Finite State Machine Approach to Digital Event Reconstruction

Pavel Gladyshev and Ahmed Patel

Department of Computer Science,
University College Dublin,
Belfield, Dublin 4, Ireland
pavel@gladyshev.info , ahmed.patel@ucd.ie

Abstract

This paper presents a rigorous method for reconstructing events in digital systems. It is based on the idea, that once the system is described as a finite state machine, its state space can be explored to determine all possible scenarios of the incident. To formalize evidence, the evidential statement notation is introduced. It represents the facts conveyed by the evidence as a series of witness stories that restrict possible computations of the finite state machine. To automate event reconstruction, a generic event reconstruction algorithm is proposed. It computes the set of all possible explanations for the given evidential statement with respect to the given finite state machine.

Keywords: digital, forensics, event, reconstruction, algorithm, state machine

1 Introduction

A sound forensic analysis is expected to rely on a credible scientific theory that explains why and how expert conclusions follow from the available evidence. Digital forensic analyzes are currently lacking such a theory. To improve this situation, this paper demonstrates how finite state machines can be used to formalize event reconstruction in digital investigations. It gives mathematical definition of the event reconstruction process and proposes a generic event reconstruction algorithm based on that definition.

Consider the following idea. Many real-world digital systems, such as digital circuits, computer programs, or network protocols, can be described mathematically as finite state machines. A finite state machine can be depicted as a graph, whose nodes represent possible system states, and whose arrows represent possible transitions from state to state (see Figure 1). All possible computations leading to a particular state can be determined by back-tracing transitions leading to that state. In theory, the investigator could perform event reconstruction as follows:

1. Obtain a finite state model of the system under investigation.

2. Determine all possible scenarios of the incident by back-tracing transitions from the state in which the system was discovered.

3. Discard scenarios that disagree with the available evidence.

This vague idea is generalized and clarified in the rest of this paper. The presentation is organized into four sections. Section 2 illustrates the key concepts of the proposed

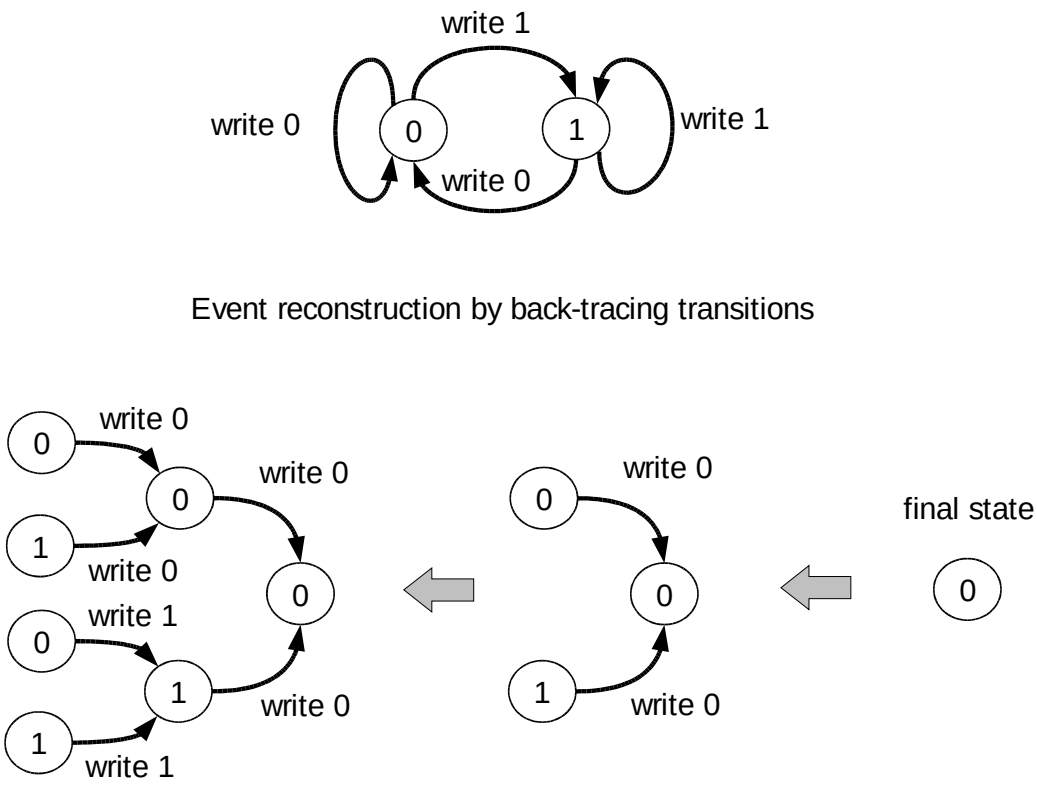

Figure 1. Event reconstruction by back-tracing transitions

reconstruction approach on a simple analysis example. Section 3 formalizes the concepts introduced in Section 2 and gives rigorous definition of the event reconstruction problem. Section 4 describes a generic event reconstruction algorithm based on that definition. Finally, Section 5 puts the work in the context of related research, discusses its possible applications and possibilities for further development.

2 Informal example of state machine analysis

This section illustrates the proposed event reconstruction approach by using it on a fictional example of networked printer analysis. First, an informal analysis is given, then it is repeated using a finite state model of the printer.

2.1 Investigation at ACME Manufacturing

The dispute. The local area network at ACME Manufacturing consists of two personal computers and a networked printer as shown in Figure 2. The cost of running the network is shared by its two users Alice (A) and Bob (B). Alice, however, claims that she never uses the printer and should not be paying for the printer consumables. Bob disagrees, he says that he saw Alice collecting printouts. The system administrator, Carl, has been assigned to investigate this dispute.

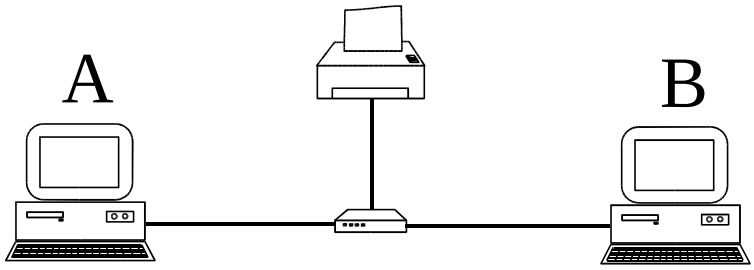

Figure 2. ACME Manufacturing LAN topology

The investigation. To get more information about how the printer works, Carl contacted the manufacturer. According to the manufacturer, the printer works as follows:

1. When a print job is received from the user it is stored in the first unallocated directory entry of the print job directory.

2. The printing mechanism scans the print job directory from the beginning and picks the first active job.

3. After the job is printed, the corresponding directory entry is marked as “deleted” , but the name of the job owner is preserved.

The manufacturer also noted that

4. The printer can accept only one print job from each user at a time.

5. Initially, all directory entries are empty.

After that, Carl examined the print job directory. It contained traces of two Bob's print jobs, and the rest of the directory was empty:

job from B (deleted)
job from B (deleted)
empty
empty
empty
...

The analysis. Carl reasons as follows. If Alice never printed anything, only one directory entry must have been used, because printer accepts only one print job from each user. However, two directory entries have been used and there are no other users except Alice and Bob. Therefore, it must be the case that both Alice and Bob submitted their print jobs at the same time. The trace of the Alice's print job was overwritten by Bob's subsequent print jobs.

In the next subsection, it is shown how the same conclusion can be mechanically derived

from the finite state model of the print job directory.

2.2 State machine analysis of the ACME investigation

Please look at Figure 3. It shows a finite state model of the print job directory. Ellipses correspond to possible states of the directory. Arrows correspond to addition (or deletion) of print jobs.

Each ellipse in Figure 3 shows the content of the print job directory in the corresponding state. For the sake of simplicity, only the first two directory entries are modeled. For example, the ellipse (A,B) represents the state in which directory contains an active job from Alice, and an active job from Bob:

job from Alice
job from Bob
empty
empty
empty
...

The initial state of the directory corresponds to the ellipse (e,e). The state discovered by Carl corresponds to the ellipse (**?**,**?**). Any possible scenario of the incident corresponds to a path from (e,e) to (**?**,**?**). All such scenarios can be found by back-tracing transitions leading into (**?**,**?**), or equivalently, by forward-tracing transitions from (e,e).

The Alice's claim that she never printed anything corresponds to a path from (e,e) to (**?**,**?**) that does not have states with "A" in them. By forward-tracing transition from (e,e), one can ensure that any path from (e,e) to (**?**,**?**) has to go through either (A,B) or (B,A) state, which means that Alice is lying.

2.3 The need for formal statement of event reconstruction problem

The example presented in this section shows that the rigor and objectivity of digital forensic analysis can be improved by using formal methods of computer science. However, formal analysis is more labor intensive than informal analysis, because additional effort is required for formalizing the system under investigation. On the other hand, the exploration of possible computations can be automated, thus, speeding-up formal analytical process.

Any tool for automation of forensic analysis will rise the question of its correctness. But before the tool's correctness can be proved, it is necessary to define precisely what it means for a tool to be correct. To answer this question, the next section proposes a formal definition of *event reconstruction problem* – a formal statement of what is the correct outcome of event reconstruction for the given evidence and digital system.

