Formalising Event Reconstruction in Digital Investigations

Pavel Gladyshev

The thesis is submitted to University College Dublin for the degree of PhD in the Faculty of Science

August 2004

Department of Computer Science

Head of department: Prof. Barry Smyth
Supervisor: Dr. Ahmed Patel

TO MY FAMILY

Contents

List of Figures

Abstract

The highly technical nature of computer crime facilitated the development of a new branch of forensic science called digital forensics. Instead of dead bodies, it collects and analyses data produced, transmitted, and stored by digital devices. The field of digital forensics is rapidly evolving. A major research challenge perceived by the digital forensic community is the need for theoretical basis validating correctness of methods and tools used by digital forensic investigators.

An important part of digital forensic analysis is event reconstruction. It is the process of determining the events that happened during the incident. In digital forensic investigations, event reconstruction is fairly complex. A single push of a button triggers a chain of events inside one or more digital devices that produce the digital evidence. Informal, unaided reasoning is not always sufficient to comprehensively analyse this chain of events.

One way to make event reconstruction more objective and rigorous is to employ mathematics. As a first step in this direction, this research aimed to give formal meaning to the problem of event reconstruction. More specifically, the objectives of this research were (1) to define a formal model of event reconstruction, and (2) to demonstrate that that model can be used as a basis for formalisation and automation of selected examples of digital forensic analysis.

To achieve these objectives, the following approach was adopted. First, a study of digital forensic techniques and legal theory was undertaken to clarify the requirements to and place of event reconstruction in digital forensic analysis. Then, a review of existing event reconstruction techniques was carried out. The review has shown that none of these technique are fully adequate in digital forensic context. Next, a formal model of event reconstruction was defined. The defined model possesses the following features:

- The system under investigation is modeled as a finite state machine.
- A special-purpose formalism called "evidential statement" is used for describing the evidence.
- The outcome of event reconstruction is given precise mathematical meaning in terms of the finite state machine model of the system.

The usefulness of the proposed model was then demonstrated by developing a generic event reconstruction algorithm, based on the defined model, and using that algorithm to formalise and automate selected examples of digital forensic analysis. Finally, several possible directions for future research have been suggested.

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at this, or any other, University or institute of tertiary education.

Pavel Gladyshev
August, 4 2004

Chapter 1

Introduction

... Detection is, or ought to be, an exact science and should be treated in the same cold and unemotional manner. ...

Arthur Conan Doyle

1.1 Motivation

The highly technical nature of computer crimes facilitated a wholly new branch of forensic science called digital forensics. Instead of dead bodies, digital forensic scientists collect and analyse data produced, transmitted, and stored by digital devices. The aim of digital forensic analysis remains the same – to clarify events of the incident and, ultimately, identify its perpetrators.

At the time of writing, the field of digital forensics is rapidly evolving. Despite having a variety of practical techniques and tools, it provides little theoretical basis to support correctness of investigation findings. The development of such a theoretical basis is seen as an important research problem. In particular, the first Digital Forensic Research Workshop (2001) stated in its report that

> "What is missing in the digital realm is any real theoretical data about the details of transformations involved in moving from reality to a digitally processed representation. ... Trained and certified forensic serologists can comment on the correctness of DNA evidence via explanations that incorporate findings from molecular biology, population genetics, and probability theory. Most analysis in Digital Forensic Science[1] cannot make similar claims." [4]

Since then, significant progress has been made in some areas, such as testing of information copying tools [60] and specification of data examination and analysis tools [22]. Little work, however, has been done on the theory of event reconstruction.

Event reconstruction is the process of determining the events that happened during the incident. It is a fundamental activity in any investigation, because unless investigator determines what happened and how it happened, there is simply no basis for determining why it happened or who may have done it.

In "ordinary" forensics the link between evidence and perpetrator's actions is often straightforward – a fingerprint on the wall indicates that someone has touched the wall, the unique shape of papillar lines can be used to identify the person in question. Common sense reasoning is usually sufficient to analyse events of the incident.

In digital forensics, however, the link between evidence and perpetrator's actions is more complex. A single push of a button triggers a chain of events inside one or more digital devices that produce the digital evidence. Informal, unaided reasoning is not always sufficient to comprehensively analyse this chain of events. According to Stephenson [76], the problem of "inconsistencies in interpreting digital evidence in complex attacks" is a specific problem to be

[1] Currently there is no universally agreed term for digital forensics. Some authors call it digital forensic science, computer forensics, or forensic computing.

solved by the digital forensic community. This research contributes to solving this problem.

1.2 Research objectives

One way to make event reconstruction more rigorous is to employ mathematics. In order to do it, the task of event reconstruction has to be formalised as a mathematical problem. Once this is done, the reconstruction can be performed completely in the formal domain and, possibly, automated.

The aim of this research is (1) to formalise event reconstruction in a general setting, that is, assuming nothing specific about the digital system under investigation or about the purpose of event reconstruction, and (2) to show that this formalisation can be used to describe and automate spelected examples of digital forensic analysis.

1.3 Research idea

To see how the research aim can be achieved, consider the following idea. Many real-world digital systems, including digital circuits, computer programs, and 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. All possible computations leading to a particular state can be determined by backtracing transitions leading to that state (see Figure 1.1). 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 backtracing transitions from the state in which the system was discovered.
3. Discard scenarios that disagree with the available evidence.

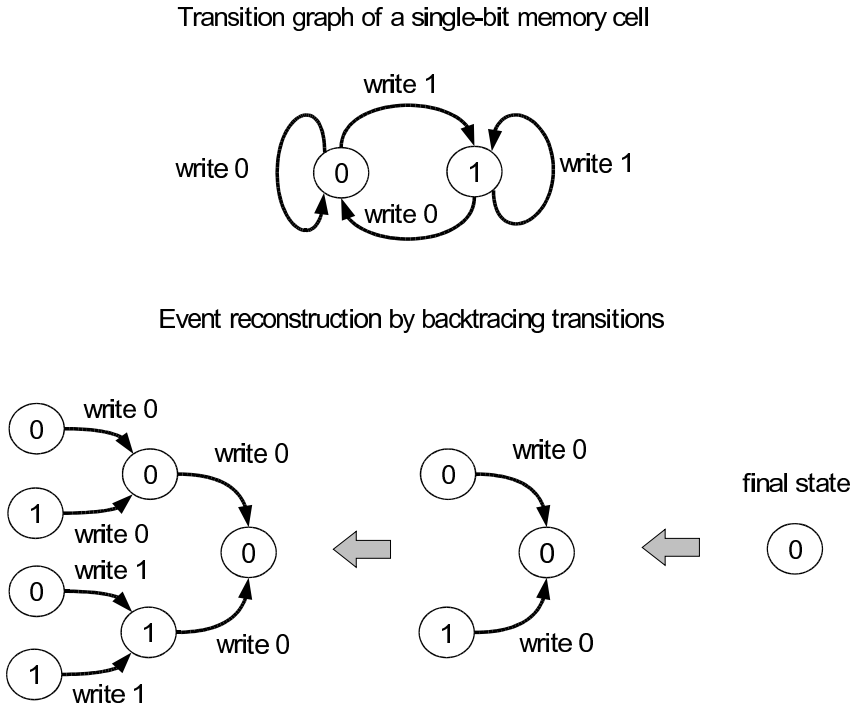


Figure 1.1: Event reconstruction by backtracing transitions

This vague approach is generalised and formalised in this dissertation. The results of this work have been published in [39], which is given in Appendix D.

1.4 Dissertation structure

The structure of the dissertation is schematically shown in Figure 1.2. First, the background of forensic event reconstruction is researched.

Chapter 2 describes relevant concepts from the legal theory. It reviews concepts of legal process, evidence, and proof, and the role of forensic expert in the legal process. It also gives definition of digital evidence and highlights difficulties associated with its use in litigation.

Chapter 3 describes the technical side of digital forensic investigation. It reviews the digital forensic investigative process, classifies its analysis techniques, and highlights the need for effectiveness and efficiency of digital forensic techniques.

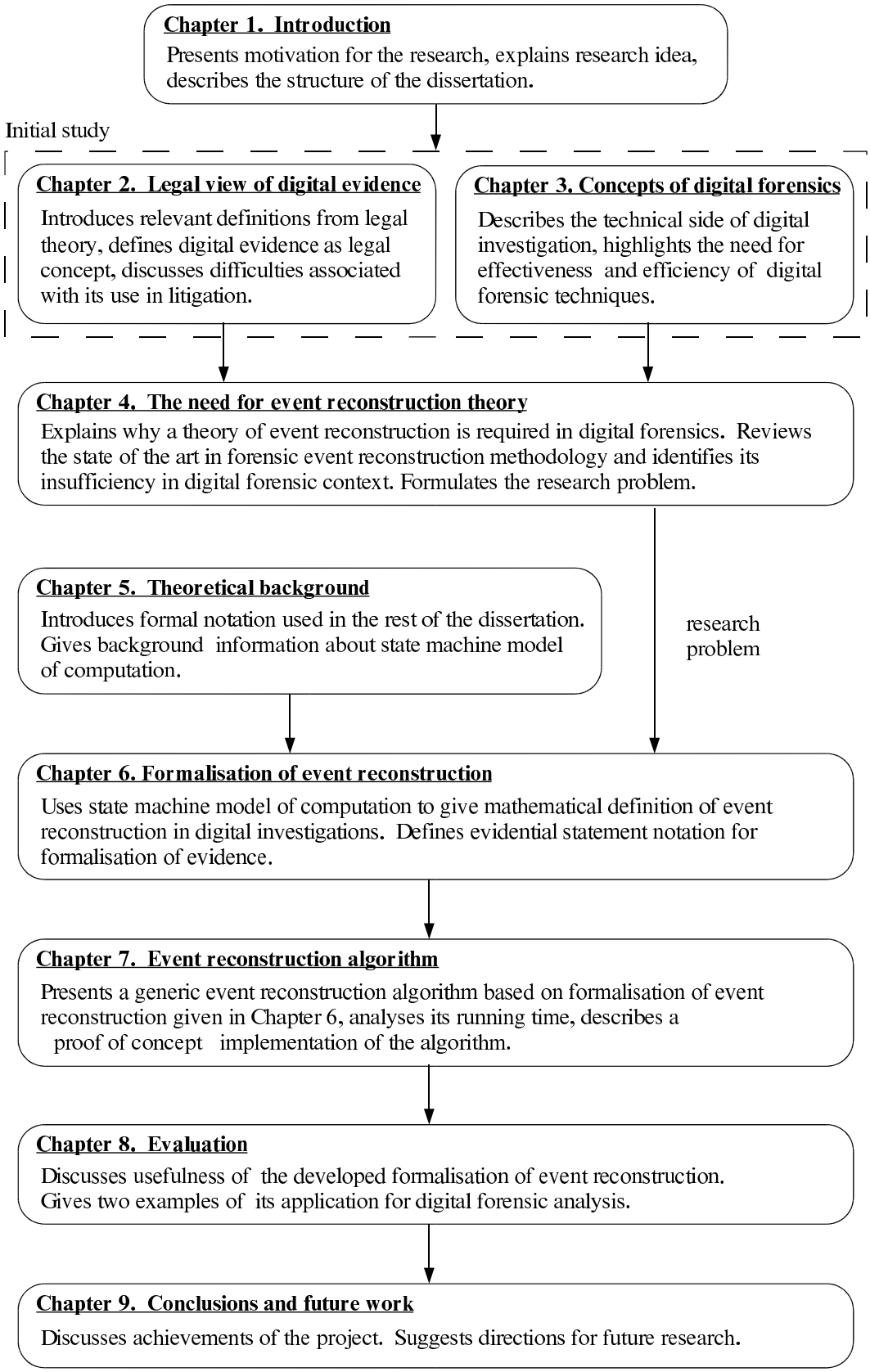


Figure 1.2: Dissertation structure

Second, once the background study is completed, the problem statement is given in Chapter 4. The chapter reviews existing semi-formal reconstruction techniques from related forensic sciences and identifies their deficiency in digital forensic context. Based on this analysis the research problem is formulated at the end of the chapter.

Third, the mathematical model of event reconstruction is developed. After the formal notation and necessary theoretical background is given in Chapter 5, Chapter 6 develops a mathematical definition of event reconstruction problem. It builds a system of formal objects that describe the system under investigation, transition backtracing, and the evidence.

The key result presented in Chapter 6 is the evidential statement notation. It describes the evidence as a system of observations about the past behavior of a finite state machine. The problem of event reconstruction is then defined as finding all possible explanations for the given evidential statement with respect to the given finite state machine.

Fourth, the usefulness of the developed formalism needs to be demonstrated. Chapters 7 and 8 serve that purpose. Chapter 7 constructs a generic event reconstruction algorithm, which is based on the developed formalisation of event reconstruction. It also analyses the running time of the algorithm, and describes a "proof of concept" implementation of the algorithm in Common Lisp. Chapter 8 then uses that algorithm and its implementation to formalise and automate selected examples of digital forensic analysis.

Finally, the conclusions for the entire work, as well as directions for future research are given in Chapter 9.

1.5 Summary of achievements

To conclude this chapter, given below is a list of the key achievements of this project.

- For the first time, a precise mathematical definition of event reconstruction in digital investigations has been given. Apart from providing basis for automation of event reconstruction, it contributes to the development of digital forensics theory as a discipline.
- A generic event reconstruction algorithm has been designed and implemented. Unlike many digital forensic tools, it has solid theoretical foundation, which can be used to defend its admissibility in legal proceedings.
- As example applications of the developed formalisation, two instances of digital forensic analysis have been formalised and automated. They demonstrate why and how formal event reconstruction can be used in practical investigations.
- It has been demonstrated on a practical example, that formal approach to event reconstruction can discover weak points and hidden assumptions in informal event reconstructions.

Chapter 2

Legal view of digital evidence

Before developing a model or a theory, it is important to understand the requirements of the domain in which the model or the theory is going to be used. The ultimate purpose of digital forensic analysis is to assist in finding and convicting perpetrators of crime. So, it is important to understand the requirements imposed on the forensic analysis by the legal process. This is the purpose of this chapter.

2.1 Legal concepts

For the benefit of readers unfamiliar with legal theory, this section introduces fundamental legal concepts explaining how disputes are resolved in courts and how evidence is used in this process.

2.1.1 The nature of disputes resolved in courts

Disputes, which are resolved in courts, involve two parties. One party, called *plaintiff* or *prosecutor*, contends that certain events in the past happened, and that under the applicable law they make the other party, called *defendant* or *accused*, obligated to perform some act — e.g. go to prison. The defendant disputes one or more of the factual contentions of the accusing party. If the

disputed events constitute violation of substantive law by the accused, the dispute is *criminal*. Otherwise, the dispute is *civil*.

Facts to be proved

To resolve a dispute, the court must first establish necessary facts and then apply the law to the facts to make a decision. Facts that need proof include:

- *facts in issue*, facts on which the disputing parties disagree;
- *circumstantial facts*, whose existence can be used to prove or disprove facts in issue;
- facts that must be proved in order for appropriate law to be applied or for evidence to be admitted into court proceedings.

2.1.2 The nature of legal proof

In court proceedings, facts are proved to the finder of fact[1] by demonstrating evidence of the fact. The proof in court differs from mathematical proof in two important ways:

1. No inference procedure is prescribed for the finder of fact by the law. In [2], the term evidence is defined as "any matter of fact, the effect, tendency, or design of which is to produce a persuasion in the mind of existence or non-existence of some other matter of fact". Thus, the finder of fact is expected to use common sense. An advantage of this arrangement is that all sorts of evidence can be considered by the court. A disadvantage is that the way evidence is *presented* has impact on the inferences made from that evidence.

[1] The finder of fact is either the jury or the judge, depending on the type of the trial.

2. Court is limited in time and resources when resolving a dispute. As a result, court can accept a highly probable, but not necessarily correct, hypothesis to be true[2]. It means that each of the disputing parties adopts a strategy aimed at discovery and interpretation of evidence to prove its own position, disprove the other party's position, or both. Neither party is interested in the discovery of the full truth.

2.1.3 Standards of proof

The degree of certainty that must be achieved by the finder of fact in order to accept the truth of a fact is termed the *standard of proof*. The two major standards are the *criminal standard* and the *civil standard*. Criminal standard is generally used in criminal proceedings, and civil standard is generally used in civil proceedings.

According to the criminal standard, the finder of fact must be persuaded "beyond reasonable doubt" to accept the truth of a fact.

According to the civil standard, the fact is considered to be true if the evidence for the fact outweighs evidence against the fact.

2.1.4 Presumptions of fact

In legal proof the conclusion about truth or falsity of a fact is almost never final. After a fact is proved, it is *presumed* true. If new evidence is discovered which clearly disproves a previously proved fact, the finder of fact must change its opinion about that fact.

For most facts nothing is presumed about them until they are proved. Some facts, however, are presumed true from the start of court proceedings. The law defines which facts are to be presumed true. For example, in criminal disputes

[2] In science, the process of determining correctness or falsity of a theory can go on indefinitely

the accused is presumed innocent until the prosecutor proves otherwise.

2.1.5 Burden of proof

When a fact is being proved, one of the disputing parties carries the *burden of proof*. That party is responsible for persuading the finder of fact into believing that the fact is true. Which party carries the burden of proof depends on the type of dispute and on the legislation applied in the case. In general, the party that proclaims existence of a fact carries the burden of proving it.

2.1.6 Characteristics of evidence

Two main characteristics of evidence are relevance and weight. The term *relevance* refers to the relationship between evidence and the fact being proved. A piece of evidence is relevant when it makes the fact in question more or less probable. If the evidence does not change probability of the fact, the evidence is irrelevant. The *weight* of evidence is the measure of how much the evidence changes the probability of the fact.

The relevance and weight of a piece of evidence are determined by the court on the basis of general knowledge.

Admissibility of evidence

In countries with common law tradition each piece of evidence must pass admissibility test before it can be used in court.

The admissibility test is specified by the law. The admissibility of a piece of evidence depends on the type of dispute and on how the evidence is related to the fact being proved. In general, a piece of evidence is inadmissible if has no relevance to the fact being proved. However, a relevant and weighty item of evidence may be excluded because it violates some formal rule.

Evidential integrity

The weight of a piece of evidence depends on how probable the evidence is if the fact is true and on how less probable it is if the fact is false. A piece of evidence that is equally likely to originate from tampering as from existence of the fact being proved, has no weight in proving the fact.

To preserve the weight of evidence, the possibility of tampering with it must be minimised. This is called preserving *evidential integrity*. Evidential integrity is preserved by handling and examining evidence in ways that do not change it. All handling and examination must be performed or witnessed by individuals to whom the finder of fact trusts to be objective and competent to do so.

Proving evidence integrity is usually a part of admissibility test. To prove that no tampering occurred, the history of each piece of evidence is recorded from the moment it is seized to the moment it is presented in court. This record is called the *chain of custody*.

2.1.7 Classes of evidence

Legal evidence can be classified in several ways. In jurisprudence, evidence is classified according to what type of fact it proves, what form it takes, and what law governs its use. The major classes of legal evidence defined in [2] are as follows.

- *Circumstantial evidence* is any evidence that proves not a fact in issue, but some other (circumstantial) fact, which can be used by the finder of fact to infer existence or non-existence of a fact in issue.

- *Direct evidence* is a first hand evidence of a fact. It is usually a testimony of a participant of the disputed events, who perceived the fact with one of the five senses.

- *Hearsay.* The rule against hearsay says that any assertion of fact other than one made by a person while giving oral evidence in the court proceedings is inadmissible as evidence of any fact asserted. Thus, any out of court statements including photographs, video tapes, and digital information produced and stored by a computer are hearsay and cannot be used as evidence. There are, however, multiple exceptions to the hearsay rule, which allow admission of hearsay coming from sufficiently reliable sources. In particular, data recorded by a machine in the normal course of operation is usually admissible as evidence of the recorded events if there was no human intervention in the recording process.

- *Documentary evidence* is admissible hearsay in form of documents, photographs, tapes, etc. presented to the court as evidence of contents.

- *Real evidence* refers to items of evidence which are presented for examination by the senses of the finder of fact (e.g. knife covered in blood).

- *Testimonial evidence* is any oral or written statement made on oath or affirmation for the purpose of legal proceedings.

- *Expert evidence* is a special form of testimonial evidence, in which an expert gives evidence of his opinion. Expert evidence is required when the matters in question are outside of the competence of the finder of fact. When expert is called to give evidence, it must be established that he or she is competent to do so.

The evidence is also classified according to the function it performs in the trial. It is customary to identify

- *Inculpatory evidence* that proves the guilt of a party,

- *Exculpatory evidence* that proves the innocence of a party,

- Evidence that proves or disproves integrity of a piece of evidence.

2.2 Forensic science and evidence

The term *forensic science* refers to "the application of scientific techniques to legal investigations" [29]. There are two main reasons for use of science in court.

1. A scientific fact may be a fact in issue. This happens, for example, when a new drug is claimed to be dangerous.

2. Science can be used at investigation stage to obtain objective, *circumstantial* evidence which is based on logic and scientific theory rather than on common sense. This application of science takes form of specialised techniques such as DNA profiling, or blood group analysis.

In either case, forensic analysis is likely to be outside of the competence of the finder of fact. Thus, forensic evidence is a special case of expert evidence.

2.2.1 Requirements to scientific evidence

When scientific evidence is given, it is possible that a qualified expert may have based his findings on a novel scientific theory that lacks sufficient experimental support to draw reliable conclusions.

The United States has a body of legislation specifically addressing this problem. The article 702 of the federal rules of evidence requires from the trial judge to establish with respect to any scientific testimony submitted to a federal court in the U.S. whether the reasoning or methodology underlying the testimony is scientifically valid.

The U.S. Supreme Court in the Daubert vs. Merrill Dow Pharmaceuticals, Inc. case [30] specified a number of non-mandatory, non-exclusive criteria for determining scientific validity of the reasoning underlying the expert testimony. In Supreme Court's opinion, the key question to answer is

- whether the theory or technique employed by the expert can be (and has been) tested.

In addition, the judge should consider

- the known or potential rate of error associated with the theory or technique, and
- the existence and maintenance of standards controlling the technique's operation.

Finally, the judge may consider

- whether the theory or technique have been subjected to peer review and publication, and
- whether the theory or technique enjoys widespread acceptance,

because "a *known* technique that has been able to attract only minimal support within the community, may properly be viewed with skepticism [30]."

In 1999, the U.S. Supreme Court ruled on Kumho Tire Co. vs. Carmichael case [52] that the Daubert criteria *may* be used by the trial judge when admitting any expert testimony.

Although not all countries with the common law tradition have analogues of Daubert criteria, the law generally requires that the expert findings must be based on a well established knowledge or theory.

2.3 Digital evidence

Digital evidence is defined by [81] as "any information of probative value that is either stored or transmitted in a digital form". It includes files stored on computer hard drive, digital video, digital audio, network packets transmitted over local area network, etc.

Depending on what facts the digital evidence is supposed to prove, it can fall into different classes of evidence.

- Digital images or software presented in court to prove the fact of *possession* are real evidence.

- E-mail messages presented as proof of their content are documentary evidence.
- Log files, file time stamps, all sorts of system information used to reconstruct sequence of events are circumstantial evidence.
- Digital documents notarised using digital signature may fall into testimony category[3].

The use of digital information in legal disputes is complicated by a number of technical problems, which reduce weight of computer based evidence or even make it irrelevant. The following subsections introduce each of the problems.

2.3.1 Anonymity of digital information

Digital information generated, stored, and transmitted between computing devices does not bear any physical imprints connecting it to the individual who caused its generation. Unless the information is a recording from external sensors capable of perceiving individualising characteristics (e.g. speech recording, video, or photographs) or was generated using some secret known to a single person (e.g. digital signature) there is nothing *intrinsic* linking digits to a person.

2.3.2 Context of digital information

Digital information is a sequence of digits encoding some knowledge. The encoding, and hence the meaning of digits is determined by the context in which the information is produced and used. Before inferences can be made, the context determining the meaning of information must be clarified.

[3] provided the law of the country permits use of digitally signed documents as a substitute to paper based documents

If the information is produced for use by the third party devices or computer programs, it must follow some documented format. The format prescribes how the information is to be interpreted.

