 57].

Code Availability

The `prototype` directory contains the Python program, the finite models, and tests. It requires Python 3.10 or later. The README explains how to reproduce the findings.

Acknowledgements

Chakraborty is supported by a studentship from UCL’s EPSRC-funded Centre for Doctoral Training in Cybersecurity (EP/S022503/1).

References

- [1] P. Chakraborty, T. Caulfield, D. Pym, Causality and decision-making: A logical framework for systems and security modelling (2025). [arXiv:2508.01758](#).
- [2] R. Krishna, M. S. Iqbal, M. A. Javidian, B. Ray, P. Jamshidi, CADET: Debugging and Fixing Misconfigurations using Counterfactual Reasoning (2021). URL <https://arxiv.org/abs/2010.06061>
- [3] A. Geiger, H. Lu, T. Icard, C. Potts, Causal abstractions of neural networks, in: Advances in Neural Information Processing Systems 34 (NeurIPS 2021), 2021, pp. 9574–9586.
- [4] J. Y. Halpern, Actual Causality, The MIT Press, 2016. doi:10.7551/mitpress/10809.001.0001.
- [5] J. Pearl, Causality: Models, Reasoning and Inference, 2nd Edition, CUP, 2009.
- [6] F. Eberhardt, R. Scheines, Interventions and causal inference, Philosophy of Science 74 (5) (2007) 981–995. doi:10.1086/525638.
- [7] R. Massidda, A. Geiger, T. Icard, D. Bacciu, Causal abstraction with soft interventions, in: Proceedings of the Second Conference on Causal Learning and Reasoning, Vol. 213 of Proceedings of Machine Learning Research, PMLR, 2023, pp. 68–87.
- [8] E. F. Rischel, S. Weichwald, Compositional abstraction error and a category of causal models, in: Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, Vol. 161 of Proceedings of Machine Learning Research, PMLR, 2021, pp. 1013–1023.
- [9] R. Lorenz, S. Tull, Causal models in string diagrams, CoRR abs/2304.07638 (2023).
- [10] P. Pruthi, D. Jensen, Compositional models for estimating causal effects, in: Proceedings of the Fourth Conference on Causal Learning and Reasoning, Vol. 275 of Proceedings of Machine Learning Research, PMLR, 2025, pp. 1365–1404.
- [11] C. A. R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677. doi:10.1145/359576.359585.
- [12] R. Milner, Communication and Concurrency, Prentice-Hall, Inc., USA, 1989.
- [13] R. Milner, Communicating and mobile systems: the π -calculus, Cambridge University Press, USA, 1999.

- [14] T. Caulfield, M.-C. Ilau, D. Pym, Engineering Ecosystem Models: Semantics and Pragmatics, in: *Simulation Tools and Techniques*, Springer International Publishing, Cham, 2022, pp. 236–258. doi:10.1007/978-3-030-97124-3_21.
- [15] M. Bujorianu, T. Caulfield, M.-C. Ilau, D. Pym, Interfaces in Ecosystems: Concepts, Form, and Implementation, in: *Simulation Tools and Techniques: 16th EAI International Conference, SIMUtools 2024, Bratislava, Slovakia, December 9–10, 2024, Proceedings*, Vol. 603 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, 2025, pp. 27–47. doi:10.1007/978-3-031-87345-4_3.
- [16] G. Anderson, D. Pym, A calculus and logic of bunched resources and processes, *Theor. Comput. Sci.* 614 (C) (2016) 63–96. doi:10.1016/j.tcs.2015.11.035.
- [17] M. Collinson, D. Pym, Algebra and logic for resource-based systems modelling, *Mathematical Structures in Computer Science* 19 (5) (2009) 959–1027. doi:10.1017/S0960129509990077.
- [18] D. Galmiche, T. Lang, D. Pym, Minimalistic System Modelling: Behaviours, Interfaces, and Local Reasoning, in: A. A. Juan, J.-L. Guisado-Lizar, M.-J. Morón-Fernández, E. Perez-Bernabeu (Eds.), *Simulation Tools and Techniques: 16th EAI International Conference, SIMUtools 2024, Bratislava, Slovakia, December 9–10, 2024, Proceedings*, Vol. 603 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, Cham, 2025, pp. 48–71. doi:10.1007/978-3-031-87345-4_4.
- [19] G. Winskel, Events, causality and symmetry, *The Computer Journal* 54 (1) (2009) 42–57. doi:10.1093/comjnl/bxp052.
- [20] I. Beer, S. Ben-David, H. Chockler, A. Orni, R. J. Treffler, Explaining counterexamples using causality, in: *Computer Aided Verification*, Vol. 5643 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 94–108. doi:10.1007/978-3-642-02658-4_11.
- [21] G. Gössler, D. Le Métayer, A general framework for blaming in component-based systems, *Science of Computer Programming* 113 (2015) 223–235. doi:10.1016/j.scico.2015.06.010.
- [22] G. Caltais, S. Leue, M. R. Mousavi, (de-)composing causality in labeled transition systems, in: *Proceedings First Workshop on Causal Reasoning for Embedded and Safety-Critical Systems Technologies*, Vol. 224 of *Electronic Proceedings in Theoretical Computer Science*, 2016, pp. 10–24. doi:10.4204/EP TCS.224.3.
- [23] G. Gössler, J.-B. Stefani, Causality analysis and fault ascription in component-based systems, *Theoretical Computer Science* 837 (2020) 158–180. doi:10.1016/j.tcs.2020.06.010.