3 Formalization of event reconstruction problem

This section formally defines the event reconstruction problem. It is based on the idea that the knowledge used by forensic expert to reconstruct past events in a digital system can be divided into two categories:

- Knowledge of the system functionality – the expert knowledge

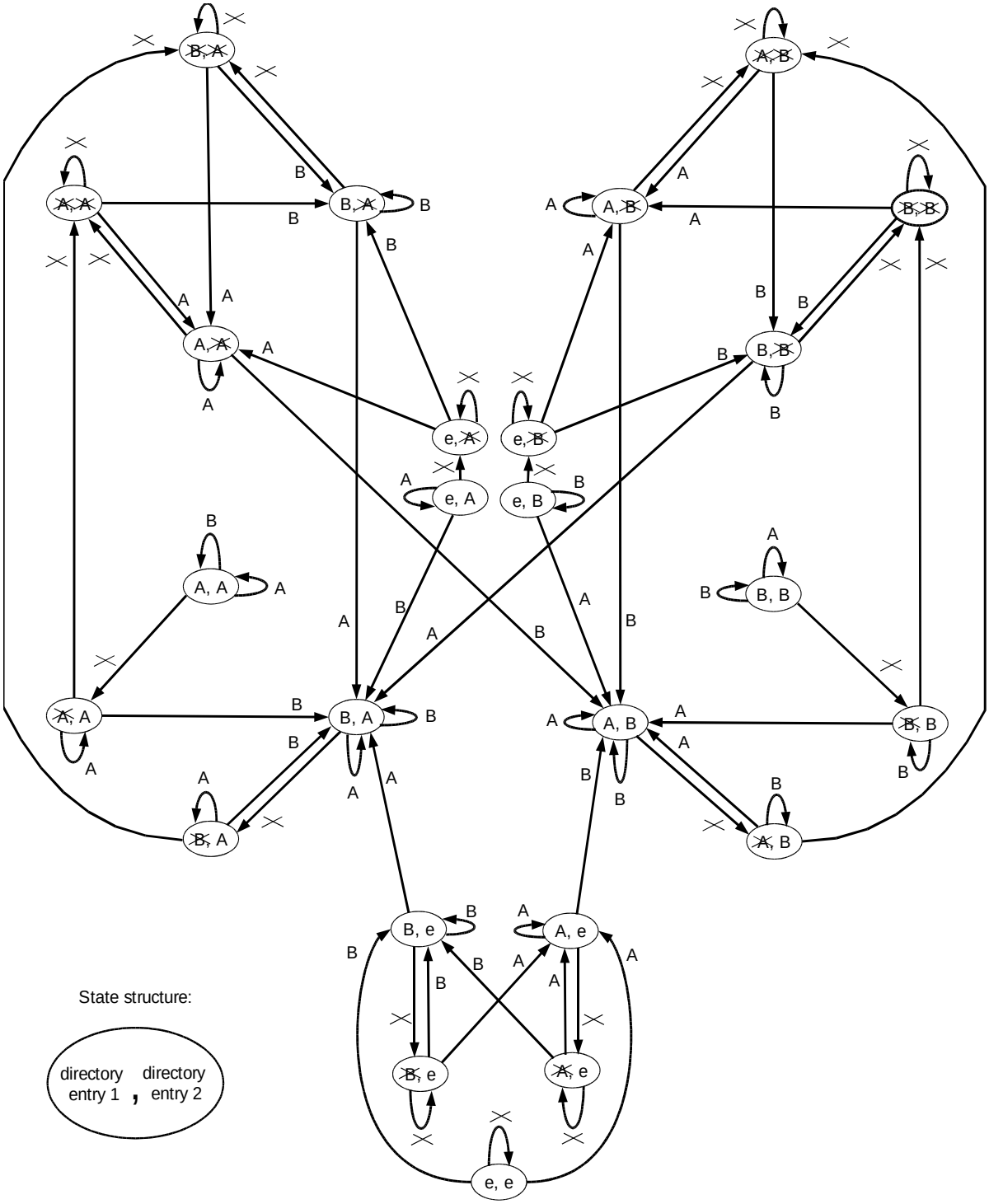

Figure 3. Transition graph of the print job directory model

- Evidence – description of the system's final state and clues to the system's behavior in the past, such as witness statements, printouts, etc.

The proposed theory formalizes the knowledge of the system functionality as a finite state machine and defines *evidential statement* notation for describing the evidence and investigative assumptions. The event reconstruction problem is then defined as finding all possible explanations for the given evidential statement with respect to the given finite state machine.

3.1 Notation

Sets are denoted by capital Roman letters	$A, B, C \ldots$
Set of integers is denoted	$\mathbb{Z}$
Empty set is denoted	$\emptyset$
Sets are defined either by explicit enumeration	$A = \{a, b, c\}$
or by a set former	$A = \{x \mid P(x)\}$
Set product of A and B is denoted	$A \times B$
The n^{th} power of set A is denoted	A^n
The powerset (the set of all subsets) of set A is denoted	2^A

Symbols $\supseteq, \supset, \subseteq, \subset, \cup, \cap, \notin, \in$ are used in their usual mathematical meaning.

Sequences are denoted by lower case letters	$a, b, c, d \ldots$
Sequences are defined by listing their elements	$s = (1, 2, 3)$
Empty sequence is denoted	ε
The length of sequence s is denoted	$\lvert s \rvert$
Elements of sequence s are denoted	s_i ,where $0 \leq i < \lvert s \rvert,\ i \in \mathbb{Z}$
Head of sequence s is its first element	s_0
Tail of sequence s is the rest of s	$(s_1, s_2, \ldots s_{\lvert s \rvert - 1})$
Concatenation of sequences s and q is denoted	$s \,.\, q$
Function δ that maps set A into set B is denoted	$\delta : A \rightarrow B$

3.2 Finite state machine

Finite state machine is a sequence of four elements $T = (Q, I, \phi, q)$, where

I is a finite set of all possible events,
Q is a finite set of all possible states,
$\phi : I \times Q \to Q$ is a transition function that determines the next state for every possible combination of state and event,
$q \in Q$ is the current system state

Transition is the process of state change. Transitions are instantaneous.

A (finite) computation is a non-empty, finite sequence of steps $c = (c_0, c_1, \cdots, c_{|c|-1})$, where each step is a pair $c_j = (c_j^\iota, c_j^q)$, where $c_j^\iota \in I$ is event, $c_j^q \in Q$ is a state, and any two steps c_k and c_{k-1} are related via transition function:

$$\text{for all } k \text{, such that } 1 \le k < |c| \text{ , } c_k^q = \phi(c_{k-1}^\iota, c_{k-1}^q)$$

The set of all finite computations of the finite state machine T is denoted C_T.

3.3 Run

To formalize transition back-tracing, the concept of run is defined. A *run* is a possibly empty sequence of finite computations, in which the next computation is obtained from the previous computation by discarding its first element. Please look at Figure 4, which graphically illustrates this concept.

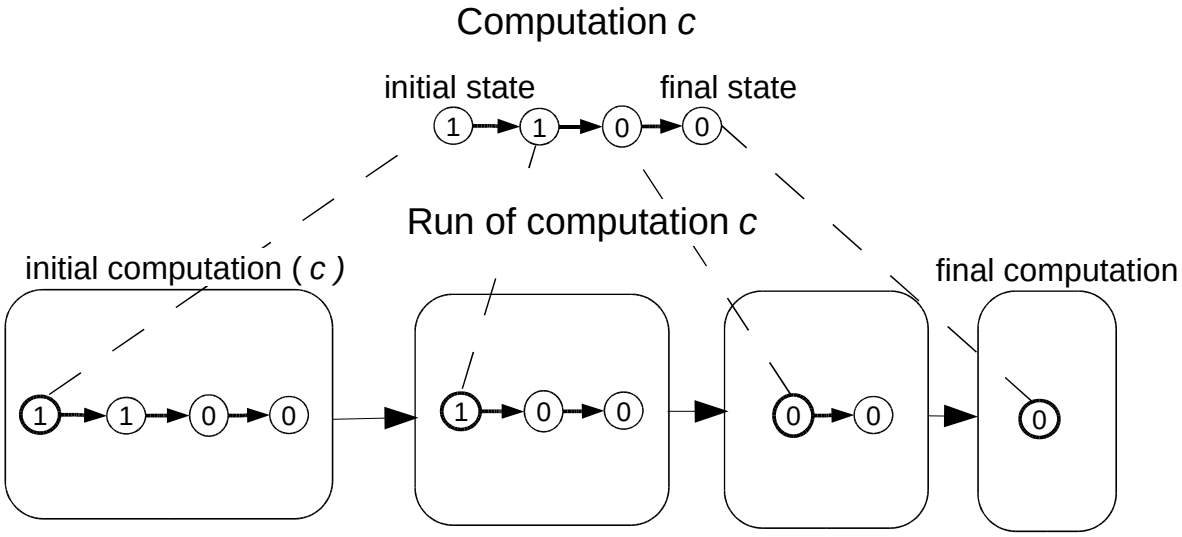

Figure 4. Run of computation

A run is a sequence of computations $r \in (C_T)^{|r|}$, such that if r is non-empty, its first element is a computation $r_0 \in C_T$, and for all $1 \le i < |r|$, $r_i = \psi(r_{i-1})$, where function ψ discards the first element of the given computation.

For two computations $x \in C_T$ and $y \in C_T$, $y = \psi(x)$ if and only if $x = x_0 \cdot y$

The set of all runs of the finite state machine T is denoted R_T.

The run of computation c is a run, whose first computation is c.

Observe that any run r is completely determined by its length and its first computation.

3.4 Partitioned run

Partitioned run is a finite sequence of runs $pr \in (R_T)^{|pr|}$, such that concatenation of its elements in the order of listing is also a run: $(pr_0 \cdot pr_1 \cdot pr_2 \cdot \cdots \cdot pr_{|pr|-1}) \in R_T$.

Set of all partitioned runs is denoted PR_T.

A *partitioning* of run $r \in R_T$ is a partitioned run denoted pr_r, such that concatenation of its elements produces r:

$$(pr_{r\,0} \cdot pr_{r\,1} \cdot pr_{r\,2} \cdot \cdots \cdot pr_{r\,|pr_r|-1}) = r$$

3.5 Formalization of back-tracing

The inverse of ψ is function $\psi^{-1}: C_T \to 2^{C_T}$. For any computation $y \in C_T$, it identifies a subset of computations, whose tails are y:

$$\text{for all } x \in \psi^{-1}(y), \quad y = \psi(x)$$

In other words, ψ^{-1} back-traces the given computation.

