Fundamenta Informaticae 173 (2020) 217–251
DOI 10.3233/FI-2020-1922
IOS Press

Causal Reasoning for Safety in Hennessy Milner Logic

Georgiana Caltais[*][†]
University of Konstanz, Germany
georgiana.caltais@uni-konstanz.de

Mohammad Reza Mousavi
University of Leicester, UK
mm789@leicester.ac.uk

Hargurbir Singh
University of Konstanz, Germany
hargurbir.singh@uni-konstanz.de

Abstract. Determining and computing root causes in system failures is a significant issue in science and engineering. In this paper, we introduce a notion of causality for explaining counterexamples in system analysis based on formal models. The counter-examples are produced by checking for hazardous situations expressed in the Hennessy-Milner Logic, in the context of Labelled Transition System models. We also introduce CauseJMu, a tool for automatically identifying such causal computations within a system model. CauseJMu relies on encoding causality in terms of an extension of Hennessy-Milner Logic to recursive formulae with data. The encodings enable deciding whether a certain computation is causal or not, using the mCRL2 model checker.

Keywords: Cauality, Counterfactual causal reasoning, Concurrency, Labelled Transition Systems, Hennessy Milner Logic, Safety, Model-checking, Process algebra, mCRL2

[*]The work of Georgiana Caltais and Hargurbir Singh was supported by the DFG project "CRENKAT", proj. no. 398056821.
[†]Address for correspondence: University of Konstanz, Germany

1. Introduction

Explaining phenomena, particularly failures, using counterfactual causal reasoning has been a challenging line of research in science and engineering [1, 2, 3]. An instance of this line of research in computer science has been about explaining counterexamples found in the process of system verification, and in particular model-checking, through counterfactual reasoning [4, 5, 6]. Apart from the challenge of establishing well-behaved, expressive, and intuitive notions of causality, devising automated tools helping engineers to understand how to relate system faults and failures is not a trivial task.

A notion of causality that is frequently used in relation to technical systems relies on counterfactual reasoning. Lewis [1] formulates the counterfactual argument, which defines when an event is considered a cause for some effect (or hazardous situation) when the following constraints are satisfied:

a) whenever the event that is presumed to be the cause occurs, the effect occurs as well (this corresponds to a sufficiency condition), and

b) when the presumed cause does not occur, the effect will not occur either (counterfactual argument, or a necessity condition).

Counterfactual reasoning involves scrutinising alternative worlds: one world satisfying the sufficiency condition, where both the cause and the effect occur, and another world satisfying the necessity condition, in which neither the cause nor the effect occur.

Nevertheless, the Lewis-style counterfactual argument is considered too simple for explaining causes within complex logical structures of multiple events. In their seminal work [2, 3], Halpern and Pearl define a notion of complex logical events based on boolean equation systems and propose a number of conditions (the so-called AC conditions) under which an event can be considered causal for an effect.

The Halpern and Pearl model has been related in various forms to models of computation. Leitner-Fischer and Leue [4, 5] adopt the Halpern and Pearl model of actual causation to the context of transition systems and trace models for concurrent system computations. In addition to the Halpern and Pearl model, the aforementioned approach to causality [4, 5] considers the temporal order of events as well as the non-occurrence of events as being causal. The causality checking technique has been applied to various case studies in the area of analysing critical systems for safety violations [5]. In this setting, an ordered sequence of events is computed as being the actual cause of a safety property violation.

Most relevant for our work are the results in [6]. Caltais, Leue, and Mousavi [6] define a notion of counterfactual causal reasoning in the context of Labelled Transition Systems (LTSs) and Hennessy Milner Logic (HML) [7]. In short, LTSs represent system models, whereas HML formulae specify the system hazards, or (undesired) effects. The notion of causality complies to the characteristics of causation proposed by Haplern and Pearl [2, 3] and is further adapted to the setting of concurrent systems in the spirit of the earlier definition by Leitner-Fischer and Leue [5]. Intuitively, an execution within an LTS is causal whenever it leads to a state where a certain effect, or hazard, is enabled. This definition of causality [6] exploits what is referred to as the “non-occurrence of events” [5], and identifies relevant system execution fragments that, whenever performed, change the occurrence of the effect from true to false. Then, similar to the earlier approaches [2, 3, 5], the new definition [6] emphasises that only relevant events included in the causal explanation may influence the occurrence of the effect and other events have no influence on the effect as long as the causal events are present. Finally, only minimal causal executions are considered as valid explanations with respect to a system hazard.

Considerable amount of work on fault analysis, fault localisation and software debugging techniques, such as delta debugging [8], nearest neighbor queries [9], counterexample explanation in model checking [10, 11] and why-because-analysis [12] are based on counterfactual arguments.

Compositional verification of causality, has been recently addressed in the literature of causality [13, 14, 15, 16, 17, 6]. We later show by means of a short, but illustrative example, that the notion of causality introduced in this paper is not compositional with respect to the CCS-like parallel composition of LTSs [18]. Nevertheless, reasoning about causality within an LTS model can be reduced to reasoning about causality at the level of its components, whenever the latter are interleaved, non-communicating LTSs.

Event structures [19, 20] have been exploited to model causal dependencies in concurrent models. Nevertheless, in our approach we are interested in minimal traces that lead to hazardous situations and which satisfy the causal definition introduced in this paper. We do not take into consideration the order of events along these traces and, hence, we do not distinguish between interleaving and true concurrency. We intend to investigate further whether using event structures may lead to more efficient methods for calculating causes.

Our contributions. The work in this paper is an extension of the results presented in FROM 2018:

- We refine the notion of causality initially introduced in [6] to more faithfully accommodate the AC conditions in [2, 3].
- We provide proofs of compositionally for this new definition, for the case of interleaved, non-communicating LTSs. We also explain why compositionally does not hold for the general case of communicating systems, by means of examples.
- We provide a revised encoding of causality in terms of modal μ-calculus with data [21], extending HML. This paves the way to the implementation of a model-checking procedure for computing causalities using the mCRL2 toolbox [22, 21].
- We introduce CauseJMu, a tool exploiting the aforementioned encoding, in order to automatically compute causal explanations. CauseJMu is an application based on the interplay between mCRL2 and Java.
- We discuss the results of running CauseJMu in order to explain hazardous situations and ways to avoid them, by means of a railway level-crossing example.

Paper structure. Section 2 is dedicated to preliminaries. We provide a brief overview of LTSs (as semantic models of systems) and HML (as the logic to specify hazardous situations). We also introduce there the level-crossing as our running example. In Section 3, we provide a brief account of the definition of causality proposed by Halpern [3], and introduce our adoption of this notion in the context of LTSs and hazards expressed in HML. In Section 4 we present our compositionality results. Section 5 introduces the encoding of causality in terms of modal μ-calculus formulae with data. Implementing causality in mCRL2 and automating the causality checking procedure within CauseJMu is discussed in Section 6. In Section 7 we draw some conclusions and present the directions of our future work.

2. Preliminaries

In this section, we provide a brief overview of Labelled Transition Systems and Hennessy Milner Logic, used throughout the paper in order to formally model systems and associated properties describing hazardous situations. We discuss two (equivalent) ways of modelling systems: Labelled Transition Systems as a semantic model for system modelling and process algebra as a syntactic model (that can be given an LTS semantics); the latter serves as the input language for the mCRL2 model-checker. A level-crossing example is provided for a better illustration of the aforementioned concepts.

2.1. Labelled Transition Systems (LTSs)

Let A be a possibly infinite set of labels, usually referred to as *alphabet*. Let $(-)^*$ be the Kleene star operator. We use $w, w_0, w_1, \ldots$ to range over words in A^*. We write $\varepsilon \in A^*$ for the empty word and wa for the word obtained by concatenating $w \in A^*$ and $a \in A$. We call a word w' to be a *sub-word* of a word w, if q' is obtained by deleting n letters ($n \geq 1$) at some not-necessarily-adjacent positions in w, written $w' \in sub(w)$ or simply $subword(w', w)$. The empty sequence ε is a sub-word of any non-empty word w. As expected, we call w a *supra-word* of w' whenever w' is a sub-word of w. Naturally, ε is not a supra-word of any word in A^*. We write $\mid w \mid$ for the *length* of a word w, inductively defined as expected, where $\mid \varepsilon \mid= 0$.

Consider two disjoint alphabets A and B, and assume $w \in (A \cup B)^*$. We write $w_A = w \downarrow A$, respectively, $w_B = w \downarrow B$ for the maximal subwords of w such that $w_A \in A^*$, respectively, $w_B \in B^*$. Furthermore, we write $w_A \mid\mid w_B$ for the set of all words $w \in (A \cup B)^*$ such that $w \downarrow A = w_A$, $w \downarrow B = w_B$ and $\mid w \mid=\mid w_A \mid + \mid w_B \mid$.

Definition 2.1. (Labelled Transition Systems (LTSs))
A *labelled transition system* (LTS) is a triple $(\mathbb{S}, s_0, A, \rightarrow)$, where $\mathbb{S}$ is a finite set of states, $s_0 \in \mathbb{S}$ is the initial state, A is the action alphabet and $\rightarrow \subseteq \mathbb{S} \times A \times \mathbb{S}$ is the transition relation.

We write $\twoheadrightarrow\ \subseteq \mathbb{S} \times A^* \times \mathbb{S}$, to denote the reachability relation, *i.e.,* the smallest relation satisfying: $\frac{}{p \xrightarrow{\varepsilon}\!\!\!\!\!\twoheadrightarrow p}$, and $\frac{p \xrightarrow{w}\!\!\!\!\!\twoheadrightarrow p' \quad p' \xrightarrow{a} p''}{p \xrightarrow{wa}\!\!\!\!\!\twoheadrightarrow p''}$.

Definition 2.2. (Lists)
Consider two natural numbers $a, b \in \mathbb{N}$ such that $a \leq b$; a *closed interval* $[a, b] \subseteq \mathbb{N}$ of natural numbers, is defined as $\{i \mid i \in \mathbb{N}, a \leq i \leq b\}$.

A *list* l of type T, denoted by $l : List[T]$, is a function of type $\mathbb{N} \rightarrow D$, such that either its domain is the empty set, in which case the list is *empty*, or its domain is a closed interval $[0, i]$, for some natural number $i \in \mathbb{N}$, in which case the list is called *finite*, or its domain is the set of all natural numbers, in which case the list is called *infinite*. The cardinality of the domain of a finite list is called its *size*. Two lists are *size compatible*, when they have the same size.

For any list of type $List[T]$, we write $[\,]$ to denote the empty list of that type. We write $\mathcal{D} = [w_0, \ldots, w_n]$ for a finite list of words $w_i \in A^*$, with $0 \leq i \leq n$. A notation of shape $\mathcal{D} = [w_0,\, w_1, \ldots]$ refers to an infinite list $\mathcal{D}$ of words $w_i \in A^*$, for $i \geq 0$. Moreover, we write $w : \mathcal{D}$ as an alternative to a list with w as the first element, and $\mathcal{D}$ as the list of "remaining" elements; for instance:

$$w_1 : [w_2, w_3] = [w_1, w_2, w_3].$$

We say that lists $\mathcal{D}_0, \ldots, \mathcal{D}_n$ are *size-compatible* if they are finite lists of the same length, or if they are all infinite lists.

By reusing the notation, we often write $[D]$ to denote type $List[D]$.

For instance, $[\,]$ and $[\,]$ are size-compatible, $[w_0, w_1, w_2]$ and $[w'_0, w'_1, w'_2]$ are size-compatible, $[w_0, w_1, \ldots]$ and $[w'_0, w'_1, \ldots]$ are size-compatible, whereas $[\,]$ and $[w]$ are not size-compatible.

Definition 2.3. (Computations)
A *computation* of size n is a sequence

$$\mathcal{D}, (l_0, \mathcal{D}_0), \ldots,\, (l_n, \mathcal{D}_n)$$

with $n \in \mathbb{N}$, $l_i \in A$ and $\mathcal{D}, \mathcal{D}_i \in [A^*]$, for $0 \leq i \leq n$. We call $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n \in [A^*]$ the *decorations* of the computation.

Consider a computation π such that:

$$\pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots,\, (l_n, \mathcal{D}_n);$$

whenever $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n$ are size-compatible, we write $traces(\mathcal{D}, (l_0, \mathcal{D}_0) \ldots (l_n, \mathcal{D}_n))$ or, in short, $traces(\pi)$, to denote the extensions of $l_0 \ldots l_n$ with words from $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n$ as follows:

$$traces([\,], (l_0, [\,]) \ldots (l_n, [\,])) = \{l_0 \ldots l_n\}$$
$$traces(w : \mathcal{D}, (l_0, w_0 : \mathcal{D}_0) \ldots (l_n, w_n : \mathcal{D}_n)) = \{w l_0 w_0 \ldots l_n w_n\} \cup traces(\mathcal{D}, (l_0, \mathcal{D}_0) \ldots (l_n, \mathcal{D}_n))$$

For instance,

$$traces([\varepsilon, \varepsilon, \varepsilon], (a, [w_{a0}, w_{a1}, w_{a2}]), (b, [\varepsilon, \varepsilon, \varepsilon]), (c, [\varepsilon, w_{c1}, \varepsilon])) = \{a w_{a0} bc,\, a w_{a1} bc w_{c1},\, a w_{a2} bc\}$$

for $a, b, c \in A$ and $w_{a0}, w_{a1}, w_{a2}, w_{c1} \in A^*$.

Consider an LTS $T = (\mathbb{S}, s_0, A, \rightarrow)$; we say that π is a *computation* of T whenever the following hold:

- $s_0 \xrightarrow{l_0} s_1 \ldots \xrightarrow{l_n} s_{n+1}$,
- $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n$ are size-compatible, and
- for all $w \in traces(\pi)$ there exists $s \in \mathbb{S}$ such that $s_0 \overset{w}{\twoheadrightarrow} s$.

We use $\pi, \mu, \ldots$ to range over computations.

The set of *sub-computations* of $\pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$, denoted by $sub(\pi)$ is the set of all computations $\pi' = \mathcal{D}', (l'_0, \mathcal{D}'_0), \ldots, (l'_m, \mathcal{D}'_m)$ such that $l'_0 \ldots l'_m \in sub(l_0 \ldots l_n)$. Note that all elements of $sub(\pi)$ should be computations themselves.

Intuitively, a computation provides us with a rich structure in which not only singleton actions in the computation interleaved with any choice of equally indexed sequences of actions in the lists provide a trace of the LTS. This is illustrated by the following example.