- [24] C. Baier, et al., From verification to causality-based explications, in: *LIPIcs* 198: 48th Int. Colloq. Automata, Languages, and Programming (ICALP 2021), 2021, pp. 1:1–1:20. doi:10.4230/LIPIcs.ICALP.2021.1.
- [25] J. Parreaux, J. Piribauer, C. Baier, Counterfactual causality for reachability and safety based on distance functions, in: *Proceedings of the Fourteenth International Symposium on Games, Automata, Logics, and Formal Verification*, Vol. 390 of *Electronic Proceedings in Theoretical Computer Science*, 2023, pp. 132–149. doi:10.4204/EPTCS.390.9.
- [26] C. Dubslaff, K. Weis, C. Baier, S. Apel, Causality in configurable software systems, in: *Proceedings of the 44th International Conference on Software Engineering*, Association for Computing Machinery, 2022, pp. 325–337. doi:10.1145/3510003.3510200.
- [27] M. Tanhaei, C-ADL: A causal architecture description language for design-time root cause analysis and counterfactual reasoning in distributed systems, *Journal of Systems and Software* 239 (2026) 112902. doi:10.1016/j.jss.2026.112902.
- [28] N. Wiener, *Cybernetics*, 2nd Edition, M.I.T. Press, New York,, 1961.
- [29] W. R. Ashby, *An Introduction to Cybernetics*, J. Wiley, New York, 1956.
- [30] H. A. Simon, On the definition of the causal relation, *The Journal of Philosophy* 49 (16) (1952) 517–528.
- [31] H. A. Simon, Causal ordering and identifiability, in: W. C. Hood, T. C. Koopmans (Eds.), *Studies in Econometric Method*, no. 14 in *Cowles Commission Monograph*, John Wiley & Sons, New York, 1953, pp. 49–74.
- [32] H. A. Simon, The architecture of complexity, *Proceedings of the American Philosophical Society* 106 (6) (1962) 467–482.
- [33] Y. Iwasaki, H. A. Simon, Causality and model abstraction, *Artificial Intelligence* 67 (1) (1994) 143–194. doi:10.1016/0004-3702(94)90014-0.
- [34] M. Gladyshev, N. Alechina, M. Dastani, D. Doder, Dynamics of causal dependencies in multi-agent settings, in: *Engineering Multi-Agent Systems (EMAS 2023)*, Vol. 14378 of *Lecture Notes in Computer Science*, Springer, Cham, 2023, pp. 95–112. doi:10.1007/978-3-031-48539-8_7.
- [35] C. Ioannidis, D. Pym, J. Williams, I. Gheyas, Resilience in information stewardship, Tech. Rep. RN/14/16, University College London, Department of Computer Science, research Note (2014).