Although function ψ^{-1} can be used to formalize back-tracing, it is inconvenient, because it takes a single computation and produces a set of computations. As a result, it cannot be applied to its own output. A more convenient alternative is function $\Psi^{-1}: 2^{C_T} \to 2^{C_T}$, which is applied to a set of computations:

$$\text{for } Y \subseteq C_T, \quad \Psi^{-1}(Y) = \bigcup^{\forall y \in Y} \psi^{-1}(y)$$

The meaning of functions ψ, ψ^{-1}, and Ψ^{-1} is illustrated in Figure 5.

Back-tracing of computations is defined as a finite number of compositions of Ψ^{-1} with itself applied to a subset of computations.

Additional convenience of function Ψ^{-1} is that it's software implementation can manipulate *implicit* symbolic descriptions of computation sets, whereas implementation of ψ^{-1} requires explicit representation of computations.

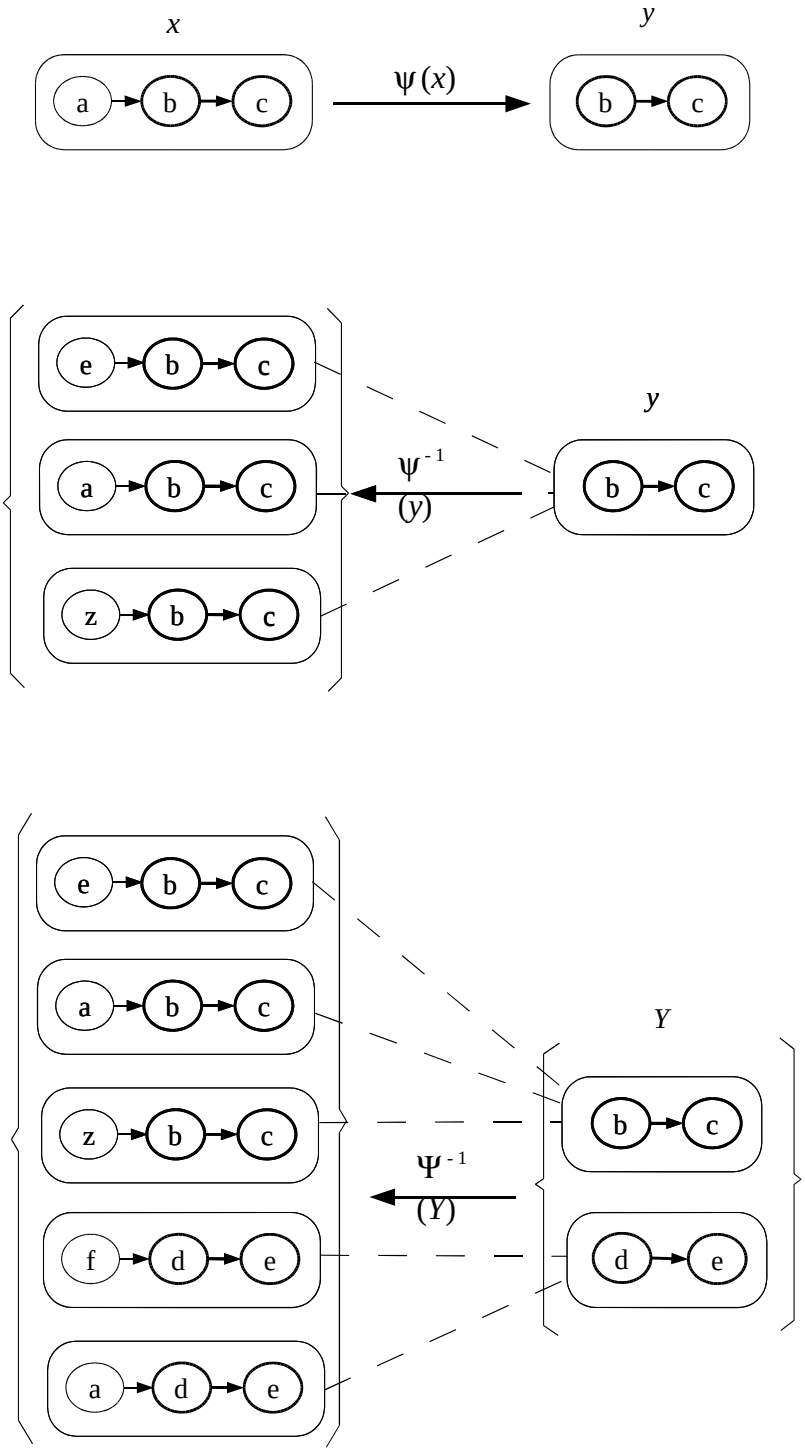

Figure 5. Functions ψ, ψ^{-1} , and Ψ^{-1}

3.6 Formalization of evidence

In a way, every piece of evidence tells its own "story" of the incident. The aim of event reconstruction can be seen as combining stories told by witnesses and by various pieces of evidence to make the description of the incident as precise as possible. This story-oriented view of event reconstruction is the basis of evidence formalization presented below.

3.6.1 Observation

Observation is a statement that system behavior exhibited some property p *continuously* for *some time.* Formally, it is defined as a triple $o = (P, min, opt)$, where P is the set of all computations of T that possess observed property, min and opt are non-negative integers that specify duration of observation.

An explanation of observation o is a run $r \in R_T$ such that every element of run r possesses observed property: for all $0 \le i < |r|$, $r_i \in P$, and the length of run r satisfies min and opt : $min \le |r| \le (min + opt)$.

The meaning of observation o is the set $R_o \subseteq R_T$ of all runs that explain o.

Observations can be divided into several types:

- *Fixed length observation* is observation of the form $(P, x, 0)$. Any run explaining it has length x.
- *Zero-observation* is observation of the form $(P, 0, 0)$. The only run explaining it is the empty sequence ε
- *No-observation* is observation $\$ = (C_T, 0, infinitum)$ that puts no restrictions on computations that could have happened during the incident. The *infinitum* is an integer constant that is greater than the length of any computation that may have happened during the incident.

3.6.2 Observation sequence

An *observation sequence* is a non-empty sequence of observations listed in chronological order:

$$os = (observation_A, observation_B, observation_C, \ldots)$$

An observation sequence represents uninterrupted eyewitness story. The next observation in the sequence begins immediately when the previous observation finishes. Gaps in the story are represented by no-observations.

An explanation of observation sequence os is a partitioned run pr such that The length of pr is equal to the length of os : $|pr| = |os|$, and each element of pr explains the corresponding observation of os : for all $0 \le i < |os|$, $pr_i \in R_{os_i}$

Note that the same run can explain the same observation sequence in a number of ways, each

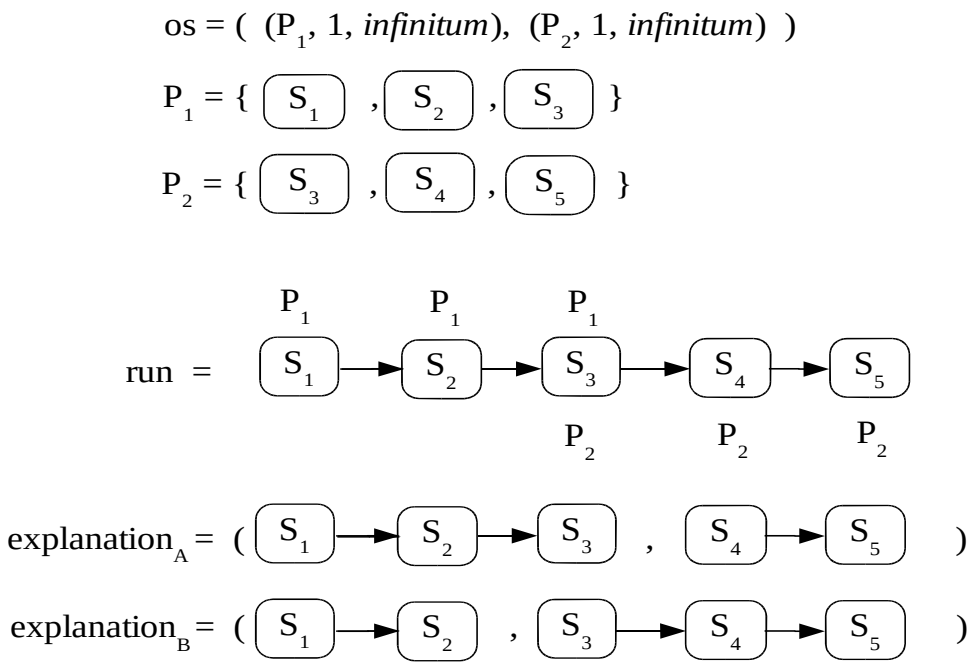

Figure 6. A run that gives two explanations of the same observation sequence

corresponding to a different partitioning of the run. Figure 6 illustrates this possibility of multiple explanations.

The meaning of observation sequence os is the set $PR_{os} \subseteq 2^{(R_T)^{|os|}}$ of *all* partitioned runs that explain os.

A run r *satisfies* an observation sequence os if and only if there exists a partitioning of r that explains os. There may be more than one partitioning of r that explains os.

A computation c *satisfies* an observation sequence os if and only if there is a run r that satisfies os and $r_0 = c$.

3.6.3 Evidential statement

Evidential statement is a non-empty sequence of observation sequences

$$es = (os_A, os_B, , os_C, \cdots)$$

Evidential statement combines restrictions imposed by all of its observation sequences – a computation satisfying one observation sequence must also satisfy all other observation sequences in the evidential statement.