Example 2.4. Computation Consider the Car LTS in Figure 1; computations $\pi_0 = [\,]$, $(Ca, [\,])$, $(Go_{in}, [\,])$, $(Cc, [\,]), (Cl, [\,])$ and $\pi_1 = [\varepsilon, \varepsilon, \varepsilon, \ldots]$ $(C_a, [\varepsilon, Gc_{in}, \quad Gc_{in}\ Gc_{in}, \ldots])$ are both valid computations for the Car LTS.

However, neither $\pi_2 = []$, $(C_a, [\varepsilon, Gc_{in}])$, nor $\pi_3 = [\varepsilon, \varepsilon]$, $(C_a, [\varepsilon, Go_{in}])$, $(C_c, [\varepsilon, \varepsilon])$ are valid computations for the LTS. The former computation, i.e., π_2, is not valid because the decorations are not equally sized (the first decoration is the empty list, while the second one has two elements. The latter computation is not valid because the particular interleaving $C_a\ C_c$ is a trace of π_3 but is not admitted as a trace of the Car LTS.

$$\begin{tikzcd}
\text{Car LTS:} & c_0 \arrow[r, "Ca"] & c_1 \arrow[loop above, "Gc\_in"] \arrow[r, "Go\_in"] & c_2 \arrow[r, "Cc"] & c_3 \arrow[r, "Cl"] & c_4
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Train LTS:} & t_0 \arrow[r, "Ta\_out"] & t_1 \arrow[r, "Tc"] & t_2 \arrow[r, "Tl\_out"] & t_3
\end{tikzcd}$$

$$\begin{tikzcd}
\text{Gate LTS:} & g_0 \arrow[loop left, "Go\_out"] \arrow[r, bend right, "Ta\_in"'] & g_1 \arrow[loop right, "Gc\_out"] \arrow[l, bend right, "Tl\_in"']
\end{tikzcd}$$

Figure 1. The Car, Train and Gate as LTSs

In other words, for a computation to be valid, $traces(\pi)$ should consist of traces in the LTS given by the pairwise extension of $l_0 \ldots l_n$ with equally indexed elements of the decorations $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n$. This decorations will later be used for those events (sequences of actions) that can disable the effect and hence, their absence is provided as a part of the causal explanation. In such a causal explanation, given a computation π as defined above, the sequence of actions does lead to the effect, but the sequences $w \in traces(\pi)$ determine executions $s_0 \overset{w}{\twoheadrightarrow} s$ in T, such that the effect does not occur in s.

2.1.1. A basic process algebra

LTSs can be specified in a process algebraic syntax. Process algebras [23, 21] have been successfully exploited for the formal specification and verification of parallel and communicating reactive systems. Typically, process algebras specify a notion of *syntax*, which is accompanied by a corresponding (structural operational) *semantics* [24, 25]. As soon as a notion of behavioural equivalence of systems is established, reasoning on systems can be performed (a) in an equational fashion, via sound and complete axiomatisations that enable showing the equality of two terms built according to the aforementioned syntax, or (b) via equivalence checking of the state based systems (LTSs) generated according to the operational semantics, or (c) by model-checking LTSs with respect to properties expressed in a logical formalism [26, 22, 21].

The syntax of a simple process algebra is defined by the following grammar:

$$p ::= \delta \mid a \mid a.p \mid p + p \mid p \mid\mid p \mid X \mid X \doteq p \quad (a \in A, X \in \mathit{Var}) \tag{1}$$

Intuitively, A stands for the action alphabet of the process (e.g., described semantically by an LTS), $a.p$ stands for action prefixing: first an action a is performed and then process p will follow, $+$ is non-deterministic choice and $||$ is parallel composition. δ is the empty process that has no behaviour. Communication between processes is performed based on synchronisations between actions that play a sender, respectively, receiver role. In this paper we write a_in whenever the LTS is ready to receive/read action a, and a_out to indicate that the LTS is ready to send action a. Depending on the context, we write either a_ack or a whenever we acknowledge the successful communication based on a_in and a_out. Recursion is specified using recursive variables $X \in \mathit{Var}$ and recursive equations of the form $X \doteq p$, where variable X (or any other recursive variable) may appear in p.

The operational semantics of processes associates an LTS to each process; it can be specified via the following rules in the Plotkin style of structural operational semantics [24] and follows the same principles as the semantics of the calclus of communicating systems (CCS) [18]:

$$\begin{gathered}
\frac{\cdot}{a.p \xrightarrow{a} p} \qquad \frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} p'}{X \xrightarrow{a} p'} X \doteq p \\[1em]
\frac{p \xrightarrow{a} p'}{p \mid\mid q \xrightarrow{a} p' \mid\mid q} \quad \frac{q \xrightarrow{a} q'}{p \mid\mid q \xrightarrow{a} p \mid\mid q'} \quad \frac{p \xrightarrow{a\_in} p' \quad q \xrightarrow{a\_out} q'}{p \mid\mid q \xrightarrow{a\_ack)} p' \mid\mid q'}
\end{gathered} \tag{2}$$

The semantics of a process is the LTS that has the process term as its initial state, the set of actions A and the transition relation that is the smallest relation satisfying the above-given inference rules. When it is clear from the context instead of writing the outcome of synchronisation as a_{ack}, we simply write a. Consider two LTSs:

$$T = (\mathbb{S}, s_0, A, \rightarrow) \qquad T' = (\mathbb{S}', s_0', B, \rightarrow').$$

We abuse notation and write $T \mid\mid T'$ in lieu of $s_0 \mid\mid s_0'$, and $T + T'$ in lieu of $s_0 + s_0'$.

Two processes that run in parallel but do not communicate with each other (*i.e.*, never use the last rule in (2)) are *interleaved*. Without loss of generality, we assume that interleaved LTSs have disjoint action alphabets.

2.2. Hennessy Milner Logic (HML)

As previously mentioned, the behaviour of LTSs can be analysed by model-checking properties given in a temporal logic [26, 22, 21]. Of particular interest for this paper are properties formalising the occurrence of effects in terms of satisfiability of formulae in Hennessy Milner logic [7].

Definition 2.5. (Hennessy-Milner logic (HML))
The syntax of Hennessy-Milner logic (HML) [7] is given by the following grammar:

$$\phi, \psi ::= \top \mid \langle a \rangle \phi \mid \neg \phi \mid \phi \wedge \psi \qquad (a \in A).$$

We define the satisfaction relation $\models$ over LTSs states and HML formulae as follows.

Let $T = (\mathbb{S}, s_0, A, \rightarrow)$ be an LTS. Let ϕ, ϕ' range over HML formulae. It holds that:

$s \models \top$ for all $s \in \mathbb{S}$

$s \models \neg\phi$ whenever s does not satisfy ϕ; also written as $s \not\models \phi$

$s \models \phi \wedge \phi'$ if and only if $s \models \phi$ and $s \models \phi'$

$s \models \langle a \rangle \phi$ if and only if $s \xrightarrow{a} s'$ for some $s' \in \mathbb{S}'$ such that $s' \models \phi$

Disjunction and "box" ($[a]\phi$) are the duals to conjunction and "diamond" ($\langle a \rangle \phi$) and are defined as follows:

$$\phi \vee \phi' = \neg(\phi \wedge \phi') \qquad [a]\phi = \neg\langle a \rangle \neg\phi.$$

2.3. A level crossing example

Consider a level crossing example, where a gate is used to signal the approach/departure of a train, with the purpose of assisting the cars in the vicinity of the crossing to safely drive over the railway. The gate can send a message indicating the status of being closed (Gc_out) or open (Go_out), respectively, depending on whether it received a message from the train, signalling its approach to the crossing (Ta_in) or its departure from the crossing (Tl_in), respectively. The train displays a rather simple behaviour: it can send a message stating its approach to the crossing (Ta_out), afterwards it can actually enter the crossing (Tc) and then send a message signalling its departure from the crossing (Tl_out). The car can approach the crossing (Ca), wait as long as the gate is closed (Gc_in), eventually observe the gate being open (Go_in), enter the crossing afterwards (Cc) and the leave the crossing (Cl).

The LTSs describing the behaviours of the car, train and gate are illustrated in Figure 1. Note that the car can enter the crossing only after the gate is open, whereas the gate enters the state of being open only after a train signals its departure. Orthogonally, after receiving an approach message from a train, the gate enters the state of being closed, in which case the car can only observe the gate being closed without performing any further (driving related) actions.

The LTSs in Figure 1 are derived based on the structural operational semantics rules in (2), applied on the corresponding process terms (recursively) defined as follows:

$$\begin{array}{lrcl} \text{Car:} & C & \doteq & Ca.C1; \\ & C1 & \doteq & Gc\_in.C1 + Go\_in.Cc.Cl.\delta; \\ \\ \text{Train:} & T & \doteq & Ta\_out.Tc.Tl\_out.\delta; \\ \\ \text{Gate:} & G & \doteq & Go\_out.G + Ta\_in.G1; \\ & G1 & \doteq & Gc\_out.G1 + Tl\_in.G; \end{array} \tag{3}$$

The level crossing system is defined as the parallel composition of the processes in (3). The associated process term is $C \mid\mid T \mid\mid G$. The corresponding LTS is illustrated in Figure 2. The initial state is the upper left node in Figure 2 and it corresponds to $C \mid\mid T \mid\mid G$. The remaining states and transitions are subsequently derived according to the semantic rules in (2).

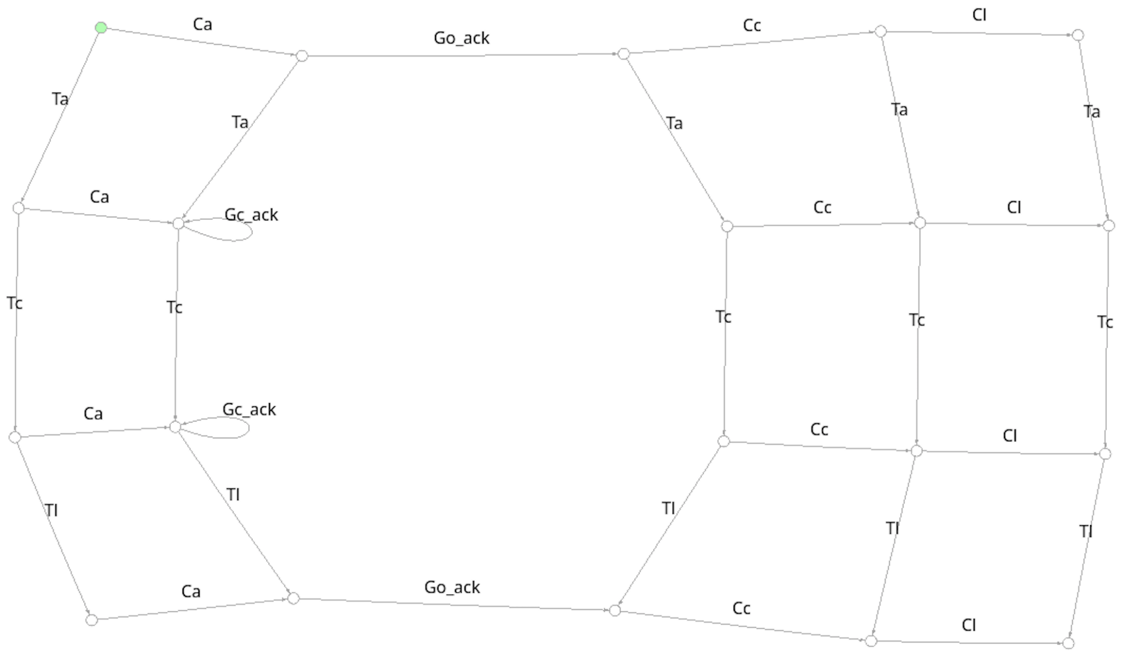

Figure 2. The Level Crossing LTS

The HML property formalising the collision between the train and the car inside the crossing is $\varphi = \langle Tc \rangle \top \wedge \langle Cc \rangle \top$; this property should not hold in any reachable state of the LTS. However, it is easy to see there is a design flaw in the railway system. By executing the trace $Ca\ Go\_ack\ Ta$ we can reach a state where the hazard, or the effect φ is satisfied. However, at a closer look we observe that by executing, for instance, the trace $Ca\ Go\_ack\ Cc\ Cl\ Ta$, the setting shifts from a hazardous to a safe one. In other words, if the car leaves before the train approaches the crossing, the hazard cannot happen.

The work in this paper aims at automatically identifying design flaws within system models and providing ways to avoid hazardous situations as above, by exploiting a notion of causality defined in the next section.

3. Causality for Hennessy-Milner Logic formulae

In this section we provide an overview of the seminal work on causality by Haplern and Pearl [2, 3] together with our adoption of this notion in the setting of LTSs models and effects, or hazards, specified as HML formulae.

3.1. The Halpern-Pearl (HP) defintion

Halpern and Pear [2, 3] define actual causes with respect to certain effects, in the context of models (possible worlds) described by means of variables and associated values. Intuitively, the causal influence between some variables is captured by structural equations that describe how the outcome is determined by relating values to variables.

We proceed with a brief overview of causality as introduced by Halpern [3]. A causal model M is a tuple $(\mathcal{S}, \mathcal{F})$ where $\mathcal{S}$ is a signature, which explicitly lists the so-called *endogenous* and *exogenous* variables and characterises their possible values, and $\mathcal{F}$ defines a set of modifiable structural equations, relating the values of the variables. Endogenous variables are those whose values are described by the structural equations [2]. According to Halpern and Pearl [2], exogenous variables are taken as given. The structural equations model the effect of exogenous variables on the endogenous ones, and the effect of the endogenous variables on each other.

A signature $\mathcal{S}$ is a tuple $(\mathcal{U}, \mathcal{V}, \mathcal{R})$, where $\mathcal{U}$ is a set of exogenous variables, $\mathcal{V}$ is a set of endogenous variables, and $\mathcal{R}$ associates with every variable $Y \in \mathcal{U} \cup \mathcal{V}$ a non empty set $\mathcal{R}(Y)$ of possible values for Y (*i.e.*, the set of values over which Y ranges).

$\mathcal{F}$ associates with each variable $X \in \mathcal{V}$ a function denoted

$$F_X : (\times_{U \in \mathcal{U}} \mathcal{R}(U)) \times (\times_{Y \in \mathcal{V} \setminus \{X\}} \mathcal{R}(Y)) \to \mathcal{R}(X)$$

which determines the value of X given the values of all the other variables in $\mathcal{U} \cup \mathcal{V}$. For an example inspired by Halpern and Pearl [2, 3], consider $F_X(u, y, z) = u + y$, usually written as $X = U + Y$, for exogenous variable U and endogenous variables X, Y and Z . Thus, if $Y = 3$ and $U = 2$, then $X = 5$, regardless of how Z is set.