If the information is produced for internal use by some device or computer program, there is usually no publicly available description of how to interpret it. If this is the case, the investigator must understand the internal operation of the device or program to interpret the information.

2.3.3 Automated interpretation of digital information

Manual interpretation of the digital information can be extremely labor consuming (consider manual reconstruction of a picture stored in a file) or even impossible – how would one manually interpret recorded speech? The use of automated tools for interpreting digital information is unavoidable. A precondition for use of any such tool is the assurance that the tool gives correct interpretation of the information.

2.3.4 Danger of damaged information

Like many other types of evidential material, digital information stored on magnetic and optical media can be damaged by a variety of causes. Dampness, strong magnetic fields, ultraviolet radiation, and incompetent use of storage devices and examination tools are some of the possibilities. But unlike other types of evidential material, digital information is highly sensitive to minor changes. A single bit change may cause dramatic change in its interpretation. At the same time, minor changes may be very hard to detect in a large quantity of digital information, particularly if the damaged information has valid interpretation. To minimise the impact of this problem, typical storage devices use checksumming and similar means allowing them to reasonably reliably detect accidental information damage.

2.4 Summary

This chapter reviewed major legal concepts and terms surrounding the use of digital information in litigation. The notions of legal dispute, proof, and evidence have been introduced. Classes and properties of evidence have been reviewed, as well as specific requirements to expert evidence. A definition of digital evidence has been given, and difficulties associated with its use in litigation have been discussed.

Note that from legal point of view digital evidence is not very different from other forms of evidence. Like any other form of evidence, it has to be relevant to the dispute, and it has to pass admissibility test[4]. The latter usually includes demonstration of evidence integrity (i.e. proving that the evidence has not been tampered with).

Note also that advanced analysis of digital evidence, such as event reconstruction, often requires specialist knowledge and, therefore, falls into the category of expert evidence. As expert evidence, it may have to pass Daubert criteria or similar admissibility test that verifies that its analysis methodology is scientifically valid. Thus, passing admissibility test for expert evidence, is an important requirement for event reconstruction in digital investigations.

Finally, note that admissibility of the event reconstruction *methodology* is different from admissibility of the *information* used in the event reconstruction process. Since the focus of this dissertation is on the methodology, the admissibility of the information is basically ignored. More precisely, the rest of this dissertation assumes that any information used in the event reconstruction process has already been proved admissible.

[4] in countries with common law tradition

Chapter 3

Concepts of digital forensics

Digital forensics is a branch of forensic science concerned with the use of digital information (produced, stored and transmitted by computers) as source of evidence in investigations and legal proceedings. Digital Forensic Research Workshop has defined digital forensics as

> "The use of scientifically derived and proven methods toward the preservation, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations." [4]

This chapter introduces concepts of digital forensics and digital forensic analysis techniques described in the literature.

3.1 Investigative process

Investigative process of digital forensics can be divided into several stages. According to [4] and [71], there are four major stages: preservation, collection, examination, and analysis.

- *Preservation.* Preservation stage corresponds to "freezing the crime scene". It consists in stopping or preventing any activities that can damage digital information being collected. Preservation involves operations such as preventing people from using computers during collection, stopping ongoing deletion processes, and choosing the safest way to collect information.

- *Collection.* Collection stage consists in finding and collecting digital information that may be relevant to the investigation. Since digital information is stored in computers, collection of digital information means either collection of the equipment containing the information, or recording the information on some medium. Collection may involve removal of personal computers from the crime scene, copying or printing out contents of files from a server, recording of network traffic, and so on.

- *Examination.* Examination stage consists in an "in-depth systematic search of evidence" relating to the incident being investigated. The output of examination are data objects found in the collected information. They may include log files, data files containing specific phrases, timestamps, and so on.

- *Analysis.* The aim of analysis is to "draw conclusions based on evidence found".

Other works including [25], [19], and [72] proposed similar stages summarised in Table 3.1.

The aims of preservation and collection are twofold. First they aim to provide examination and analysis with as much relevant information as possible. Second they aim to ensure integrity of the collected information.

Preservation and collection are not discussed in this dissertation, because this research is primarily concerned with the analysis stage of the investigative process. In the rest of this dissertation it is simply assumed that all neces-

Stage	Analogue in [25]	Analogue in [19]	Analogue in [72]
preservation	preservation	preservation	preservation
collection	recognition, collection, documentation	collection	identification, retrieval
examination	classification, comparison, individualisation	formulating leads, focused searches	extraction, processing
analysis	reconstruction	temporal analysis	interpretation

Figure 3.1: Stages of investigative process

sary information has been collected, and that the integrity of the collected information has been preserved.

Interested readers can obtain more information about collection and preservation stages of investigative process from the following sources. The best practice guides, such as [3], [8], and [17], specify standard procedures and check lists of things to be done by investigators during preservation and collection stages. An approach to testing correctness of tools used during collection of information is described in [19]. A more formal approach is being developed by the U.S. National Institute for Standards and Technology (NIST) in [60]. Methods for detecting tampering after collection are described in [12] and [68]. They are based on checksumming and one-way hashing of collected information.

3.2 Examination and analysis techniques

This section describes techniques and tools used at examination and analysis stages. Techniques are grouped into subsections according to the type of question they answer.

3.2.1 Search techniques

This group of techniques searches collected information to answer the question whether objects of given type, such as hacking tools, or pictures of certain kind, are present in the collected information. According to the level of search automation, this dissertation classifies techniques into *manual browsing* and automated searches. Automated searches include *keyword search*, *regular expression search*, *approximate matching search*, *custom searches*, and *search of modifications*.

Manual browsing

Manual browsing means that the forensic analyst browses collected information and singles out objects of desired type. The only tool used in manual browsing is a viewer of some sort. It takes a data object, such as file or network packet, decodes the object and presents the result in a human-comprehensible form.

Manual browsing is slow. Most investigations collect large quantities of digital information, which makes manual browsing of the entire collected information unacceptably time consuming.

Keyword search

Keyword search is automatic search of digital information for data objects containing specified key words. It is the earliest and the most widespread technique for speeding up manual browsing. The output of keyword search is the list of found data objects (or locations thereof).

Keywords are rarely sufficient to specify the desired type of data objects precisely. As a result, the output of keyword search can contain *false positives*, objects that do not belong to the desired type even though they contain specified keywords. To remove false positives, the forensic scientist has to manually browse the data objects found by the keyword search.

Another problem of keyword search is *false negatives*. They are objects

of desired type that are missed by the search. False negatives occur if the search utility cannot properly interpret the data objects being searched. It may be caused by encryption, compression, or inability of the search utility to interpret novel data format[1].

A strategy for choosing key words and phrases is described in Chapter 6 of [82]. In summary, it prescribes (1) to choose words and phrases highly specific to the objects of the desired type, such as specific names, addresses, bank account numbers, etc.; and (2) to specify all possible variations of these words.

Regular expression search

Regular expression search is an extension of keyword search. Regular expressions described in [43] provide a more flexible language for describing objects of interest than keywords. Apart from formulating keyword searches, regular expressions can be used to specify searches for Internet e-mail addresses, and files of specific type. Forensic utility EnCase [68] performs regular expression searches.

Regular expression searches suffer from false positives and false negatives just like keyword searches, because not all types of data can be adequately defined using regular expressions.

Approximate matching search

Approximate matching search is a development of regular expression search. It uses matching algorithm that permits character mismatches when searching for keyword or pattern. The user must specify the degree of mismatches allowed. Approximate matching can detect misspelled words, but mismatches also in-

[1] There is a constant lag between development of new data formats and the ability of forensic search tools to interpret them. An attempt to reduce time gap between appearance of new data formats and their incorporation into search tools was made in [36].

crease the number of false positives. One of the utilities used for approximate search is `agrep` described in [83].

Custom searches

The expressiveness of regular expressions is limited. Searches for objects satisfying more complex criteria are programmed using a general purpose programming language. For example, the *FILTER_I* tool from new Technologies Inc. uses heuristic procedure described in [7] to find full names of persons in the collected information. Most custom searches, including *FILTER_I* tool suffer from false positives and false negatives.

Search of modifications

Search of modification is automated search for data objects that have been modified since specified moment in the past.

Modification of data objects that are not usually modified, such as operating system utilities, can be detected by comparing their current hash with their expected hash. A library of expected hashes must be build prior to the search. Several tools for building libraries of expected hashes are described in the "file hashes" section of [59].

Modification of a file can also be inferred from modification of its timestamp. Although plausible in many cases, this inference is circumstantial. Investigator assumes that a file is always modified simultaneously with its timestamp, and since the timestamp is modified, he infers that the file was modified too. This is a form of event reconstruction and is further discussed in the following subsection.

3.2.2 Reconstruction of events

Search techniques are commonly used for finding incriminating information, because "currently, mere possession of a digital computer links a suspect to all

the data it contains" (see page 10 of [4]).

However, the mere fact of presence of objects does not prove that the owner of the computer is responsible for putting the objects in it. Apart from the owner, the objects can be generated automatically by the system. Or they can be planted by an intruder or virus program. Or they can be left by the previous owner of the computer. To determine who is responsible, the investigator must reconstruct events in the past that caused presence of the objects.

Reconstruction of events inside a computer requires understanding of computer functionality. Many techniques emerged for reconstructing events in specific operating systems. This dissertation classifies these techniques according to the primary object of analysis. Two major classes are identified: *log file analysis* and *file system analysis*.

Log file analysis

A log file is a purposefully generated record of past events in a computer system. It is organised as a sequence of entries. A log file entry usually consists of a timestamp, an identifier of the process that generated the entry, and some description of the reason for generating an entry.

It is common to have multiple log files on a single computer system. Different log files are usually created by the operating system for different types of events. In addition, many applications maintain their own log files.

Log file entries are generated by the system processes when something important (from the process's point of view) happens. For example, a TCP wrapper process described in [72] generates one log file entry when a TCP connection is established and another log file entry when the TCP connection is released.

The knowledge of circumstances, in which processes generate log file entries, permits forensic scientist to infer from presence or absence of log file entries that certain events happened. For example, from presence of two log file entries generated by TCP wrapper for some TCP connection X, forensic scientist can

conclude that

- TCP connection X happened
- X was established at the time of the first entry
- X was released at the time of the second entry

This reasoning suffers from implicit assumptions. It is assumed that the log file entries were generated by the TCP wrapper, which functioned according to the expectations of the forensic scientist; that the entries have not been tampered with; and that the timestamps on the entries reflect real time of the moments when the entries were generated. It is not always possible to ascertain these assumptions, which results in several possible explanations for appearance of the log file entries. For example, if possibility of tampering cannot be excluded, then forgery of the log file entries could be a possible explanation for their existence. The problem of uncertainty in digital evidence is discussed at length in [24]. To combat uncertainty caused by multiple explanations, forensic analyst seeks *corroborating evidence*, which can reduce number of possible explanations or give stronger support to one explanation than another.

Determining temporal order with timestamps. Timestamps on log file entries are commonly used to determine temporal order of entries from different log files. The process is complicated by two time related problems, even if the possibility of tampering is excluded.

First problem may arise if the log file entries are recorded on different computers with different system clocks. Apart from individual clock imprecision, there may be an *unknown* skew between clocks used to produce each of the timestamps. If the skew is unknown, it is possible that the entry with the smaller timestamp could have been generated after the entry with the bigger timestamp.

Second problem may arise if resolution of the clocks is too *coarse*. As a

result, the entries may have identical timestamps, in which case it is also not possible to determine whether one entry was generated before the other.

File system analysis

Log files is not the only source of evidence that can be used for event reconstruction. Other data objects can also be used. This subsection describes how structural information stored by the file system can be used for event reconstruction.

In most operating systems, a data storage device is represented at the lowest logical level by a sequence of equally sized storage blocks that can be read and written independently. Most file systems divide all blocks into two groups. One group is used for storing user data, and the other group is used for storing structural information.

Structural information includes structure of directory tree, file names, locations of data blocks allocated for individual files, locations of unallocated blocks, etc. Operating system manipulates structural information in a certain well-defined way that can be exploited for event reconstruction.

Detection of deleted files. Information about individual files is stored in standardised *file entries* whose organisation differs from file system to file system. In Unix file systems, the information about a file is stored in a combination of i-node and directory entries pointing to that i-node. In Windows NT file system (NTFS), information about a file is stored in an entry of the Master File Table.

When a disk or a disk partition is first formatted, all such file entries are set to initial "unallocated" value. When a file entry is allocated for a file, it becomes *active*. Its fields are filled with proper information about the file. In most file systems, however, the file entry is not restored to the "unallocated" value when the file is deleted. As a result, presence of a file entry whose value is different from the initial "unallocated" value, indicates that that file entry

once represented a file, which was subsequently deleted.

File attribute analysis. Every file in a file system — either active or deleted — has a set of attributes such as name, access permissions, timestamps and location of disc blocks allocated to the file. File attributes change when applications manipulate files via operating system calls.

File attributes can be analysed in much the same way as log file entries. Timestamps are a particularly important source of information for event reconstruction. In most file systems a file has at least one timestamp. In NTFS, for example, every *active* (i.e. non-deleted) file has three timestamps, which are collectively known as MAC-times.

- Time of last Modification (M)
- Time of last Access (A)
- Time of Creation (C)

Imagine that there is a log file that records every file operation in the computer. In this imaginary log file, each of the MAC-times would correspond to the last entry for the corresponding operation (modification, access, or creation) on the file entry in which the the timestamp is located. To visualise this similarity between MAC-times and the log file, the mactimes tool from the coroner's toolkit [34] sorts individual MAC-times of files — both active and deleted — and presents them in a list, which resembles a log file.

Signatures of different activities can be identified in MAC-times like in ordinary log files. Given below are several such signatures, which have been published.

Restoration of a directory from a backup. According to [10], the fact that a directory was restored from a backup can be detected by inequality of timestamps on the directory itself and on its sub-directory '.' or '..'. When the directory is first created, both the directory timestamp and the timestamp

on its sub-directories '.' and '..' are equal. When the directory is restored from a backup, the directory itself is assigned the old timestamp, but its sub-directories '.' and '..' are timestamped with the time of backup restoration.

Exploit compilation, running, and deletion. In [79], the signature of compiling, running, and deleting an exploit program is explored. It is concluded that "when someone compiles, runs, and deletes an exploit program, we expect to find traces of the deleted program source file, of the deleted executable file, as well as traces of compiler temporary files."

Moving a file. When a file is being moved in Microsoft FAT file systems, the old file entry is deleted, and a new file entry is used in the new location. According to [74], the new file entry maintains same block allocation information as the old entry. Thus, the discovery of a deleted file entry, whose allocation information is identical to some active file, supports possibility that the file was moved.

Reconstruction of deleted files. In most file systems file deletion does not erase the information stored in the file. Instead, the file entry and the data blocks used by the file are marked as unallocated, so that they can be reused later for another file. Thus, unless the data blocks and the deleted file entry have been re-allocated to another file, the deleted file can usually be recovered by restoring its file entry and data blocks to active status.

Even if the file entry and some of the data blocks have been re-allocated, it may still be possible to reconstruct parts of the file. The lazarus tool described in [35], for example, uses several heuristics to find and piece together blocks that (could have) once belonged to a file. Lazarus uses the following heuristics about file systems and common file formats.

- In most file systems, a file begins at the beginning of a disk block;
- Most file systems write file into contiguous blocks, if possible;

- Most file formats have a distinguishing pattern of bytes near the beginning of the file;
- For most file formats, same type of data is stored in all blocks of a file.

Lazarus analyses disc blocks sequentially. For each block, lazarus tries to determine (1) the type of data stored in the block – by calculating heuristic characteristics of the data in the block; and (2) whether the block is a first block in a file – using well known file signatures. Once the block is determined as a "first block", all subsequent blocks with the same type of information are appended to it until new "first block" is found.

This process can be viewed as a very crude and approximate reconstruction based on some knowledge of the file system and application programs. Each reconstructed file can be seen as a statement that that file was once created by an application program, which was able to write such a file.

Since lazarus makes very bold assumptions about the file system, its reconstruction is highly unreliable, which is acknowledged by [35]. Despite that fact, [35] states that lazarus works well for small files that fit entirely in one disk block. The effectiveness of tools such as lazarus can probably be improved by using more sophisticated techniques for determining the type of information contained in a disk block. One such technique that employs support vector machines has been recently described in [31].

3.2.3 Time analysis

Timestamps are readily available source of time, but they are easy to forge. Several attempts have been made to determine time of event using sources other than timestamps. Currently, two such methods have been published. They are *time bounding* and *dynamic time analysis*.

Time bounding

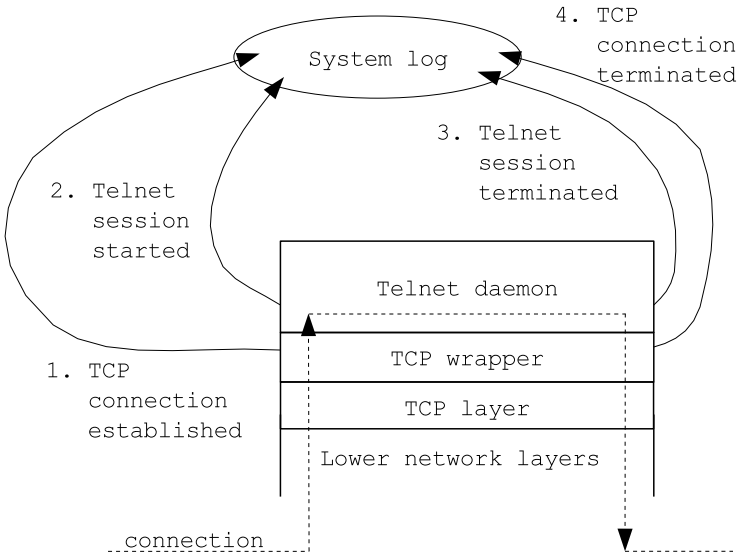
It was shown in Section 3.2.2 that timestamps can be used for determining temporal order of events. The inverse of this process is also possible – if the temporal order of events is known *a priori*, then it can be used to estimate time of events. Suppose that three events A, B, and C happened. Suppose also that it is known that event A happened before event B, and that event B happened before event C. The time of event B must, therefore, be bounded by the times of events A and C.

An example of time bounding is given in [72]. It considers a telnet session, which is carried over a TCP connection. The times of establishing and releasing the TCP connection are recorded by the TCP wrapper. Since (1) the TCP connection has to be established before the telnet session can be started, and (2) the telnet session must terminate before or together with the termination of the TCP connection, the time of the telnet connection is bounded by the times recorded by the TCP wrapper. This reasoning is illustrated in Figure 3.2.

Dynamic time analysis

A different approach to using external sources of time is described in [80]. It exploits the ability of web servers to insert timestamps into web pages, which they transmit to the client computers. As a result of this insertion, a web page stored in a web browser's disk cache has two timestamps. The first timestamp is the creation time of the file, which contains the web page. The second timestamp is the timestamp inserted by the web server.

According to [80], the offset between the two timestamps of the web page reflects the deviation of the local clock from the real time. It is proposed to use that offset to calculate the real time of other timestamps on the local machine. To improve precision, it is proposed to use the average offset calculated for a number of web pages downloaded from different web servers.



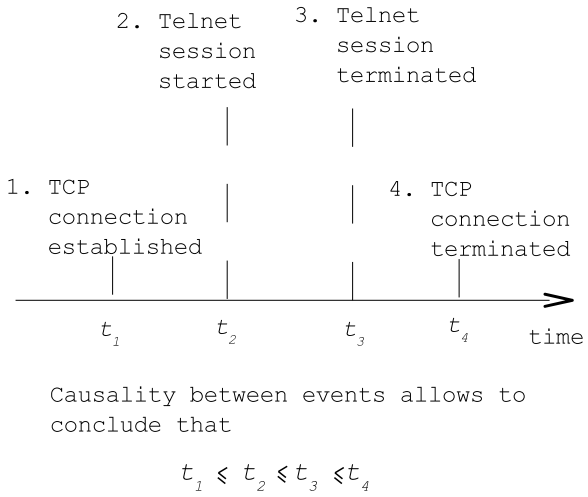


Figure 3.2: An example of time bounding

This analysis assumes that (1) timestamps are not tampered with, and that (2) the offset between system clock and real time is constant at all times (or at least that it does not deviate dramatically).

3.3 Summary

This chapter presented a review of digital forensic concepts. It identified the stages of digital investigative process and described the major types of digital forensic techniques used for examination and analysis of digital evidence.

As a final point, note that the need for effective and efficient digital forensic analysis has been a major driving force in the development of digital forensics. Manual browsing was initially the only way to do digital forensics. It was later augmented with various search utilities and, more recently, with tools such as mactimes and lazarus that support more in-depth analysis of digital evidence. Due to the limited time and manpower available to a forensic investigation, there is a constant demand for tools and techniques that increase the accuracy of digital forensic analysis and minimise the time required for it.

The next chapter looks in more detail at the problem of event reconstruction in digital investigations. It explains why a theory of event reconstruction is required in digital forensics, analyses the state of the relevant art, and, ultimately, formulates the research problem addressed in this dissertation.

Chapter 4

The need for a theory of event reconstruction

This chapter begins with an explanation why a theory of event reconstruction is needed in digital forensics. Such a theory is needed to reduce reasoning errors, to support automation, and to fulfil legal requirements. This is the subject of Section 4.1.

Currently there is no such theory, but there are semi-formal techniques that structure event reconstruction in "ordinary" investigations. A review of key points of these techniques is given in Section 4.2. It is shown that the reviewed techniques do not fulfil the needs of digital forensics, because they do not facilitate automation, and their event reconstruction process remains informal and incomplete.

It is then argued in Section 4.3.2 that digital forensic investigations are more suitable for formalisation than "ordinary" investigations, and that complete formalisation of certain reconstruction techniques in digital forensics is both possible and desirable.

The chapter concludes with a problem statement, which defines objectives for the rest of the dissertation.

4.1 Why digital forensics need a theory of event reconstruction

As shown in Section 3.2.2, reconstruction of events in computer systems is an important task in digital forensic analysis. The development of a theory supporting event reconstruction techniques is needed in the digital forensic analysis for a number of reasons.

1. To *improve efficiency of analysis.* Formalisation of reconstruction techniques would facilitate their automation, which may reduce time required to perform them.

2. To *improve effectiveness of analysis.* Informal reasoning employed by existing reconstruction techniques increases the possibility of erroneous conclusions. The development of a formal reconstruction procedure based on an established theory of computer science would reduce the possibility of reasoning error, thus increasing effectiveness of the analysis.

3. To *satisfy admissibility requirements.* The presence of a sound theory that explains working of reconstruction techniques is warranted by the legal requirements for admissibility of expert evidence outlined in Section 2.2.1.

4.2 State of the art

Although the field of digital forensics is rapidly evolving, few publications to date explored the theoretical side of analysis and corroboration of digital evidence. The major developments include

- a classification of uncertainties accompanying digital evidence, and a method for reasoning about these uncertainties [24];

- 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 by specifying their translation function and error rate [22];
- the analysis of the possibility of using formal description of file systems for extracting data from binary images of disk drives [61];
- a demonstration that it is feasible to describe the outcome of investigation using a rigorous formal notation – colored petri nets [75].

With the exception of [75], none of these works addressed the problem of event reconstruction directly. In [75] it was argued that colored petri nets provide a convenient way to *illustrate* the results of otherwise informal event reconstruction, but no theory of event reconstruction process has been proposed.

Although currently there is no theory of event reconstruction in digital forensics, several attempts to bring structure and formality to the event reconstruction have been made in criminalistics and other branches of forensic science. They are discussed in the following subsections.

4.2.1 Attack trees

An important part of reconstruction process is identification of possible incident scenarios. Attack trees described in [73] is a semi-formal approach to discovery, documentation, and analysis of possible incident scenarios.

An attack tree is a diagram that describes different scenarios achieving some goal. The goal is represented by the root node. Other nodes represent subgoals that must be achieved – either alone, or in conjunction with other subgoals – to achieve the goal.