An explanation of evidential statement es is a sequence of partitioned runs spr, such that all elements of spr are partitionings of the same run:

$$spr_{0,0} \cdot spr_{0,1} \cdot \cdots \cdot spr_{0,|spc_0|-1} =$$
$$= spr_{1,0} \cdot spr_{1,1} \cdot \cdots \cdot spr_{1,|spc_1|-1} =$$
$$\vdots$$
$$= spr_{|es|-1,0} \cdot spr_{|es|-1,1} \cdot \cdots \cdot spr_{|es|-1,|spc_1|-1} = r$$

and the length of spr is equal to the length of es

$$|spr| = |es|$$

and each element of spr explains corresponding observation sequence of es :

$$\text{for all } 0 \le i < |es|, \; spr_i \in PR_{es_i}$$

The meaning of evidential statement es is the set of all sequences of partitioned runs $SPR_{es} \subseteq (PR_{es_0} \times PR_{es_1} \times \cdots \times PR_{es_{|es|-1}})$ that explain es .

Evidential statement is *inconsistent* if it has empty set of explanations $SPR_{es} = \emptyset$.

Figure 7 illustrates the relationship between the evidential statement and other formal notions introduced in this section.

3.6.4 Definition of event reconstruction problem

In terms of the above defined formalization of evidence, event reconstruction problem is defined as *calculating the meaning* SPR_{es} *of the given evidential statement* es *with respect to the given finite state machine* T .

3.7 Formalization of event reconstruction: An example

To illustrate the use of formal machinery defined above, this section formalizes event reconstruction problem for the example investigation described in Section 2.

Formalization of system functionality. First, it is necessary to define a state machine that describes the functionality of the printer that was investigated by Carl at the ACME Manufacturing. Such a state machine was given in Figure 3. It's set of possible states is defined as

$$DIR = \{\text{A}, \text{B}, \text{A\_deleted}, \text{B\_deleted}, \text{empty}\}$$
$$Q = DIR \times DIR$$

Note that in Figure 3, "A_deleted" is denoted as "X", "B_deleted" is denoted as "Y", and "empty" is denoted as e.

The set of possible events is defined as $I = \{\text{add\_A}, \text{add\_B}, \text{take}\}$. Note that in Figure 3, events are shown on the arrows. Event "add_A" is denoted as "A", event "add_B" is denoted as "B", and event "take" is denoted as a "X".

Transition function $\phi : I \times Q \to Q$ is graphically defined as follows. For every event

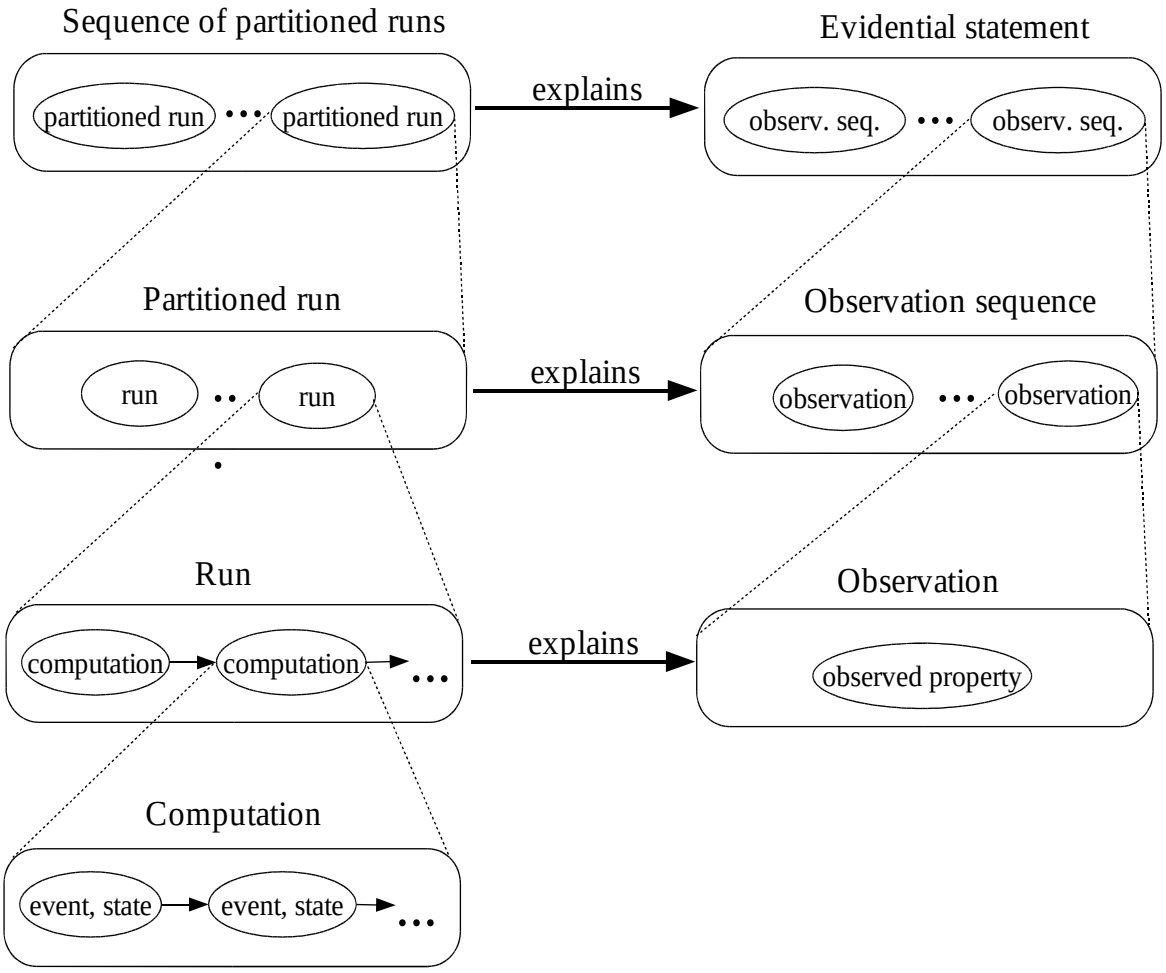

Figure 7. Evidential statement and related notions

$\iota \in I$ and state $(x, y) \in Q$, Figure 3 defines the next state $(x', y') = \phi(\iota, (x, y))$ by the arrow that leads from oval (x, y) to oval (x', y') and is labeled with ι.

Formalization of evidence. First, consider properties observed by the witnesses. The initial state of the print job directory, which was observed by the printer manufacturer, is described by the property

$$P_{empty} = \{ c \mid c \in C_T \ , \ c_0^q = (\text{empty}, \text{empty}) \}$$

which says that both directory entries at the moment of observation are empty. The final state of the printer, which was observed by Carl during printer examination, is described by the property

$$P_{\text{B\_deleted}} = \{ c \mid c \in C_T \ , \ c_0^q = (\text{B\_deleted}, \text{B\_deleted}) \}$$

which says that both directory entries at the moment of observation contain deleted print jobs from Bob.

The complete “stories” told by Carl and the printer manufacturer are captured by two observation sequences. The first observation sequence describes Carl's story:

$$os_{Carl} = ((C_T, 0, \mathit{infinitum}), (P_{\text{B\_deleted}}, 1, 0))$$

it says that Carl observed nothing about the state of the print job directory, until he examined the printer and found that both directory entries contained deleted print jobs from Bob.

The manufacturer story is that, initially, all directory entries were empty, but then the printer was sold and nothing was observed about its subsequent states:

$$os_{manufacturer} = ((P_{empty}, 1, 0), (C_T, 0, \mathit{infinitum}))$$

These observation sequences form the evidential statement

$$es_{ACME} = (os_{Carl}, os_{manufacturer})$$

The evidential statement combines the knowledge contained in the two observation sequences. The task of event reconstruction is to find all computations that satisfy both observation sequences simultaneously.

3.8 Testing investigative hypotheses by including them into evidential statement

The purpose of event reconstruction is usually to prove or disprove some claim about the incident. To show that the claim *may* be true, the investigator has to show that there are *some* explanations of evidence that agree with the claim. To disprove the claim, the investigator has to show that there are no explanations of evidence that agree with the claim.

In the investigation described in Section 2, the claim is that Alice never printed anything. To formally disprove that claim, the investigator has to show that all explanations of the evidential statement es_{ACME} involve Alice printing something at one point or another. A straightforward approach would be to compute all possible explanations for es_{ACME} and check them all manually. However, this approach is impractical when the number of explanations is large. An alternative approach is to formulate the claim as an observation sequence, include it into the evidential statement, and try to find explanations that agree with both the evidence and the claim.

For example, Alice's claim can be formalized as observation sequence, which says that Alice did not print anything until Carl examined the printer:

$$P_{Alice} = \{ c \mid c \in C_T \ , \ c_0^t \neq \text{add\_A} \}$$
$$os_{Alice} = ((P_{Alice}, 0, \mathit{infinitum}), (P_{\text{B\_deleted}}, 1, 0))$$

The extended evidential statement for the ACME investigation is then

$$es'_{ACME} = (os_{Alice}, os_{Carl}, os_{manufacturer})$$

If there are explanations of es'_{ACME} they must agree with both the evidence and the Alice's claim, which means that the claim may be true. If there are no explanations of es'_{ACME} but there are some explanations of es_{ACME} the claim must be false, because it makes evidential statement inconsistent.

4 Event reconstruction algorithm

This section describes an algorithm for computing the meaning of the given evidential statement with respect to the given state machine. The algorithm is presented in four steps. First, a procedure for computing the meaning of fixed-length observation sequences is presented. Second, a procedure for computing the meaning of generic observation sequences is presented. Third, it is shown how the meanings of individual observation sequences can be combined into the meaning of the evidential statement. Finally, a "proof-of-concept" implementation of the algorithm in Common Lisp is described.