Given a signature $\mathcal{S} = (\mathcal{U}, \mathcal{V}, \mathcal{R})$, a *primitive event* is a formula of the form $X = x$, for $X \in \mathcal{V}$ and $x \in \mathcal{R}(X)$. A *causal formula* over $\mathcal{S}$ is of the form

$$[Y_1 \leftarrow y_1, \ldots, Y_k \leftarrow y_k]\varphi$$

where φ is a Boolean combination of primitive events, Y_i are distinct endogenous variables and $y_i \in \mathcal{R}(Y_i)$ for all $i \in \{1, \ldots, k\}$. Such a formula is sometimes abbreviated as $[\vec{Y} \leftarrow \vec{y}]\varphi$ and, intuitively, it states that φ holds if Y_i is set to y_i for all $i \in \{1, \ldots, k\}$.

We call a *context* a setting $\vec{u}$ for the exogenous variables in $\mathcal{U}$, and write $(M, \vec{u})$ to denote the model M for which the variables $\mathcal{U}$ are set to $\vec{u}$. External interventions are modelled by setting the value of some variable X to x in a causal model $M = (\mathcal{S}, \mathcal{F})$. This results in a new causal model, denoted $M_{X \leftarrow x}$, which is identical to M, except that the equation for X in $\mathcal{F}$ is replaced by $X = x$. Several external interventions $X_1 \leftarrow x_1, \ldots, X_n \leftarrow x_n$ are abbreviated as $\vec{X} \leftarrow \vec{x}$, and the new associated model is denoted by $M_{\vec{X} \leftarrow \vec{x}}$.

We write $(M, \vec{u}) \vDash \varphi$ whenever the causal formula φ is true in a causal model M given the context $\vec{u}$. The satisfiability relation $\vDash$ is defined inductively. $(M, \vec{u}) \vDash X = x$ if the variable X has value x in the unique (since Halpern and Pearl [2, 3] deal with acyclic models) solution to the equations in M in context $\vec{u}$. Conjunctions and negations are defined in the standard way. Moreover, $(M, \vec{u}) \vDash [\vec{Y} \leftarrow \vec{y}]\varphi$ if $(M_{\vec{Y} \leftarrow \vec{y}}, \vec{u}) \vDash \varphi$.

The types of events allowed as actual causes are of the form $X_1 = x_1 \wedge \ldots \wedge X_k = x_k$, also abbreviated as $\vec{X} = \vec{x}$. Formally, an actual cause is defined in terms of three actual cause conditions (AC) as follows:

Definition 3.1. (Actual Cause / The HP Definition [3])
$\vec{X} = \vec{x}$ is an *actual cause* of φ in $(M, \vec{u})$ if the following conditions hold:

AC1: $(M, \vec{u}) \models (\vec{X} = \vec{x})$ and $(M, \vec{u}) \models \varphi$.
Intuitively, we say that both the actual cause $\vec{X} = \vec{x}$ and the effect specified by φ can be observed in the actual world of $(M, \vec{u})$.

AC2: There exists a partition of $\mathcal{V}$ into two disjoint subsets $\vec{Z}$ and $\vec{W}$, with $\vec{Z} \cap \vec{W} = \emptyset$ and $\vec{X} \subseteq \vec{Z}$ such that

1. There is a setting $\vec{x'}$ and $\vec{w}$ of the variables in $\vec{X}$ and $\vec{W}$ for which the following holds: $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x'}, \vec{W} \leftarrow \vec{w}]\neg\varphi$.
   Intuitively, this corresponds to a necessity condition indicating that whenever the actual cause is not observed in the world of $(M, \vec{u})$, the effect disappears as well (*i.e.*, $\neg\varphi$ holds).
2. If there is $\vec{z}$ such that $(M, \vec{u}) \models \vec{Z} = \vec{z}$, then, for all $\vec{W'} \subseteq \vec{W}$ and $\vec{Z'} \subseteq \vec{Z}$ the following holds:
   $(M, \vec{u}) \models [\vec{X} \leftarrow \vec{x}, \vec{W'} \leftarrow \vec{w}, \vec{Z'} \leftarrow \vec{z}]\varphi$.
   Intuitively, this corresponds to a sufficiency condition indicating that whenever the actual cause is observed, the effect holds as well.

AC3: $\vec{X}$ is minimal, in the sense that no subset of $\vec{X}$ satisfies conditions AC1 and AC2.
Intuitively, minimality ensures that only those elements that are essential with respect to the effect φ are part of the cause.

3.2. Adoption of HP to LTSs and HML

We next introduce an adoption of actual causes into the setting of system models encoded as LTSs, and effects specified as properties φ in HML. This paves the way to counterfactual causal reasoning in concurrency and serves as a basis for our compositionality results in the subsequent sections.

Our approach is based on an interpretation of the HP Definition similar to the work of Leitner-Fischer and Leue [5, 27], where causes are intended to encode (minimal) explanations of counterexamples witnessing the violation of (safety) properties within state-based systems.

For a parallel with the railway crossing example in Section 2.3, we are interested in defining and computing actual causes of an effect φ encoded as an HML property stating that both the train and the car are in the crossing at the same time. Our causal model corresponds to the railway crossing LTS model, based on which we can analyse all possible worlds / executions of the train, car and gate components.

Exogenous variables are not controlled by the system and hence, cannot be included in the counterfactual reasoning and the causal explanation with respect to the effect that we analyse. Intuitively, for the level crossing example, an exogenous variable could be the intention of the train driver to leave the crossing; since this is not an action we can control, only its effect, *i.e.*, the train's departure is something that we will reason about. Orthogonally, endogenous variables represent all events that can have a potentially causal effect. In our context, these could be: car being in the crossing, train being in the crossing or gate stuck open.

A special kind of endogenous variables are the so-called *contingencies* [2, 3]. A set of contingent variables, typically denoted by $W \subseteq \mathcal{V}$, in Definition 3.1 enable expressing so-called "contingent dependencies". Contingencies specify the valuation of (contingent) variables not included in the causal explanation such that it becomes evident that the change of valuations in the causal explanation alone can control the occurrence and non-occurrence of hazard. For an intuition, consider two cars C_1 and C_2 that can crash with a train. We say that a collision depends on C_1 entering the crossing under the contingency that C_2 did not enter the crossing. Fixing the position of C_2 so that it is outside the crossing, makes it evident that it was C_1 alone which can influence the existence or removal of the hazard. Contingencies are given by those events that are not specified explicitly by the cause and its decoration.

We encode causalities via computations $\pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$ underlined by "causal traces" $l_0 \ldots l_n$ and lists of decorations $\mathcal{D}, \mathcal{D}_i$, with $i \in \{0, \ldots, n\}$. Causal traces are the counterpart of actual causes $\vec{X} = \vec{x}$ as in Definition 3.1, and encode the shortest sequences of events which, if executed, lead to the hazard, or effect φ. As we will see, causal traces comply to conditions similar to AC1–AC3 encoding necessity, sufficiency and minimality.

Enriching a causal trace with execution steps captured within decorations $\mathcal{D}, \mathcal{D}_i$ determines the elimination of the hazard. This property is a reminiscent of the work of Leitner-Fischer and Leue [5], and is not part of the original HP Definition. THere [5], events that can transform a hazardous situation into a safe one are called "causal by their non-occurrence". In the level crossing example, the car leaving the crossing before the train enters it, could be considered causal by its non-occurrence.

Our adoption of the HP Definition is as follows:

Definition 3.2. (Causality for LTSs and HML)
Consider a transition system $T = (\mathbb{S}, s_0, A, \rightarrow)$. *Actual causes* for an HML property φ in T, denoted by $\mathit{Causes}(\varphi, T)$, is the set of all computations $\pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$ such that:

AAC1: $\exists s \in \mathbb{S} \,:\, s_0 \xrightarrow{l_0 \ldots l_n}\!\!\!\!\rightarrow s \wedge s \models \varphi$.

AAC2.1: $\forall \chi' \in A^*, s \in \mathbb{S} \,:\, s_0 \xrightarrow{\chi'}\!\!\!\!\rightarrow s \wedge s \models \neg\varphi \Rightarrow l_0 \ldots l_n \notin \mathit{sub}(\chi') \vee \chi' \in \mathit{traces}(\pi)$.

AAC2.2: $\forall \chi' = \chi l_0 \chi_0 \ldots l_n \chi_n \in \{l_0 \ldots l_n\} \cup (A^* \setminus \mathit{traces}(\pi)), s \in \mathbb{S} \,:\, s_0 \xrightarrow{\chi'}\!\!\!\!\rightarrow s \Rightarrow s \models \varphi$.

AAC2.3: $\forall \chi' \in \mathit{traces}(\pi), s \in \mathbb{S} \,:\, s_0 \xrightarrow{\chi'}\!\!\!\!\rightarrow s \Rightarrow s \models \neg\varphi$

AAC3: $\forall \pi' \in \mathit{sub}(\pi) : \pi'$ does not satisfy **AAC1 –AAC2.3**

where $l_i \in A$ and $\mathcal{D}, \mathcal{D}_i \in [A^*]$, for $n \in \mathbb{N}$ and $0 \leq i \leq n$.
We call $l_0 \ldots l_n$ a *causal trace*.

Intuitively, **AAC1** identifies a scenario where both the causal trace $l_0 \dots l_n$ and the effect φ can be observed within the model T. **AAC2.1** says that it is necessary to observe the causal trace along arbitrary executions χ' in order to guarantee the effect φ. More precisely, whenever the effect cannot be observed, the causal trace $l_0 \dots l_n$ is not part of χ' or is interleaved with events that are causal by their non-occurrence. **AAC2.2** states that it is sufficient to observe the causal trace along arbitrary executions χ' in order to enable the hazard φ, unless $l_0 \dots l_n$ is interleaved with events causal by their non-occurrence in χ'. **AAC2.3** requires causal traces interleaved with sequences causal by their non-occurrence to remove the hazard. **AAC3** is the minimality condition that requires π to be the shortest computation satisfying conditions **AAC1** –**AAC2.3**.

Additionally, observe that our definition does not make use of contingent dependencies W as in Definition 3.1. Similar to the earlier work [5, 27], our approach is based on a complete exploration of the LTS model and enables the explicit identification of all potential causes. We guarantee the termination of the LTS exploration, as we traverse loop-like executions only once. This approach is in accordance with the minimality of causal traces. In the same spirit, we consider loop-like decorations causal by their non-occurrence only once. Executing such loop decorations once or more than once leads to the same state and, hence, to the same non-hazardous scenario.

$$\begin{tikzcd}
s_0 \arrow[rr, "d"] \arrow[d, "a"] & & s_7 \arrow[d, "e"] \\
s_1 \arrow[r, "b"] \arrow[d, "h"] & s_3 \arrow[d, "h"] & s_8 \arrow[d, "h"] \\
s_5 & s_6 & s_9
\end{tikzcd}
\qquad
\begin{tikzcd}
s_0 \arrow[r, "d"] \arrow[d, "a"] & s_7 \arrow[d, "e"] \\
s_1 & s_8
\end{tikzcd}$$

Figure 3. An LTS and Its Causal Projection w.r.t. $\langle h \rangle \top$

Assume the effect, or hazard stating that action h can be triggered as a first step; *i.e.*, $\varphi = \langle h \rangle \top$. Consider the LTS on the left hand side of Figure 3. Observe that ab can be regarded as a potential causal trace with respect to φ as it satisfies **AAC1** : $s_0 \overset{ab}{\twoheadrightarrow} s_3$ and $s_3 \models \varphi$. Additionally, the computation $\pi = (a, [\varepsilon]), (b, [h])$ looks like a good candidate of actual causality, as it also satisfies **AAC2.3** : $s_0 \overset{abh}{\twoheadrightarrow} s_6$ and $s_6 \not\models \varphi$. Moreover, π satisfies **AAC2.2** as well: the only trace that contains the events ab and is not in $traces(\pi)$ is ab itself, $s_0 \overset{ab}{\twoheadrightarrow} s_3$ and $s_3 \models \varphi$. **AAC2.1** is also satisfied: all traces χ' that lead to non-hazardous situations determine the executions: $s_0 \overset{\chi'=\varepsilon}{\twoheadrightarrow} s_0$, $s_0 \overset{\chi=ah}{\twoheadrightarrow} s_5$, $s_0 \overset{\chi=abh}{\twoheadrightarrow} s_6$, $s_0 \overset{\chi=d}{\twoheadrightarrow} s_7$, $s_0 \overset{\chi=deh}{\twoheadrightarrow} s_9$. Observe that traces $\chi' \in \{\varepsilon, ah, d, deh\}$ are not supra-words of ab, whereas $\chi' = abh$ is in $traces(\pi)$. Last, but not least, note that π does not satisfy the minimality condition. It can be easily verified that $\pi' = (a, [h, bh])$ satisfies **AAC1** –**AAC2.3** and $\pi' \in sub(\pi)$. Hence, π' is a cause. Similarly, we can show that $\pi'' = (d, [\varepsilon]), (e, [h])$ is another actual cause for φ. Note that π'' is entailed by de which a longer causal trace than a that is the causal trace of π'. Nevertheless, de and a explain two very different witnesses with respect to φ.

4. Compositionality results

In this section, we show that the proposed notion of causality in this paper is not compositional with respect to the CCS-like operators in (2). Subsequently, we show that compositionality can be recovered if one confines the specification to a parallel composition of interleaved, non-communicating components.

We proceed by first introducing the notion of *causal projection* that enables expressing the compositionality results by capturing only the sub-LTSs underlying the causal traces with respect to an HML formula.

Definition 4.1. (Causal projection)
A *causal projection* of $T = (\mathbb{S}, s_0, A, \rightarrow)$ with respect to an HML property φ, is $T' = (\mathbb{S}', s_0, A, \rightarrow')$ such that $\rightarrow' = \bigcup_{\substack{\pi \in \mathit{Causes}(\varphi, T) \\ \pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n) \\ ex = s_0 \xrightarrow{l_0} s_1 \ldots s_{n-1} \xrightarrow{l_n} s_n \in \twoheadrightarrow}} \{(s_i, l_i, s_{i+1}) \in ex\}$ and $\mathbb{S}' = \bigcup_{(s_i, l_i, s_{i+1}) \in \rightarrow'} \{s_i, s_{i+1}\}$.