Figure 4.1 shows an attack tree of opening a safe without authorisation (this example is taken from [73]). The goal of the tree is to open a safe. The goal can be achieved in a number of different ways, such as picking the lock,

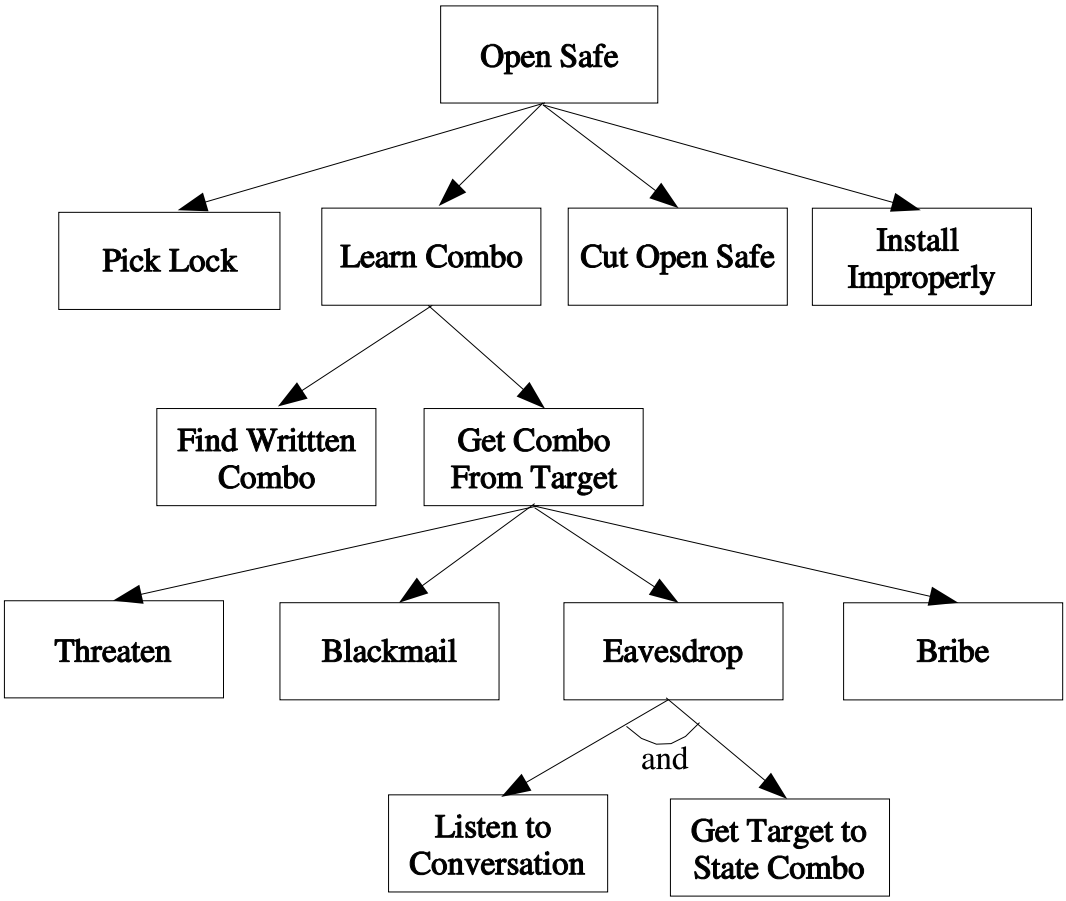


Figure 4.1: Attack tree describing different ways to open a safe

cutting through the safe wall, learning the combo, or making a security hole when the safe is installed. Different ways of learning a combo are elaborated under the "learn combo" node.

Basic attack tree is built from two types of node: AND nodes, and OR nodes. In this dissertation, AND node is graphically distinguished from an OR node by an arc that joins arrows coming out of the AND node. Both AND and OR nodes describe sub-goals that need to be fulfilled to fulfil the main goal of attack. To fulfil an OR node, any one of its child nodes must be fulfilled. To fulfil an AND node, all of its child nodes must be fulfilled. A possible scenario corresponds to every group of leaf nodes, whose joint fulfilment results in the fulfilment of the tree's root node.

To build an attack tree for a given goal, the analyst repeatedly splits the tree's goal into subgoals until the required level of detail is achieved. The process of splitting the goal into subgoals is informal. The analyst must use his or her intuition and common sense.

Once the attack tree is built, the analyst can calculate various properties of the discovered scenarios. The simplest property to calculate is whether a scenario is possible. Finding possible scenarios is a three-step process. In the first step, the analyst assigns a value "possible" or "impossible" to each leaf node. To decide whether a particular leaf node is possible or impossible, the analyst uses available evidence. In the second step, the analyst decides for every other node whether it is possible or not using the following rules

- an OR node is possible if *at least one* of its child nodes is possible;
- an AND node is possible if *all* of its child nodes are possible.

In the third step, all possible scenarios are identified by tracing them from root to leafs along chains of "possible" nodes.

Attack trees permit other types of scenario comparison. In particular, they can be used to calculate probability and cost of different scenarios.

4.2.2 Visual investigative analysis

Another semi-formal technique used for event reconstruction is Visual Investigative Analysis (VIA). It is a charting technique that

> uses a network approach to display graphically the sequences of occurrences and the relationships of all the elements of a criminal incident [65].

VIA emerged from the need to visualise complex crimes, in which many criminal activities go in parallel and interact with each other. Examples of such crimes are organised crime business manipulations and planned bankruptcies.

Graphical notation

VIA chart is a directed acyclic graph whose arrows represent activities and whose nodes represent states of the world in which activities start and finish. The following graphical notation is used to draw VIA charts.

- *Solid line arrows.* A solid line arrow represents single activity. The description of the activity is placed above the arrow. Activity always start at one node and ends at another. To improve clarity of the chart, only one solid line arrow is permitted between any two nodes.

- *Circles and triangles.* Circles and triangles are nodes. A node represents the world state in which one or more activities start or finish. *Terminal* is the last node in a sequence of activities. There is usually only one terminal on a chart. Terminals are drawn as triangles. All other nodes are drawn as circles.

- *Dotted lines and arrows.* A dotted line or arrow denotes a *dummy* activity. Dummy activity consumes no time and does nothing. It simply says that two nodes denote the same world state. Dotted lines are used for drawing parallel activities whose starting and ending world states are the same.

All nodes in VIA chart are given reference numbers. An activity is referred to by its starting and ending node numbers.

An example VIA chart is given in Figure 4.2. It shows a bank robbery at a very coarse level of detail.

Creation of VIA chart

Creation of VIA chart is a highly informal process. It consists of three interleaving stages: identification of activities, ordering of activities, and additional investigation.

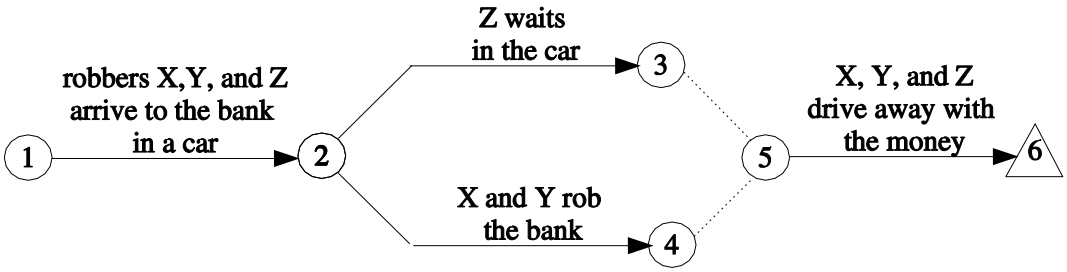


Figure 4.2: Example VIA chart

First stage is identification of activities. It consists in studying investigation reports and making a list of activities to be portrayed on a VIA chart.

Second stage is ordering of activities. The ordering is determined by answering three questions about each activity:

1. what activities precede this one;
2. what activities are concurrent with this one;
3. what activities follow this one.

During the second stage it may become clear that additional investigation is necessary to answer some of the questions. Such investigation is the task of the third stage. Several iterations of activity identification, activity ordering, and additional investigations may be required before VIA chart is complete. The ability to suggest additional investigations is a major advantage of the visual investigative analysis over unstructured investigations.

Visual investigative analysis does not provide a way to portray possible scenarios. Instead, the process of charting is interleaved with additional investigation until all possible scenarios except one are eliminated.

4.2.3 Multilinear events sequencing

Multilinear event sequencing (MES) is a set of semi-formal techniques for conducting investigations [13]. It is based on the same ideas as visual investigative

analysis and attack trees.

Multilinear event sequencing diagrams

The heart of MES are Multilinear event sequence diagrams (MES-diagrams). A MES-diagram portrays the crime or accident being investigated as a sequence of causally connected *events*, which represent activities.

MES-diagram shows events as rectangular *event blocks*. Each event block contains the following information about its event

- description of the action;
- actor – the object or person that performed the action;
- time of event.

Event blocks are connected via arrows that represent causation. If event X was necessary for event Y to occur, then MES-diagram contains an arrow from X to Y.

MES-diagrams have two axes. The horisontal axis represent time. The vertical axis lists actors involved in the accident. Event blocks are placed on the diagram according to their time and actor.

Any relevant information that does not fit event block structure can be placed on MES-diagram as a *condition*. Condition is an oval with some text inside. A condition is linked to an event block via an arrow.

Figure 4.3 shows MES-diagram of the bank robbery example from the previous section.

Creation of MES-diagrams

The process of creating MES-diagrams is similar to the process of creating VIA charts. First, events are identified. Second, causal connections between events are determined. Two techniques are defined to make creation of MES-diagrams more formal:

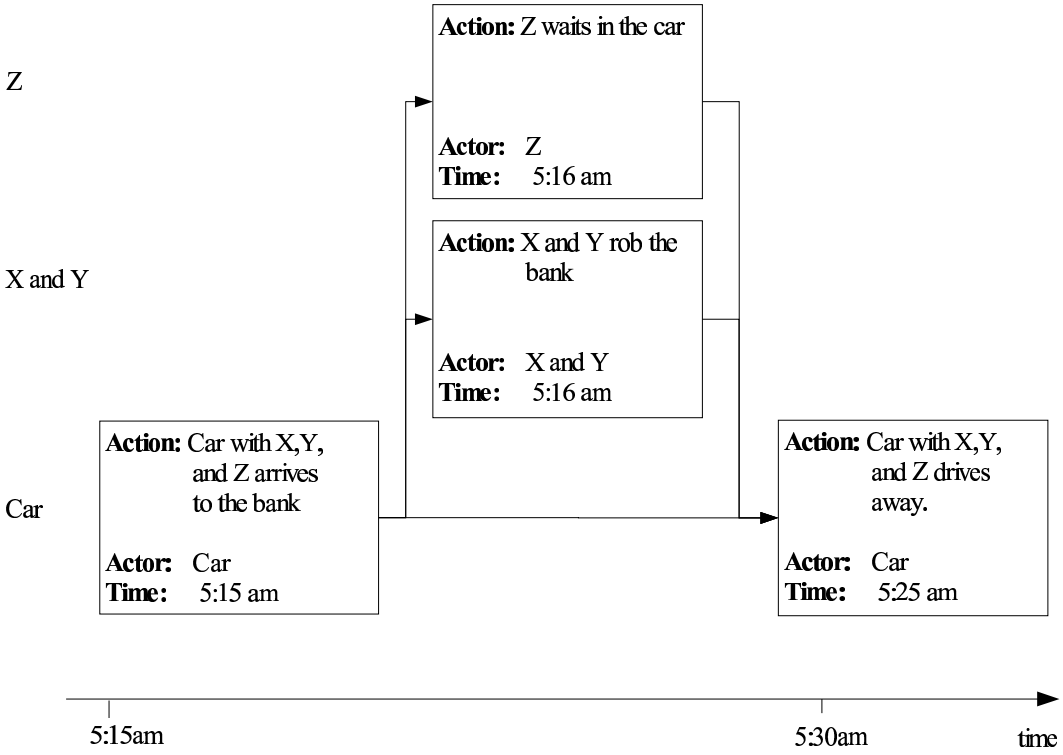


Figure 4.3: Example MES-diagram

1. *MES-trees.* Identification of events in MES is informal, but MES defines a technique called MES-trees [14] for elaborating possible scenarios of the crime or accident. MES-trees are very similar to attack trees described in Seton 4.2.1.

2. *Counterfactual reasoning.* Counterfactual reasoning is a way to establish causal dependency between events. It refers to the following philosophical argument due to David Hume [45]:

   > We may define a cause to be an object followed by another, and where all the objects, similar to the first, are followed by objects similar to the second. Or, in other words, where, if the first object had not been, the second never had existed.

   A mathematical formalisation of counterfactual reasoning has been given in [55] and [56].

Counterfactual reasoning is used on MES-diagram as follows. After events are identified, counterfactual reasoning is applied to every pair of events X and Y on the diagram. If X is a causal factor of Y, then an arrow is drawn from X to Y.

4.2.4 Why-because analysis

Why-Because Analysis (WBA) is a method for determining causes of complex accidents [53]. Like MES, it uses counterfactual reasoning to establish causality, but the approach taken by WBA is more formal.

Why-because graph

The graphical notation used by WBA to represent accidents is Why-because graph (WB-graph). It is a directed acyclic graph whose nodes represent elements of the accident, and whose arrows represent causal relationship.

Four kinds of nodes are defined for building WB-graphs. A node can be either of the following

- system state
- event – change from one state to another
- process – undifferentiated mix of events and states
- non-event – causally important absence of some event

Figure 4.4 shows WB-graph for the tarts rhyme from [23]:

> The Queen of Hearts, she made some tarts, All on a summer's day:
> The Knave of Hearts, he stole those tarts, And took them quite away!

Five nodes of WB-graph represent respectively:

1. Process: The queen of hearts makes tarts

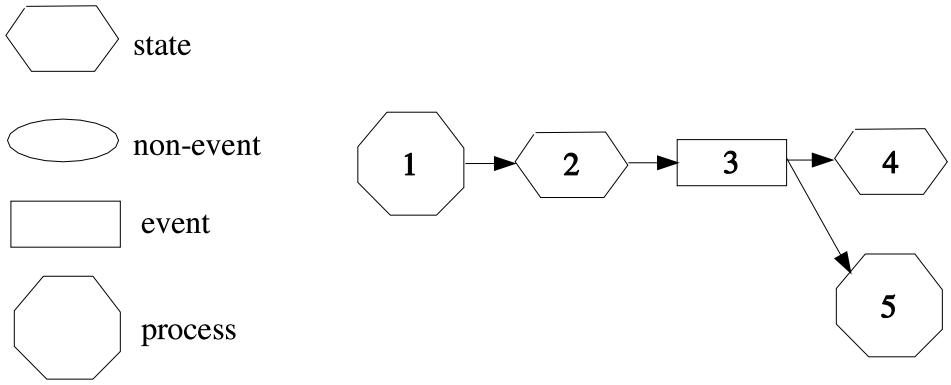


Figure 4.4: WB-graph for the tarts rhyme

2. State: the tarts are present

3. Event: Knave of hearts steels the tarts

4. State: the tarts are missing

5. Process: Knave of hearts takes the tarts away

The process of creating a WB-graph is similar to the process of creating MES-diagram. First, nodes of the graph are identified. Second, causal relationship between nodes is established using informal counterfactual reasoning.

Formal verification of WB-graph

After the WB-graph is constructed, it can be formally verified using *explanatory logic* defined by WBA.

Explanatory logic. Explanatory logic (EL), which is described in [57] and [58], has been purposefully developed for formal verification of WB-graphs. To achieve necessary expressiveness, EL combines three distinct formalisms: The Temporal Logic of Actions, the Standard Deontic Logic, and VCU logic.

- The Temporal Logic of Actions (TLA) described in [54] is a well known formalism for describing state based systems. TLA is used by EL for

modeling the real world. World states, events, processes, and required scientific knowledge are all described by TLA formulae.

- Standard Deontic Logic (SDL), which is described in [63], formalises "reasoning pertaining to obligation, permission, prohibition, and other normative matters [42]." SDL is used by EL to formalise laws and regulations. TLA is used in conjunction with SDL to formalise actions and objects mentioned in laws and regulations.

- VCU logic of D.K.Lewis [56] formalises counterfactual reasoning.

A well-formed EL formula is a VCU formula, whose basic propositions are TLA or SDL/TLA formulae.

Causal sufficiency of WB-graph. Formal verification of WB-graph consists in proving that every node in the graph (except for the nodes that represent root causes) is given a *causally sufficient* explanation.

Let $A_1, A_2, \ldots, A_n,$ and B be nodes in WB-graph, and let $A_1, A_2, \ldots, A_n$ be the nodes linked to B via an arrow (the arrow goes from A_i to B). Then the set $A_1, A_2, \ldots, A_n$ is a *causally sufficient* explanation of B if

1. $A_1, A_2, \ldots, A_n$ are *necessary* causal factors of B, which means that B cannot happen without prior happening of $A_1, A_2, \ldots, A_n$.

2. The happening of all $A_1, A_2, \ldots,$ and A_n will cause B to happen under any circumstances.

EL defines a set of proof rules for proving causal sufficiency. For example, proof rule (4.7) of [58] proves causal sufficiency through procedural necessity. It says that to prove that *circumstances* and *procedures* are causally sufficient explanation of event E, it suffices to prove that

1. Event E happened, and

2. *circumstances* occurred shortly before the event E, and

3. *procedures* were followed at all times, and

4. The *procedures* always lead to E if *circumstances* occur

The truth of items 1, 2, and 3 is usually assumed. The assumptions are motivated by available evidence. The correctness of item 4 is proved formally from formal definition of events and procedures.

EL and reconstruction process. EL formalises verification of reconstruction results, but it does not attempt to formalise the reconstruction process itself. The following observations support this conclusion.

First of all, the existence of WB-graph nodes (events, states, processes, and non-events) is *assumed* when formal verification starts. The initial discovery of the nodes is informal, and is not automated in WBA.

Second, WBA does not attempt to explore all possible causes of an event. It is explicitly stated in [58] that

> It would be a logical problem, interesting perhaps, but not always directly relevant to an incident investigation, to determine precisely which of the procedures might have been necessary to the occurrence of the event. We would not wish to enforce this investigation in all circumstances.

As a result, EL is designed to prove causal sufficiency of given set of nodes $A_1, A_2, \ldots, A_n$ with respect to another node B, rather than to explore all possible causes of B.

4.3 Summary and research problem statement

4.3.1 Analysis of the state of the art

Techniques described in this section can be readily used in digital forensics, but they do not *fulfil* the demand for a reconstruction theory. They reduce

reasoning errors by structuring the reconstruction process, but they neither automate reconstruction, nor ensure completeness of reconstruction. There are several reasons for that insufficiency.

- *Techniques described in this section were developed to assist human investigators rather than to create automated tools.* They do organise reconstruction process into a sequence of steps, but the investigator is expected to use his or her intuition to perform individual steps. Formal notation is used only to present and verify reconstruction results.

- *Techniques described in this section do not formalise the* entire *knowledge used by investigators.* The knowledge used by investigators in "ordinary" investigations is difficult to formalise in its entirety. First of all, it is a lot of knowledge – apart from "common sense" knowledge, it includes the laws of physics, engineering, psychology, medicine, etc. Second, much of that knowledge is too vague to be expressed and manipulated in formal logic.

4.3.2 Research problem statement

In some respects event reconstruction in digital forensics is easier than event reconstruction in "ordinary" investigations. The domain of digital forensics — computers — is only a part of the domain of "ordinary" investigations. In addition, computer functionality can be adequately described using mathematics and formal logic. However, formalisation of the entire knowledge used by digital forensic scientists is still impractical because of its continuing enlargement.

Some reconstruction techniques of digital forensics require only limited knowledge of computers. For example, identification of deleted and moved files on a disk volume requires only the knowledge of the file system, it does not require the knowledge of Internet protocols or anything else. Formalisation of such techniques is practically possible, because the body of knowledge to be formalised is limited and well defined.

This research does not aim to formalise all reconstruction techniques of digital forensic analysis. *The aim of this research is (1) to formalise event reconstruction in a general setting, that is, assuming nothing specific about the digital system under investigation or about the purpose of event reconstruction, and (2) to show that this formalisation can be used to describe and automate selected examples of digital forensic analysis.*

The rest of this dissertation describes an attempt to achieve these aims. After the necessary theoretical backgorund is defined in Chapter 5, Chapter 6 develops a mathematical definition of event reconstruction problem. Chapters 7 and 8 then demonstrate that the developed formalisation can be used to describe and automate specific techniques of digital forensic analysis. Chapter 7 constructs and implements a generic event reconstruction algorithm, which is based on the developed formalisation of event reconstruction. Chapter 8 uses that algorithm to formalise and automate examples of file system analysis and event time bounding analysis.

Chapter 5

Theoretical background

This chapter provides necessary background in computer science for the development and analysis of event reconstruction model in the rest of the dissertation. The chapter is divided into two sections.

Section 5.1 describes the formal notation used in the dissertation. It is a combination of usual mathematical notation with expressions of ACL2 logic. Mathematical notation is described in Section 5.1.1. ACL2 notation is described in Section 5.1.2.

Sections 5.2 introduces the state machine model of computation, which serves as theoretical basis for formalisation of event reconstruction presented in subsequent chapters.

5.1 Formal notation

5.1.1 Mathematical notation

Sets. Sets are denoted by capital Roman letters, e.g. A, B, C. Sets are defined in two ways:

1. by listing their members between { and }, e.g. $A = \{1, 2\}$;

2. by a set former $A = \{a|p(a)\}$, which specifies the set of all objects a such that $p(a)$ is true. Sometimes $p(a)$ is given verbal definition.

Empty set. Empty set is denoted $\emptyset$.

Powerset. Powerset of a set A is denoted 2^A.

Cardinality. Cardinality of a set A is denoted $|A|$.

Membership. Statement that a is a member of set A is denoted $a \in A$. Statement that a is not a member of set A is denoted $a \notin A$.

Subset. Statement that set A is a subset of set B is denoted $A \subseteq B$.

Union, intersection, and set difference. Union, intersection, and set difference of two sets A and B are denoted $A \cup B$, $A \cap B$, and $A \setminus B$ respectively.

Integers. Set of integers is denoted $\mathbb{Z}$.

Rationals. Set of rational numbers is denoted $\mathbb{R}$.

List (Sequence). A list (sequence) is defined by listing its elements in round brackets, e.g. $(0, 1, 1, 0, 0)$.

Length. Length of a list a is denoted $|a|$.

Numbering of elements in a list. Elements in a list are numbered from 0. The i-th element of a list a is denoted a_i. If $a = (1, 2, 3)$ then $a_0 = 1,\ a_1 = 2,\ a_2 = 3$

Concatenation. Concatenation of two lists a and b is denoted $a \cdot b$. For example $(1, 2, 3) \cdot (4, 5) = (1, 2, 3, 4, 5)$

Sum of elements of a list. The sum of all elements of a list a is denoted Σa. The sum of elements a_i such that $m \leq i \leq n$ is denoted $\sum_{i=m}^{n} a_i$. More precisely,

$$\sum_{i=m}^{n} a_i = \begin{cases} a_m + \ldots + a_n & \text{, if } m < n \\ a_m & \text{, if } m = n \\ 0 & \text{, if } m > n \end{cases}$$

Empty list. Empty list is a list with no elements. Let a denote a list, and let ϵ denote an empty list, then

$$|\epsilon| = 0$$

$$\epsilon \cdot a = a \cdot \epsilon = a$$

Language. A language is a set of all finite lists composed of elements of some set. Language is denoted A^*, where A is a set, from elements of which lists are composed. Language includes empty list ϵ. If $A = \{0, 1\}$, then

$$A^* = \{\epsilon, (0), (1), (0, 0), (0, 1), (1, 0), (1, 1), \ldots\}$$

Any language over a non-empty set of elements is infinite.

Tuple. Lists represent tuples. A tuple is defined by listing its elements in brackets, e.g. $(1, 2)$

Set product. Set product is denoted $A \times B$. It is a set of all possible pairings between elements of A and elements of B:

$$A \times B = \{(a, b) |\ a \in A,\ b \in B\}$$

Functions. Usual mathematical syntax is used for functions. For example, term $f(x, y)$ denotes application of function f to arguments x and y.