4.1 Computing the meaning of sequences of fixed-length observations

Recall function $\Psi^{-1}: 2^{C_T} \to 2^{C_T}$ introduced in Section 3.5. It takes a set of computations $Y \in 2^{C_T}$ and produces the set of all computations, whose tails are in Y. In other words, it returns all possible back-tracings of computations in Y.

Function Ψ^{-1} provides basic operation for automation of back-tracing. Together with set intersection, it can be used to calculate the meaning of observation sequences that consist of fixed-length observations only. The idea is to take the set of all computations C_T as the starting point and iteratively back-trace it into the past using Ψ^{-1}. At each step, computations that do not possess observed property are discarded. This is achieved by intersecting the set of back-tracings with the set of computations that possess property observed at the current step. The result of intersection is then used as input for the next invocation of Ψ^{-1}, and so on. The process continues until either all observations are explained, or the set of computations becomes empty. Please look at Figure 8, which illustrates this process for observation sequence

$$example = ((A,3,0),(B,2,0))$$

If the set of computations produced at the last step of reconstruction is non-empty, its elements satisfy observation sequence *example* by construction. The set of partitioned runs $PR_{example}$ that explain *example* can be generated from these computations using function ψ and the *fixed* length of observations in *example*.

A map of partitioned runs (MPR) is a representation for a set of partitioned runs. It is a tuple $pm = (len, C)$ where C is the set of initial computations, len is a sequence of partition lengths. A single MPR represents the set of all partitioned runs whose initial computation is in C, and whose partitions have lengths $len_0, len_1, \cdots, len_{|len|-1}$.

Observe that the meaning of a fixed length observation sequence can be expressed by a single MPR.

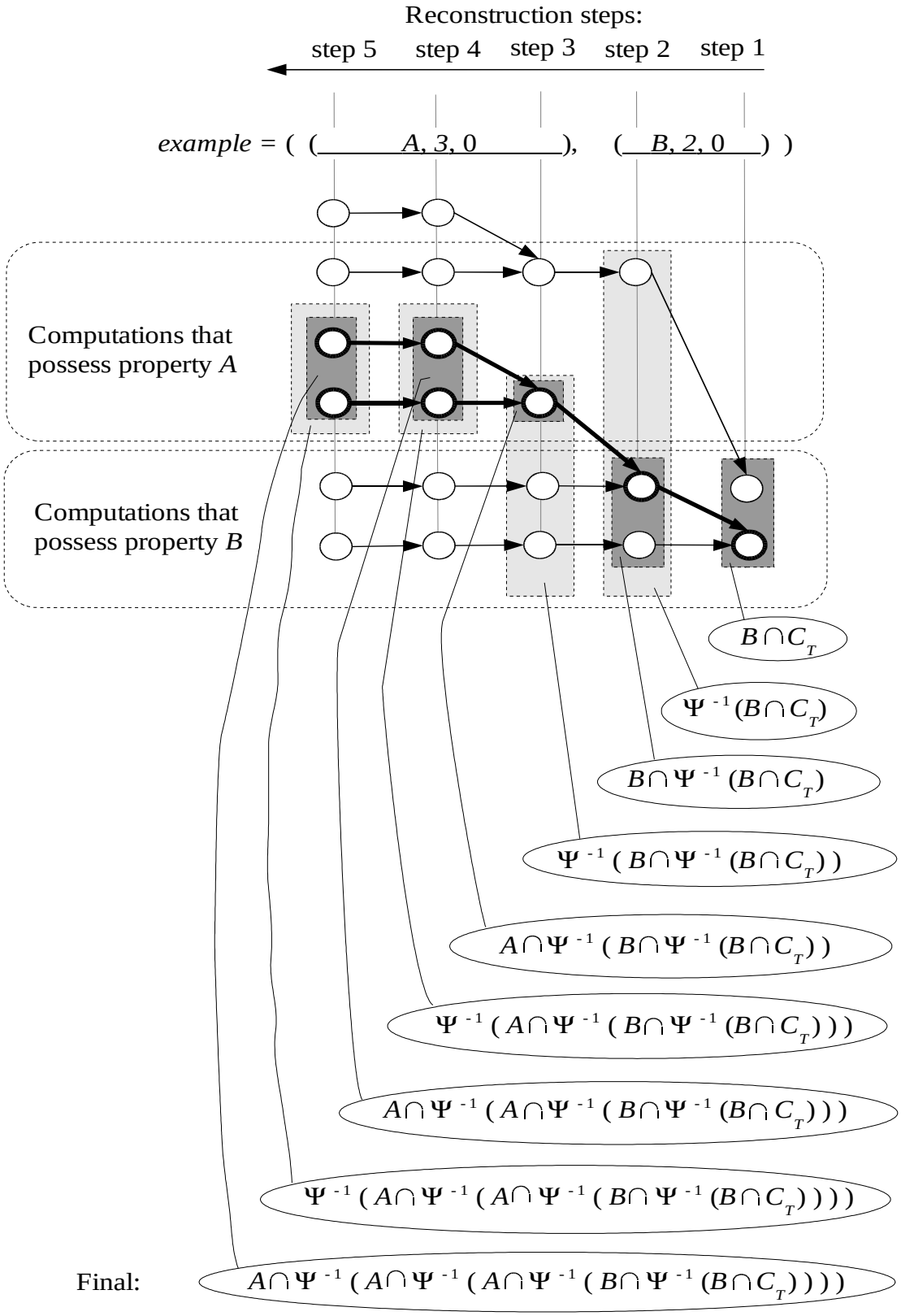

Figure 8. Finding explanations of a fixed-length observation sequence

4.2 Computing the meaning of generic observation sequences

The reconstruction process described above works, because the property observed at every step is known. This is because the length of run satisfying a fixed length observation is equal to the observation's min parameter. For a generic observation $o=(P, min, opt)$, whose $opt \neq 0$, the length of explaining run is not fixed, but is bounded between min and $min+opt$. As a result, single observation sequence represents many variants of linking observed properties to reconstruction steps. Consider, for example, observation sequence $example2 = ((A,1,3),(B,1,2))$, which says that

- initially, property A was observed for at least 1 and at most 4 step,
- then property B was observed for at least 1 and at most 3 steps.

This observation sequence represents twelve possible variants of linking properties to reconstruction steps:

AB	*ABB*	*ABBB*
AAB	*AABB*	*AABBB*
AAAB	*AAABB*	*AAABBB*
AAAAB	*AAAABB*	*AAAABBB*

Every one of these variants can be represented by a fixed-length observation sequence. Note that the meaning of *example2* is the union of explanations of each variant. Thus, the meaning of *example2* can be calculated as follows:

1. Convert *example2* to a set of fixed-length observation sequences.
2. Calculate the meaning of each fixed-length observation sequence in as described above.
3. Calculate the union of explanations of the fixed-length observation sequences.

Observe that the meaning of *example2* can be represented as a set of MPRs – each MPR representing the meaning of one of the fixed-length observation sequences.

4.3 Computing the meaning of evidential statement

The meaning of an evidential statement can be computing using a two-step procedure. First, the meanings of individual observation sequences are computed as described in the previous sections. Then the meanings of observation sequences are combined into the meaning of the entire evidential statement.

To combine the meanings of observation sequences, note that, to satisfy the evidential statement, a run must satisfy all of its observation sequences. Thus, the problem is to identify the subset of runs, whose partitionings are present in the meanings of all observation sequences.

Let $pm_a=(len_a, C_a)$ and $pm_b=(len_b, C_b)$ be two MPRs. A run r can be partitioned by both pm_a and pm_b if and only if two conditions hold:

1. the initial computation of run r belongs to initial computation sets of both MPRs: $r_0 \in C_a$ and $r_0 \in C_b$, and
2. both MPRs have equal total number of computation steps: $\Sigma len_a = \Sigma len_b$.

If $\Sigma len_a \neq \Sigma len_b$, then the two MPRs have no common runs. Otherwise, the common runs are determined by the common set of initial computations $C_a \cap C_b$.

A *map of sequence of partitioned runs (MSPR)* $mspr = ((len_0, len_1, \cdots, len_n), C)$ is a representation for a set of sequences of partitioned runs. C is the set of initial computations, and $len_0, \cdots len_n$ are lists of lengths that describe how to partition runs generated from the elements of C. MSPR is *proper* if and only if $\Sigma len_0 = \Sigma len_1 = \cdots = \Sigma len_n$.

The combination of two MPRs is defined by function $comb$ that takes two MPRs and returns a proper MSPR:

$$comb(pm_x, pm_y) = \begin{cases} \emptyset & \text{, if } \Sigma len_x \neq \Sigma len_y \text{ or } C_x \cap C_y = \emptyset \\ ((len_x, len_y), C_x \cap C_y) & \text{, otherwise} \end{cases}$$

Suppose that the meanings of two observation sequences os_a and os_b are represented by two sets of MPRs called PM_a and PM_b respectively. The meaning of evidential statement $es = (os_a, os_b)$ is expressed by the set of proper MSPRs, which is obtained by combining every MPR from PM_a with every MPR from PM_b:

$$\forall x \in PM_a, \forall y \in PM_b, SPM_{es} = \cup comb(x, y)$$

This process can be extended to arbitrary number of observation sequences, thus providing a way to calculate meaning of an arbitrary evidential statement.

Implementation note. The computation method described above has been deliberately made inefficient to clarify the concepts underlying it. If observations have large opt parameters, it will generate large number of fixed-length observation sequences, which may overflow computer memory. To address this problem, generation and reconstruction of fixed length observation sequences can be combined into a single process, which constructs fixed length observation sequence only as far as necessary to perform the next reconstruction step. It might also be possible to devise problem-specific checks that detect and abandon fruitless back-tracings early in the reconstruction process.

4.4 Implementation of the algorithm

The algorithm described above has been implemented with minor modifications as a "proof-of-concept" Common Lisp program, whose source code is given in the Appendix. The program can compute meanings of evidential statements about the print job directory model from Section 3.7. It was developed using CMU Common Lisp18c running on a Pentium PC. The following sections describe program interface and its application to the example analysis from Section 2.