We write $T \downarrow \varphi$ for the causal projection of T with respect to φ. Intuitively, a causal projection is an LTS whose traces are exactly all causal traces (formally defined as the trace $l_0 \ldots l_n$ in Definition 3.2). For instance, the causal projection of the LTS in Figure 3, with respect to $\langle h \rangle \top$, is illustrated on the right hand side of the same Figure 3. Note that ab is not a trace of causal projection, because ab does not provide a minimal cause.

4.1. (De)-composing causality in communicating LTSs

In this section we provide a small example of two LTSs illustrating that, unfortunately, causality is in general not compositional with respect to the CCS-like operators in (2). More precisely, we show that the proposed notion of causality is not compositional for communicating LTSs.

$\varphi_1 = \langle h \rangle \top, \quad \varphi_2 = \langle h' \rangle \top$

$T_1 :\quad s_0 \xrightarrow{a} s_1 \xrightarrow{h} s_2 \xrightarrow{c\_in} s_3$

$T_2 :\quad p_0 \xrightarrow{d} p_1 \xrightarrow{c\_out} p_2 \xrightarrow{h'} p_3$

$T_1 \mid\mid T_2 :$

$$\begin{tikzcd}
s_0 \mid\mid p_0 \arrow[r, "a"] \arrow[d, "d"] & s_1 \mid\mid p_0 \arrow[r, "h"] \arrow[d, "d"] & s_2 \mid\mid p_0 \arrow[d, "d"] & & \\
s_0 \mid\mid p_1 \arrow[r, "a"] & s_1 \mid\mid p_1 \arrow[r, "h"] & s_2 \mid\mid p_1 \arrow[r, "c\_ack"] & s_3 \mid\mid p_2 \arrow[r, "h'"] & s_3 \mid\mid p_3
\end{tikzcd}$$

$T_1 \downarrow \varphi_1 :\quad s_0 \xrightarrow{a} s_1$

$T_2 \downarrow \varphi_2 :\quad p_0 \xrightarrow{d} p_1 \xrightarrow{c\_out} p_2$

$(T_1 \mid\mid T_2) \downarrow (\varphi_1 \vee \varphi_2) :\quad s_0 \mid\mid p_0 \xrightarrow{a} s_1 \mid\mid p_0$

$(T_1 \mid\mid T_2) \downarrow (\varphi_1 \wedge \varphi_2) :\quad$ the empty LTS

Figure 4. Impossibility Results

Consider the LTSs in Figure 4, and their parallel composition. Assume the effects $\varphi_1 = \langle h\rangle\top$ and $\varphi_2 = \langle h'\rangle\top$. It is easy to see based on the causal projections with respect to φ_1 in T_1, φ_2 in T_2, $\varphi_1 \vee \varphi_2$ in $T_1 \mid\mid T_2$ and $\varphi_1 \wedge \varphi_2$ in $T_1 \mid\mid T_2$, that reasoning on causality at the level of component LTSs does not determine causality within the composed LTS, and the other way around. In the context of the LTSs in Figure 4, this is a consequence of the fact that the effect h' can be seen after acknowledging on c_ack, whereas acknowledging on c_ack is only possible after the effect h.

We conclude that in general causality at the level of composed LTSs cannot be equivalently expressed as an interleaving, disjunction or conjunction of causalities within the component LTSs.

4.2. (De)-composing causality in interleaved LTSs

In this section, we show that our notion of causality is compositional in the context of interleaved, non-communicating LTSs. Consider two LTSs

$$T = (\mathbb{S}, s_0, A, \rightarrow) \qquad T' = (\mathbb{S}', s'_0, B, \rightarrow')$$

and two HML formulae φ_1 and, respectively, φ_2 built over A and, respectively, B. We show that computing causalities witnessing effects of shape $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ in the interleaving $T \mid\mid T'$ can be reduced to computing causalities w.r.t. φ_1 in T, and φ_2 in T'.

Let ϕ be an HML property over the alphabet of an LTS $T = (\mathbb{S}, s_0, A, \rightarrow)$. We call ϕ an *immediate effect* whenever $s_0 \models \phi$. Note that we are not interested in reasoning about immediate effects that trivially reflect the hazard in the first state of the system. We are rather interested in understanding how individual components play a role in propagating the hazard within a system consisting of several interleaved LTSs.

4.2.1. (De)-composing disjunction

Theorem 4.5 formalises the compositionality result for disjunction of hazards. Intuitively, Theorem 4.5 establishes an isomorphism ($\simeq$) between the causal projection of the interleaving $T \mid\mid T'$ with respect to $\varphi_1 \vee \varphi_2$, and the union of the causal projections $T \downarrow \varphi_1$ and $T' \downarrow \varphi_2$, at component level. The proof of Theorem 4.5 relies on the intermediate results captured in Lemma 4.4. The latter holds according to Lemma 4.2 and Lemma 4.3.

Lemma 4.2. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s'_0, B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. If $\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$ is a computation satisfying **AAC1 –AAC2.3** w.r.t. $\phi \vee \psi$ in the interleaving $T \mid\mid T'$, then there exists
$\pi = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ a computation satisfying **AAC1 –AAC2.3** w.r.t. ϕ in T or
$\pi' = \overline{\mathcal{D}}', (l'_p, \overline{\mathcal{D}}'_p), \ldots, (l'_q, \overline{\mathcal{D}}'_q)$ a computation satisfying **AAC1 –AAC2.3** w.r.t. ψ in T'
where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l'_p \ldots l'_q$.

Proof:
We first construct the decorations $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}'_j$ above, for $i \in \{k, \ldots, m\}$ and $j \in \{p, \ldots, q\}$. The construction is as follows:

- initialise $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}'_j$ with [ ] for $i \in \{k, \ldots, m\}$ and $j \in \{p, \ldots, q\}$
- for all $\tilde{\chi} = \chi l_0 \chi_0 \ldots l_n \chi_n \in traces(\mu) \setminus \{l_0 \ldots l_n\}$ do:
  1. let $\overline{\overline{\chi}} = \overline{\chi} l_k \overline{\chi}_k \ldots l_m \overline{\chi}_m = \tilde{\chi} \downarrow A$
  2. let $\overline{\overline{\chi'}} = \overline{\chi}' l'_p \overline{\chi}'_p \ldots l'_q \overline{\chi}'_q = \tilde{\chi} \downarrow B$
  3. insert $\overline{\chi}, \overline{\chi}_i$ into $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i$ accordingly, for all $i \in \{k, \ldots, m\}$
  4. insert $\overline{\chi}', \overline{\chi}'_j$ into $\overline{\mathcal{D}}', \overline{\mathcal{D}}'_j$ accordingly, for all $j \in \{p, \ldots, q\}$.
- for all $\tilde{\chi} = \chi l_0 \chi_0 \ldots l_n \chi_n \in (A \cup B)^* \setminus traces(\mu)$ do:
  5. construct $\overline{\overline{\chi}}$ as in 1. above, and $\overline{\overline{\chi'}}$ as in 2. above
  6. if $\forall s : s_0 \overset{\overline{\overline{\chi}}}{\twoheadrightarrow} s \Rightarrow s \models \neg\phi$ then proceed as in 3. above
  7. if $\forall s' : s'_0 \overset{\overline{\overline{\chi'}}}{\twoheadrightarrow}{}' s' \Rightarrow s' \models \neg\psi$ then proceed as in 4. above.

Then, we proceed by reductio ad absurdum and show that π or, respectively, π' have to satisfy **AAC1** –**AAC2.3** with respect to ϕ in T or, respectively, ψ in T'. In order to show this, we exploit the construction of $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}'_j$, Definition 3.2 and the hypothesis as follows.

- Assume **AAC1** is not satisfied by π and π'. Equivalently, the following hold:

$$\forall s \in \mathbb{S} : s_0 \xrightarrow{l_k \ldots l_m}\!\!\!\!\!\rightarrow s \Rightarrow s \models \neg\phi$$
$$\forall s' \in \mathbb{S}' : s'_0 \xrightarrow{l'_p \ldots l'_q}\!\!\!\!\!\rightarrow{}' s' \Rightarrow s' \models \neg\psi$$

  Hence,

$$\forall s \in \mathbb{S}, s' \in \mathbb{S}' : s_0 \mid\mid s'_0 \xrightarrow{l_0 \ldots l_n}\!\!\!\!\!\rightarrow s \mid\mid s' \Rightarrow s \mid\mid s' \models \neg(\phi \vee \psi)$$

  also holds, which is in contradiction with the hypothesis that μ satisfies **AAC1** . Therefore, either π or π' have to satisfy **AAC1** .

- Assume **AAC2.1** is not satisfied by π and π'. Consequently, the following hold:

$$\exists \overline{\chi} \in A^*, s \in \mathbb{S} : s_0 \overset{\overline{\chi}}{\twoheadrightarrow} s \wedge s \models \neg\phi \wedge sub(l_k \ldots l_m, \overline{\chi}) \wedge \overline{\chi} \notin traces(\pi)$$
$$\exists \overline{\chi}' \in B^*, s' \in \mathbb{S}' : s'_0 \overset{\overline{\chi}'}{\twoheadrightarrow}{}' s' \wedge s' \models \neg\psi \wedge sub(l'_p \ldots l'_q, \overline{\chi}') \wedge \overline{\chi}' \notin traces(\pi')$$

  Hence, the following also holds

$$\exists \tilde{\chi} \in \overline{\chi} \mid\mid \overline{\chi}', s \in \mathbb{S}, s' \in \mathbb{S}' : s_0 \mid\mid s'_0 \overset{\tilde{\chi}}{\twoheadrightarrow} s \mid\mid s' \wedge sub(l_0 \ldots l_n, \tilde{\chi}) \wedge \tilde{\chi} \notin traces(\mu)$$

  given the construction of $traces(\pi)$ and $traces(\pi')$ in steps $1.-7.$ above. This is in contradiction with the hypothesis that μ satisfies **AAC2.1** . Therefore, either π or π' have to satisfy **AAC2.1** .

- Assume **AAC2.2** is not satisfied by π and π'. Consequently, the following hold:

$$\exists\overline{\overline{\chi}} = \overline{\chi} l_k \overline{\chi}_k \dots l_m \overline{\chi}_m \in \{l_k \dots l_m\} \cup (A^* \setminus \mathit{traces}(\pi)), s \in \mathbb{S} : s_0 \overset{\overline{\overline{\chi}}}{\twoheadrightarrow} s \wedge s \vDash \neg\phi$$
$$\exists\overline{\overline{\chi}}' = \overline{\chi}' l'_p \overline{\chi}'_p \dots l_q \overline{\chi}'_q \in \{l'_p \dots l'_q\} \cup (B^* \setminus \mathit{traces}(\pi')), s' \in \mathbb{S}' : s'_0 \overset{\overline{\overline{\chi}}'}{\twoheadrightarrow}{}' s' \wedge s' \vDash \neg\psi$$

Hence, by the construction of $\mathit{traces}(\pi)$ and $\mathit{traces}(\pi')$ in steps $1. - 7.$ above, the following also holds:

$$\exists\tilde{\chi} \in \{l_0, \dots, l_n\} \cup ((A \cup B)^* \setminus \mathit{traces}(\mu)), s \in \mathbb{S}, s' \in \mathbb{S}' :$$
$$s_0 \mid\mid s'_0 \overset{\tilde{\chi}}{\twoheadrightarrow} s \mid\mid s' \wedge s \mid\mid s' \vDash \neg(\phi \vee \psi)$$

with the additional observation that $\tilde{\chi} \in \overline{\overline{\chi}} \mid\mid \overline{\overline{\chi}}'$. This is in contradiction with the hypothesis that μ satisfies **AAC2.2** . Therefore, either π or π' have to satisfy **AAC2.2** .

- Additionally, note that both π and π' satisfy **AAC2.3** by the construction in steps $1. - 7.$ above. Therefore, based on the observations so far, π and π' can satisfy (✓) or falsify ($\times$) **AAC1** –**AAC2.3** as illustrated in the table below.

	AAC1			**AAC2.1**			**AAC2.2**			**AAC2.3**		
	Cases			*Cases*			*Cases*			*Cases*		
	1	2	3	4	5	6	7	8	9	10	11	12
π	$\times$	✓	✓	$\times$	✓	✓	$\times$	✓	✓	✓	✓	✓
π'	✓	$\times$	✓	✓	$\times$	✓	✓	$\times$	✓	✓	✓	✓

We further proceed by analysis of tuples

$$(-, -, -, -) \in \{1, 2, 3\} \times \{4, 5, 6\} \times \{7, 8, 9\} \times \{10, 11, 12\}.$$

It is straightforward to see that, for instance, tuples of shape $(3, 6, 9, -)$ stand for both π and π' satisfying **AAC1** –**AAC2.3** . Similarly, $(1, 4, 7, -)$ and its symmetric $(2, 5, 8, -)$, respectively, stand for π' and π, respectively, satisfying **AAC1** –**AAC2.3** .

Furthermore, note that tuples of shape $(1, -, 8, -)$ and, symmetrically $(2, -, 7, -)$, lead to a contradiction. Based on $(1, -, 8, -)$, for example, we reach the following contradiction:

$$\forall s \in \mathbb{S} : s_0 \xrightarrow{l_k \dots l_m}\!\!\!\!\rightarrow s \Rightarrow s \vDash \neg\phi$$
$$s_0 \xrightarrow{l_k \dots l_m}\!\!\!\!\rightarrow \overline{s} \Rightarrow \overline{s} \vDash \phi$$

for some $\overline{s} \in \mathbb{S}$ that has to exist, as by hypothesis, $l_0 \dots l_n$ is a valid trace in $T \mid\mid T'$.

All remaining tuples either witness π or π' satisfying **AAC1** –**AAC2.3** , or lead to contradictions similar to the one above.

$\square$

Similar proof mechanisms have been employed for showing that Lemmata 4.3, 4.6 and 4.7 also hold. Since the proofs are almost identical, we omit such detailed proofs, and only provide the construction of decorations as in steps 1. − 7. above, because these decorations are built in a different manner from what we have shown in Lemma 4.2.

Lemma 4.3. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s_0', B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. If $\pi = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ is a computation satisfying **AAC1 –AAC2.3** w.r.t. ϕ in T, then there exists a computation $\mu = \mathcal{D}, (l_k, \mathcal{D}_k), \ldots, (l_m, \mathcal{D}_m)$ satisfying **AAC1 – AAC2.3** w.r.t. $\phi \vee \psi$ in the interleaving $T \mid\mid T'$.