- [36] C. Ioannidis, D. Pym, J. Williams, Is public co-ordination of investment in information security desirable?, *Journal of Information Security* 7 (2016) 60–80. doi:10.4236/jis.2016.72005.
- [37] S. Bongers, P. Forré, J. Peters, J. M. Mooij, Foundations of structural causal models with cycles and latent variables, *The Annals of Statistics* 49 (5) (2021) 2885–2915. doi:10.1214/21-AOS2064.
- [38] R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time temporal logic, *J. ACM* 49 (5) (2002) 672–713. doi:10.1145/585265.585270.
- [39] L. Hammond, J. Fox, T. Everitt, R. Carey, A. Abate, M. Wooldridge, Reasoning about causality in games, *Artificial Intelligence* 320 (2023) 103919. doi:10.1016/j.artint.2023.103919.
- [40] M. Mishra, J. Fox, M. Wooldridge, Characterising interventions in causal games, in: *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*, Vol. 244 of *Proceedings of Machine Learning Research*, PMLR, 2024, pp. 2560–2572.
- [41] A. Datta, D. Garg, D. Kaynar, D. Sharma, A. Sinha, Program actions as actual causes: A building block for accountability, in: *2015 IEEE 28th Computer Security Foundations Symposium*, IEEE, 2015, pp. 261–275. doi:10.1109/CSF.2015.25.
- [42] J. Y. Halpern, J. Pearl, Causes and Explanations: A Structural-Model Approach. Part I: Causes, *Brit. J. Phil. Sci.* 56 (4) (2005) 843–887.
- [43] J. Woodward, What is a Mechanism? A Counterfactual Account, *Philosophy of Science* 69 (S3) (2002) 366–377. doi:10.1086/341859.
- [44] J. Woodward, *Making Things Happen: A Theory of Causal Explanation*, Oxford University Press, New York, 2003. doi:10.1093/0195155270.001.0001.
- [45] P. Machamer, L. Darden, C. F. Craver, Thinking About Mechanisms, *Philosophy of Science* 67 (1) (2000) 1–25. doi:10.1086/392759.
- [46] H. A. Simon, C. I. Barnard, *Administrative Behavior: A study of Decision-making Processes in Administrative Organization*, Macmillan Co., 1947.
- [47] M. Ulrich, Addressing cascading failures, Chapter 22 of *Site Reliability Engineering: How Google Runs Production Systems*, accessed 7 August 2026 (2016). URL <https://sre.google/sre-book/addressing-cascading-failures/>
- [48] Microsoft, Retry storm antipattern, *Azure Architecture Center*, accessed 7 August 2026 (2025). URL <https://learn.microsoft.com/en-us/azure/architecture/antipatterns/retry-storm/>

- [49] A. Ikram, et al., Root cause analysis of failures in microservices through causal discovery, in: *Adv. in Neural Inf. Proc. Sys.*, Vol. 35, 2022, pp. 31158–31170.
- [50] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, H. Wang, Retrieval-augmented generation for large language models: A survey (2024). [arXiv:2312.10997](#).
- [51] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, D. Kiela, Retrieval-augmented generation for knowledge-intensive NLP tasks, in: *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020, pp. 9459–9474.
- [52] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, C. Potts, DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines (2023). [arXiv:2310.03714](#).
- [53] G. Aucher, J. van Benthem, D. Grossi, Modal logics of sabotage revisited, *J. Log. Computat.* 28 (2) (2017) 269–303. [doi:10.1093/logcom/exx034](#).
- [54] P. Blackburn, M. de Rijke, Y. Venema, *Modal logic*, CUP, 2001.
- [55] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, A. Simaitis, Prism-games: A model checker for stochastic multi-player games, in: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vol. 7795 of *Lecture Notes in Computer Science*, Springer, 2013, pp. 185–191. [doi:10.1007/978-3-642-36742-7_13](#).
- [56] F. Eberhardt, Introduction to the Epistemology of Causation, *Philosophy Compass* 4 (6) (2009) 913–925. [doi:10.1111/j.1747-9991.2009.00243.x](#).
- [57] B. Schölkopf, F. Locatello, S. Bauer, N. R. Ke, N. Kalchbrenner, A. Goyal, Y. Bengio, Toward Causal Representation Learning, *Proceedings of the IEEE* 109 (5) (2021) 612–634.