4.4.1 Overview of the program interface

The program provides a set of constants, macros, and functions for defining observation sequences, computing their meaning, combining the meanings of observation sequences into meanings of evidential statements, and visualizing the meanings of evidential statements.

Observed properties are defined using two macros: `defprop1` and `defprop2`.

Macro (`defprop1` *`name1`* (*`c0`*) *`exp1`*) defines constant with name *`name1`* that represents the set of computations, whose first element *`c0`* satisfies logical expression *`exp1`*. Formally, it defines property of the form $P_{name1} = \{c \mid c \in C_T \text{ , } \exp1(c_0)\}$. For example, property $P_{\text{B\_deleted}}$ that describes the final state of the printer is defined as follows

```
(defprop1 *B_DELETED* (x)
  (and (equal (first (second x)) 'B_deleted)
       (equal (second (second x)) 'B_deleted)))
```

Macro (`defprop2` *`name2`* (`c0` *`c1`*) *`exp2`*) defines constant with name *`name2`* that represents the set of computations, whose first element *`c0`* and second element *`c1`* satisfy logical expression *`exp2`*. Formally it defines property $P_{name2}=\{c|c\in C_T, \exp2(c_0, c_1)\}$.

Observation sequences are represented by Lisp lists. Observation sequences os_{Carl} and $os_{manufacturer}$ from Section 3.7 can be defined as follows

```
(defvar *CARL* `((,*C_T* 0 ,*inf*) (,*B_DELETED* 1 0)))
(defvar *MANU* `((,*EMPTY* 1 0) (,*C_T* 0 ,*inf*)))
```

where `*C_T*` represents C_T , `*inf*` represents *infinitum* , `*B_DELETED*` represents property $P_{\text{B\_deleted}}$, and `*EMPTY*` represents property P_{empty} .

The meaning of observation sequence is computed using function `solve-os`. It takes an observation sequence as input and returns a list of MPRs that describes the meaning of the given observation sequence. For example, the meaning of os_{Carl} is computed by

```
(solve-os *CARL*)
```

The meanings of evidential statement is combined from meanings of individual observation sequences using functions `singleton-es-sol` and `add-sol`. Function `singleton-es-sol` transforms the meaning of a single observation sequence *os* into the meaning of singleton evidential statement $es = (os)$. Function `add-sol` takes the meanings of observation sequence *os* and evidential statement *es* to produce the meaning of combined evidential statement *os.es* . For example, the meaning of es_{ACME} is computed by the following code

```
(defvar *SOL-CARL* (solve-os *CARL*))
(defvar *SOL-MANU* (solve-os *MANU*))
(add-sol *SOL-MANU* (singleton-es-sol *SOL-CARL*))
```

To visualize the meaning of evidential statement, function `draw` is provided. It takes the meaning of evidential statement and creates a tree of possible scenarios[1]. An example tree is

1 The output of `draw` is a file for DOT utility [4]. The latter should be manually invoked to draw the tree.

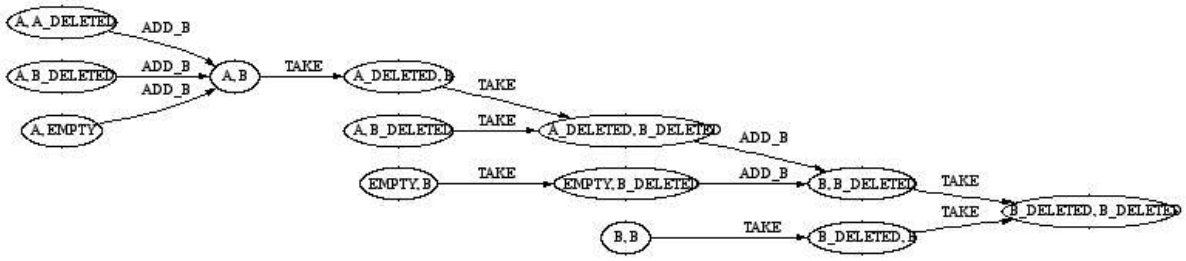

Figure 9. Sample output of the program. It papershows all computations ending in (B_deleted, B_deleted) that do not contain repetitive transitions and transitions caused by event "Add_A"

shown in Figure 9.

4.4.2 Automated analysis of the ACME investigation

The code that automates analysis of the ACME investigation is given in the Appendix on page 27. It computes the meanings of two evidential statements. One evidential statement describes only the evidence. The other evidential statement describes the evidence and the Alice's claim that she never used the networked printer. These statements correspond to the evidential statements es_{ACME} and es'_{ACME} from Section 3.8. The meanings of the two evidential statements are stored into variables `*SOL-ES-ACME*` and `*SOL-ES-PRIMED-ACME*` respectively.

When the program stops, it can be manually verified that `*SOL-ES-ACME*` contains some explanations while `*SOL-ES-PRIMED-ACME*` is empty. It means that, addition of Alice's claim into consistent evidential statement es_{ACME} produced an inconsistent evidential statement es'_{ACME}, which means that Alice must be lying.

The problem of speculative transitions. Initial attempts to automate analysis of ACME investigation has shown that the presence of loops in the transition graph dramatically decreases the performance of the program given in the Appendix. The problem is caused by speculative transitions – transitions that may or may not have happened.

An example of speculative transition is an attempt of the printing mechanism to take the next print job from the empty print job directory. Such transition does not change the state of the print job directory, because the directory is empty. However, if there is no evidence that it did or did not happen, there is no reason to believe that it never happened, or that it happened once, twice, or more times. Every such possibility corresponds to a separate explanation of the evidential statement.

The impact of the problem was reduced by exploiting the nature of the hypothesis being tested. The hypothesis is that Alice *never* printed anything. The truth or falseness of that statement is not affected by how many times sequences of transitions are repeated. Reflecting this insight, the analysis has been restricted to computations in which speculative sequences of transitions happen at most once. To achieve this, two additional observation

sequences have been added.

5 Discussion and Conclusions

Although the field of digital forensic science is rapidly maturing, few publications to date explored the use of formality for analysis and corroboration of digital evidence. The major developments include

- a semi-formal classification of uncertainties accompanying digital evidence, and a method for reasoning about there uncertainties [2];
- the view of digital forensic tools as translators of information between different layers of abstraction inherent in computer software, and a way of defining such tools [1];
- the analysis of the possibility of using formal description of file systems for extracting data from binary images of disk drives [3];
- a demonstration that it is feasible to describe the outcome of investigation using a rigorous formal notation (colored petri nets) [5].

This paper contributed to this growing body of knowledge a demonstration that event reconstruction and hypothesis testing in digital investigations can be performed with mathematical rigor and objectivity. It provided explicit formalization of the link between the evidence, the model of digital system, and the possible scenarios of the incident. Based on that formalization, it presented an algorithm for computing possible scenarios of the incident and for testing investigative hypotheses. These results provide formal basis for the development and verification of forensic analysis tools. However, more development is required for the presented ideas to benefit everyday investigations. Some of the possible developments are discussed below.

5.1 Possible applications of the proposed reconstruction approach

The most straightforward application of the proposed event reconstruction approach is the development of a general-purpose event reconstruction tool along the lines of the program given in the Appendix. When using such a tool, the human investigator would provide the formal description of the digital system, the evidence, and the investigative hypotheses. The tool would calculate and visualize possible incident scenarios consistent with the given formal description. The rigor offered by such a tool would be welcome in investigations where reliability and comprehensiveness of event reconstruction are crucial to the success of subsequent legal action.

Another possible application of the proposed event reconstruction approach is proving correctness of existing forensic analysis techniques. Many advanced digital forensic techniques can be viewed as special cases of event reconstruction. For example, the recovery of deleted files can be viewed as reconstruction of events in the file system up to the moment when the given file was deleted. Such specialized event reconstruction can be defined (with respect to the file system model) by the evidential statement

$$es_x = (a_0, a_1, \cdots, a_n, (\$, (x, 1, 0)))$$

where $(x, 1, 0)$ formalizes the knowledge of the system's final state, and observation sequences $a_0 \cdots a_n$ formalize assumptions made by the technique's designers.
To prove the technique's correctness one should prove that for all possible final states, the meaning of es_x is linked to the technique's output out_x according to some well defined interpretation relation $\overset{R}{\sim}$:

$$\text{forall } x, \; SPR_{es_x} \overset{R}{\sim} out_x$$

The interpretation relation $\overset{R}{\sim}$ can be that out_x is equal to some part of SPR_{es_x}, or that it can be derived from SPR_{es_x} by some function. To perform such proofs, a suitable body of lemmas should be developed.

5.2 Further formalization of event reconstruction

The proposed formalization of event reconstruction has captured the basic sense of event reconstruction – the reconstruction process must find all possible *sequences* of events that agree with the evidence. Although finding the sequence of events is fundamental, there are many other characteristics of events that may be interesting to the consumer of investigation. For example, the consumer might want to know the odds of a particular investigative hypothesis, or the likely real times of reconstructed events. To compute answers to such questions, the formalization of event reconstruction has to be extended with additional attributes that describe statistical and real-time properties of the system and incident. The possibility of such extensions will be researched and published in future papers.

6 References

1. Carrier, B. “Defining Digital Forensic Examination and Analysis Tool Using Abstraction Layers”, , International Journal of Digital Evidence vol. 1, no. 4, Economic Crime Institute, at Utica College, Utica, USA, 2003.
2. Casey, E. “Error, Uncertainty and Loss in Digital Evidence” , International Journal of Digital Evidence vol. 1, no. 2, Economic Crime Institute, at Utica College, Utica, USA, 2002.
3. Geber, M.B., Leeson, J.J. “Shrinking the Ocean: Formalizing I/O Methods Modern Operating Systems”, International Journal of Digital Evidence vol. 1, no. 2, Economic Crime Institute, at Utica College, Utica, USA, 2002.
4. Gansner, E., North,S.C. “An open graph visualization system and its applications to software engineering”. Software Practice and Experience, 1999.
5. Stephenson, P. “Modeling of Post-Incident Root Cause Analysis”, International Journal of Digital Evidence vol. 2, no. 2, Economic Crime Institute, at Utica College, Utica, USA, 2003.