Proof:
We first construct the decorations $\mathcal{D}, \mathcal{D}_i$ of μ above, for $i \in \{k, \ldots, m\}$. The construction is as follows:

- initialise $\mathcal{D}, \mathcal{D}_i$ with $[\,]$ for $i \in \{k, \ldots, m\}$
- for all $\overline{\overline{\chi}} = \overline{\chi} l_k \overline{\chi}_k \ldots l_m \overline{\chi}_m \in \mathit{traces}(\pi)$, for all $\tilde{\chi} \in B^*, s' \in \mathbb{S}'$ such that $s_0' \xrightarrow{\tilde{\tilde{\chi}}}\!\!{}' s' \Rightarrow s' \models \neg\psi$, for all $\chi \in \overline{\overline{\chi}} \mid\mid \tilde{\chi}$, let $\chi = \chi' l_k \chi_k' \ldots l_m \chi_m'$. Add χ', χ_i' to $\mathcal{D}, \mathcal{D}_i$ accordingly, for all χ as before and $i \in \{k, \ldots, m\}$.

The rest of the proof follows by the construction of $\mathcal{D}, \mathcal{D}_i$, Definition 3.2 and the hypothesis. □

Lemma 4.4. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s_0', B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. The following holds:
$\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n) \in \mathit{Causes}(\phi \vee \psi, T \mid\mid T')$ if and only if
$\pi = \overline{\mathcal{D}}, (l_0, \overline{\mathcal{D}}_0), \ldots, (l_n, \overline{\mathcal{D}}_n) \in \mathit{Causes}(\phi, T)$ or
$\pi' = \overline{\mathcal{D}}', (l_0, \overline{\mathcal{D}}_0'), \ldots, (l_n, \overline{\mathcal{D}}_n') \in \mathit{Causes}(\psi, T')$
where μ is a computation of the interleaving $T \mid\mid T'$, π is a computation of T and π' is a computation of T'.

Proof:
Showing the "*if*" case is as follows. By Lemma 4.2 we know that there exists a computation $\overline{\pi} = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ satisfying **AAC1 –AAC2.3** w.r.t. ϕ in T or a computation $\overline{\pi'} = \overline{\mathcal{D}}'$, $(l_p', \overline{\mathcal{D}}_p'), \ldots, (l_q', \overline{\mathcal{D}}_q')$ satisfying **AAC1 –AAC2.3** w.r.t. ψ in T'
where $l_k \ldots l_m = l_0 \ldots l_n \downarrow A$ and $l_p' \ldots l_q' = l_0 \ldots l_n \downarrow B$.

Assume, without loss of generality, that $\overline{\pi}$ satisfies **AAC1 –AAC2.3** w.r.t. ϕ in T. It has to hold that $l_0 \ldots l_n = l_k \ldots l_m$. Otherwise, according to Lemma 4.3, we can build μ' over $l_k \ldots l_m$ such that $\mu' \in \mathit{sub}(\mu)$ satisfies **AAC1 –AAC2.3** , thus contradicting the hypothesis $\mu \in \mathit{Causes}(\phi \vee \psi, T \mid\mid T')$.

Let $\pi = \overline{\pi}$. Assume that π does not satisfy **AAC3** Hence, there exists $\pi'' \in \mathit{sub}(\pi)$ such that π'' satisfies **AAC1 –AAC2.3** w.r.t. ϕ in T. By Lemma 4.3 it holds that there exists a computation μ'' satisfying **AAC1 –AAC2.3** w.r.t. $\phi \vee \psi$ in $T \mid\mid T'$. Note that $\mu'' \in \mathit{sub}(\mu)$, hence, we reached a contradiction of the hypothesis $\mu \in Causes(\phi \vee \psi, T \mid\mid T')$. Therefore, it must be the case that π satisfies **AAC3** as well. In other words, $\pi \in Causes(\phi, T)$.

Showing the "*only if*" case is as follows. Assume, without loss of generality, that

$$\pi = \overline{\mathcal{D}}, (l_0, \overline{\mathcal{D}}_0), \dots, (l_n, \overline{\mathcal{D}}_n) \in Causes(\phi, T).$$

By Lemma 4.3 we know that there exists a computation $\mu = \mathcal{D}$, $(l_0, \mathcal{D}_0)$, $\dots$, $(l_n, \mathcal{D}_n)$ satisfying **AAC1** –**AAC2.3** w.r.t. $\phi \vee \psi$ in the interleaving $T \mid\mid T'$. Assume μ violates **AAC3** . Hence, there exists $\mu' \in sub(\mu)$ satisfying **AAC1** –**AAC2.3** w.r.t. $\phi \vee \psi$ in the interleaving $T \mid\mid T'$. By Lemma 4.2, there exists $\tilde{\pi}$ satisfying **AAC1** –**AAC2.3** w.r.t. ϕ in T. Note that $\tilde{\pi} \in sub(\pi)$. This is in contradiction with the hypothesis $\pi \in Causes(\phi, T)$. In other words, μ has to satisfy **AAC1** –**AAC3** . Hence, $\mu \in Causes(\phi \vee \psi, T \mid\mid T')$. □

Theorem 4.5. ((De-)composing Disjunction)
Consider LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s'_0, B, \rightarrow')$ such that $A \cap B = \emptyset$. Assume two HML formulae ϕ and ψ over A and B, respectively. Whenever ϕ and ψ are not immediate effects, the following holds:

$$T \mid\mid T' \downarrow (\phi \vee \psi) \;\simeq\; T \downarrow \phi + T' \downarrow \psi. \tag{4}$$

Proof:
Consider:

$$\begin{array}{rcl} (\mathbb{S}_{\mid\mid}, s_0 \mid\mid s'_0, A \cup B, \rightarrow_{\mid\mid}) &=& (T \mid\mid T') \downarrow (\phi \vee \psi) \\ (\mathbb{S}_{+}, s_0 + s'_0, A \cup B, \rightarrow_{+}) &=& (T \downarrow \phi) + (T' \downarrow \psi) \end{array}$$

The result is a direct consequence of Lemma 4.4 and the semantics of the non-deterministic choice operator $(+)$, where the isomorphic structure $(\simeq)$ is underlined by:

$$\begin{array}{rcl} & f : \mathbb{S}_{\mid\mid} \rightarrow \mathbb{S}_{+} & \\ f(s_0 \mid\mid s'_0) &=& s_0 + s'_0 \\ f(p \mid\mid q) &=& \begin{cases} p & \text{if} \quad q = s'_0 \wedge p \neq s_0 \\ q & \text{if} \quad p = s_0 \wedge q \neq s'_0 \end{cases} \end{array} \qquad \begin{array}{rcl} & f^{-1} : \mathbb{S}_{+} \rightarrow \mathbb{S}_{\mid\mid} & \\ f^{-1}(s_0 + s'_0) &=& s_0 \mid\mid s'_0 \\ f^{-1}(p) &=& \begin{cases} p \mid\mid s'_0 & \text{if} \quad p \in \mathbb{S} \wedge p \neq s_0 \\ s_0 \mid\mid p & \text{if} \quad p \in \mathbb{S}' \wedge p \neq s'_0 \end{cases} \end{array}$$

□

4.2.2. (De)-composing conjunction

Theorem 4.9 formalises the compositionality result for conjunction of effects. For an intuition, Theorem 4.9 establishes an isomorphism between the causal projection of the interleaving $T \mid\mid T'$ w.r.t. $\varphi_1 \wedge \varphi_2$, and the interleaving of the causal projections $T \downarrow \varphi_1$ and $T' \downarrow \varphi_2$.

In proving Theorem 4.9, we exploit Lemma 4.8. The latter is a consequence of Lemma 4.6 and Lemma 4.6.

Lemma 4.6. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s_0', B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. If $\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$ is a computation satisfying **AAC1 –AAC2.3** w.r.t. $\phi \wedge \psi$ in the interleaving $T \mid\mid T'$, then there exist
$\pi = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ a computation satisfying **AAC1 –AAC2.3** w.r.t. ϕ in T and
$\pi' = \overline{\mathcal{D}}', (l_p', \overline{\mathcal{D}}_p'), \ldots, (l_q', \overline{\mathcal{D}}_q')$ a computation satisfying **AAC1 –AAC2.3** w.r.t. ψ in T'
where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l_p' \ldots l_q'$.

Proof:
We first construct the decorations $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}_j'$ above, for all $i \in \{k, \ldots, m\}$ and $j \in \{p, \ldots, q\}$. The construction is as follows:

- initialise $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}_j'$ with [ ] for all $i \in \{k, \ldots, m\}$ and $j \in \{p, \ldots, q\}$
- for all $\overline{\overline{\chi}} \in \mathit{traces}(\mu)$ such that $\forall s \in \mathbb{S} : s_0 \xrightarrow{\overline{\overline{\chi}} \downarrow A} s \Rightarrow s \vDash \neg\phi$ add $\overline{\chi}, \overline{\chi}_i$ to $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i$, for all $i \in \{k, \ldots, m\}$, where $\overline{\overline{\chi}} \downarrow A = \overline{\chi} l_k \overline{\chi}_0 \ldots l_m \overline{\chi}_m$
- for all $\overline{\overline{\chi}} \in \mathit{traces}(\mu)$ such that $\forall s' \in \mathbb{S}' : s_0' \xrightarrow{\overline{\overline{\chi}} \downarrow B}' s' \Rightarrow s' \vDash \neg\psi$ add $\overline{\chi}, \overline{\chi}_i$ to $\overline{\mathcal{D}}', \overline{\mathcal{D}}_i'$, for all $i \in \{p, \ldots, q\}$, where $\overline{\overline{\chi}} \downarrow B = \overline{\chi} l_p \overline{\chi}_p \ldots l_q \overline{\chi}_q$.

Then, we proceed by reductio ad absurdum and show that π and, respectively, π' have to satisfy **AAC1 –AAC2.3** w.r.t ϕ in T and, respectively, ψ in T'. In showing this, we exploit the construction of $\overline{\mathcal{D}}, \overline{\mathcal{D}}_i, \overline{\mathcal{D}}', \overline{\mathcal{D}}_j'$, Definition 3.2 and the hypothesis. □

Lemma 4.7. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s_0', B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. If $\pi = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ is a computation satisfying **AAC1 –AAC2.3** w.r.t. ϕ in T, and $\pi' = \overline{\mathcal{D}}', (l_p', \overline{\mathcal{D}}_p'), \ldots, (l_q', \overline{\mathcal{D}}_q')$ is a computation satisfying **AAC1 –AAC2.3** w.r.t. ψ in T' then there exists $\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$ a computation satisfying **AAC1 –AAC2.3** w.r.t. $\phi \wedge \psi$ in the interleaving $T \mid\mid T'$, where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l_p' \ldots l_q'$.

Proof:
We first construct the decorations $\mathcal{D}, \mathcal{D}_i$ of μ above, for $i \in \{0, \ldots, n\}$. The construction is as follows:

- initialise $\mathcal{D}, \mathcal{D}_i$ with [ ] for $i \in \{0, \ldots, n\}$
- consider $\chi \in \mathit{traces}(\pi)$ and $\chi' \in B^*$; if $\tilde{\tilde{\chi}} = \tilde{\chi} l_0 \tilde{\chi}_0 \ldots l_n \tilde{\chi}_n \in \chi \mid\mid \chi'$ and if $\tilde{\tilde{\chi}}$ is a valid trace in $T \mid\mid T'$, add $\tilde{\chi}, \tilde{\chi}_i$ to $\mathcal{D}, \mathcal{D}_i$ accordingly, for $i \in \{0, \ldots, n\}$
- consider $\chi' \in \mathit{traces}(\pi')$ and $\chi \in A^*$; if $\tilde{\tilde{\chi}} = \tilde{\chi} l_0 \tilde{\chi}_0 \ldots l_n \tilde{\chi}_n \in \chi \mid\mid \chi'$ and if $\tilde{\tilde{\chi}}$ is a valid trace in $T \mid\mid T'$, add $\tilde{\chi}, \tilde{\chi}_i$ to $\mathcal{D}, \mathcal{D}_i$ accordingly, for $i \in \{0, \ldots, n\}$

The rest of the proof follows by the construction of $\mathcal{D}, \mathcal{D}_i$, Definition 3.2 and the hypothesis. □

Lemma 4.8. Consider two non-communicating LTSs $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s'_0, B, \rightarrow')$. Assume two HML formulae ϕ and ψ over A and B, respectively, that are not immediate effects in T and T', respectively. The following holds:
$\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n) \in \mathit{Causes}(\phi \wedge \psi, T \mid\mid T')$ if and only if
$\pi = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m) \in \mathit{Causes}(\phi, T)$ and
$\pi' = \overline{\mathcal{D}}', (l'_p, \overline{\mathcal{D}}'_p), \ldots, (l'_q, \overline{\mathcal{D}}'_q) \in \mathit{Causes}(\psi, T')$
where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l'_p \ldots l'_q$, μ is a computation of the interleaving $T \mid\mid T'$, π is a computation of T and π' is a computation of T'.

Proof:
Showing the "*if*" case is as follows. By Lemma 4.6 we know that there exist
$\overline{\pi} = \overline{\mathcal{D}}, (l_k, \overline{\mathcal{D}}_k), \ldots, (l_m, \overline{\mathcal{D}}_m)$ a computation satisfying **AAC1** –**AAC2.3** w.r.t. ϕ in T and $\overline{\pi'} = \overline{\mathcal{D}}', (l'_p, \overline{\mathcal{D}}'_p), \ldots, (l'_q, \overline{\mathcal{D}}'_q)$ a computation satisfying **AAC1** –**AAC2.3** w.r.t. ψ in T' where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l'_p \ldots l'_q$. Assume, without loss of generality, that π violates **AAC3** . In other words, assume there exists a computation $\overline{\pi} \in \mathit{sub}(\pi)$ such that $\overline{\pi}$ satisfies **AAC1** –**AAC2.3** . Hence, by Lemma 4.7 we can build a computation $\mu' \in \mathit{sub}(\mu)$ based on $\overline{\pi}$ and π', that satisfies **AAC1** –**AAC2.3** w.r.t. $\phi \wedge \psi$ in $T \mid\mid T'$. This violates the hypothesis $\mu \in \mathit{Causes}(\phi \wedge \psi, T \mid\mid T')$. Therefore, our last assumption was wrong and the following hold: $\pi \in \mathit{Causes}(\phi, T)$ and $\pi' \in \mathit{Causes}(\psi, T')$.