O–notation. O–notation provides a way to give an asymptotic upper bound on a function. For a given function of m non-negative integer arguments $g(x_0, \ldots, x_m)$, expression $O(g(x_0, \ldots, x_m))$ denotes the set of functions

$$\begin{aligned} O(g(x_0, \ldots, x_m)) = \{f(x_0, \ldots, x_m) \mid & \text{ there exist positive constants} \\ & c, n_0, \ldots, n_m \text{ such that} \\ & 0 \leq f(x_0, \ldots, x_m) \leq cg(x_0, \ldots, x_m) \text{ for all } x_0, \ldots, x_m \\ & \text{such that } x_0 \geq n_0, \text{ and } x_1 \geq n_1, \text{ and } \ldots x_m \geq n_m\} \end{aligned}$$

When it is said that function $f(x_0, \ldots, x_m)$ is $O(g(x_0, \ldots, x_m))$ it means that it is a member of $O(g(x_0, \ldots, x_m))$. When $O(g(x_0, \ldots, x_m))$ is used in mathematical expressions it stands for some unnamed member of $O(g(x_0, \ldots, x_m))$.

ACL2 functions. ACL2 functions, including functions defined in Appendix A, are permitted in mathematical formulae. Both the usual mathematical syntax and the prefix notation of Lisp are allowed. For example, $car(l) =$ `(car l)`.

5.1.2 ACL2 notation

ACL2 (A Computational Logic for Applicative Common Lisp) is a first-order logic of total functions. It uses side-effect free subset of Common Lisp as a syntax for the logic. For a precise definition of syntax and semantics of ACL2 logic see [49]. A tutorial introduction to ACL2 can be found in [48]. A collection of ACL2 case studies is given in [47].

Atomic data types. ACL2 logic provides atomic data objects of several types. *Numbers* include the integers, rationals, and complex rational numbers. *Strings* are sequences of characters such as `"abc"`. *Symbols* can be thought as atomic words, such as `lisp` or `append`.

Ordered pairs. ACL2 provides ordered pairs or *conses*. An element of a cons is either a cons or an atom. Binary trees and lists are represented as

conses.

Lists. A *list* is either an atomic object or a cons whose second element is a list. Atomic objects are called empty lists.

True lists. Lists terminated with special symbol `nil` are called *true lists*.

Constants. In ACL2 statements, Common Lisp syntax is used for constants. For example

- `0 10 -1 -5` are integer constants
- `"hello world!"` is a string constant. String constants are delimited with `"` sign
- `'lisp 'append` are symbol constants
- `'( a . b )` and `'((lisp . 1) . (append . 2))` are examples of ordered pair constants
- `'(0 1 0)` is an example of a true list constant. It is a shorthand for the ordered pair constant `'(0 . (1 . (0 . nil )))`

Booleans. Falsity in boolean expressions is denoted by symbol `nil`. Any non-`nil` value in boolean expressions means truth. Symbol `t` is returned as truth constant by most ACL2 functions.

Expressions. Common Lisp expressions are used to construct ACL2 statements. An ACL2 expression is either

- a constant,
- a variable symbol (whose value must be determined by the context),
- a function application of the form `(fn a1 a2 ... an)`, where `fn` is the function name and `a1`, `a2`, ... `an` are function arguments,

- an event `(event-name a1 a2 ... an)` that modifies logical world of ACL2, or
- a backquote expression

Named constants. Named constants are shorthands for ordinary constants. They are defined using `defconst` event: `(defconst *name* constant)`, where `*name*` is the name for the constant `constant`.

Backquote expression. Backquote expression is a list constant prepended with ‘ instead of ’. Atoms in backquote expressions may be prepended with comma, in which case they are treated as names of constants[1]. The value of a backquote expression is the result of substituting the values of the comma-prepended constants into the list constant. Consider the following example.

```
(defconst *LIST-A* '(a b c))
(defconst *LIST-B* `(1 2 ,*LIST-A* 3))
```

The value of `*LIST-B*` is constant `'(1 2 (a b c) 3)`.

ACL2 functions. ACL2 provides a number of built-in functions. Most of them are standard Common Lisp functions. Built-in ACL2 functions used in the following chapters are defined in Appendix A.

Macros. Macros are used in ACL2 as shorthands for long expressions. Standard macros used in the following chapters are described in Appendix A.

New function definition. New functions are introduced into ACL2 logical universe using `defun` event. The general form of function definition is `(defun name args dcl body)`, where `name` is the name for the new function,

[1] There are other interpretations of comma-prepended atoms, which are not used in this dissertation. See [49] for details

`args` is a list of formal parameters, `dcl` is an optional list of declarations, and `body` is an ACL2 expression. A function definition extends ACL2 logical universe with an axiom of the form

$$name(args) = body$$

The following statement defines function concatenating two lists.

```
(defun app (a b)
  (if (consp a)
    (cons (car a) (app (cdr a) b))
    b))
```

Before recursive definition is admitted into the logic, the recursion must be proved to terminate. This is achieved by proving that some measure of function arguments decreases according to some well-founded relation with each recursive iteration.

Theorems. Theorems are introduced using `defthm` event. General form of theorem definition is `(defthm name body inst)`. Where `name` is the name for the new theorem, and `body` is an ACL2 expression in one of permitted forms (see [48] for details). `inst` denotes optional instructions to the theorem prover.

The meaning of theorem is that for an arbitrary substitution of terms for the variables in `body`, the result of evaluating `body` is `t`. For example, `(defthm 2x2 (equal (* 2 2) 4))` is a theorem in ACL2. The following theorem states associativity of concatenation.

```
(defthm assoc-of-app
  (equal (app (app a b) c)
         (app a (app b c))))
```

Recursion as limited form of quantification. Although ACL2 logic provides support for explicit quantification, ACL2 theorem prover offers little automation for reasoning with quantifiers. As a result, ACL2 theorems are usually formulated in terms of recursive test functions instead of quantified expressions.

For example, to state in ACL2 that all elements of list `l` are non-negative integers one usually defines a recursive function

```
(defun natural-listp (l)
  (if (consp l)
    (and (integerp (car l))
         (< 0 (car l))
         (natural-listp (cdr l)))
    t))
```

which returns true only if its argument is a list of non-negative integers. Once function `natural-listp` is defined, expression `(natural-listp l)` can be used in theorems to state desired property.

Proofs. ACL2 proofs are constructed using ACL2 proof rules. See [49] for a comprehensive description of ACL2 proof rules.

Propositional axiom. For every formula ϕ, derive $(\neg\phi \vee \phi)$.

Propositional proof rules:

- *Expansion:* derive $(\phi_1 \vee \phi_2)$ from ϕ_2.
- *Contraction:* derive ϕ from $(\phi \vee \phi)$.
- *Associativity:* derive $((\phi_1 \vee \phi_2) \vee \phi_3)$ from $(\phi_1 \vee (\phi_2 \vee \phi_3))$.
- *Cut:* derive $(\phi_2 \vee \phi_3)$ from $(\phi_1 \vee \phi_2)$ and $(\neg\phi_1 \vee \phi_3)$.

Other propositional rules of inference such as modus popens can be derived from the above rules and the propositional axiom.

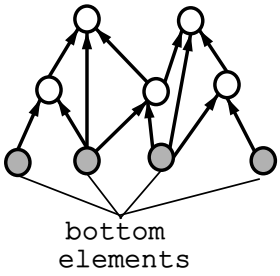


Figure 5.1: Well founded order

Substitution of equals for equals. Derive

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$$

from

$$(x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$$

Instantiation. From theorem ϕ and substitution σ of terms for variables in ϕ derive the result of substitution.

Opening of function calls. To "open" a function call $f(x, y, z)$ means to replace term $f(x, y, z)$ by the right hand side of the function definition $f(a, b, c) = body(a, b, c)$ with x, y, and z substituted for a, b, and c respectively.

Induction. ACL2 induction principle is a variation of structural induction described in [21].

Suppose there is a set $A = \{a\}$ and a well founded relation $\succ$ defined on the elements of A. Relation $\succ$ is well founded, if any non-empty subset of A contains at least one "bottom" element $\hat{a}$, such that no other element of the subset is $\succ$-smaller than $\hat{a}$. The relation $\succ$ may be partial. The well-foundedness of $\succ$ arranges elements of A in a directed acyclic graph (see Fig. 5.1).

To prove that some property p holds for all elements of A it is sufficient to prove that

1. *Base case:* the property p holds for all bottom elements of A;
2. *Induction step:* the property p holds for arbitrary $a \in A$, assuming that p holds for all elements that are $\succ$-smaller than a.

Instead of assuming that p holds for all $\succ$-smaller elements, one can assume that p holds for *some* of the $\succ$-smaller elements. One can also split single base case and single induction step into several base cases and several induction steps, all of which must be proved.

This induction principle can be applied directly to ACL2 function definitions. Every definition of ACL2 function is associated with a measure of the function arguments and a well-founded relation. To show that every function definition terminates, ACL2 proves that the measure decreases according to the relation in every recursive call. The measure together with the relation impose well-founded ordering on the values of function arguments. The values for which recursion terminates are bottom elements.

Given a formula and a function definition, one can use the following technique to generate base cases and induction steps. First, identify all execution paths through the body of the function. Second, write a base case for every execution path with no recursive calls. Third, write an induction step for every path that contains one or more recursive calls. Each induction step has as many induction hypotheses as there are recursive calls in the corresponding execution path. Each induction hypothesis asserts correctness of the target formula for parameters of the corresponding recursive call.

To prove `(booleanp (natural-listp l))` by structural induction, one would analyse definition of `natural-listp` given on page 56 and produce one base case and one induction step:

Base case:

```
(implies (atom l)
         (booleanp (natural-listp l)))
```

Induction step:

```
(implies (and (not (atom l))
              (booleanp (natural-listp (cdr l))))
              (booleanp (natural-listp l)))
```

Both statements are trivially proved by opening calls of `natural-listp` and simplification.

5.2 State machine model of computation

This section describes the state machine model of computation and its application to analysis of computing systems. Section 5.2.1 defines basic state machine and reviews some of its extensions. Section 5.2.2 discusses approaches to the development of state machine models of computing systems. Section 5.2.3 discusses methods for automated analysis of finite state machine models of computing systems.

5.2.1 Basic state machine model and its variations

The notion of state machine was introduced by Alan Turing in his work on computable numbers [77]. It served as a model of human performing a computation.

Basic state machine can be defined as a triple $T = (I, Q, \delta)$, where

- I is the set of input symbols;
- Q is the set of states, which the machine can assume;

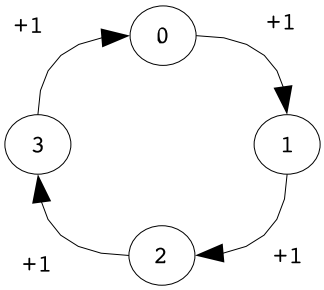


Figure 5.2: Counting state machine

- $\delta : Q \times I \rightarrow Q$ is the transition function.

State machine is called finite if all of its elements are finite.

Operation of state machine. State machine consumes a sequence of input symbols. For each symbol, the machine changes its state. The new state is determined by the transition function from the current state of the machine and the input symbol being consumed. The process of state change is known as *transition*, and the sequence of transitions is called a *computation*.

Transition graph. A common way to depict finite state machine is to draw its transition graph also known as transition diagram. The nodes in the graph represent states, the arrows represent transitions. The labels on the arrows represent input symbols that cause transitions. Figure 5.2 shows a finite state machine that counts from 0 to 3. It has four states: 0, 1, 2, and 3, and a single input symbol +1, which forces the machine to advance to the next state. Suppose that 0 is the initial state of the machine, then after processing the sequence $(+1, +1, +1)$ the machine will be in state 3.

State machine resembles sequential circuit. The operation of state machine closely resembles the operation of sequential circuit, which is the basic building block of modern computers. A sequential circuit consists of a combinatorial circuit and a vector of memory elements. Memory elements store the

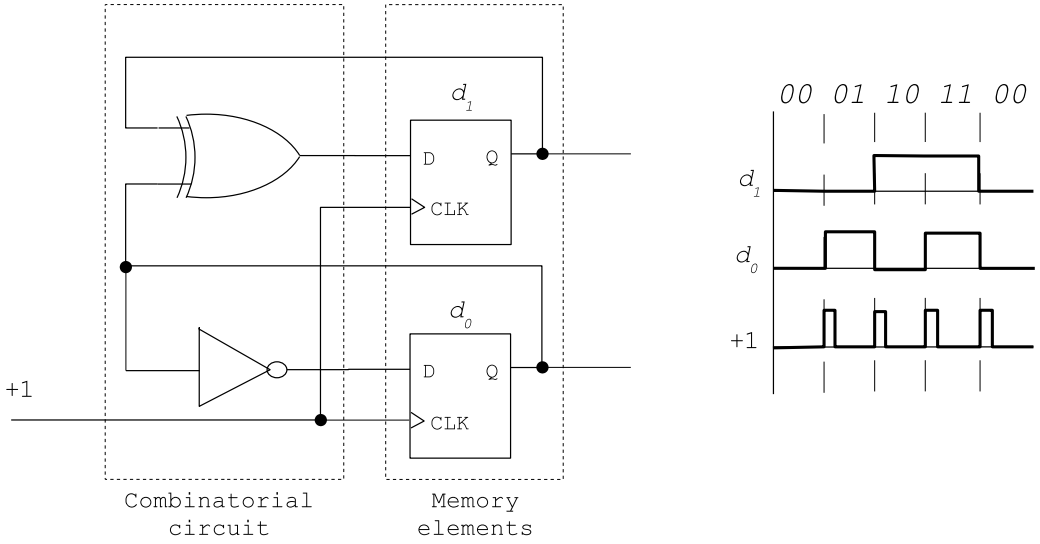


Figure 5.3: A 2-bit binary counter

current state of the circuit, while the combinatorial circuit implements its transition function. Only two distinct voltage levels are allowed in the sequential circuit — high and low.

Figure 5.3 shows an example sequential circuit that implements a 2-bit binary counter. It works as follows.

- The combinatorial circuit inverts the output of memory element d_0 and feeds it back to the input D of d_0. That is, if the output of d_0 is high, its input D will be low, and vice versa.
- The combinatorial circuit produces high voltage at the input D of d_1 if the outputs of d_1 and d_0 are different, and low voltage if the outputs of d_1 and d_0 are the same.
- When the voltage at the input CLK of memory elements raises from low to high, the voltage at their D inputs is latched in the memory elements and appears at their outputs.

If the low voltage level at the circuit output is associated with digit '0', and the high voltage level is associated with digit '1', and if the outputs of d_0 and

d_1 are associated with digits in a binary number (d_0 being the least significant digit, and d_1 being the most significant digit), then each clock pulse adds 1 to the number represented by the memory elements.

As long as the details of state transition process are not important to the analysis of sequential circuit, finite state machine shown in Figure 5.2 provides a good abstraction of the 2-bit binary counter. Finite state machines are commonly used for specification and minimisation of sequential circuits (see for example Chapter 10 of [41]).

Modeling concurrent systems as state machines

State machines provide a natural way to model systems whose components change their states synchronously like sequential circuits. Nevertheless, state machines can also be used to model concurrent systems, whose components change their states asynchronously (at different moments in time). State machine models of such systems are based on the interleaving model of concurrency, which can be summarized as follows:

- states of all components of a concurrent system form a global system state;
- the result of concurrent updates to the global state can always be simulated by an equivalent *sequence* of atomic updates to the global state.

The entire system is modeled as a single state machine, whose state is a vector of states of individual components, and whose transitions perform atomic updates of the global state. Figure 5.4 gives an example of such a model.

The interleaving model of concurrency has been successfully used in practice, particularly in the domain of hardware and software verification. See [28] for examples.

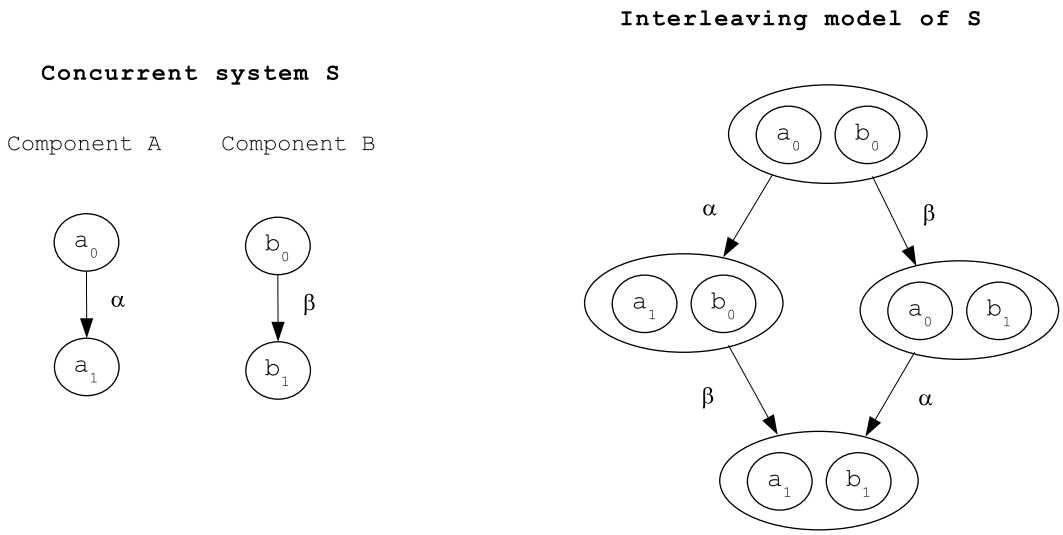


Figure 5.4: Interleaving model of concurrent system

Variations of state machine

Many extensions and modifications of the basic state machine are known. Given below are some examples that appeared in the literature. Due to the abundance of such extensions, the list given below is characterising rather than comprehensive.

- *Transducers.* Transducer is a state machine, which has output. For each input symbol is produces an output symbol. The concept of transducer emerged in [44] as a way to formalise synthesis of sequential digital circuits. The two best known types of transducers are Mealy machines [62] and Moore machines [64]. *Mealy machine* is a tuple with six elements:

$$Me = (I, Q, \delta, q_0, O, \sigma)$$

  where I,Q, and δ are defined as above, and

  - q_0 is the initial state of the machine;
  - O is the set of output symbols;
  - $\sigma : I \times Q \rightarrow O$ is the output function.

Initially, *Me* resides in state q_0. Simultaneously with each transition, *Me* produces an output symbol which is determined by σ from the current state of the machine and the current input symbol. *Moore machine* is also a tuple of six elements:

$$Mo = (I, Q, \delta, q_0, O, \kappa)$$

where I, Q, δ, q_0, and O are defined as above and $\kappa : Q \rightarrow O$ is the output function. Like Mealy machine, *Mo* generates one output symbol after each transition. However, the output symbol of *Mo* does not depend on the current input symbol.

- *Acceptors.* Acceptor is a state machine, some of whose states are designated as "accepting" states. Such a state machine *accepts* a sequence of input symbols if, after processing the sequence, it stops in an accepting state. Similarly, it rejects an input sequence if it stops in a non-accepting state. Formally acceptor is a tuple with five elements

  $$Ac = (I, Q, \delta, q_0, Q_a)$$

  where I,Q, δ, and q_0 are defined as above, and Q_a is the set of accepting states.

  The notion of acceptor (or accepting automaton) was introduced in [51] to formalise McCulloch-Pitts model of neural net. Finite accepting automata turned out to be compact representations for many useful sets of objects [16]. An object is in the set, if symbolic encoding of the object is accepted by an accepting automaton representing the set.

- *Non-deterministic automata.* Non-deterministic automaton is an enhancement of the acceptor automaton concept. Non-deterministic automaton is obtained from the basic (deterministic) acceptor by replacing

transition function δ with a transition *relation*

$$\widetilde{\delta} \subseteq ((I \times Q) \times Q)$$

Rather than specifying a single next state for a combination of current state and input symbol, transition relation specifies *several* possible next states. As a result, a single input sequence corresponds to several possible computations. Non-deterministic automaton accepts an input sequence if there is at least one possible computation corresponding to the input sequence that ends in an accepting state.

Non-deterministic automata were introduced by Rabin and Scott in [70] as a way to simplify formal descriptions of acceptors. They also showed that a non-deterministic automaton can always be simulated by a deterministic automaton, which accepts exactly the same set of input sequences as the non-deterministic automaton.

- *State machines with external memory.* This type of state machine model consists of a state machine that controls one or more external storage devices. The best known example of such model is the family of Turing machines described in [77].

  A basic Turing machine consists of a finite state machine which controls a single read/write head. The head can move along an infinite tape, which is divided into sections and each section can store one symbol from a finite alphabet. The head can read the symbol from the section directly under it, write new symbol into the section, and move to the left or to the right by one section (see Figure 5.5).

  Turing machine functions as follows. First the symbol is read from the tape and input into the state machine. The state machine transits into a new state and emits a command for the head, which specifies (a) the new symbol to be written on the tape, and (b) in what direction the

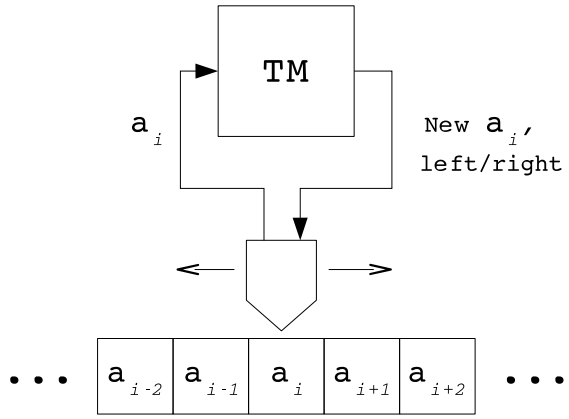


Figure 5.5: Turing machine

head should move after writing the symbol.

The state machine that controls the head can be formally defined as a tuple

$$TM = (A,\, Q,\, \delta,\, q_0,\, \theta,\, \{\text{left}, \text{right}\})$$

where

- A is the set of symbols that can be written on the tape
- Q is the set of states of the state machine
- q_0 is the initial state
- $\delta : A \times Q \rightarrow Q$ is the transition function
- $\theta : A \times Q \rightarrow A \times \{\text{left}, \text{right}\}$ is the output function, such that for every combination of tape symbol and state it determines the new tape symbol to be written, and the direction of the head's movement.

Turing machines were introduced to formalise the notion of computation and served as a major instrument in building theories of computability and complexity.

In summary, the basic state machine model can be extended in many ways to suit modeling needs of a specific domain. Both finite and infinite state machine

models have been defined.

Finite state machines are appropriate for digital investigations

From philosophical standpoint the infinite models — such as Turing machines or machines with infinite state space — are not required for reasoning about the *real* computing systems. The real computing systems are finite. They are built using finite number of sequential circuits, which in the normal course of operation have finite number of states; they communicate over channels with finite throughput; and they operate at a finite clock frequency for a finite amount of time. Digital investigations are concerned only with real computing systems. Thus, for the purposes of digital investigations it should suffice to model computing systems as finite state machines.

For this reason, the following chapters assume that *the system under investigation can be formalised as a finite state machine*, and that *only finite computations need to be considered for event reconstruction.*

5.2.2 Creation of system models

Creation of a suitable formal model of a system is important first step in any formal analysis. This section reviews potential problems with formal models, and suggests possible approaches to the development of models for digital investigations.

Potential problems with formal models All formal models, including state machine models, suffer from two kinds of problems: problems caused by the closed world assumption, and errors in specification of possible system behaviour.

Closed world assumption. If some state or event is not represented in the formal model of a system, the subsequent formal analysis will have no basis for reasoning about such a state or event. The analysis will have to assume

that such a state or event does not exist. This assumption is known as the *closed world assumption*. Observe, that if some important event is omitted, the analysis is not comprehensive, and the conclusions obtained by such analysis are not necessarily sound.

Errors in specification of system behaviour. This refers to misrepresentation of system behaviour within the chosen formal framework. In the state machine setting it amounts to allowing impossible computation or disallowing possible computations. Note, however, that the problem arises only with those computations, whose presence or absence may affect the outcome of the analysis. Obviously, if it can be proved that presence or absence of some computation does not affect the outcome of the analysis, that computation may be safely excluded from the model.

Verification of formal models. Two distinct methods exist for checking that the model is free from above described errors. First, model correctness can be tested experimentally, by comparing model predictions with the experimental results. Second, model correctness can be proved by showing its equivalence to another model of the same system, where the latter model is believed to be correct.

Approaches to the development of system models

Identified below are two possible approaches to the development of formal models for digital investigations: completely manual and transformational, which obtains the model by transforming another model.