Appendix

```
; 1. Helper functions

; append given suffix to each element
; of given list.  Return the list of results
(defun combine (lst suff)
  (if (atom lst)
    nil
    (cons (cons (car lst) suff)
  (combine (cdr lst) suff))))

; append elements of l2 to element of l1
; in all possible combinations
(defun product (l1 l2)
  (if (atom l2)
    nil
    (append (combine l1 (car l2)) (product l1
(cdr l2)))))

; convert each element of given list into
; singleton list
(defun listify-elements (l)
  (if (atom l)
    nil
    (cons (list (car l))
          (listify-elements (cdr l)))))

; Test if given object is integer zero or
; not an integer
(defun zp (x)
  (if (integerp x) (eq x 0) t))

; 2. Transition function

(defun st (c s)
  (let ((d1 (first s))
        (d2 (second s)))
    (cond ((equal c 'add_A)
           (if (or (equal d1 'A)
                   (equal d2 'A))
               s
               (if (or (equal d1 'empty)
                       (equal d1 'A_deleted)
                       (equal d1 'B_deleted))
                   (list 'A d2)
                   (if (or (equal d2 'empty)
                           (equal d2 'A_deleted)
                           (equal d2 'B_deleted))
                       (list d1 'A)
                       s))))
          ((equal c 'add_B)
           (if (or (equal d1 'B)
                   (equal d2 'B))
               s
               (if (or (equal d1 'empty)
                       (equal d1 'A_deleted)
                       (equal d1 'B_deleted))
                   (list 'B d2)
                   (if (or (equal d2 'empty)
                           (equal d2 'A_deleted)
                           (equal d2 'B_deleted))
                       (list d1 'B)
                       s))))
          ((equal c 'take)
           (if (equal d1 'A)
               (list 'A_deleted d2)
               (if (equal d1 'B)
                   (list 'B_deleted d2)
                   (if (equal d2 'A)
                       (list d1 'A_deleted)
                       (if (equal d2 'B)
                           (list d1 'B_deleted)
                           s)))))
          (t s))))


; 3. Inverse transition function

(defun rev-st (s)
  (let ((d1 (first s))
        (d2 (second s)))
    (append
      (if (equal d1 'A_deleted)
          (list (list 'take (list 'A d2)))
          nil)
      (if (equal d1 'B_deleted)
          (list (list 'take (list 'B d2)))
          nil)
      (if (and (not (equal d1 'A))
               (not (equal d1 'B))
               (equal d2 'A_deleted))
          (list (list 'take (list d1 'A)))
          nil)
      (if (and (not (equal d1 'A))
               (not (equal d1 'B))
               (equal d2 'B_deleted))
          (list (list 'take (list d1 'B)))
          nil)
      (if (and (or (equal d1 'A_deleted)
                   (equal d1 'B_deleted)
                   (equal d1 'empty))
               (or (equal d2 'A_deleted)
                   (equal d2 'B_deleted)
                   (equal d2 'empty)))
          (list (list 'take s))
          nil)
      (if (and (equal d1 'A)
               (not (equal d2 'A)))
          (list (list 'add_A (list 'empty d2))
                (list 'add_A
                      (list 'A_deleted d2))
                (list 'add_A
                      (list 'B_deleted d2)))
          nil)
      (if (and (equal d1 'B)
               (not (equal d2 'B)))
          (list (list 'add_B (list 'empty d2))
                (list 'add_B
                      (list 'A_deleted d2))
                (list 'add_B
                      (list 'B_deleted d2)))
          nil)
      (if (and (equal d1 'B)
               (equal d2 'A))
          (list (list 'add_A (list d1 'empty))
                (list 'add_A
                      (list d1 'A_deleted))
                (list 'add_A
                      (list d1 'B_deleted)))
          nil)
      (if (and (equal d1 'A)
               (equal d2 'B))
          (list (list 'add_B (list d1 'empty))
                (list 'add_B
                      (list d1 'A_deleted))
                (list 'add_B
                      (list d1 'B_deleted)))
          nil)
      (if (or (equal d1 'A) (equal d2 'A))
          (list (list 'add_A s))
          nil)
      (if (and (equal d1 'A) (equal d2 'A))
          (list (list 'add_B s))
          nil)
      (if (and (equal d1 'B) (equal d2 'B))
          (list (list 'add_A s))
          nil)
      (if (or (equal d1 'B) (equal d2 'B))
          (list (list 'add_B s))
          nil))))


; 4. Computations

; proper value of directory entry
(defun valuep (v)
  (or (equal v 'empty)
      (equal v 'A)
      (equal v 'B)
      (equal v 'B_deleted)
      (equal v 'A_deleted)))

; proper state of print job directory
(defun statep (s)
  (and (equal (cdr (cdr s)) nil)
       (valuep (first s))
       (valuep (second s))))

; proper event
(defun eventp (e)
```

```
  (or (equal e 'add_A)
      (equal e 'add_B)
      (equal e 'take)))

; proper computation
(defun cp (c)
  (if (atom c)
    t
    (if (atom (cdr c))
      (and (null (cdr c))
           (eventp (caar c))
           (statep (cadar c)))
      (and (eventp (caar c))
           (statep (cadar c))
           (equal (st (caar c) (cadar c))
                  (cadadr c))
           (cp (cdr c))))))

; Set of all single-step computations
(defvar *ALL-SINGLE-STEP-COMPS*
  (product
   '(take add_A add_B)
    (listify-elements
     (product
      '(empty A B B_deleted A_deleted)
       (listify-elements
        '(empty A B B_deleted A_deleted)))))))


; 6. Sets of computations
;
; A pattern denotes all computations that
; begin with it.  A set of computations is
; represented by a list of patterns.

; The set of all single-step patterns
(defvar *ALL-SINGLE-STEP-PATT*
  (listify-elements *ALL-SINGLE-STEP-COMPS*))


; Checks if given computation c matches
; given pattern p.
(defun matches (c p)
  (if (atom p)
    t
    (if (atom c)
      nil
      (and (equal (car c) (car p))
           (matches (cdr c) (cdr p))))))


; Checks if given computation matches given
; description
(defun in (c d)
  (if (atom d)
    nil
    (or (matches c (car d))
        (in c (cdr d)))))

; Intersection of two descriptions
(defun intpp (p1 p2)
  (if (matches p1 p2)
    (list p1)
    (if (matches p2 p1)
      (list p2)
      nil)))

(defun intpd (p d)
  (if (atom d)
    nil
    (append (intpp p (car d))
            (intpd p (cdr d)))))

(defun intdd (d1 d2)
  (if (atom d1)
    nil
    (append (intpd (car d1) d2)
            (intdd (cdr d1) d2))))

; union of two descriptions
(defun uindd (d1 d2)
  (append d1 d2))

; test for emptiness of a description
(defun emp (d)
  (if (atom d)
    t
    (and (not (cp (car d)))
         (emp (cdr d)))))


; 7. Back-tracing function (psi^-1)


(defun revcomp (c)
  (if (atom c)
    *ALL-SINGLE-STEP-PATT*
    (combine (rev-st (cadar c)) c)))


; 8. Back tracing function (PSI^-1)

(defun rev (lst)
  (if (atom lst)
    nil
    (append (revcomp (car lst))
            (rev (cdr lst)))))


; 9. Computing the meaning of a fixed-length
;    observation sequence.

; compute explanation of a single-step
; observation sequence
(defun revers (os d)
  (if (emp d)
    nil
    (if (atom os)
      d
      (revers (cdr os)
              (intdd (car os) (rev d))))))


; convert fixed-length observation
; into a list of single-step observations
(defun single-step-obs (p n)
  (if (zp n)
    nil
    (cons p (single-step-obs p (1- n)))))


; convert fixed length observation sequence
; into single-step observation sequence
(defun single-step-os (fos)
  (if (atom fos)
    nil
    (append
      (single-step-obs
        (first (car fos))
        (second (car fos)))
      (single-step-os (cdr fos)))))


; computing meaning of a fixed-length
; observation sequence
(defun solve-fix-os (fos)
  (list
    (list
      fos (revers
            (reverse (single-step-os fos))
            '(nil)))))


; 10. Computing the meaning of a generic
;     observation sequence

; compute meaning of a list of fixed-length
; observation sequences
(defun solve-fix-os-list (fix-os-list)
  (if (atom fix-os-list)
    nil
    (append
      (solve-fix-os (car fix-os-list))
      (solve-fix-os-list (cdr fix-os-list)))))

; convert generic observation into a
; set of fixed-length observations
```

```
(defun fix-obs (p min opt)
  (if (zp opt)
    (list (list p min opt))
    (cons (list p (+ min opt) 0) (fix-obs p min
(1- opt)))))

; convert a generic observation sequence
; into a list of fixed-length observation
; sequences
(defun fix-os (os)
  (if (atom os)
    (list nil)
    (product (fix-obs (first (car os)) (second
(car os)) (third (car os)))
             (fix-os (cdr os)))))

; calculate meaning of a generic
; observation sequence
(defun solve-os (os)
  (solve-fix-os-list (fix-os os)))


; 11. Tools for specifying observed properties

; Combs through given list and discards
; elements, which are not computations
(defun select-comps (l acc)
  (if (atom l)
    acc
    (if (cp (car l))
      (select-comps (cdr l) (cons (car l) acc))
      (select-comps (cdr l) acc))))

; All two-step patterns
(defvar *ALL-TWO-STEP-PATT*
  (select-comps
    (product *ALL-SINGLE-STEP-COMPS*
             *ALL-SINGLE-STEP-PATT*) nil))

; Defines observed property that restricts
; system state at the moment of observation
;