Showing the "*only if*" case is as follows. By Lemma 4.7 we know that there exists

$$\mu = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots, (l_n, \mathcal{D}_n)$$

a computation satisfying **AAC1** –**AAC2.3** w.r.t. $\phi \wedge \psi$ in the interleaving $T \mid\mid T'$, where $l_0 \ldots l_n \in l_k \ldots l_m \mid\mid l'_p \ldots l'_q$. Assume μ does not satisfy **AAC3** . In other words, assume there exists $\tilde{\mu} \in \mathit{sub}(\mu)$ satisfying **AAC1** –**AAC2.3** w.r.t. $\phi \wedge \psi$ in the interleaving $T \mid\mid T'$. By Lemma 4.6, we can build $\tilde{\pi} \in \mathit{sub}(\pi)$ a computation satisfying **AAC1** –**AAC2.3** w.r.t. ϕ in T and $\tilde{\pi}' \in \mathit{sub}(\pi')$ a computation satisfying **AAC1** –**AAC2.3** w.r.t. ψ in T'. This contradicts the hypothesis $\pi \in \mathit{Causes}(\phi, T)$ and $\pi' \in \mathit{Causes}(\psi, T')$. Hence, our last assumption that μ does not satisfy **AAC3** was wrong. It follows that $\mu \in \mathit{Causes}(\phi \wedge \psi, T \mid\mid T')$. □

Theorem 4.9. ((De-)composing Conjunction)
Consider $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s'_0, B, \rightarrow')$ such that $A \cap B = \emptyset$. Assume two HML formulae ϕ and ψ over A and B, respectively. The following holds:

$$T \mid\mid T' \downarrow (\phi \wedge \psi) \;=\; (T \downarrow \phi) \mid\mid (T' \downarrow \psi). \qquad (5)$$

Proof:
The result is a direct consequence of Lemma 4.8 and the semantics of the parallel composition operator ($\mid\mid$) for non-communicating LTSs. □

5. Encoding causality

This section is dedicated to encoding causality in terms of the so-called modal formulae with data, thus paving the way to identifying causalities using the mCRL2 model-checker.

5.1. Modal formulae with data

In what follows, we provide an overview of the fragment of modal μ-calculus with data [21] that is relevant for our work. This logic extending the Hennesy-Milner Logic in Definition 2.5 with data types and fixed points:

$$\begin{array}{rcl} af & ::= & t \mid a \mid \mathit{true} \mid \mathit{false} \\ R & ::= & \varepsilon \mid af \mid R \cdot R \mid R + R \mid R^* \mid R^+ \\ \phi & ::= & \mathit{true} \mid \mathit{false} \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \to \phi \mid \forall d : D.\phi \mid \exists d : D.\phi \mid \langle R \rangle \phi \mid [R]\phi \\ & & \mu X(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).\phi \mid \\ & & \mathcal{V} X(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).\phi \mid X(t_1, \ldots, t_n) \end{array} \tag{6}$$

In (6) above, atomic propositions af define a set of actions. Hence, a stands for the set consisting of action a (within an LTS system model). Additionally, true stands for the set of all actions, whereas false stands for the empty set. Any expression t of Boolean type is a formula.

Regular formulae R enable sequences of actions in modalities, and are defined in the standard way. The formula ε represents the empty sequence of actions; intuitively, by performing no action we remain in the same state of the LTS model. $R_1 \cdot R_2$ refers to the sequences of actions obtained by concatenating expressions in R_1 with expressions in R_2. $R_1 + R_2$ represents the union of sequences of actions, R^* denotes zero or more repetitions of the sequences in R, whereas R^+ stands for one or more repetitions.

We can write, for instance, $[\mathit{true}^*]\langle a \rangle \mathit{false}$ in order to express that action a cannot be triggered in our LTS, or $\langle a.b + c.d \rangle \mathit{true}$ to say that ab or cd can be executed from the current state.

More formally:

$$\begin{array}{rclcrcl} \langle R_1 + R_2 \rangle \phi & = & \langle R_1 \rangle \phi \vee \langle R_2 \rangle \phi & \qquad & \langle R_1 \cdot R_2 \rangle \phi & = & \langle R_1 \rangle \langle R_2 \rangle \phi \\ [R_1 + R_2]\phi & = & [R_1]\phi \wedge [R_2]\phi & & [R_1 \cdot R_2]\phi & = & [R_1][R_2]\phi. \end{array}$$

Negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$) of formulae have the usual interpretation. Existential and universal quantifiers for ranging over data domains are also introduced. μ and, respectively, $\mathcal{V}$ stand for the minimal and, respectively, the maximal fixed point operators. As also indicated in [21], intuitively, the formula $\mu X.\phi$ (respectively, $\nu X.\phi$) is valid for all those states in the smallest (respectively, largest) set X that satisfies the equation $X = \phi$.

For an example, the formula

$$\nu Y.\mu X.(p \wedge \langle a \rangle Y) \vee \langle a \rangle X \tag{7}$$

says that on some a-path there are infinitely many states where p holds. The formula

$$\nu X.\mu Y([a]Y \vee [b]X) \tag{8}$$

states that each sequence of a and b actions contains only finite subsequences of as.

Definition 5.1. (Quantifier-free Positive Normal Form)
A modal formula with data as in (6) is in *positive normal form* if negations occur only in front of atomic propositions. Observe that any modal formula can be put into positive normal form by use of de Morgan laws and α-conversion [28]. Moreover, for finite data, the $\forall$ and $\exists$ operators can be considered as short-hands for $\wedge$ and $\vee$. A modal formula in positive normal form, built over finite data and without occurrences of $\forall$ and $\exists$, is called a formula in *quantifier-free positive normal form*.

Definition 5.2. (Alternation Depth)
The *alternation depth* of a formula ϕ in quantifier-free positive normal form, written $AD(\phi)$, is the number of alternating fixed points inductively defined as follows:

$$\begin{array}{rcl}
AD(f) & = & 0 \\
 & & \text{for } f \in \{af, \neg af, \varepsilon, X(t_1, \ldots, t_n)\} \\
AD(@f) & = & AD(f) \\
 & & \text{for } @ \in \{[R], \langle R \rangle\} \\
AD(f \boxtimes g) & = & max(AD(f), AD(g)) \\
 & & \text{for } \boxtimes \in \{\vee, \wedge\} \\
AD(\mu X(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).f) & = & 1 + max\{AD(g) \mid g \text{ is a } \nu \text{ sub-formula of } f\} \\
AD(\nu X(d_1 : D_1 := t_1, \ldots, d_n : D_n := t_n).f) & = & 1 + max\{AD(g) \mid g \text{ is a } \mu \text{ sub-formula of } f\}
\end{array} \tag{9}$$

For instance:

$$AD(\ (\mu X_1.\nu X_2.X_1 \vee X_2)\ \wedge\ (\mu X_3.X_3 \vee [a]true)\ )\ =\ 2$$

5.2. Encoding AAC1–AAC3

In this section we introduce an encoding of actual causality as in Definition 3.2, via modal formulae with data [21]. This paves the way to the automatic identification of causes in the mCRL2 model checker [22, 21].

Some of the "ingredients" necessary for the aforementioned encoding are as follows.

Consider a computation

$$\pi = \mathcal{D}, (l_0, \mathcal{D}_0), \ldots (l_n, \mathcal{D}_n) \tag{10}$$

over an LTS $T = (\mathbb{S}, s_0, A, \rightarrow)$. We write l for the underlying trace $l_0 \ldots l_n$, and $\mathcal{LD}$ for the list of size-compatible lists $\mathcal{D}, \mathcal{D}_0, \ldots, \mathcal{D}_n$. In our modal formulae, $traceslL\mathcal{D}$ stands for $traces(\pi)$, whereas $sizeCompatible(\mathcal{LD})$ checks whether the lists in $\mathcal{LD}$ are size-compatible. Additionally, recall that our approach relies on a complete exploration of our system models. We write $allExec$ to represent the list of all valid executions within T. (According to the remarks in Section 3.2, $allExec$ and $traceslL\mathcal{D}$ are finite.)

We write A^* for the "type" of words over actions in A (*e.g.*, $l : A^*$), and $List[A^*]$ for the "type" of lists of words over actions in A (*e.g.*, $\mathcal{D}_i : List[A^*]$, for $i \in \{0, \ldots, n\}$). We write $head(L)$ to denote the first element of a list L, and $tail(L)$ to represent the list obtained by removing its first element. An expression of shape $e \in L$ checks wether a list L contains the element e. $\sharp L$ is the length of L.

Checking whether π in (10) is a cause for an effect given by the HML formula φ in the sense of Definition 3.2, reduces to checking the satisfiability of the modal formulae with data below, encoding **AAC1** – **AAC3** . (Note that PC, NEC, SUF, NOC and MIN are variable names used in the context of fixed point operators encoding a so-called "positive" causality in the spirit of $\mathbf{AAC1}$, necessity, sufficiency, actions that are causal by their non-occurrence and the minimality condition, respectively.)

The encoding of **AAC1** is:

$$\mu PC(l : List[A^*]) \,.\, \langle l \rangle \varphi \tag{11}$$

In accordance with **AAC1** in Definition 3.2, the formula in (11) identifies a setting in which by executing the causal trace l a state satisfying the effect φ is reached. The fixed point notation μPC in (11) is used for consistency reasons. The formula in (11) can be equivalently rewritten as $\langle l \rangle \varphi$.

The encoding of **AAC2.1** is:

$$\begin{aligned} &\nu NEC(l : A^*, tracesl\mathcal{LD} : List[A^*], allExec : List[A^*]) \,.\, (allExec == [\,]) \;\vee \\ &\qquad (\;\; (\langle head(allExec) \rangle \neg\varphi \rightarrow \\ &\qquad\qquad (\neg subword(l, head(allExec)) \vee head(allExec) \in tracesl\mathcal{LD})\;) \;\wedge \\ &\qquad\quad NEC(l, tracesl\mathcal{LD}, tail(allExec)) \;\;) \end{aligned} \tag{12}$$

Formula (12) encodes a necessity condition. It states that whenever an execution of the LTS leads to a state where the effect φ cannot be observed (*i.e.*, $\langle head(allExec) \rangle \neg\varphi$), one of the following holds: (a) The causal trace l is not part of the aforementioned execution (*i.e.*, $\neg subword(l, head(allExec))$). (b) The execution is interleaved with events causal by their non-occurrence which shift the setting from a hazardous one to a safe one ($i.e., head(allExec) \in tracesl\mathcal{LD}$). The recursive call on $tail(allExec)$ guarantees all executions within the LTS are checked in the spirit of **AAC2.1** .

The encoding of **AAC2.2** is:

$$\begin{aligned} &\nu SUF(l : A^*, tracesl\mathcal{LD} : List[A^*], allExec : List[A^*]) \,.\, (allExec == [\,]) \;\vee \\ &\qquad (\;\; ((subword(l, head(allExec)) \wedge head(allExec) \notin tracesl\mathcal{LD}) \rightarrow \\ &\qquad\qquad [head(allExec)]\varphi\;) \;\wedge \\ &\qquad\quad SUF(l, tracesl\mathcal{LD}, tail(allExec)) \;\;) \end{aligned} \tag{13}$$

Formula (13) encodes a sufficiency condition, and is the dual of (12). Intuitively, it states that whenever the causal trace occurs within an execution $exec \in allExec$ of the LTS, and $exec$ is not interleaved with events causal by their non-occurrence, it must be the case that $exec$ always leads to a hazard.

The encoding of **AAC2.3** is:

$$\begin{aligned} &\nu NOC(tracesl\mathcal{LD} : List[A^*]) \,.\, (tracesl\mathcal{LD} == [\,]) \;\vee \\ &\qquad ([head(tracesl\mathcal{LD})]\neg\varphi \;\wedge\; NOC(tail(tracesl\mathcal{LD}))) \end{aligned} \tag{14}$$

Formula (14) encodes the events causal by their non-occurrence that shift the context from a hazardous to a non-hazardous one. More precisely, all executions $exec$ that are obtained by interleaving the causal trace l with events causal by their non-occurrence (*i.e.*, $exec \in tracesl\mathcal{LD}$) always lead to states where the effect φ cannot be observed (*i.e.*, $[exec]\neg\varphi$).

The encoding of **AAC3** is:

$$\begin{aligned}
&\mu MIN(l : A^*, traceslLD : List[A^*], allExec : List[A^*])\,.\forall l' : A^*.\forall \mathcal{LD}' : List[List[A^*]]\,.\\
&\quad(\ sizeCompatible(\mathcal{LD}') \wedge (\sharp l' == \sharp\mathcal{LD}') \wedge (subword(l', l))\ ) \rightarrow\\
&\quad\neg PC(l')\vee\\
&\quad\neg NEC(l', traceslLD', allExec)\vee\\
&\quad\neg SUF(l', traceslLD', allExec)\vee\\
&\quad\neg NOC(traceslLD')
\end{aligned} \tag{15}$$

In (15) we write, for instance, $PC(l')$ to represent the modal formula in (11) for the execution l'. Formula (15) encodes the minimality condition for π in (10). It fails whenever all conditions $\mathbf{AAC1}-\mathbf{AAC2.3}$ are satisfied by a sub-computation of π, built over an arbitrary execution l' and the arbitrary size-compatible lists of events causal by their non-occurrence in $\mathcal{LD}'$.

Remark 5.3. Observe that formulae (11)-(14) have alternation depth at most 1.

Theorem 5.4. (Correctness of Encoding)
Formulae (11)–(15) correctly encode conditions **AAC1**–**AAC3** in Definition 3.2.

Proof:
Condition **AAC1** refers to the *existence* of a trace that satisfies φ; condition **AAC3** requires the *existence* of an execution underlined by $\pi' \in sub(\pi)$ that violates at least one of **AAC1**–**AAC3**. Hence, we encode **AAC1** and **AAC3** as *least fixed points* in (11) and (15), respectively. It is trivial to see that **AAC1** is satisfied if and only if the least fixed point in (11) is satisfied. In case of **AAC3**, the encoding of minimality in (11) is achieved by ensuring that all $subword$s of the cause at hand l violate at least one of the conditions encoded below. Symmetrically, **AAC2.1**–**AAC2.3** require the satisfaction of some conditions *for all* χ' of a certain shape. Hence, we encode **AAC2.1**–**AAC2.3** as *greatest fixed points* in (12), (13) and (14), respectively. Universal quantifications ($\forall\chi'$) in **AAC2.1**–**AAC2.3** are handled within the associated modal formulae via recursive calls parametrised over all executions or decorated traces of the system under analysis. The rest of the translation is purely syntactic. □

6. Causality at work

In this section, we present our approach to causality checking from a more practical perspective. We introduce the automated tool CauseJMu built on top of mCRL2 and Java, and discuss its performance w.r.t. causality checking in the context of the level crossing example in Section 2.3. We proceed by first discussing the mCRL2 toolset, used in order to actually encode the LTS system models, the associated safety properties and the modal formulae capturing causality as in Definition 3.2. The mCRL2 model-checker is then invoked within CauseJMu to compute causalities.