Manual model construction. In this approach all modeling is performed by a human expert who specifies the model using some formal language. This approach is laborious, but the resulting model should be admissible in court, because by building it the expert expresses his expert opinion about how

the system works. Testing is an obvious way to improve confidence in such a model. Proving equivalence of the model to another model of the system is another possibility.

Model construction by transforming another model. A different way to obtain a model of a system is to transform another model of the same system using a well defined set of transformation rules. The rules must be such that they preserve properties of analytical interest. This approach is particularly appealing, because most of computer systems are already defined using some formal language — either programming language like C and Java, or hardware definition language like VHDL and Verilog. Automatic construction of finite state machine models directly from source code is an area of active research. Prototype systems for automatic construction of finite state models have been reported in [32].

Other approaches to model construction Since the main aim of this research is to formalise event reconstruction, further investigation of approaches to creating finite state machine models of systems for forensic purposes is left for future work. The reader is referred to [66] for further discussion of practical aspects of specifying state machine models of systems.

5.2.3 Analysis of finite computations

Many analyses of finite state machines can be reduced to a search for computations that satisfy certain property. If the length of possible computations is limited, all such computations can be found by a depth-limited search in the state space of the machine.

Let A be a finite state machine $A = (I, Q, \delta)$, and let k be an upper bound on the number of transitions in possible computations of A. A naive algorithm for analysis of computations of A is given in Figure 5.6. First, it computes the set C_{A_k} of all computations of A bounded by k, then it checks

1: $C_{current} \leftarrow Q$
2: $C_{A_k} \leftarrow C_{current}$
3: **for** $j \leftarrow 1$ to k step 1 **do**
4: $\quad C_{next} \leftarrow \emptyset$
5: $\quad$ **for** every computation $c \in C_{current}$ **do**
6: $\quad\quad q \leftarrow$ the last state in c
7: $\quad\quad$ **for** every input symbol $\iota \in I$ **do**
8: $\quad\quad\quad p \leftarrow \delta(q, \iota)$
9: $\quad\quad\quad$ Make new computation c' by suffixing c with a transition $q \xrightarrow{\iota} p$
10: $\quad\quad\quad C_{next} \leftarrow C_{next} \cup \{c'\}$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad C_{A_k} \leftarrow C_{A_k} \cup C_{next}$
14: $\quad C_{current} \leftarrow C_{next}$
15: **end for**
16: **for** every computation $c \in C_{A_k}$ **do**
17: $\quad$ Check c against analysis criteria
18: **end for**

Figure 5.6: A naive algorithm for finite computation analysis

every computation in C_{A_k} against the analysis criteria.

If operations in lines 1–16 and 18 take constant time, and the time of checking in line 17 is $g(k)$, then it can be shown that for a given A the worst running time of the algorithm is $O((g(k) + \gamma)|Q||I|^{k+1})$, where γ is an implementation dependent constant.

Despite exponential complexity of the naive algorithm, algorithms with lower complexity have been constructed for many kinds of finite state machine analyses. Most of this work was done in the domain of automatic verification of reactive systems also known as model checking [28]. Three key methods were used for reducing complexity. Each of them is discussed in the following paragraphs.

Avoiding explicit construction of computations Many properties of computations can be inferred from the transition graph without constructing possible computations. For example, to verify that every state in every possible

computation of A satisfies some property, it suffices to check that all states in Q satisfy that property. This provides an algorithm with running time $O(|Q|)$ rather than $O((g(k) + \gamma)|Q||I|^{k+1})$

This idea was actively developed by the model checking community. One commonly used formalism for expressing correctness criteria is propositional temporal logic CTL*, which was defined in [33]. By avoiding explicit construction of computations, it was possible to construct an algorithm that checks given finite state machine against given CTL* formula f in time $O((|Q| + R)^{O(|f|)})$, where $|Q|$ is the number of possible states, $|f|$ is the length of the formula, and R is the number of arcs in the transition graph of the state machine (see pages 46–69 of [28] for details).

Symbolic representation of state sets Another approach to reducing complexity of analysis algorithms is to represent sets of computations *implicitly*, for example as formulae in some decidable logic. When such a formula is evaluated against a state it is either true or false. Thus, the formula can be viewed as a representation for the set of all states that make it true.

When sets of states are represented symbolically, state transitions are implemented by formula transformations. A set of states is processed at once. In model checking, the result of such transformation is usually defined as the set of all states reachable by single transition from states in the input set. That is, for a set of states $X \subseteq Q$

$$\mathit{Transform}(X) = \bigcup_{x \in X} \{\delta(x, i)\} \text{ for all } i \in I \text{ for which } \delta(x, i) \text{ is defined}$$

For example, the set of all states reachable from set X in k transitions can be computed by k consecutive transformations of the formula representing X. Similarly, model checking algorithms for various propositional temporal logics can be defined in terms of state set transformer (see Chapter 5 in [28]).

The hope of symbolic techniques is that the time and space required for

formula manipulation is less than the time required for manipulation of states represented *explicitly* as distinct data objects. To fulfil this hope, symbolic representations of state sets must be supported by efficient algorithms for checking emptiness of, intersecting, and otherwise transforming these representations. Symbolic representations commonly used in model checking include

- Ordered Binary Decision Diagrams (OBDD) [18] and their variations.
- Propositional logic formulae in conjunction with efficient satisfability checking algorithms [15].

Other representations such as regular expressions [50] and integer constraints [20] are used in the domains where OBDDs and propositional formulae are insufficiently expressive, such as verification of real time systems.

Symbolic model checkers have been reported to outperform explicit model checkers by many orders of magnitude in the number of states which they can handle (see [46]).

Reduction of the finite state machine model This group of techniques is based on the observation that the analysis of a particular property often uses only a part of the information contained in the model. In this case, the analysis can be performed on a reduced model, in which the redundant information is removed. The reduced model often requires less time and space to analyse. However, the construction of the reduced model makes sense only if it can be performed efficiently — only if the gain in the computing resources provided by the reduced model is greater than the amount of computing resources spent on its construction.

Several techniques for model reduction have been proposed in model checking. The most successful examples are partial order reduction, and data abstraction.

Partial order reduction. In model checking, concurrent systems are modeled using the interleaving model of concurrency described in Section 5.2.1. The need to represent all possible interleavings of concurrent transitions can make the model very large. For n concurrent transitions there are $n!$ possible interleavings. Partial order reduction is a technique for simplifying analysis of such models.

Partial order reduction is based on the observation that many concurrent transitions are *independent* from each other — they neither enable nor disable each other, and they lead to the same global state irrespective of the order in which they are executed. It turns out that to verify many properties it suffices to consider only one possible ordering of independent transitions [27]. A model checker that uses partial order reduction detects independent transitions, and — if the property which is begin verified permits partial order reduction — it considers only one interleaving of such transitions.

The use of partial order reduction for simplifying model checking was first proposed in [67]. More general model checking algorithms based on the same ideas appeared later in [78], [69], and [40].

Data Abstraction. Data abstraction is used for simplifying model checking of systems that involve data processing. Data abstraction is based on the observation that many specifications involve fairly simple relationships among data values. In such cases, the large number of actual data values can be mapped into a small number of *abstract* data values, which represent key groups of actual data values. The model is then re-stated in terms of abstract data values. The new model often has less states than the original model. Figure 5.7 gives an illustration of data abstraction.

For a survey of "classical" abstraction techniques see Chapter 13 of [28]. Current research in this area concentrates on automatic derivation of abstract models directly from the source code of industrial programming languages and hardware-definition languages [26].

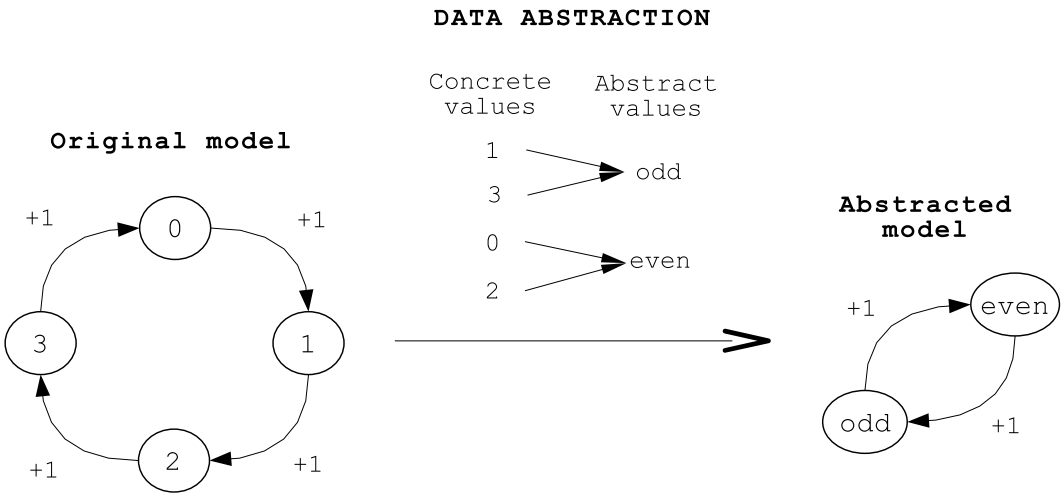


Figure 5.7: Data abstraction

5.3 Summary

This chapter defined formal notation used in the rest of the dissertation and presented background information about the state machine model of computation and its application to analysis of computing systems. It was argued that finite state machines and finite computations provide sufficient basis for formalisation of event reconstruction in digital investigation. The next chapter builds on the ideas presented in this chapter. It defines a formalism for describing evidence as properties of computations, and gives a formal definition of event reconstruction problem.

Chapter 6

Formalisation of event reconstruction problem

This chapter uses state machine model of computation to formalise event reconstruction problem in digital investigations. The chapter is organised into two parts. First, the key concepts are introduced informally on a fictional example of networked printer analysis in Section 6.1. Second, the introduced concepts are more rigorously formalised in Section 6.2.

6.1 Informal example of state machine analysis

This section illustrates the possibility of using state machines for event reconstruction in digital investigations. It considers a fictional example of networked printer analysis. First, an informal analysis is given, then it is illustrated using a finite state model of the printer.

6.1.1 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 6.1. The

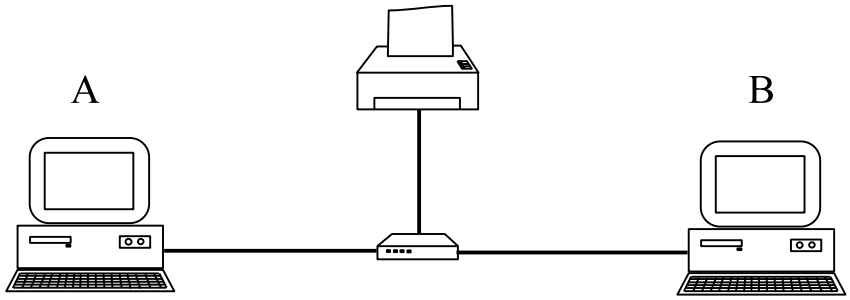


Figure 6.1: ACME Manufacturing LAN topology

cost of running the network is shared by its two users Alice (A) and Bob (B). Alice, however, claims that she never prints anything 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.

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 derived from the finite state model of the print job directory.

6.1.2 Informal analysis illustrated with a state machine

Please look at Figure 6.2. 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 6.2 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 A
job from B
empty
empty
empty
...

The initial state of the directory corresponds to the ellipse (e,e). The state

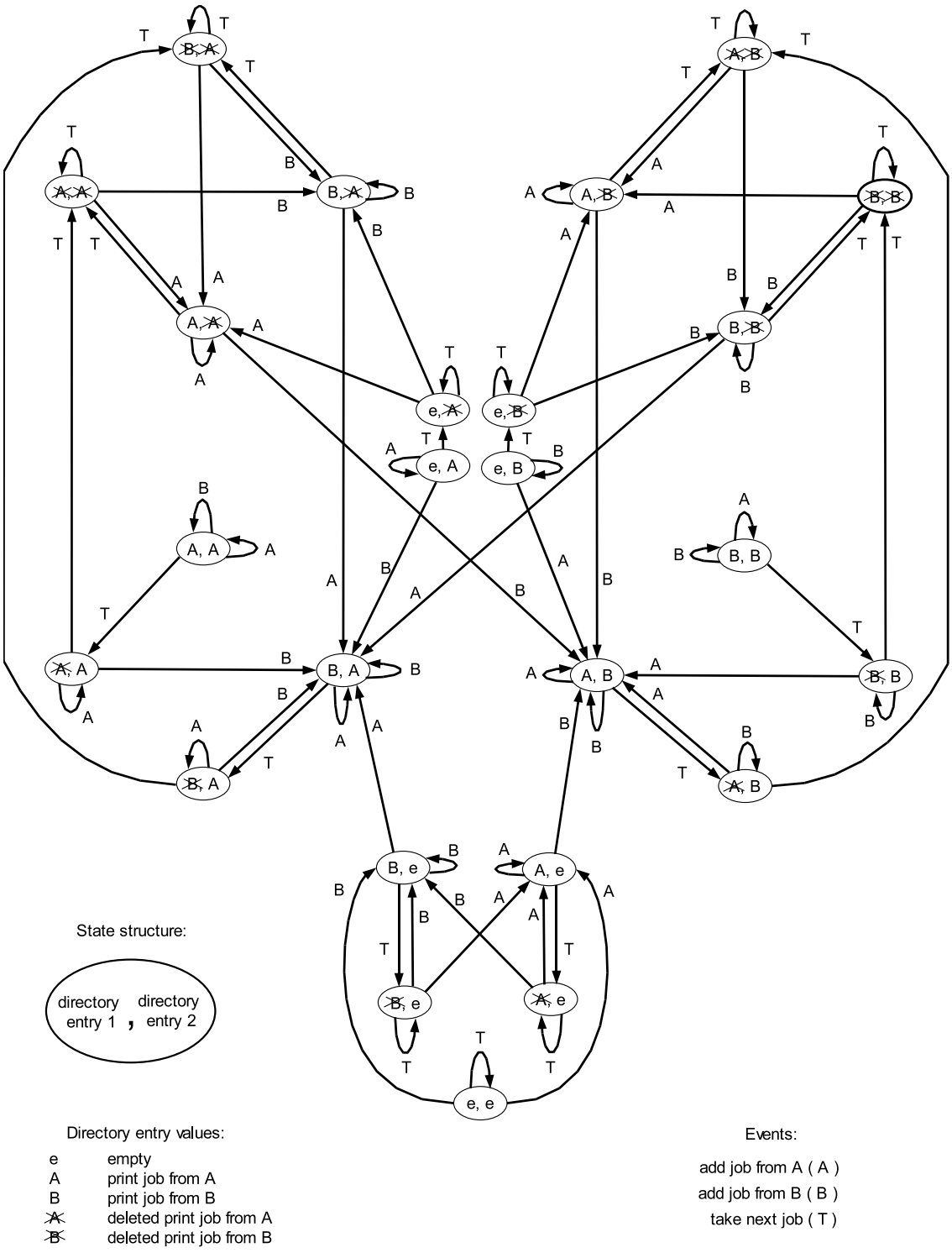


Figure 6.2: Transition graph of the print job directory model

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 backtracing 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 the (A,B) state, which means that Alice is lying.

6.1.3 Evidential statements

As illustrated by the foregoing example, finite state machines can be used as a basis for automatic event reconstruction. However, finite state machines alone are insufficient to completely automate the event reconstruction process. It is also necessary to formalise the available evidence, such as Carl's observation of the final state of the print job directory. So in addition to using state machines to model system functionality, this dissertation defines the *evidential statement* notation for describing the evidence about an incident.

The idea of evidential statements is to formalise pieces of evidence as statements about the properties and change of system state in the past.

Consider, for example, Carl's observation of the print job directory. The knowledge of the incident that he obtained *directly* from the examination of the printer can be described as follows:

1. the state of the print job directory at the moment of examination was (⊠, ⊠)

2. before the print job directory reached that state, it visited zero or more states about which nothing is evident from the examination (it could have been any states).

Similarly, the manufacturer's knowledge of the initial state of the print job directory can be described as follows:

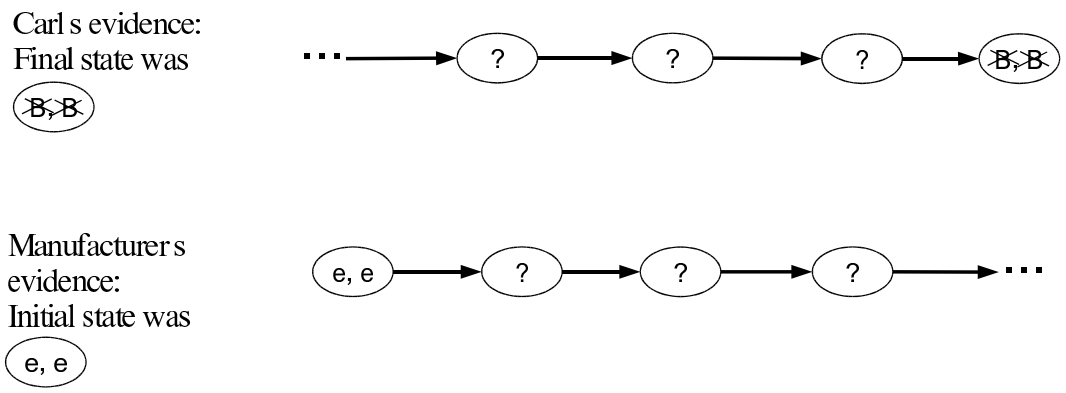


Figure 6.3: Evidence in ACME investigation

1. the initial state of the print job directory was (e, e),

2. after that, the print job directory could have visited zero or more states, but the manufacturer knows nothing about those states.

Both descriptions are illustrated graphically in Figure 6.3. Observe that, essentially, both pieces of evidence restrict possible sequences of transitions that could have occurred during the incident. As a result, event reconstruction can be viewed as the process of finding all sequences of transitions that satisfy these restrictions.

Motivation for the development of new formal notation

Checking that computations of a given state machine satisfy given set of restrictions is the basic problem of model checking. Since both digital forensics and model checking are concerned with the analysis of discrete digital systems, it may seem feasible to use existing formal verification methods in digital investigations. Additional argument for using these methods is that the goals of *some* investigations can be formulated as verification problems (in ACME investigation, for example, the goal is to verify that Alice never printed anything). There are however, considerable problems with the use of existing formal verification methods in forensic context:

1. *Insufficiency of explanations.* The output of formal verification is basically a "true" or "false" answer with respect to the given logical formula. Although model checking tools do provide a counterexample if the formula is false, little other information is given.

   At the same time, forensic analysis of evidence is expected to produce more than a simple "yes" or "no" answer. When preparing for the court hearing, for example, attorneys may want to know about alternative explanations of the available evidence. At the trial, the expert may be challenged to give a comprehensive explanation of how the particular piece of evidence fits with the facts in issue.

   As a result, formal methods in digital forensics should provide more informative explanation of how possible scenarios are linked to the evidence.

2. *The absence of the notion of evidence from formal verification.* The concept of evidence is fundamental in the legal context. Apart from restricting possible sequences of transitions, a piece of evidence can have its own properties related to its discovery. For example, an eyewitness observation may have real-world time associated with it, which may be used to perform event time bounding analysis described in Section 3.2.3. The existing formal verification methods do not provide explicit ways to represent evidence and to reason about its properties.

As a result of the aforementioned problems, it has been decided to create a new formal notation rather than use an existing one. The new notation provides explicit representation for evidence, and defines the basic analytical problem as finding the set of all possible explanations of the given evidence.

6.1.4 Assumption about reliability of evidence

The importance of evidence reliability has been highlighted in the previous section and in Chapter 2. However, due to limited timeframe of this research

work, it has been decided not to address the issue of evidence reliability in this research. Throughout the rest of this dissertation, it is implicitly assumed that all evidence is absolutely reliable. This assumption simplifies the problem of event reconstruction by avoiding reasoning with uncertainty. *This simplification is justified, because the evidence may be incorporated into analysis gradually. First the investigator can perform formal analysis with only the most reliable evidence. If results are unsatisfactory, the analysis can be repeated with less reliable evidence included.*

6.2 Formalisation of event reconstruction problem

This section formally defines the event reconstruction problem. The definition 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
- Evidence — description of the system's final state and clues to the system's behaviour in the past, such as witness statements, printouts, etc.

The proposed definition represents the knowledge of the system functionality as a finite state machine and uses *evidential statement* notation for describing the evidence and investigative assumptions. The event reconstruction is defined as finding all possible explanations for the given evidential statement with respect to the given finite state machine. Appendix C.1 contains formalisation of the evidential statement notations and related notions in ACL2 logic.

6.2.1 Finite state machine

Finite state machine is a tuple of four elements $T = (Q, I, \phi)$, 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 event and state.

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

A *(finite) computation* is a non-empty, finite sequence of steps, 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:

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

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

Observe that, since there is no upper bound on the possible length of a computation, C_T is infinite.

6.2.2 Run

To formalise transition backtracing, 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 6.4, which graphically illustrates this concept.

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 integer $1 \leq i < |r|$, $r_i = \psi(r_{i-1})$, 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.

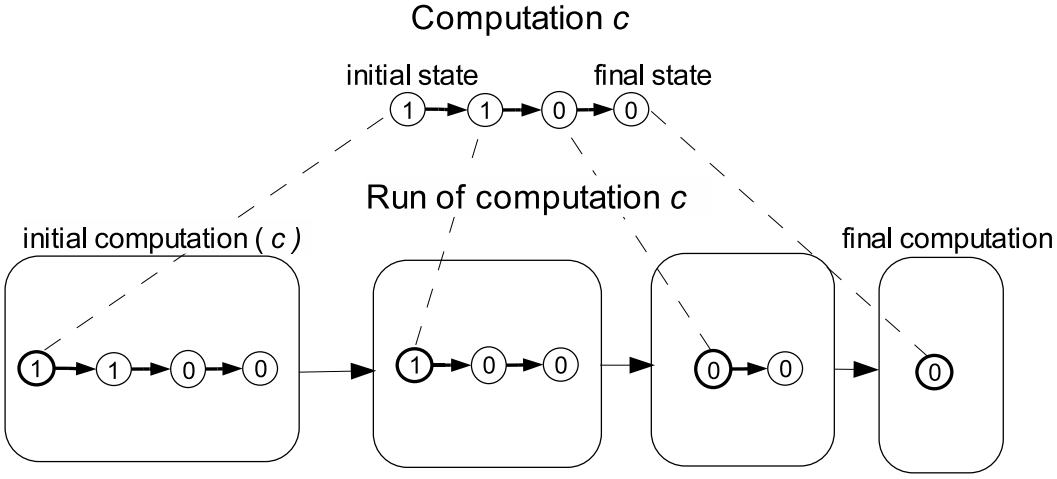


Figure 6.4: Run of computation

6.2.3 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:

$$(pr_0 \cdot pr_1 \cdot pr_2 \cdot \ldots \cdot pr_{|pr|-1}) \in R_T$$

The *set of all partitioned runs* of the finite state machine T is denoted PR_T.

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

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

6.2.4 Formalisation of backtracing

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:

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

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

Although function ψ^{-1} can be used to formalise backtracing, 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:

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

The meaning of functions ψ, ψ^{-1}, and Ψ^{-1} is illustrated in Figure 6.5.