; This is achieved by
;   (1) defining a tester function
;   (2) combing tester function through all
;       single-step patterns
;
(defmacro defprop1-helper (const-name
                           tester-name
                           vars
                           body)
  `(progn
     (defun ,tester-name (comp-list acc)
       (if (atom comp-list)
         acc
         (if ((lambda ,vars ,body)
                (caar comp-list))
           (,tester-name
              (cdr comp-list)
              (cons (car comp-list) acc))
           (,tester-name (cdr comp-list) acc))))

     (defvar ,const-name
       (,tester-name
          *ALL-SINGLE-STEP-PATT* nil))))

; User interface to defprop1-helper macro.
; It automatically constructs a name for the
; tester function.
(defmacro defprop1 (name vars body)
  (let
    ((tester-name
       (intern
         (concatenate
           'string
           (symbol-name name)
           "-TESTER"))))
   `(defprop1-helper
      ,name
      ,tester-name
      ,vars
      ,body)))


; Defines observed property that restricts
; system state at the moment of observation
; AND AT THE FOLLOWING STATE
;
; This is achieved by
;   (1) defining a tester function
;   (2) combing the tester function
;       through all two-step patterns

(defmacro defprop2-helper (const-name
                           tester-name
                           vars
                           body)
  `(progn
     (defun ,tester-name (comp-list acc)
       (if (atom comp-list)
         acc
         (if ((lambda ,vars ,body)
                (caar comp-list)
                (cadar comp-list))
           (,tester-name
              (cdr comp-list)
              (cons (car comp-list) acc))
           (,tester-name
              (cdr comp-list)
              acc))))

     (defvar ,const-name
       (,tester-name
          *ALL-TWO-STEP-PATT*
          nil))))

; User interface to defprop2-helper macro.
; It automatically constructs a name for the
; tester function.
(defmacro defprop2 (name vars body)
  (let
    ((tester-name
       (intern
         (concatenate
           'string
           (symbol-name name)
           "-TESTER"))))
     `(defprop2-helper
        ,name
        ,tester-name
        ,vars
        ,body)))


; 12. Combining the meanings of two
;     observation sequences

; Calculates total length of the given
; fixed-length observation sequence
(defun fos-total-len (fos)
   (if (atom fos)
     0
     (+ (second (car fos))
        (fos-total-len (cdr fos)))))

; Takes a partitioned run map (os-chunk)
; and converts it into a singleton
; sequence of partitioned run maps
(defun singleton-es-chunk (os-chunk)
   (list
      (list (car os-chunk))
      (cadr os-chunk)))

; Takes a set of partitioned run maps
; (os-sol) and converts each element
; into a singleton sequence of
; partitioned run maps
(defun singleton-es-sol (os-sol)
   (if (atom os-sol)
     nil
     (cons (singleton-es-chunk (car os-sol))
           (singleton-es-sol (cdr os-sol)))))

; Takes given sequence of partitioned
; run maps (es-chunk)and combines it with
; given partitioned run map (os-chunk).
; Returns either nil -if combination
; is impossible, or an updated es-chunk.
(defun comb (os-chunk es-chunk)
   (if (equal
         (fos-total-len (car os-chunk))
         (fos-total-len (caar es-chunk)))
     (let
      ((intersection
```

```
          (intdd (cadr os-chunk)
                 (cadr es-chunk))))
       (if (emp intersection)
         nil
         (list
           (list
             (cons (car os-chunk)
                   (car es-chunk))
             intersection))))
     nil))
```

```
; Combines given set of sequences of
; partitioned run maps (es-sol) with
; the given partitioned run map.
(defun add-chunk-to-sol (es-sol os-chunk)
  (if (atom es-sol)
    nil
    (append
     (comb os-chunk (car es-sol))
     (add-chunk-to-sol (cdr es-sol)
                       os-chunk))))

; Combines given set of sequences of
; partitioned run maps (es-sol) with the
; given set of partitioned run maps (os-sol)
(defun add-sol (os-sol es-sol)
   (if (atom os-sol)
     nil
     (append
       (add-chunk-to-sol es-sol
                         (car os-sol))
       (add-sol (cdr os-sol) es-sol))))
```

```
; 13 ACME Investigation Analysis

; OBSERVED PROPERTIES

(defprop1 *B_DELETED* (x)
  (and
    (equal (first (second x)) 'B_deleted)
    (equal (second (second x)) 'B_deleted)))

(defprop1 *EMPTY* (x)
  (and
    (equal (first (second x)) 'EMPTY)
    (equal (second (second x)) 'EMPTY)))

(defprop1 *NO-ADD-A* (x)
    (not (equal (first x) 'ADD_A)))

(defprop2 *NO-STUTTERING* (x y)
    (not (equal (second x) (second y))))

(defprop1 *NO-ADD_B-IN-FINAL-STATE* (x)
    (if (and (equal (first (second x))
                    'B_deleted)
             (equal (second (second x))
                    'B_deleted))
      (not (equal (first x) 'ADD_B))
      t))

; Infinitum
(defvar *inf* 8)

(defvar *C_T* *ALL-SINGLE-STEP-PATT*)


; COMPUTING THE MEANING OF OBSERVATION
; SEQUENCES


; Carl's observation sequence
(defvar *A* (solve-os
   `((,*C_T* 0 ,*inf*)
     (,*DELETED* 1 0))))


; Manufacturer's observation sequence
 (defvar *E* (solve-os
  `((,*EMPTY* 1 0)
   (,*C_T* 0 ,*inf*))))


; Alice's Claim
 (defvar *B* (solve-os
  `((,*NO-ADD-A* 0 ,*inf*)
    (,*DELETED* 1 0))))

; Additional observation sequences that
; control proliferation of possible
; explanations

; Assume that every transition changed
; the state of the print job directory.
 (defvar *C* (solve-os
   `((,*NO-STUTTERING* 0 ,*inf*)
    (,*FINAL-STATE* 1 0))))

; Assume that Bob did not print anything,
; once the print job directory got into the
; final state
(defvar *D* (solve-os
   `((,*NO-ADD_B-IN-FIN* 0 ,*inf*)
    (,*DELETED* 1 0))))

(defvar *SOL-ES-ACME*
  (add-sol
    *E*
    (add-sol
      *D*
      (add-sol
        *C*
        (singleton-es-sol *A*)))))

(defvar *SOL-ES-PRIMED-ACME*
   (add-sol *B* *SOL-ES_ACME*))


; COMPUTE THE MEANINGS OF EVIDENTIAL
; STATEMENTS

; Compute the meaning of the evidential
; statements es_ACME and es'_ACME
; (with additional restrictions *D* and *E*)
```

```
; 14. Drawing reconstruction results using DOT

; construct a 'prettified' string that
; describes a list of objects
(defun pretty (l)
  (if (null l)
    ""
    (if (atom l) (symbol-name l)
      (let ((s (car l)))
        (concatenate 'string
          (if (symbolp s)
            (symbol-name s)
              (if (consp s)
                (concatenate
                  'string "(" (pretty s) ")")
                " ??? "))
          (if (not (null (cdr l)))
              " "
              "")
          (pretty (cdr l)))))))

; Constructs an underscore-separated
; identifier that represents a list of
; objects -- for use as identifier in DOT
; file
(defun stringify (l acc)
   (if (atom l)
     acc
     (stringify
       (cdr l)
       (concatenate
          'string
           acc "_" (symbol-name (car l))))))

; Flatten a tree of objects
(defun flatten (l)
  (if (atom l)
     l
    (if (consp (car l))
      (append
         (flatten (car l)) (flatten (cdr l)))
         (cons (car l) (flatten (cdr l))))))

; Draw a reversed computation
(defun draw-comp (file comp name1)
   (if (atom comp) nil
      (progn                ; draw first oval
        (format
           file
           "n~A [label=\"~A\"];~%"
           name1
           (pretty (cadar comp)))
        (if (atom (cdr comp))
          nil
          (let
            ((name2
              (concatenate
                'string
                name1
               (stringify
                 (flatten (cadr comp))
                  ""))))
            (progn
             (format        ; draw second oval
                file
                "n~A [label=\"~A\"];~%"
                name2
                (pretty (cadadr comp)))
             (format        ; draw link
                file
              "n~A -> n~A [label=\"~A\"];~%"
             name2
             name1
             (symbol-name (car (cadr comp))))
             (draw-comp   ; draw the rest
                file (cdr comp) name2)))))))


; Draw a list of computations
(defun draw-comps (file comps)
   (if (atom comps)
     nil
     (let ((rv (reverse (car comps))))
       (progn
         (draw-comp
             file
             rv
             (stringify
                (cdr (flatten (car rv))) ""))
          (draw-comps file (cdr comps))))))

; Draw computations satisfying evidential
; statement
(defun draw-sol (file es-sol)
   (if (atom es-sol)
     nil
     (progn
       (draw-comps file (cadar es-sol))
       (draw-sol file (cdr es-sol)))))

; TOP LEVEL FUNCTION

(defun draw (es-sol)
   (with-open-file (f "t.dot"
                      :direction :output
                      :if-exists :supersede)
     (format
        f
        "strict digraph G { ~% size=\"8,11\";~%
rankdir=LR;~%")
     (draw-sol f es-sol)
     (format f "}~%")
   ))

; 15. Draw the result of reconstruction

; Compute the meaning of the evidential
; statement that consists of all observation
; sequences without manufacturer's observation
; that defines the initial state of the
; print job directory

(defvar *SOL*
  (add-sol
    *D*
    (add-sol
      *C*
      (add-sol
        *B*
        (singleton-es-sol *A*)))))

; draw the result
(draw *SOL*)
```