6.1. The mCRL2 toolset

mCRL2 [22, 21] is a formal toolset successfully exploited for the modelling, validation and verification of concurrent systems and protocols. In this section we provide a brief overview of mCRL2, relevant for our work on causality checking. Namely, we discuss how concurrent systems can be specified and model-checked in mCRL2.

Systems are specified in mCRL2 in a process algebraic fashion [23], according to the approach in Section 2.1. The keyword `act` introduces the actions of the LTSs, whereas `proc` introduces the LTS process terms built according to the grammar in (1). The keyword `comm` specifies the communicating actions a_in, a_out and the name of the action a_ack (or a, depending on the context) acknowledging the communication as in (2). `allow` introduces the actions the LTS is allowed to perform. Moreover, `allow` can be used in order to disable the independent (not synchronised) execution of communicating actions a_in, a_out. `init` defines (the composition of) the LTSs we want to model-check.

```
act Ta_out, Tc, Tl_out, Go_out, Ta_in, Gc_out,
      Tl_in, Ca, Gc_in, Go_in, Cc, Cl, Ta, Tl, Go_ack, Gc_ack;

proc
    T  = Ta_out.Tc.Tl_out.delta;
    G  = Go_out.G+Ta_in.G1;
    G1 = Gc_out.G1+Tl_in.G;
    C  = Ca.C1;
    C1 = Gc_in.C1+Go_in.Cc.Cl.delta;

init
    allow( { Ta, Tl, Tc, Ca, Cc, Cl, Gc_ack, Go_ack },
               comm( { Ta_out|Ta_in -> Ta, Tl_out|Tl_in -> Tl,
                       Gc_out|Gc_in -> Gc_ack, Go_in|Go_out -> Go_ack },
                       T || G || C )
     );
```

Figure 5. Level Crossing mCRL2 Specification: spec.mcrl2

In Figure 5, we give the mCRL2 specification of the level crossing example in Section 2.3. Without loss of generality, assume this specification is written in the file *spec.mcrl2*.

Actions `Ta_out, Ta_in, Go_out`, *etc.*, are the actions of the LTSs formalizing the models of the car, train and the gate in (3). We write `delta` for the empty process δ as in (1). The associated LTS process terms `C`, `T` and, respectively, `G` execute in parallel: `T || G || C`. The `comm` section defines, for instance, communications acknowledging the arrival of the train `Ta_out|Ta_in -> Ta`, or the opening of the gate `Go_in|Go_out -> Go_ack`. The actions allowed to be independently executed by `T || G || C` correspond to the train arriving, train leaving, train entering the crossing, car arriving, car entering the crossing, car leaving, acknowledging the gate closing and acknowledging the gate opening, respectively: `{ Ta, Tl, Tc, Ca, Cc, Cl, Gc_ack, Go_ack }`.

Note that the actual LTS model as in Figure 2 can be derived from the mCRL2 specification in Figure 5, in accordance with the set of semantic rules in (2). This is achieved by applying a series of tools in the mCRL2 toolbox, in a "pipe-and-filter" fashion. The sequence of steps can be summarised as follows:

$$\mathit{spec.mcrl2} \xrightarrow{mcrl22lps} \mathit{spec.lps} \xrightarrow{lps2lts} \mathit{spec.lts} \xrightarrow{ltsconvert} \mathit{spec.fsm} \tag{16}$$

The first step translates the mCRL2 specification to a linear process specification in the *.lps* format. The next step generates an LTS from a linear process. Eventually, the LTS is converted into the *.fsm* format which contains transition and state information, in a human readable format.

A modal formula with data like in (6) can be defined in an *.mcf* file. For instance, in the context of the level crossing example in Section 2.3, we can create a file called *safety.mcf*, and define a safety formula specifying that a train and a car cannot be in the crossing at the same time:

```
[true*]!(<Tc>true && <Cc>true)
```

Note that the action names `Tc` and `Cc` must be defined in the model specification file *.mcrl2*. Additionally, observe the slightly different notation used for denoting negation and conjunction of formulae: `!` and `&&`, respectively. Model-checking the above formula against the model in Figure 5 is possible via the following sequence of steps:

$$\mathit{spec.lps} \xrightarrow{lps2pbes} \mathit{spec.pbes} \xrightarrow{pbes2bool} \mathit{true}/\mathit{false} \tag{17}$$

The *lps2pbes* tool generates a parameterised boolean equation system from the linear process specification. The formula saved in the *safety.mcf* file will be used as a parameter of the *lps2pbes* tool. The latter can generate a counterexample based on the *spec.bpes* file in case the formula is not satisfied by the system model.

For a thorough description of the mCRL2 toolset, and the associated user manual, we refer the interested reader to: `https://www.mcrl2.org/web/user_manual/index.html`.

```
sort myAct = struct ma | mb | mc;

var l1, l2 : List(myAct);

map subword: List(myAct) # List(myAct) -> Bool;

eqn
   (l1 == []) -> subword(l1, l2) = true;
   (l1 != []) && (l2 == []) -> subword(l1, l2) = false;
   (l1 != []) && (l2 != []) -> subword(l1, l2) =
         ( (head(l1) == head(l2) && subword(tail(l1), tail(l2))) ||
           (head(l1) != head(l2) && subword(l1, tail(l2))) );
```

Figure 6. Specifying Functions in mCRL2

Last, but not least, a *.mcrl2* file can include variables and equational specifications of functions with parameters of different sorts. For an example, we provide in Figure 6 the specification of a function `subword` determining whether all the elements of a given list `l1` occur in the same order within a list `l2`. The keywords `sort` and `struct` introduce the data type `myAct` which, intuitively, stands for an alphabet consisting of the constants `ma, mb, mc`. The keyword `var` introduces a list of variables of a certain sort. The keyword `map` introduces the signature of the `subword` function with two parameters of type list of elements in `myAct`. Note that `List(...)` and `Bool` are predefined in mCRL2. Functions `head` and `tail` are predefined as well, and return the head, respectively the list obtained after removing the first element of a list. The empty list is naturally denoted by `[]`. The keyword `eqn` introduces the definition of `subword` via three conditional (recursive) equations; the condition is the expression on the left hand side of `->`.

6.2. CauseJMu: a Java-mCRL2 tool for causality

In this section we introduce CauseJMu, a tool for automatically computing causality as in Section 3.2, with respect to effects in HML. Our tool is available for download at: `https://www.sen.uni-konstanz.de/research/research/causejmu`.

CauseJMu is a tool based on the interplay between Java and mCRL2. On the one hand, mCRL2 is exploited for formally specifying the LTS under analysis, and a safety property that can be disproved by reaching a hazard/effect in HML. On the other hand, mCRL2 performs the model-checking of both the safety property and the modal formulae with data (11)–(15) encoding the definition of causality in Section 3.2, whenever a counterexample is identified. All computations π built on top of such counterexamples that determine the *true* result of the model-checking procedure are valid causal explanations.

As already mentioned in Section 3.2, our approach to causality relies on a complete exploration of the LTS model in order to identify all possible counterexamples witnessing hazardous situations. Nevertheless, the model checking problem is solved in mCRL2 through solving a parity game [29, 30], where winning strategies of one player are then used to construct a subgraph of the original LTS in which the same strategies can be used to explain the model checking result. In this setting, there is no natural notion of continuing a search after a first counterexample,

Therefore, CauseJMu runs a Breadth First Search (BFS) / Depth First Search (DFS) based algorithm [26] implemented in Java in order to identify all counterexamples underlying possible causal traces within the LTS model (generated with mCRL2 as in (16)). Additionally, for each counterexample, the Java-implemented algorithm identifies the size-compatible lists of executions, or decorations, that can switch the setting from a hazardous to a non-hazardous one. In other words, the Java component of CauseJMu automatically computes the set of computations of a given LTS.

It is also worth mentioning that CauseJMu is a multi-threaded application. In short, each such thread is in charge of handling a set of computations. For each computation, one thread (a) translates conditions **AAC1** –**AAC3** in Definition 3.2 into mCRL2 format, based on the corresponding encodings (11)–(15) and (b) invokes the mCRL2 model-checking procedure for each of the encodings. Whenever the formulae (11)–(15) are satisfied, the corresponding computation is stored as a valid cause.

The user is left to specify a valid model within a *.mcrl2* file, and a hazard encoded as an HML formula. Moreover, the user is required to specify the maximum length of the causal traces to be computed. The length can be a positive integer value, or "any length" whenever $*$ is introduced. Similarly, the user has to specify the maximum length of the decorations causal by their non-occurrence, via a positive integer values or $*$. The number of threads handling causality checking can also be specified. Naturally, one thread corresponds to a "sequential" run of CauseJMu. The rest is handled by CauseJMu in a fully automated manner, *e.g.*, the interplay between Java and mCRL2, the automated generation of formulae (11)–(15), the model-checking in mCRL2, etc. The set of causal computations as in Definition 3.2 are eventually displayed to the user. The main steps of the causality checking approach are sketched in Algorithm 1.

Algorithm 1: Main steps in CauseJMu

```
Input:  The .mcrl2 model file path, the hazard formula $\varphi$, the maximum causal trace length, the
        maximum length of decorations causal by their non-occurrence, the number of threads
Output: The set SC of causal computations

 1. Run mCRL2 and generate the LTS model by following the steps in (16);
 2. Run the Java BFS/DFS-implementation to generate all computations within the LTS;
 3. Split the set of computations amongst the Java threads;
 4. For each thread:
 5.   For each computation $\pi$, invoke the Java procedure generating the .mcf file encoding (11);
 6.   Model-check with mCRL2 all the .mcf files by following the steps in (17);
 7.   Store all computations $\pi$ underling counterexamples identified in step 6.;
 8.   For each computation $\pi$ stored in step 7.:
 9.     Invoke the Java procedure generating the .mcf files encoding (12)–(15);
10.     Model-check with mCRL2 all the .mcf files by following the steps in (17);
11.     If all formulae in step 10. are satisfied, store $\pi$ in the set SC of causal computations.
```

In what follows, we discuss a few implementation details. In step 9., we only generate the *.mcf* files encoding (12)–(14). We skip the minimality condition **AAC3** encoded by (15). Instead, at the very end of our causality checking algorithm, we select only those computations π that satisfy **AAC1** –**AAC2.3** (*i.e.*, those computations π for which the corresponding *.mcf* encodings (11)–(14) are satisfied), whose underlying causal traces are not supra-words of other causal traces. All computations π that satisfy **AAC1** –**AAC2.3** and entail minimal causal traces comply to Definition 3.2 by construction. In step 7., we only choose those computations π whose underlying causal traces are not supra-words of computations π' already stored in the set *SC* of causal computations. This decision is based on the minimality condition in **AAC3** . Additionally, after running the BFS/DFS algorithm in step 2., all identified traces are sorted according to their length. This contributes to speeding up the causality checking procedure as the shortest causes are identified first.

Automating the generation of formulae (11)–(15) in Java is not only a nice-to-have feature, but a necessity. This is because actions (introduced by the `act` keyword in the *.mcrl2* model specification files) cannot be interpreted as data within modal formulae defined in (6). Also, data cannot be encapsulated within the $\langle\rangle$ and $[]$ modalities.

For an example, consider a trivial system $s_0 \xrightarrow{a} s_1 \xrightarrow{h} s_2$, and the effect $\varphi = \langle h \rangle \top$. It is easy to see that the computation $(a, [h])$ is the only cause with respect to φ. The associated *.mcrl2* file is:

```
act a h;
proc P = a.h;
init allow( { a, h }, P);
```

Nevertheless, for instance, formula (14) encoding non-occurrence of events cannot be straightforwardly specified into a *.mcf* file. Instead, the *.mcrl2* file is enriched with a new sort `myAct` defining a new constant for each action of the model:

```
sort myAct = struct ma | mh;

act a h;
proc P = a.h;
init allow( { a, h }, P);
```

Then, formula (14) is encoded into a *.mcf* file as follows:

```
nu NOC(tracesLD : List(List(myAct)) = [[ma, mh]]) . val(tracesLD == []) ||
    (  (mu X(trace : List(myAct) = head(tracesLD)).
            (val(trace == []) && !<h>true) ||
            (val(trace != []) && (head(trace) == ma) && <a>X(tail(trace))) ||
            (val(trace != []) && (head(trace) == mh) && <h>X(tail(trace)))
        ) && NOC(tail(tracesLD))  )
```

As $\langle ma \cdot mh \rangle \mathit{true}$ is not a valid modal formula with data in mCRL2, the corresponding correct formula $\langle a \cdot h \rangle \mathit{true}$ is encoded via the fixed point `mu X(trace : List(myAct) = head(tracesLD)) ...` above. For larger systems, manually encoding formulae (11)–(15) into mCRL2 is not feasible.

6.2.1. Experimental evaluation

We ran CauseJMu for the level crossing example in Section 2.3, on a machine operating on Linux 4.19.5, Intel Core i5 7200U, 8 GB RAM, 64-bit architecture.

As in Figure 2, the LTS corresponding to the parallel execution of the car (C), train (T) and gate (G) processes consists of 20 states and 31. In this case, CauseJMu considered 71 traces as candidates for causality, and invoked mCRL2 for model-checking about 100 formulae.

CauseJMu correctly identified one causal computation, with the underlying causal trace

$$Ca\ Go\_ack\ Ta$$

and the size-compatible lists of executions causal by their non-occurrence, of length 30. In other words, CauseJMu correctly identified $Ca\ Go\_ack\ Ta$ as the shortest trace leading to the hazard $\langle Cc \rangle \top \wedge \langle Tc \rangle \top$, and 30 ways of avoiding the hazardous situation despite the execution of the aforementioned causal trace. Intuitively, $Ca\ Go\_ack\ Ta$ describes a scenario in which the car approaches the crossing, the gate is open (and hence, the car can enter the crossing) and the train approaches the crossing as well. All this leads to a state in which both the car and the train can be in the crossing at the same time. One way of avoiding the accident (and identified by CauseJMu) is, for instance, given by the following trace: $Ca\ Go\_ack\ Cc\ Cl\ Ta$. In other words, car enters and immediately leaves the crossing before train arrives at the crossing.