Backtracing of computations is defined as a finite number of compositions Ψ^{-1} applied to a subset of computations:

$$\Psi^{-1}(\Psi^{-1}(\ldots \Psi^{-1}(Y)\ldots))$$

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

6.2.5 Formalisation 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 formalisation presented below.

Observation

Observation is a statement that system behaviour exhibited some property *p continuously for some time*. Syntactically, observation is 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 restrict duration of observation. Informally speaking, observation o characterises a set of runs

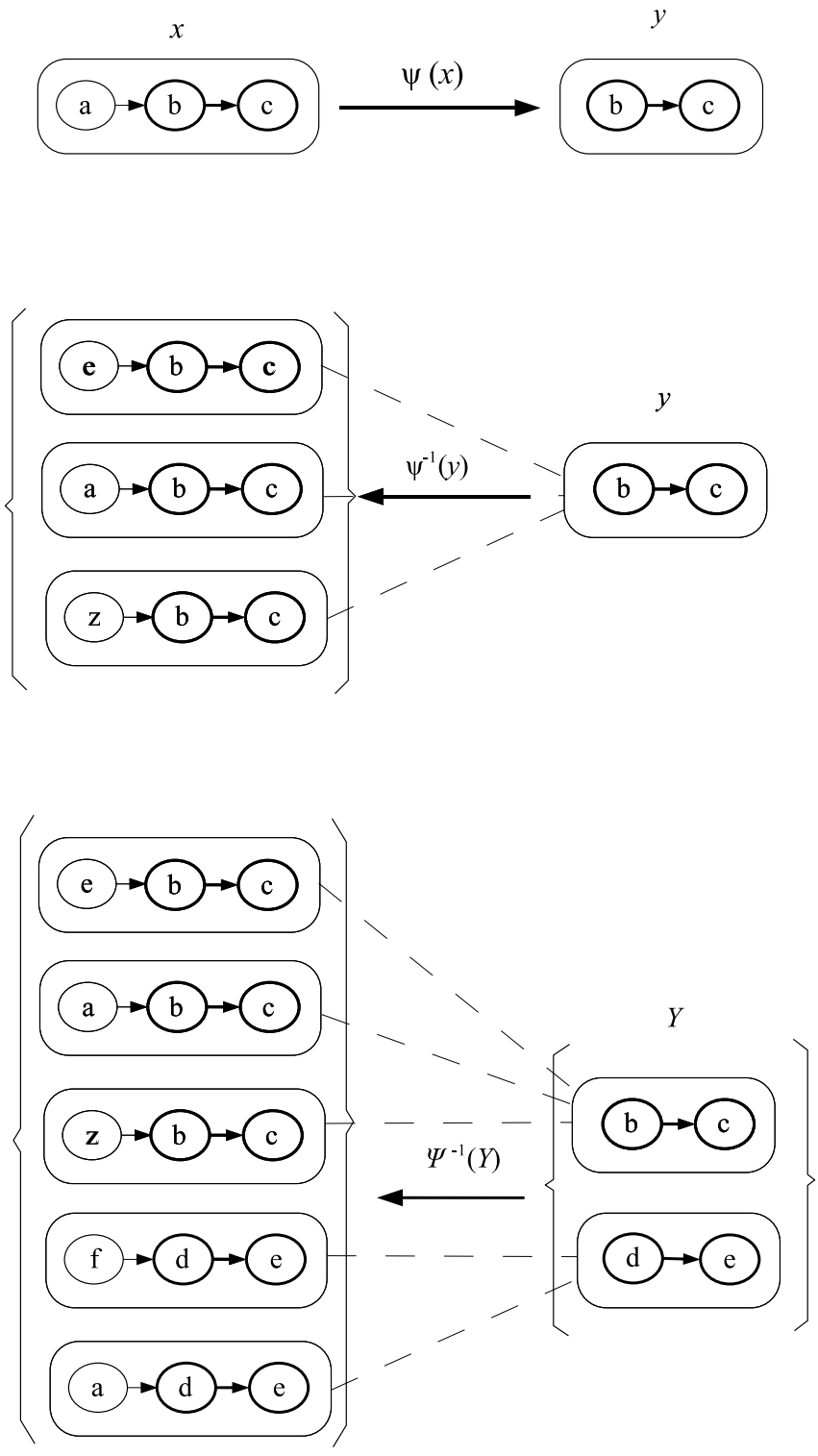


Figure 6.5: Functions ψ, ψ^{-1}, and Ψ^{-1}

R_o, whose lengths are limited by *min* and *opt* and whose computations satisfy P.

Since witness observations are external to the system, transitions that do not change observed property of the state are invisible to the witness. It means that, in general, a sequence of states could have been observed rather than a single state. Elements *min* and *opt* restrict the length of runs comprising R_o. Element *min* specifies the minimal length of runs in $r \in R_o$, and *opt* specifies maximal "excess" of length in addition to *min*:

$$min \leq |r| \leq (min + opt)$$

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

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

A note on *min* and *opt*. The introduction of restrictions on the length of observations is motivated by the following reason. Although the witness may not always tell how many transitions have been observed, it is sometimes possible to set a limit. Consider Carl's observation of the print job directory. The length of run is at least one, because the deleted print jobs from Bob were actually observed. Moreover, the length of runs corresponding to the final state observation is exactly one, because there were no more transitions from the final state.

The upper limit on the length of observation is introduced to ensure termination of reconstruction process. The introduction of the upper limit is permissible, because digital forensic analysis reconstructs only final computations. Therefore, introduction of a sufficiently large upper limit can be used to model infinity. The constant *infinitum* is introduced for this purpose:

The *infinitum* is an integer constant that is greater than the length of any computation that may have happened during the incident.

Types of observations. 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 $(C_T, 0, \textit{infinitum})$ that puts no restrictions on computations that could have happened during the incident.

Observation sequence

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

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

Informally, 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*:

$$\text{for all } 0 \leq i < |os|, \quad pr_i \in R_{os_i}$$

Note that the same run can explain the same observation sequence in a number

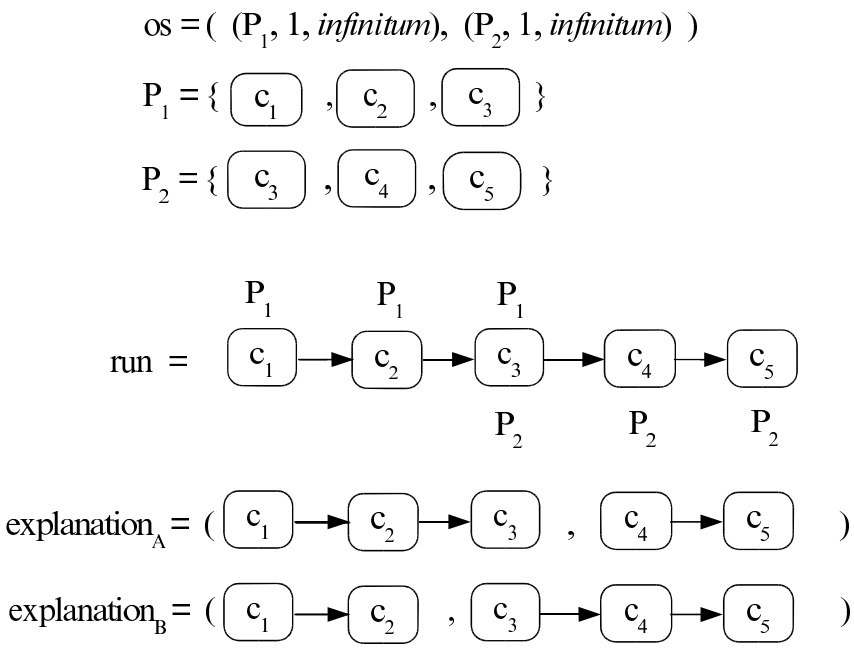


Figure 6.6: A run that gives two explanations to an observation sequence

of ways, each corresponding to a different partitioning of the run. Figure 6.6 illustrates this possibility of multiple explanations.

The *meaning of observation sequence* os is the set $PR_{os} \subseteq (R_T)^{|os|}$ 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 $r_0 = c$.

Evidential statement

Evidential statement is a non-empty sequence of observation sequences

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

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:

$$\begin{aligned} & spr_{00} \cdot spr_{01} \cdot \ldots \cdot spr_{0|spr_0|-1} = \\ = & \; spr_{10} \cdot spr_{11} \cdot \ldots \cdot spr_{1|spr_1|-1} = \\ & \vdots \\ = & \; spr_{|es|-1_0} \cdot spr_{|es|-1_1} \cdot \ldots \cdot spr_{|es|-1_{|spr_{|es|-1}|-1}} = r \end{aligned}$$

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*:

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

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

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

Figure 6.7 illustrates the relationship between the evidential statement and other formal notions introduced in this section.

Definition of event reconstruction problem

In terms of the above defined formalisation 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.

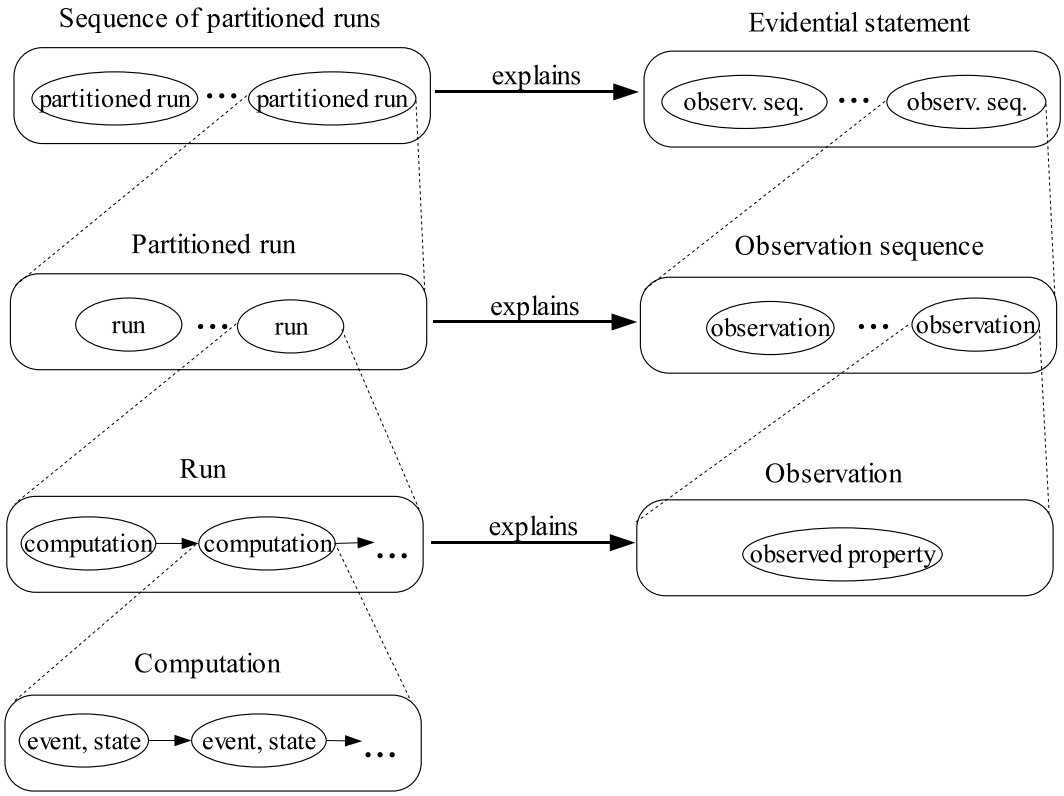


Figure 6.7: Evidential statement and related notions

A note on the use of *infinitum* The *infinitum* constant is designed as a "no-limit" constant for the *opt* parameter of observations. It is possible, therefore, that several observations in an observation sequence will have *infinitum* as their *opt* parameter. Such observation sequence may have explanations whose lengths are several times longer than *infinitum*, because each "no-limit" observation may have explaining run, whose length is *infinitum*.

By definition given in Section 6.2.5, *infinitum* is greater than any computation than might have happened during the incident. As a result, there is no practical reason to calculate the entire SPR_{es}. If *infinitum* is used and is chosen correctly, it should suffice to calculate only a part of SPR_{es} that includes all explanations of es, whose total length is less then *infinitum*.

6.3 Summary

This chapter has demonstrated that event reconstruction in digital investigations can be formalised using state machine model of computation. The following approach to event reconstruction has been proposed.

1. Create a finite state model of the system under investigation and formalise the evidence in terms of that model.

2. Use transition backtracing or any other suitable method to find all sequences of transitions that agree with the formalised evidence.

This chapter was primarily concerned with the step one of this approach. To allow formalisation of evidence, it defined evidential statement notation. The problem of event reconstruction has been defined as finding all possible explanations of the given evidential statement with respect to the given finite state machine.

The next chapter addresses the second step of the proposed approach to event reconstruction. It presents an algorithm that computes the meaning of

the given evidential statement. It also analyses the complexity of the algorithm and describes a "proof-of-concept" implementation of the algorithm in Common Lisp.

Chapter 7

Event reconstruction algorithm

The existing algorithms for analysis of finite state models of computing systems were developed for the purpose of systems verification [28]. The event reconstruction problem formulated in Chapter 6 is different from verification problems. Instead of *checking* that certain type of computations is impossible in the system, event reconstruction aims to *construct* possible computations that explain available evidence. As shown in Chapter 5, verification algorithms avoid construction of computations for efficiency reasons. They cannot be used directly to solve the event reconstruction problem.

This chapter describes an algorithm for solving the event reconstruction problem. It computes the meaning of the given evidential statement with respect to the given finite state machine. Since no such algorithm previously existed, it was decided to construct a simple algorithm, whose properties can be easily analysed. Performance improvement of the algorithm is left for future work.

The algorithm is described in three 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. An

upper bound on the running time of the event reconstruction algorithm is then derived in Section 7.4, and a "proof-of-concept" implementation of the algorithm is described in Section 7.5.

7.1 Computing the meaning of a fixed-length observation sequence

Recall function Ψ^{-1} introduced in Section 6.2.4. It takes a set of computations $Y \subseteq C_T$ and produces the set of all computations, whose tails are in Y. In other words, it returns all possible backtracings of computations in Y.

Function Ψ^{-1} provides basic operation for automation of backtracing. 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 backtrace 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 backtracings 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 7.1, 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*.

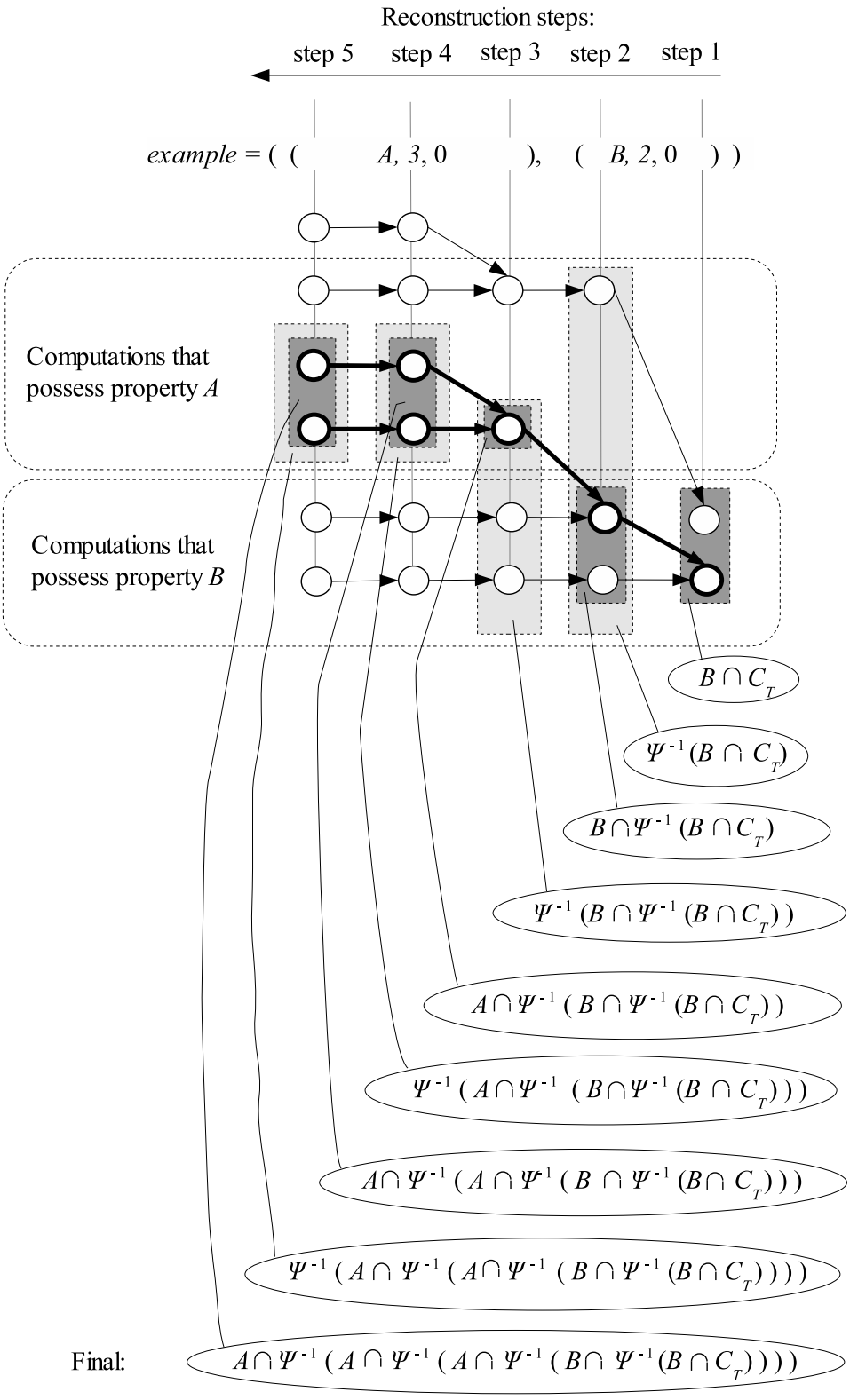


Figure 7.1: Finding explanations of a fixed-length observation sequence

```
1: function SolveFOS(fos)
2:     C_next ← C_T
3:     len ← ϵ
4:     for i ← (|fos| − 1) to 0 step -1 do
5:         observation (P, l, 0) ← fos_i
6:         len ← (l) · len
7:         for j ← 0 to l − 1 step 1 do
8:             C ← C_next ∩ P
9:             C_next ← Ψ^{-1}(C)
10:        end for
11:    end for
12:    return (C, len)
13: end function
```

Figure 7.2: Computing the meaning of a fixed-length observation sequence

A *map of partitioned runs (MPR)* is a representation for a set of partitioned runs. It is a tuple $pm = (C, len)$ where C is the set of initial computations, and 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 $len_0, len_1, \ldots len_{|len|-1}$. Observe that the meaning of a fixed length observation sequence can be expressed by a single MPR.

The algorithm for computing the meaning of the given fixed-length observation sequence is presented in Figure 7.2. It implements the idea described above and returns an MPR that expresses the meaning of the given fixed-length observation sequence.

7.2 Computing the meaning of a generic observation sequence

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 explain-

ing 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 steps,
- 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:

$$\begin{array}{rrr} AB & ABB & ABBB \\ AAB & AABB & AABBB \\ AAAB & AAABB & AAABBB \\ AAAAB & AAAABB & AAAABBB \end{array}$$

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 in three steps:

1. Convert *example2* to a set of fixed-length observation sequences.
2. Calculate the meaning of each fixed-length observation sequence 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.

The algorithm for computing the meaning of a generic observation sequence is given in Figure 7.3. The algorithm consists of two parts. First, lines 2–13 convert the given observation sequence *os* into a set of fixed-length observation

```
 1: function SolveOS(os)
 2:     F ← {ϵ}
 3:     for i ← 0 to |os| − 1 step 1 do
 4:         observation (P, min, opt) ← os_i
 5:         F_new ← ∅
 6:         for each partially constructed sequence f in F do
 7:             for j ← 0 to opt step 1 do
 8:                 f_new ← f · ( (P, min + j, 0) )
 9:                 F_new ← F_new ∪ {f_new}
10:             end for
11:         end for
12:         F ← F_new
13:     end for
14:     PM_os ← ∅
15:     for each fos in F do
16:         pm ← SolveFOS(fos)
17:         PM_os ← PM_os ∪ {pm}
18:     end for
19:     return PM_os
20: end function
```

Figure 7.3: Computing the meaning of a generic observation sequence

sequences F. After that, lines 14–18 use *SolveFOS* algorithm to compute the meaning of each fixed-length observation sequence in F. The resulting set of MPRs is returned in line 19.

7.3 Computing the meaning of an evidential statement

The meaning of an evidential statement can be computed 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 the initial computation sets of both MPRs: $r \in C_a$ and $r \in C_b$, and

2. both MPRs have equal total number of computation steps: $\Sigma\, len_a = \Sigma\, len_b$.

Clearly, if $\Sigma\, len_a \neq \Sigma\, len_b$, then the lengths of all computations represented by pm_a are different from the lengths of all computations represented by pm_b, and 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)* $spm = (C, (len_0, \ldots, len_n))$ is a representation for a set of sequences of partitioned runs. C is the set of initial computations, and $len_0, \ldots\ len_n$ are lists of lengths that describe how to partition runs generated from the elements of C. MSPR is *proper* if and only if $\Sigma\, len_0 = \ldots = \Sigma\, len_n$.

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

$$comb(pm_a, pm_b) = \begin{cases} \emptyset & \text{, if } \Sigma\, len_a \neq \Sigma\, len_b \text{ or} \\ & \quad C_a \cap C_b = \emptyset \\ (\, C_a \cap C_b, (len_a, len_b), \,) & \text{, otherwise} \end{cases}$$

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

```
 1: function SolveES(es)
 2:     SPM_es ← ∅
 3:     os ← es_0
 4:     PM_os ← SolveOS(os)
 5:     for each pm = (C, len) in PM_os do
 6:         SPM_es ← SPM_es ∪ {(C, (len))}
 7:     end for
 8:     for i ← 1 to |es| − 1 step 1 do
 9:         os ← es_i
10:         PM_os ← SolveOS(os)
11:         SPM_new ← ∅
12:         for each spm = (C_a, lenlist) in SPM_es do
13:             for each pm = (C_b, len) in PM_os do
14:                 C ← C_a ∩ C_b
15:                 if C ≠ ∅ and Σlen = Σlenlist_0 then
16:                     SPM_new ← SPM_new ∪ { (C, lenlist · (len_a) ) }
17:                 end if
18:             end for
19:         end for
20:         SPM_es ← SPM_new
21:     end for
22:     return SPM_es
23: end function
```

Figure 7.4: Computing the meaning of an evidential statement

PM_b :

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

This process can be extended to arbitrary number of observation sequences, thus providing a way to calculate meaning of an arbitrary evidential statement. The corresponding event reconstruction algorithm is given in Figure 7.4. The operation of the algorithm is divided into two parts. First, lines 2–7 compute the meaning of the first observation sequence in the evidential statement and make it the initial value for SPM_{es}. After that, the loop in lines 8–21 computes the meaning of the remaining observation sequences in the evidential statement and combines their meanings with SPM_{es}.

7.4 Running time of event reconstruction algorithm

This section derives an upper bound[1] on the running time of the event reconstruction algorithm *SolveES* described in the previous section.

7.4.1 Prefix based representation of computation sets

The running time of the event reconstruction algorithm cannot be estimated without first estimating the running time required for its basic operations $\Psi^{-1}(X)$ and $X \cap Y$. Their running times, in turn, depend on the chosen representation of computation sets.

For the purpose of this analysis, a *prefix based* representation of computation sets has been adopted. A prefix based representation of a computation set is a list

$$L_X = (x_0, \ldots, x_{|L_X|-1})$$

where $x_i \in C_T$, and $|x_i| > 0$.

The elements of L_X are called *prefixes*. Each prefix represents the set of all computations that begin with it. For example, the list of prefixes $(\,(a,b,c),\,(d,e)\,)$ represents the set of all computations of the forms $(a,b,c,\ldots)$ or $(d,e,\ldots)$. [2] This set includes, for example, computations (a,b,c), (a,b,c,d), (a,b,c,e,d), (d,e), (d,e,a). More precisely, the set of computations repre-

[1] Note that O-notation is not used in this section. This is because the running time of the event reconstruction algorithm depends on six parameters, and (since it is not clear which parameter grows faster) the use of O-notation does not bring much clarity into the resulting expression.

[2] Symbols a, b, c, d, and e stand for state–event pairs of the form (q, ι), where $q \in Q$ and $\iota \in I$.

sented by a list of prefixes L_X is

$$X = \bigcup_{i=0}^{|L_X|-1} \{\, c \,|\, c \in C_T,\, |x_i| \leq |c|, \text{ and for all integer } 0 \leq j < |x_i| \;:\; c_j = (x_i)_j \,\}$$

Appendix B discusses properties of the prefix based representation and gives algorithms for computing $\Psi^{-1}(X)$ and $X \cap Y$ of sets of computations represented as lists of prefixes. To derive an upper bound on the running time of the event reconstruction algorithm, note that the prefix based representations have the following properties.