We ran CauseJMu to determine causalities in the level crossing example, and observed the execution times depending on the number of Java threads:

No. threads	Exec. time (seconds)
1	104
2	54
3	39
4	33
5	25
6	25
10	16
56	4

If for one thread running CauseJMu took about one and a half minutes, the execution time decreased to 25 seconds for five and six threads, respectively. On 10 threads, CauseJMu performed the causality checking within 16 seconds, and on 56 threads in 4 seconds.

mCRL2 implements an efficient model-checking by solving a parity game. The complexity of model-checking a modal formula f as in (6) is:

$$O(|{\rightarrow}| \times (\mid S \mid \times \mid f \mid)^{d-1})$$

where $|{\rightarrow}|$ stands for the size of the transition relation of the LTS under analysis, $\mid S \mid$ is the number of states within the LTS, $\mid f \mid$ is the size of the formula f and d is the alternation depth. Recall that explicitly model checking the minimality formula (15) could be avoided: in step 7. of Algorithm 1 we only choose those computations π whose underlying causal traces are not supra-words of computations π' already stored in the set *SC* of causal computations. Moreover, as stated in Remark 5.3, the remaining formulae (11)–(14) are of alternation depth at most 1. Hence, the resulting model-checking complexity is linear in the size of the transition relation, for each of the considered formulae!

On the implementation side, the drawbacks are as follows. On the one hand, CauseJMu requires most of the time in order to compute all the traces and associated decorations based on the LTS model generated with mCRL2. Nevertheless, it suffices to run this preprocessing step only once for each model; the resulted information could be easily stored for later use. On the other hand, considerable amount of time is added due to the hard disk read/writes that enable the communication between Java and mCLR2. We consider overcoming this issue and further increase performance by using memory-mapped files, for instance.

A larger example consisting of two cars running in parallel with the train and the gate processes as in (3) entailed a model consisting of 100 states and 235 one-step transitions. In this case, CauseJMu considered about 11000 traces as candidates for causality, and invoked mCRL2 for model-checking around 58000 formulae. With the current implementation, on 56 threads, all causes witnessing the collision between one of the cars and the train, together with all the ways to avoid these collisions, could be computed within 713 seconds.

7. Conclusions

In this paper, we exploited counterfactual causal reasoning in order to explain counterexamples of model checking. In our setting hazardous situations are expressed in the Hennessy Milner Logic (HML) and the system model is given as a labelled transition system (LTSs). Our notion of causality is an adoption of Halpern's definition [2, 3], based on trace models in the style of Leitner-Fischer and Leue [4, 5]. We also showed that causality is compositional in the context of non-communicating, interleaved LTSs. We demonstrated by means of an example that similar results cannot be derived for communicating systems. We provided an encoding of causality in Definition 3.2, in terms of modal formulae with data. This is in order to prepare the ingredients for a faithful encoding of causality checking in the mCRL2 model-checker. Eventually, we showed how the mCRL2 toolset can be exploited using a Java wrapper, to automatically determine causal computations based on an LTS system model and an HML hazard/effect specified by the user. The integration mCRL2 and the Java wrapper was implemented in the automated tool CauseJMu. Additionally, we provided insight on the execution time of CauseJMu, when reasoning about causality within a railway level crossing.

As future work we consider optimising the implementation of CauseJMu. We would like to evaluate its performance on more (real-world) case studies, and compare the results with similar tools [31] that for example use parity games for model-checking. Additionally, we want to extend our approach to handle causalities with respect to the violation of liveness properties as well, in which case counterexamples are not finite traces, but rather loops of some sort.

We would like to understand what kind of (configurations over) communicating systems, still useful in practice, benefit from compositionality of causality. We would also like to exploit the possibility of capturing true concurrency semantics, e.g., using event structures, within our notion of causality. This will allow for identifying causal traces of shape ab and ba, for instance, whenever they represent mere interleaving of the same concurrent system and hence, is likely to lead to a more efficient computation of causes.

Acknowledgements The authors are grateful to the reviewers of Fundamenta Informaticae and FROM 2018, for their feedback and observations. We are specially grateful to Tim Willemse, for his useful comments and insight into the mCRL2 toolset.

References

[1] Lewis D. Counterfactuals. Blackwell Publishers, 1973.

[2] Halpern JY, Pearl J. Causes and Explanations: A Structural-Model Approach: Part 1: Causes. In: Breese and Koller [48], 2001 pp. 194–202. URL `https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=100&proceeding_id=17`.

[3] Halpern JY. A Modification of the Halpern-Pearl Definition of Causality. In: Yang and Wooldridge [49], 2015 pp. 3022–3033. URL `http://ijcai.org/Abstract/15/427`.

[4] Leitner-Fischer F, Leue S. Probabilistic fault tree synthesis using causality computation. *IJCCBS*, 2013. **4**(2):119–143. doi:10.1504/IJCCBS.2013.056492.

[5] Leitner-Fischer F, Leue S. Causality Checking for Complex System Models. In: Giacobazzi et al. [37], 2013 pp. 248–267. doi:10.1007/978-3-642-35873-9_16.

[6] Caltais G, Leue S, Mousavi MR. (De-)Composing Causality in Labeled Transition Systems. In: Gößler and Sokolsky [36], 2016 pp. 10–24. doi:10.4204/EPTCS.224.3.

[7] Hennessy M, Milner R. On Observing Nondeterminism and Concurrency. In: de Bakker and van Leeuwen [39], 1980 pp. 299–309. URL `http://dx.doi.org/10.1007/3-540-10003-2_79`.

[8] Zeller A. Why Programs Fail: A Guide to Systematic Debugging. Elsevier, 2009.

[9] Renieris M, Reiss S. Fault localization with nearest neighbor queries. In: 18th International Conference on Automated Software Engineering. Montreal, Canada, 2003 .

[10] Groce A, Visser W. What Went Wrong: Explaining Counterexamples. In: Workshop on Software Model Checking (SPIN), Lecture Notes in Computer Science 2648. Springer, 2003 pp. 121–135.

[11] Groce A, Chaki S, Kroening D, Strichman O. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006. **8**(3):229–247. doi:10.1007/s10009-005-0202-0.

[12] Ladkin P, Loer K. Analysing Aviation Accidents Using WB-Analysis – an Application of Multimodal Reasoning. In: AAAI Spring Symposium. AAAI, 1998 URL `https://www.aaai.org/Papers/Symposia/Spring/1998/SS-98-04/SS98-04-031.pdf`.

[13] Gößler G, Le Métayer D, Raclet J. Causality Analysis in Contract Violation. In: Runtime Verification - First International Conference, RV 2010, volume 6418 of *Lecture Notes in Computer Science*. Springer, 2010 pp. 270–284. doi:10.1007/978-3-642-16612-9_21.

[14] Gößler G, Astefanoaei L. Blaming in component-based real-time systems. In: 2014 International Conference on Embedded Software, EMSOFT 2014. ACM Press, 2014 pp. 7:1–7:10. doi:10.1145/2656045.2656048.

[15] Gößler G, Le Métayer D. A general framework for blaming in component-based systems. *Sci. Comput. Program.*, 2015. **113**:223–235. doi:10.1016/j.scico.2015.06.010.

[16] Gößler G, Stefani J. Fault Ascription in Concurrent Systems. In: Trustworthy Global Computing - 10th International Symposium, TGC, volume 9533 of *Lecture Notes in Computer Science*. Springer, 2016 pp. 79–94. doi:10.1007/978-3-319-28766-9.

[17] Befrouei MT, Wang C, Weissenbacher G. Abstraction and Mining of Traces to Explain Concurrency Bugs. In: Bonakdarpour and Smolka [35], 2014 pp. 162–177. URL `http://dx.doi.org/10.1007/978-3-319-11164-3_14`.

[18] Milner R. A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN: 3-540-10235-3. doi:10.1007/3-540-10235-3.

[19] Arbach Y, Karcher DS, Peters K, Nestmann U. Dynamic Causality in Event Structures. *Logical Methods in Computer Science*, 2018. **14**(1). doi:10.23638/LMCS-14(1:17)2018.

[20] Nielsen M, Plotkin GD, Winskel G. Petri Nets, Event Structures and Domains, Part I. *Theor. Comput. Sci.*, 1981. **13**:85–108. doi:10.1016/0304-3975(81)90112-2.

[21] Groote JF, Mousavi MR. Modeling and Analysis of Communicating Systems. MIT Press, 2014. ISBN: 9780262027717. URL `https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems`.

[22] Saberi AK, Groote JF, Keshishzadeh S. Analysis of Path Planning Algorithms: a Formal Verification-based Approach. In: Liò et al. [41], 2013 pp. 232–239. doi:10.7551/978-0-262-31709-2-ch035.

[23] Aceto L, Inglfsdttir A, Larsen KG, Srba J. Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, 2007. doi:10.1017/CBO9780511814105.

[24] Plotkin GD. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 2004. **60-61**:17–139. doi:10.1016/j.jlap.2004.05.001.

[25] Bloom B, Istrail S, Meyer AR. Bisimulation Can't be Traced. *J. ACM*, 1995. **42**(1):232–268. doi:10.1145/200836.200876.

[26] Baier C, Katoen J. Principles of model checking. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[27] Caltais G, Guetlein SL, Leue S. Causality for General LTL-definable Properties. `https://www.sen.uni-konstanz.de/typo3temp/secure_downloads/76523/0/d21a67cb313b43ca396aa4c0be674c37f89e7fea/Causality_Checking_for_Liveness_Properties.pdf`. To appear in EPTCS.

[28] Bergstra JA, Ponse A, Smolka SA (eds.). Handbook of Process Algebra. North-Holland / Elsevier, 2001. ISBN: 978-0-444-82830-9. doi:10.1016/b978-0-444-82830-9.x5017-6.

[29] Stevens P, Stirling C. Practical Model-Checking Using Games. In: Steffen [40], 1998 pp. 85–101. doi:10.1007/BFb0054166.

[30] Cranen S, Keiren JJA, Willemse TAC. Parity game reductions. *Acta Inf.*, 2018. **55**(5):401–444. doi:10.1007/s00236-017-0301-x.

[31] Beer A, Heidinger S, Kühne U, Leitner-Fischer F, Leue S. Symbolic Causality Checking Using Bounded Model Checking. In: Fischer and Geldenhuys [34], 2015 pp. 203–221. doi:10.1007/978-3-319-23404-5_14.

[32] Bradfield JC, Stirling C. Modal Logics and mu-Calculi: An Introduction. In: Bergstra et al. [28], pp. 293–330, 2001. doi:10.1016/b978-044482830-9/50022-9.

[33] Fischer MJ, Ladner RE. Propositional Dynamic Logic of Regular Programs. *J. Comput. Syst. Sci.*, 1979. **18**(2):194–211. doi:10.1016/0022-0000(79)90046-1.

[34] Fischer B, Geldenhuys J (eds.). Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings, volume 9232 of *Lecture Notes in Computer Science*. Springer, 2015. ISBN: 978-3-319-23403-8. doi:10.1007/978-3-319-23404-5.

[35] Bonakdarpour B, Smolka SA (eds.). Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings, volume 8734 of *Lecture Notes in Computer Science*. Springer, 2014. ISBN: 978-3-319-11163-6.

[36] Gößler G, Sokolsky O (eds.). Proceedings First Workshop on Causal Reasoning for Embedded and safety-critical Systems Technologies, CREST@ETAPS 2016, Eindhoven, The Netherlands, 8th April 2016, volume 224 of *EPTCS*. 2016. doi:10.4204/EPTCS.224.

[37] Giacobazzi R, Berdine J, Mastroeni I (eds.). Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings, volume 7737 of *Lecture Notes in Computer Science*. Springer, 2013. ISBN: 978-3-642-35872-2. doi:10.1007/978-3-642-35873-9.

[38] Lewis D. Causation. *Journal of Philosophy*, 1973. **70**:556–567. doi:10.2307/2025310.

[39] de Bakker JW, van Leeuwen J (eds.). Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings, volume 85 of *Lecture Notes in Computer Science*. Springer, 1980. ISBN: 3-540-10003-2.

[40] Steffen B (ed.). Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings, volume 1384 of *Lecture Notes in Computer Science*. Springer, 1998. ISBN: 3-540-64356-7. doi:10.1007/BFb0054159.

[41] Liò P, Miglino O, Nicosia G, Nolfi S, Pavone M (eds.). Proceedings of the Twelfth European Conference on the Synthesis and Simulation of Living Systems: Advances in Artificial Life, ECAL 2013, Sicily, Italy, September 2-6, 2013. MIT Press, 2013. ISBN: 9780262317092. URL `https://mitpress.mit.edu/books/advances-artificial-life-ecal-2013`.

[42] Cleaveland R. Tableau-Based Model Checking in the Propositional Mu-Calculus. *Acta Inf.*, 1990. **27**(8):725–747. doi:10.1007/BF00264284.

[43] Andersen H. Verification of Temporal Properties of Concurrent Systems. *DAIMI Report Series*, 1993. **22**(445). doi:10.7146/dpb.v22i445.6762.

[44] Bradfield JC, Walukiewicz I. The mu-calculus and Model Checking. In: Clarke et al. [45], pp. 871–919, 2018. doi:10.1007/978-3-319-10575-8_26.

[45] Clarke EM, Henzinger TA, Veith H, Bloem R (eds.). Handbook of Model Checking. Springer, 2018. ISBN: 978-3-319-10574-1. doi:10.1007/978-3-319-10575-8.

[46] Cranen S, Groote JF, Keiren JJA, Stappers FPM, de Vink EP, Wesselink W, Willemse TAC. An Overview of the mCRL2 Toolset and Its Recent Advances. In: Piterman and Smolka [47], 2013 pp. 199–213. doi:10.1007/978-3-642-36742-7_15.

[47] Piterman N, Smolka SA (eds.). Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, volume 7795 of *Lecture Notes in Computer Science*. Springer, 2013. ISBN: 978-3-642-36741-0. doi:10.1007/978-3-642-36742-7.

[48] Breese JS, Koller D (eds.). UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001. Morgan Kaufmann, 2001. ISBN: 1-55860-800-1.

[49] Yang Q, Wooldridge M (eds.). Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. AAAI Press, 2015. ISBN: 978-1-57735-738-4.