Prefix based representation of C_T. The set C_T can be represented as a list of all possible singleton prefixes L_{C_T}, whose length is

$$|L_{C_T}| = |Q||I| \tag{7.1}$$

where $|Q|$ and $|I|$ are sizes of sets of states and events respectively.

Size of prefix based representations with limited lengths of prefixes. Any set of computations P_m that restricts only the first m elements of its member computations can be represented by a list L_{P_m}, whose length is bounded by the number of all possible computation prefixes of length m:

$$|L_{P_m}| \leq |Q||I|^m \tag{7.2}$$

except for $m = 0$.

Checking set emptiness is a constant time operation. Checking emptiness of a set of computations C represented as a list of prefixes L_C amounts to checking emptiness of L_C, which can be performed in constant time.

Upper bound on the time required to compute set intersection. The time $t_{X \cap Y}$ required to compute the set intersection of two computation sets X and Y represented as lists of prefixes L_X and L_Y is bounded by

$$t_{X \cap Y} \leq cm|L_X||L_Y| \tag{7.3}$$

where c is an implementation-dependent constant, and m is the length of the longest prefix in either L_X or L_Y.

Upper bound on the length of the output of set intersection. The length $|L_{X \cap Y}|$ of the list of prefixes that represents the set intersection of two computation sets X and Y is bounded by

$$|L_{X \cap Y}| \leq |L_X||L_Y| \tag{7.4}$$

where $|L_X|$ and $|L_Y|$ are lengths of lists of prefixes that represent sets X and Y respectively.

Upper bound on the time required to compute $\Psi^{-1}(X)$. The time required to compute $\Psi^{-1}(X)$ is bounded by

$$t_{\Psi^{-1}(X)} \leq c|Q||I||L_X| \tag{7.5}$$

where c is an implementation dependent constant, $|L_X|$ is the length of the list of prefixes that represents the set X, and $|Q|$ and $|I|$ are sizes of sets of states and events respectively.

Upper bound on the length of the output of $\Psi^{-1}(X)$. The length $|L_{\Psi^{-1}(X)}|$ of the list of prefixes that represents the output of $\Psi^{-1}(X)$ is bounded by

$$|L_{\Psi^{-1}(X)}| \leq |Q||I||L_X| \tag{7.6}$$

where $|L_X|$ is the length of the list of prefixes that represents the set X, and $|Q|$ and $|I|$ are sizes of sets of states and events respectively.

7.4.2 An upper bound on the running time of *SolveFOS*

First, note that operations in lines 1–7 and 10–12 of the *SolveFOS* algorithm can be implemented to take constant time. The running times of lines 8 and 9 are bounded by inequalities 7.3 and 7.5 respectively.

Let l_{max} be an upper bound on the lengths of observations in the input observation sequence *fos*

$$\text{for all } 0 \leq i < |fos|, \quad l_i \leq l_{max} \tag{7.7}$$

where l_i is the length of the i^{th} observation in *fos*.

Let p be an upper bound on the length of prefixes used in prefix based representations of all observed properties in *fos*. Then, according to the inequality 7.2, the lengths of prefix based representations of observed properties are bounded by

$$\text{for all } 0 \leq i < |fos|, \quad |L_{P_i}| \leq |Q||I|^p \tag{7.8}$$

where L_{P_i} is a prefix based representation of the observed property P_i of the i^{th} observation in *fos*.

The running time of *SolveFOS* is equal to the following sum

$$t_{SolveFOS} = c_1 + \sum_{i=0}^{|fos|-1} \Big(c_2 + \sum_{j=0}^{l_i-1} \big(c_3 + t_{backtr_k} \big) \Big) \tag{7.9}$$

where

- c_1 represents the constant time spent in lines 1–3, 12, and in loop setup in line 4,

- c_2 represents the constant time spent in every iteration of the outer loop in lines 4–6, 11, and in loop setup in line 7,
- c_3 represents the constant time spent in every iteration of the inner loop in lines 7 and 10,
- t_{backtr_k} represents the time of the k^{th} reconstruction step performed in lines 8 and 9, where $k = j + \sum_{n=i+1}^{|fos|-1} l_n$, where l_n is the length of the n^{th} observation in fos.

The running time of the k^{th} reconstruction step is the sum of the running times of set intersection and backtracing. By monotonicity of addition and by inequalities 7.3 and 7.5, it is bounded by

$$t_{backtr_k} \leq c_4 p |L_{C_{next_k}}||L_{P_i}| + c_5 |Q||I||L_{(C_{next_k} \cap P_i)}| \tag{7.10}$$

where

- C_{next_k} is the "input" set of computations at the k^{th} reconstruction step,
- l_i is the length of observation fos_i,
- P_i is the observed property of observation fos_i,
- c_4 and c_5 are implementation dependent constants

Observe that the right-hand side of inequality 7.10 depends on the lengths of representations $L_{C_{next_k}}$ and $L_{(C_{next_k} \cap P_i)}$. To compute these values, note that $C_{next_{k+1}}$ is linked with C_{next_k} according to the following recurrence

$$\begin{aligned} C_{next_0} &= C_T \\ C_{next_{k+1}} &= \Psi^{-1}(C_{next_k} \cap P_i) \end{aligned}$$

An upper bound on the length of $L_{C_{next_{k+1}}}$ can be derived from this recurrence

using equation 7.1 and inequalities 7.4, and 7.6:

$$|L_{C_{next_0}}| = |L_{C_T}| = |Q||I|$$
$$|L_{C_{next_{k+1}}}| \leq |Q||I|(|L_{C_{next_k}}||Q||I|^p)$$

Solving this recurrence for k produces

$$|L_{C_{next_k}}| \leq |Q||I|(|Q|^2|I|^{p+1})^k = |Q|^{2k+1}|I|^{kp+k+1} \tag{7.11}$$

After substituting this result into the inequality 7.10, replacing $|L_{P_i}|$ and $|L_{(C_{next_k} \cap P_i)}|$ according to inequalities 7.8 and 7.4, and simplifying, the upper bound on the time of the k^{th} reconstruction step becomes

$$t_{backtr_k} \leq (c_4 p + c_5|Q||I|)(|Q|^{2(k+1)}|I|^{(k+1)(p+1)}) \tag{7.12}$$

An upper bound on the running time of *SolveFOS* can be obtained from the right-hand side of equation 7.9 by replacing t_{backtr_k} and l_i according to inequalities 7.12 and 7.7. After simplification, it becomes

$$t_{SolveFOS} \leq \Lambda + (c_4 p + c_5|Q||I|)\left(\frac{(|Q|^2|I|^{p+1})^{|fos|l_{max}+1} - (|Q|^2|I|^{p+1})}{(|Q|^2|I|^{p+1}) - 1}\right) \tag{7.13}$$

where

$$\Lambda = c_1 + c_2\,|fos| + c_3\,|fos|\,l_{max}$$

An upper bound on the length of the output of *SolveFOS* can be derived from the upper bound on the length of $L_{C_{next_k}}$ at the last step of event reconstruction using inequalities 7.11 and 7.4. After simplification, the resulting upper bound is

$$|L_{SolveFOS}| \leq (|Q|^2|I|^{p+1})^{|fos|l_{max}} \tag{7.14}$$

7.4.3 An upper bound on the running time of *SolveOS*

First, note that operations in lines 1–15 and 17–19 of the *SolveOS* algorithm can be implemented to take constant time. The execution of the line 16 requires time, whose upper bound is given by inequality 7.13.

Second, note that by definition of *infinitum* given in Chapter 6, the length of any computation that may have happened during the incident is bounded by *infinitum*. As a result, the length of any observation made during the incident is also bounded by *infinitum*, and

$$\text{for all } 0 \leq i < |os| \quad (min_i + opt_i) \leq infinitum \tag{7.15}$$

where min_i and opt_i are *min* and *opt* parameters of the i^{th} observation in the input observation sequence *os*.

The running time of *SolveOS* is equal to the following sum

$$t_{SolveOS} = c_6 + \sum_{i=0}^{|os|-1} (c_7 + \sum_{l=0}^{|F_i|-1} (c_8 + \sum_{j=0}^{opt_i} c_9)) + \sum_{l=0}^{|F_{|os|}|-1} (c_{10} + t_{SolveFOS}) \tag{7.16}$$

where

- c_6 represents the constant time spent in lines 1–2, 14, and 19, and in loop setups in lines 3 and 15,
- c_7 represents the constant time spent in every iteration (of the loop in lines 3–13) in lines 3–5, 12–13, and in loop setup in line 6
- c_8 represents the constant time spent in every iteration (of the loop in lines 6–11) in lines 6, and 11, and in loop setup in line 7,
- c_9 represents the constant time spent in every iteration of the inner loop in lines 7–10,
- F_i is the set of partially constructed fixed-length observation sequences at the beginning of the i^{th} iteration of the loop in lines 3–13,

- $F_{|os|}$ is the final set of fixed-length observation sequences produced by the $(|os|-1)^{\text{th}}$ iteration of the loop in lines 3–13,
- c_{10} represents the constant time spent in every iteration (of the loop in lines 15–18) in lines 15, 17, and 18.

The time of the i^{th} iteration of the outer loop in lines 3–13 depends on the size of F_i generated in the previous iteration. To obtain an upper bound on the size of F_i note that every iteration of the inner loop in lines 7–10 creates one new element for F_{i+1}, and that opt_i is bounded by the inequality 7.15. Thus, the size of F_i is bounded by the following recurrence

$$\begin{aligned} |F_0| &= 1 \\ |F_{i+1}| &\leq |F_i| \mathit{infinitum} \end{aligned}$$

Solving this recurrence for i produces

$$|F_i| \leq \mathit{infinitum}^i \tag{7.17}$$

An upper bound on $t_{SolveOS}$ can be derived from 7.16 by replacing $|F_i|$ and opt_i using inequalities 7.17 and 7.15. The resulting inequality after simplification becomes

$$t_{SolveOS} \leq c_6 + c_7\,|os| + (c_8 + c_9\,\mathit{infinitum})\left(\frac{\mathit{infinitum}^{|os|+1} - 1}{\mathit{infinitum} - 1}\right) + \sum_{l=0}^{\mathit{infinitum}^{|os|}} (c_{10} + t_{SolveFOS}) \tag{7.18}$$

Note that the length of any fixed-length observation sequence *fos* in $F_{|os|}$ is the same as the length of the original observation sequence:

$$|\mathit{fos}| = |os|$$

and that the length l of any observation in *fos* is bounded by inequality 7.15.

As a result, $t_{SolveFOS}$ in the right hand side of 7.18 can be replaced with the right hand side of inequality 7.13 with $|fos|$ replaced by $|os|$ and l_{max} replaced by *infinitum*. After simplification the resulting inequality becomes

$$t_{SolveOS} \leq c_6 + c_7 \left|os\right| + (c_8 + c_9 \left(infinitum + 1\right)) \left(\frac{infinitum^{|os|} - 1}{infinitum - 1} \right) + c_{10}\, infinitum^{|os|} + \Upsilon \quad (7.19)$$

where Υ represents the time spent on computing the meanings of individual fixed-length observation sequences:

$$\Upsilon = infinitum^{|os|} \left(c_1 + c_2 \left|os\right| + c_3 \left|os\right| infinitum + (c_4 p + c_5 |Q||I|) \left(\frac{(|Q|^2|I|^{p+1})^{|os|\, infinitum + 1} - (|Q|^2|I|^{p+1})}{(|Q|^2|I|^{p+1}) - 1} \right) \right)$$

An upper bound on the number of elements in the output of *SolveOS* can be derived as follows. Note that each iteration of the loop in lines 15–18 creates one element of the output set. Thus, by inequality 7.17, the number of elements in the output of *SolveOS* is bounded by

$$|SolveOS(os)| \leq infinitum^{|os|} \quad (7.20)$$

7.4.4 An upper bound on the running time of *SolveES*

First, note that operations in lines 1–3, 5–9, 11–13, and 15–22 of the *SolveES* algorithm can be implemented to take constant time. The running times of lines 4 and 10 are bounded by inequality 7.19. The running time of line 14 is bounded by inequality 7.3.

Let os_{max} be the upper bound on the lengths of observation sequences in

the input evidential statement es

$$\text{for all } 0 \leq i < |es|, \quad |es_i| \leq os_{max} \tag{7.21}$$

where es_i is the i^{th} observation sequence in es.

The running time of the event reconstruction algorithm *SolveES* depends on the number of observation sequences in the evidential statement es. For the purpose of the worst case analysis, it suffices to consider only evidential statements that consist of two or more observation sequences (i.e. $2 \leq |es|$), in which case the running time of *SolveES* is bounded by the following sum

$$t_{SolveES} \leq c_{11} + t_{SolveOS(es_0)} + \sum_{i=0}^{|SolveOS(es_0)|-1} c_{12} + \sum_{i=1}^{|es|-1} \Big(c_{13} + t_{SolveOS(es_i)} + \sum_{j=0}^{|SPM_i|-1} \big(c_{14} + \sum_{l=0}^{|SolveOS(es_i)|-1} (\ c_{15} + t_{(C_{a_{j_i}} \cap C_{b_l})} \ ) \big) \Big) \tag{7.22}$$

where

- c_{11} represents the constant time spent in lines 1–3, 22, and in loop setups in lines 5 and 8,
- c_{12} represents the constant time spent in every iteration of loop in lines 5–7,
- c_{13} represents the constant time spent in every iteration (of the loop in lines 8–21) in lines 8–9, 11, 20–21, and in loop setup in line 12,
- c_{14} represents the constant time spent in every iteration (of the loop in lines 12–19) in lines 12, 19, and in loop setup in line 13,
- c_{15} represents the constant time spent in every iteration (of the loop in lines 13–18) in lines 13, and 15–18,

- SPM_i is the value of SPM_{es} at the beginning of the i^{th} iteration of the loop in lines 8–21; it is the set of MSPRs obtained by combining the first i elements of es,

- $t_{SolveOS(es_i)}$ represents the time spent on computing the meaning of the i^{th} element of the evidential statement es,

- $t_{(C_{a_{j\,i}} \cap C_{b_l})}$ represents time spent on computing set intersection of the computation sets of spm and pm on the i^{th} iteration of the loop in lines 8–21, on the j^{th} iteration of the loop in lines 12–19, and on the l^{th} iteration of the loop in lines 13–18.

An upper bound on the running time of *SolveES* can be derived from the equation 7.22 using inequalities 7.21, 7.3, 7.19, 7.20, and 7.14. After replacing $|es_i|$ and $|SolveOS(es_i)|$ according to inequalities 7.21 and 7.20 respectively, moving common factors to the outside of summations, and initial simplification, the upper bound becomes

$$t_{SolveES} \leq c_{11} + c_{12}\, \mathit{infinitum}^{os_{max}} + c_{13}\,(|es| - 1) + \sum_{i=0}^{|es|-1} t_{SolveOS(es_i)} + \sum_{i=1}^{|es|} \Big(\sum_{j=0}^{|SPM_i|} \big(c_{14} + \sum_{l=0}^{\mathit{infinitum}^{os_{max}}} (c_{15} + t_{(C_{a_{j\,i}} \cap C_{b_l})})\big)\Big) \quad (7.23)$$

The value of the triple-nested sum depends on $|SPM_i|$ and on the time $t_{(C_{a_{j\,i}} \cap C_{b_l})}$. To derive an upper bound on $|SPM_i|$ note that the loop in lines 13–18 creates at most $|SolveOS(es_i)|$ elements of SPM_{i+1} for each element of SPM_i. Since the value of $|SolveOS(es_i)|$ is bounded by the inequality 7.20, the value of $|SPM_i|$ is bounded by the following recurrence

$$|SPM_1| \leq \mathit{infinitum}^{os_{max}}$$
$$|SPM_{i+1}| \leq |SPM_i|\, \mathit{infinitum}^{os_{max}}$$

Solving this recurrence for i produces

$$|SPM_i| \leq infinitum^{i\, os_{max}} \tag{7.24}$$

The time $t_{(C_{a_{j\,i}} \cap C_{b_l})}$ is bounded by the inequality 7.3

$$t_{(C_{a_{j\,i}} \cap C_{b_l})} \leq c_{16}\, |L_{C_{a_{j\,i}}}|\, |L_{C_{b_l}}|\, infinitum \tag{7.25}$$

where c_{16} is an implementation dependent constant.

Since C_b is produced by *SolveOS*, the size of $L_{C_{b_l}}$ is bounded by the inequality 7.14. The size of $L_{C_{a_{j\,i}}}$ depends on the previous iterations of the loop in lines 8–21, because each *spm* in SPM_{i+1} is obtained by combining one element of SPM_i with one element of output of $SolveOS(es_i)$. Thus, the size of $L_{C_{a_{j\,i}}}$ is bounded by the following recurrence

$$\begin{aligned} |L_{C_{a_{j\,1}}}| &\leq |L_{SolveFOS}| \\ |L_{C_{a_{j\,i+1}}}| &\leq |L_{C_{a_{j\,i}}}|\, |L_{SolveFOS}| \end{aligned}$$

Solving the recurrence for i produces

$$|L_{C_{a_{j\,i}}}| \leq |L_{SolveFOS}|^i \tag{7.26}$$

Substituting this bound into 7.25 and simplifying it using inequalities 7.21, 7.15, and 7.14 produces

$$t_{(C_{a_{j\,i}} \cap C_{b_l})} \leq (|Q|^2 |I|^{p+1})^{(i+1)\, |os_{max}|\, infinitum} \tag{7.27}$$

An upper bound on the running time of the event reconstruction algorithm can be obtained from the inequality 7.23 by replacing $t_{(C_{a_{j\,i}} \cap C_{b_l})}$, SPM_i, and $t_{SolveOS(es_i)}$ according to inequalities 7.27, 7.24, and 7.19, and simplifying. The resulting upper bound on the running time of the event reconstruction algo-

rithm is

$$t_{SolveES} \leq c_{11} + c_{12}\, infinitum^{os_{max}} + A + B + \Gamma \tag{7.28}$$

where A approximates the time spent on converting individual observation sequences of into sets of fixed-length observation sequences

$$A = |es| \left(\left(c_8 + c_9 \left(infinitum + 1 \right) \right) \left(\frac{infinitum^{os_{max}} - 1}{infinitum - 1} \right) + c_6 + c_7\, os_{max} + c_{10}\, infinitum^{os_{max}} \right)$$

B approximates the time spent on computing the meaning of the fixed length observation sequences

$$B = |es|\, infinitum^{os_{max}} \left((c_4 p + c_5 |Q||I|) \left(\frac{(|Q|^2 |I|^{p+1})^{os_{max}\, infinitum + 1} - (|Q|^2 |I|^{p+1})}{(|Q|^2 |I|^{p+1}) - 1} \right) + c_1 + os_{max}(c_2 + c_3\, infinitum) \right)$$

Γ approximates the time spent on combining the meanings of observation sequences

$$\Gamma = \frac{\left(infinitum\, (|Q|^2 |I|^{p+1})^{infinitum} \right)^{(|es|+1)\, os_{max}} - \left(infinitum\, (|Q|^2 |I|^{p+1})^{infinitum} \right)^{2\, os_{max}}}{\left(infinitum\, (|Q|^2 |I|^{p+1})^{infinitum} \right)^{os_{max}} - 1} + \frac{\left(infinitum^{(|es|-1)\, os_{max}} - 1 \right) \left(c_{14} + c_{15}\, infinitum^{os_{max}} \right) infinitum^{os_{max}}}{infinitum^{os_{max}} - 1}$$

In summary, assuming that the event reconstruction algorithm uses prefix based representation of computation sets, its running time is no more than *exponential* in the length of evidential statement, maximal length of observation sequences, the value of *infinitum* and the maximal length of prefixes used for representing observed properties. At the same time, the running time is no more than *polynomial* in the size of the state space and the number of possible events.

7.5 Implementation of the event reconstruction algorithm

The event reconstruction algorithm has been implemented as a "proof-of-concept" Common Lisp program, whose source code is collectively given by the Appendices C.2, C.3, and C.7. It was developed using CMU Common Lisp 18c running on a Pentium PC. This section describes the interface of the program.

The program provides a set of constants, macros, and functions for defining observation sequences and evidential statements, computing their meaning, and visualising the results of event reconstruction.

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 $P_{name1} = \{c \,|\, c \in C_T,\ exp1(c0)\}$.

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 $P_{name2} = \{c \,|\, c \in C_T,\ exp2(c0, c1)\}$.

Observation sequences are represented by Lisp lists. For example, observation sequence *example2* from Section 7.3 can be defined as follows:

```
(defprop1 *A* (c0) ...)
(defprop1 *B* (c0) ...)
(defconst *EXAMPLE2* `((,*A* 1 2) (,*B* 1 3)))
```

The meaning of observation sequence is computed using function `solve-os`. It takes an observation sequence and returns a list of MPRs that describes the meaning of the given observation sequence. For example, the meaning of *example2* is computed by

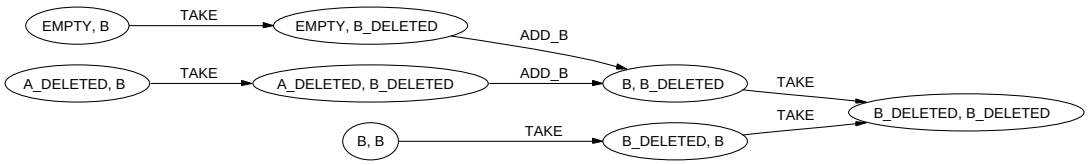


Figure 7.5: Sample output of the program

```
(solve-os *EXAMPLE2*)
```

Evidential statements are represented by Lisp lists. For example, the evidential statement $es = (os1, os2)$ can be defined as follows

```
(defconst *OS1* ...)
(defconst *OS2* ...)
(defconst *ES* '(,*OS1* ,*OS2*))
```

The *meaning of evidential statement* is computed using function `solve-es`. It takes an evidential statement as input and returns a list of MSPRs that describe the meaning of the given evidential statement. For example, the meaning of *es* is computed by

```
(solve-es *ES*)
```

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[3]. An example tree is shown in Figure 7.5 of the incident.

7.6 Summary

This chapter presented an algorithm for event reconstruction based on the formalisation of event reconstruction given in Chapter 6. The algorithm performs event reconstruction in three major steps:

[3] The output of `draw` is a file for DOT utility [37]. The latter should be manually invoked to draw the tree.

1. Convert all observation sequences in the evidential statement into sets of fixed-length observation sequences.

2. Compute the meanings of the resulting fixed-length observation sequences.

3. Combine the meanings of the fixed-length observation sequences into the meaning of the evidential statement.

The running time of the algorithm has been analysed. The derived upper bound on the running time of the algorithm is exponential in the length of evidential statement, maximal length of observation sequences, the value of *infinitum*, and the maximal length of prefixes used for representing observed properties. However, the upper bound is polynomial in the size of the state space and the number of possible events.

The algorithm has been implemented as a Common Lisp program. The next chapter further evaluates its capabilities by applying it to the analysis of example problems of digital forensic analysis.

Chapter 8

Evaluation

As defined in Chapter 4, the aims of this research are *(1) to formalise event reconstruction in a general setting, that is, assuming nothing specific about the digital system under investigation or about the purpose of event reconstruction, and (2) to show that this formalisation can be used to describe and automate selected examples of digital forensic analysis.* The first aim has been achieved in Chapter 6. A formalisation of event reconstruction problem has been developed. It can be used to automate event reconstruction as follows.

1. Formalise the system under investigation as a finite state machine, and formalise the evidence from the incident as an evidential statement;

2. use the event reconstruction algorithm from Chapter 7 to compute the meaning of the evidential statement with respect to the finite state machine.

This chapter evaluates the usefulness of this method as a forensic analysis technique, and demonstrates that it can be used to automate selected examples of digital forensic analysis.

This chapter consists of three sections. Section 8.1 defines criteria for a useful forensic analysis technique and applies them to the above-described method of event reconstruction. The issues of effectiveness, efficiency, and legal admissibility are discussed.

After that, Section 8.3 uses the developed formalisation of event reconstruction to automate two examples of digital forensic analysis. First, the simple example of networked printer anlysis from Chapter 6 is given rigorous treatment in Section 8.3.1. A more complex example is then described in Section 8.3.2, which formalises and automates forensic analysis from a published case study.

Finally, Section 8.4 reviews the problems encountered in the examples and draws conclusions about the usefulness of the developed formalisation of event reconstruction.

8.1 Evaluation criteria

What constitutes a useful forensic analysis technique? As observed in Chapter 3, forensic analysis technique is expected to be both effective and efficient in the type of analysis it performs. In addition, Chapter 2 argued that event reconstruction in digital investigations should satisfy legal requirements to expert evidence, such as Daubert criteria [30]. This gives us three criteria against which to evaluate the event reconstruction method described in the beginning of this chapter:

- *Effectiveness.* How complete and how accurate is the result of event reconstruction?
- *Efficiency.* How much manual effort does event reconstruction involve, and how long does it take to perform?
- *Conformance to admissibility requirements.* Does the event reconstruction method satisfy admissibility requirements?

The following sections discuss these criteria in more detail and evaluate how the proposed method of event reconstruction satisfies these criteria.

8.1.1 Effectiveness of event reconstruction

From effectiveness point of view, formalised and automated event reconstruction offers several advantages over semi-formal event reconstruction techniques described in Chapter 4.

Perhaps the major advantage is that formalisation and automation of event reconstruction reduces the possibility of incorrect event reconstruction. If event reconstruction is automatic, the analyst's involvement into the reconstruction process is limited. The analyst develops a model of the system and formalises the evidence. The rest is automatic process. Although this does not guarantee the absence of errors (errors, for example, can be introduced at the formalisation stage), it does remove the possibility of manual error during event reconstruction. In addition, the experience of formal methods suggests that "... Formal methods enhance existing review processes by encouraging rigorous arguments of why and in what ways the specification is correct. ..." [66]. That is, the need to formalise the incident would encourage the analyst to better understand the incident, which would reduce the possibility of errors. The examples described in Section 8.3 confirm this hypothesis.

Completeness of event reconstruction is another advantage of formalised event reconstruction. With the semi-formal event reconstruction techniques, every sequence of events has to be constructed manually. As a result, all possible sequences of events are rarely constructed. However, by computing the meaning of a single evidential statement, the analyst obtains all possible sequences of events that agree with the available evidence.

8.1.2 Efficiency of event reconstruction

To be useful in investigations, event reconstruction process should be sufficiently quick. With the developed formalisation of event reconstruction, the time required for event reconstruction is divided into the time spent on formalisation of the incident and the time spent on running the event reconstruction

algorithm.

The time required for formalisation of an incident is hard to estimate, because it depends on the circumstances of the incident. Note, however, that the developed formalisation of event reconstruction is based on the same mathematical principles as the existing methods for formal specification and verification of computing systems. As a result, formalisation of incidents is likely to require the same kind of effort as formal specification of computing systems.

The running time of the event reconstruction algorithm has been estimated in Chapter 7. An upper bound on the algorithm's running time is given by the inequality 7.28. It is exponential in the size of the evidential statement[1] and polynomial in the size of the finite state machine[2]. The exponential complexity means that the algorithm may not be able to handle large evidential statements with many observation sequences. This is a major problem limiting its application in practical investigations. Development of a more efficient event reconstruction algorithms is an important direction for future research. On the other hand, the examples descried in Section 8.3 show that the event reconstruction algorithm from Chapter 7 may still find practical applications despite its exponential complexity.

8.1.3 Legal admissibility of event reconstruction

Reconstruction of past events in a computer system requires special knowledge and, as such, falls into the category of expert evidence. The following paragraphs give reasons why formalised and automated event reconstruction based on the results of this research can pass such a test. To make the discussion

[1] More precisely, it is exponential in the length of the evidential statement, maximal length of observation sequences, the value of *infinitum*, and the maximal length of prefixes used for representing observed properties.

[2] More precisely, it is polynomial in the size of the state space and the number of possible events

more realistic, each of the following paragraphs addresses one of the Daubert criteria from Chapter 2. Note that Daubert criteria are *non-mandatory* and *non-exclusive*, that is, expert evidence may still be admissible, even if it does not conform to some of the Daubert criteria[3].

Can the technique be (and has it been) tested? The formalisation of event reconstruction developed in this dissertation relies on a well known mathematical apparatus to describe the incident and to perform event reconstruction. An algorithm for performing event reconstruction is provided. This makes the results of event reconstruction repeatable and amenable to independent verification by third-party experts. The formalisation of event reconstruction developed in this dissertation has been tested on example problems using the method described in the beginning of this chapter. The results of testing are described in Section 8.3.

What is the technique's known (or possible) error rate? The error rate associated with the developed method has not been measured. However, as discussed in Section 8.1.1, its error rate is likely to be lower than the error rate of existing semi-formal event reconstruction techniques described in Chapter 4.

If there are standards governing the application of the analysis technique, are they maintained? At the time of writing, there are no such standards.

Has the technique been subjected to peer-review and publication? The results of this research have been published in a peer-reviewed journal. The paper [39] is given in Appendix D.

[3] Ultimately, expert evidence is admissible if the judge is *convinced* that the underlying methodology is scientifically valid.

If the technique is known in the relevant scientific community, is it widely accepted? The results of this research are not widely known at the time of writing, because the paper describing them has been published very recently.

In summary, the formalisation of event reconstruction developed in this dissertation provides sufficient basis for passing the admissibility test, because it uses representation and analysis methods of a well known branch of science, it has been published in a peer-reviewed journal, and it has been tested on example problems described in the next chapter.

8.2 Comparison with other event reconstruction techniques

To conclude the first part of this chapter, Figure 8.1 compares the event reconstruction method described in this dissertation with existing semi-formal event reconstruction techniques described in Chapter 4. As the basis for comparison, it uses the three criteria defined above. The attack trees are not shown in the table, because a similar technique is already incorporated into MES technique.

As discussed in the previous sections, the method proposed in this dissertation is more effective than semi-formal event reconstruction techniques, but it has potentially lower efficiency, due to higher formalisation effort and exponential complexity of the event reconstruction algorithm.

8.3 Examples of formalised and automated event reconstruction

This section gives two examples of event reconstruction using the method described in the beginning of this chapter. Section 8.3.1 formalises and au-

Evaluation criterion	***Method proposed in this dissertation***	***VIA***	***MES***	***WBA***
Effectiveness	Reduces error rate through automation of event reconstruction and formalisation of the incident. Determines all possible scenarios of the formalised incident.	Relies on informal reasoning to perform event reconstruction. Determines some of the possible scenarios of the incident.	Relies on informal counterfactual reasoning, and MES-trees to perform event reconstruction. Determines some of the possible scenarios of the incident.	Relies on informal reasoning to perform event reconstruction, but provides EL logic for verification of causal sufficiency of reconstructed scenarios. Determines some of the possible scenarios of the incident.
Efficiency	Requires modeling of the system under investigation, and formalisation of evidence. Supports automatic event reconstruction, but the algorithm has exponential complexity.	Requires minimal formalisation (determination of possible events and activities). Does not support automatic event reconstruction.		
Daubert criteria				
Can the technique be (and has it been) tested ?	Yes, see Section 8.3	Yes, see [Mor88]	Yes, see [Ben00]	Yes, see [LL]
What is the technique s error rate?	not measured	not measured	not mesured	not measured
Are there standards governing the use of the technique?	Not yet	See [Mor88] for good practice guidelines	See [Ben00] for good practice guidelines	See [LL] for good practice guidelines
Has the technique been published in a peer-reviewed publication?	Yes, in [GP04]	Yes, in [Mor88]	Yes, in [Ben75]	Yes, in [Lad96]
If the technique is known, is it widely accepted?	Not widely known	Yes, used by the police in non-digital investigations	Yes, used for root cause analysis in accident investigations	Yes, used for root cause analysis in accident investigations

Figure 8.1: Comparison with other event reconstruction techniques

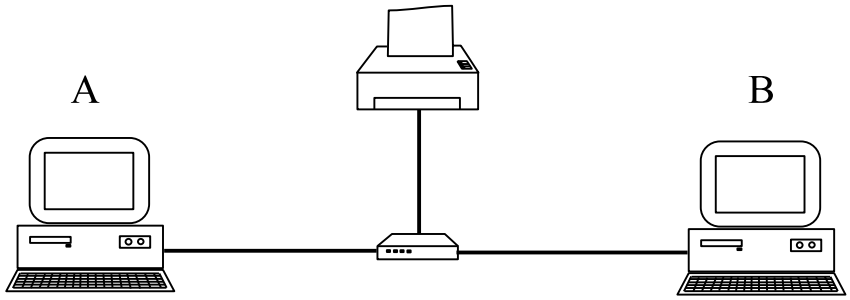


Figure 8.2: ACME Manufacturing LAN topology

tomates the simple example of networked printer anlysis from Chapter 6. A more complex example is then described in Section 8.3.2, which formalises and automates forensic analysis from a published case study.

8.3.1 Example 1. Networked printer analysis

This section illustrates the proposed formalisation of event reconstruction by applying it to the fictional example of networked printer analysis from Section 6. First, for the reader's convenience, the description of ACME investigation is repeated. Then the analysis is completely formalised and solved using the event reconstruction algorithm from Chapter 7.

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 8.2. The cost of running the network is shared by its two users Alice (A) and Bob (B). Alice, however, claims that she never prints anything 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.

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.

Automated analysis of ACME investigation

This subsection describes automated analysis of the print job directory using formalisation of event reconstruction developed in Chapter 6 and event reconstruction algorithm from Chapter 7.

Formalisation of system functionality The first step is to describe system functionality as a finite state machine. A suitable state machine was shown in Figure 6.2. For the reader's convenience it is reproduced again in Figure 8.3. Given below is a justification for the states and events chosen.

The informal analysis of the incident given in the previous section makes the following implicit assumptions about the incident

1. Alice and Bob have been the only users of the ACME Manufacturing LAN, and that security of the LAN has not been compromised.
2. Printer has always worked according to the manufacturer's description.
3. A directory entry that contains active or deleted print job is not "empty".
4. The state of the print job directory is modified only by addition of new print jobs, and by the printing mechanism fetching the print jobs from the directory.

It follows from assumptions 1, 2, and 4 that each directory entry has only five possible values: active job from Alice (A), active job from Bob (B), deleted job from Alice (A_deleted), deleted job from Bob (B_deleted), and empty.

$$ENTRY = \{\text{A, B, A\_deleted, B\_deleted, empty}\}$$

It follows from Carl's examination and assumptions 2 and 3 that only two directory entries have ever been used. Thus, the set of states needs to represent only the first two directory entries:

$$Q = ENTRY \times ENTRY$$

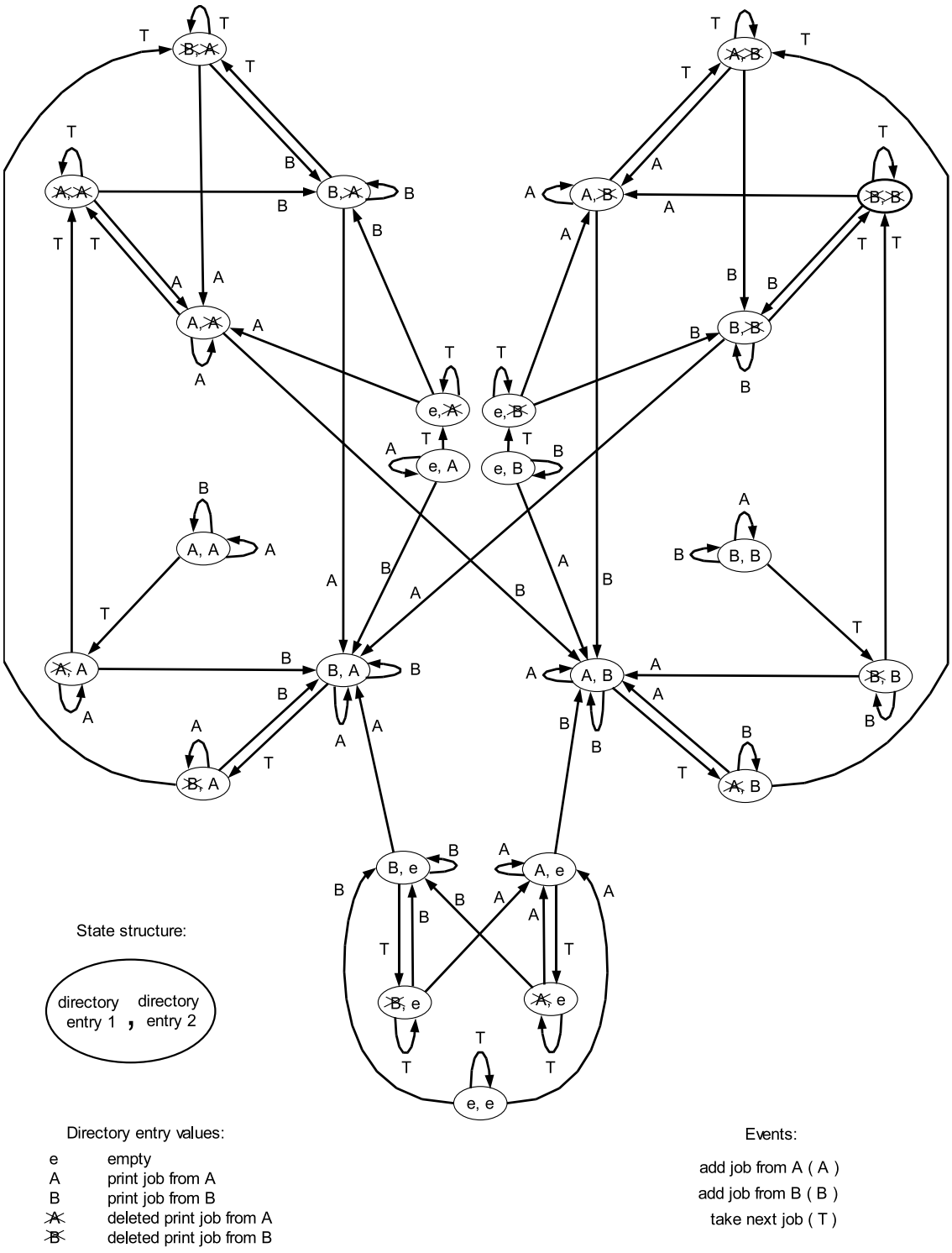


Figure 8.3: Transition graph of the print job directory model

It also follows from assumptions 1, 2, and 4 that the state of the print job directory is modified by only three possible events: submission of a print job by Alice (add_A), submission of a print job by Bob (Add_B), and the fetching of the first active print job by the printing mechanism (take).

$$I = \{\text{add\_A}, \text{add\_B}, \text{take}\}$$

Appendix C.4 presents formalisation of the state machine in ACL2 / Common Lisp. The set of states Q is defined by the recogniser function `statep`. The set of events I is defined by the recogniser function `eventp`. The transition function, whose graph is shown in Figure 8.3, is implemented by function `st`. The inverse transition function is implemented by function `rev-st`. It can be proved in ACL2 theorem prover that, when `rev-st` is given a proper state y it returns the list of all event-state pairs, whose next state according to `st` is y. The following two ACL2 theorems formalise this statement.

```
(defthm correctness-of-rev-st-wrt-st
  (implies
    (and (statep s)
         (eventp e)
         (statep y)
         (member-equal (list e s) (rev-st y)))
    (equal (st e s) y)))


(defthm completeness-of-rev-st-wrt-st
  (implies
    (and (statep s)
         (eventp e)
         (equal y (st e s)))

    (member-equal (list e s) (rev-st y))))
```

Formalisation of evidence 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

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

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

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

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 the first two 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:

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

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.

Testing investigative hypotheses The purpose of event reconstruction is usually to prove or disprove some claim about the incident. To *disprove* a claim the investigator has to show that there are no explanations of evidence that agree with the claim. To *prove* the claim the investigator has to show that all explanations of evidence agree with the claim[4]. If there are some explanations of evidence that agree with the claim, and some explanations of evidence that disagree with the claim, the claim is neither proven nor disproven. Additional evidence is required to eliminate the explanations that cause the uncertainty.

In the ACME investigation, the claim is that Alice never printed anything. To formally disprove that claim, Carl 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 if 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 formalised as observation sequence, which says that Alice did not print anything until Carl examined the printer:

$$P_{Alice} = \{c \,|\, c \in C_T,\ (c_0^q)_0 \neq \mathrm{A} \wedge (c_0^q)_1 \neq \mathrm{A}\}$$

$$os_{Alice} = (\,(P_{Alice}, 0, \mathit{infinitum}),\ (P_{\mathrm{B\_deleted}}, 1, 0)\,)$$

The extended evidential statement for the ACME investigation is then

$$es'_{ACME} = (os_{Alice}) \cdot es_{ACME}$$

[4] Note that this is equivalent to disproving the negation of the claim

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 or may not be true. If, however, 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.

Choosing the value of *infinitum* The final step in formalisation of ACME investigation analysis is to choose appropriate value of *infinitum*. Since the running time of the event reconstruction algorithm is exponential in the value of *infinitum*, the smallest possible value of *infinitum* should be chosen.

Recall that a run explaining an evidential statement must satisfy all observation sequences in it. As a result, if the maximal length of explaining run can be determined for *one* observation sequence in the evidential statement, then *infinitum* does not have to be bigger than that length.

Carl's informal analysis suggests that Alice must be lying. It means that the backtracing process started from the state (B_deleted, B_deleted) would not be able to reach the initial state, because all paths to the initial state would have states with Alice's job in them.

Taking into account these two observations, it was decided to pick a small initial value of *infinitum* and gradually increase it until the set of explanations computed by `solve-os` for os_{Alice} stops growing. This approach quickly proved problematic because of loops in the transition graphs.

The problem is illustrated by Figure 8.4, which shows computations satisfying os_{Alice} with $infinitum = 2$. Consider backtracing of computation

$$c = (\,(\text{take}, (\text{B\_deleted}, \text{B\_deleted})\,), (\text{take}, (\text{B\_deleted}, \text{B\_deleted})\,), \ldots)$$

Each transition in c represents attempt of the printing mechanism to take the next print job from the empty print job directory. It does not change the state of the print job directory, because the directory is empty. However, unless

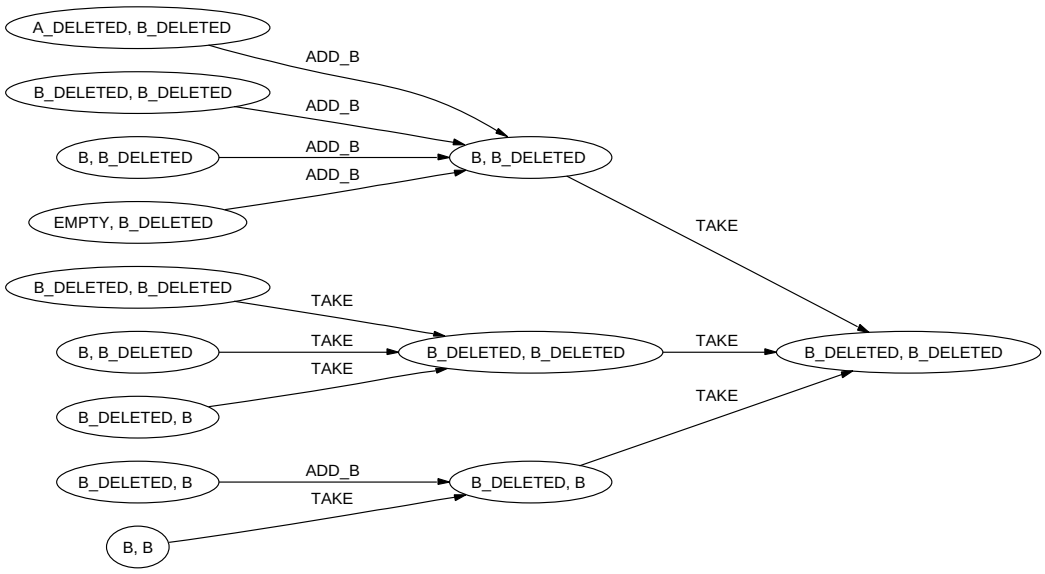


Figure 8.4: Meaning of os_{Alice} with *infinitum* = 2

there is external evidence of presence or absence of such a transition, there is no reason to believe that it never happened, or that it happened once, twice, or any other number of times. In os_{Alice} there is no such evidence. Thus, the event reconstruction algorithm dutifully reconstructs all possible sequences of $(\text{take}, (\text{B\_deleted}, \text{B\_deleted}))$ until the current value of *infinitum* is reached.

Another family of computations that cause the same problem are computations of the form

$$c = (\,(\text{Add\_B}, (\text{B\_deleted}, \text{B\_deleted})\,),\ (\text{take}, (\text{B}, \text{B\_deleted})\,),\ \ldots\,)$$

It represents printing of Bob's documents after the system first entered the state $(\text{B\_deleted}, \text{B\_deleted})$.

The problem was resolved by exploiting the nature of Alice's claim. Recall that the claim is that Alice *never* printed anything.

First observe that, if Alice had printed something, it would have changed the state of the print job directory, because Alice's print job would have been added to the print job directory. Thus, *all* single transitions that do not change

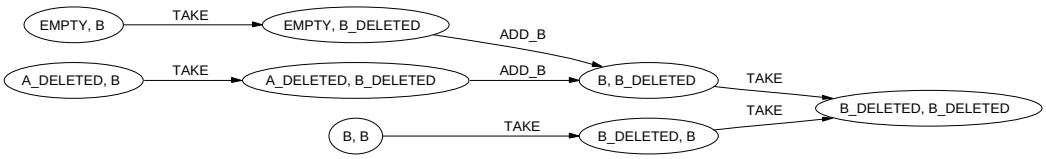


Figure 8.5: Meaning of restricted Alice's claim os'_{Alice} with *infinitum* = 3.

state of the print job directory can be excluded from the analysis.

Observe further that the truth or falseness of Alice's claim is not affected by the transition loops, which do not involve Alice printing something. The repetitive printing of Bob's documents represented by the loop $(\,(\text{Add\_B}, (\text{B\_deleted}, \text{B\_deleted})\,), (\text{take}, (\text{B}, \text{B\_deleted})\,), \ldots)$ does not involve Alice printing anything. It means that *that particular* loop can be excluded from the analysis[5].

Reflecting these insights, the property P_{Alice} was extended with two additional restrictions:

$$
\begin{aligned}
P'_{Alice} = \{c \,|\, & c \in C_T, \\
& (c_0^q)_0 \neq \text{A} \wedge (c_0^q)_1 \neq \text{A} \\
& c_0^q \neq c_1^q, \\
& c_0 \neq (\text{Add\_B}, (\text{B\_deleted}, \text{B\_deleted})) \vee c_1 \neq (\text{take}, (\text{B}, \text{B\_deleted}))\}
\end{aligned}
$$

The first additional restriction $c_0^q \neq c_1^q$ excludes from consideration single transitions that do not change the state of the print job directory. The second additional restriction $c_0 \neq (\text{Add\_B}, (\text{B\_deleted}, \text{B\_deleted})) \vee c_1 \neq (\text{take}, (\text{B}, \text{B\_deleted}))$ excludes from consideration the printing of the Bob's print jobs after the print job directory first entered the state $(\text{B\_deleted}, \text{B\_deleted})$.

[5] This must be formalised in such a way that it does not exclude computations that exit halfway through the loop

The restricted Alice's claim is described by observation sequence

$$os'_{Alice} = ((P'_{Alice}, 0, \mathit{infinitum}), (P_{\text{B\_deleted}}, 1, 0))$$

The set of explanations for the restricted Alice's claim stabilises for the values of *infinitum* >= 3. The set of computations that satisfy it for *infinitum* = 3 is shown in Figure 8.5. Note that none of these computations begin in the state (empty, empty).

The maximal length of explaining run for observation sequence os'_{Alice} is 4. Thus, *infinitum* = 4 is sufficient for the evidential statement extended with the restricted Alice's claim

$$es''_{ACME} = (os'_{Alice}) \cdot es_{ACME}$$

However, to ensure that some explanations are produced for es_{ACME}, the value of *infinitum* was increased to 6.

Running the automated test The code given in Appendix C.4 was run and the computed meanings of evidential statements es_{ACME} and es''_{ACME} were manually checked. While the meaning of es_{ACME} contained single explanation shown in Figure 8.6, the meaning of es''_{ACME} was empty, which means that Alice's claim contradicts the evidence. *The result of the automated analysis, therefore, agrees with the informal analysis.*

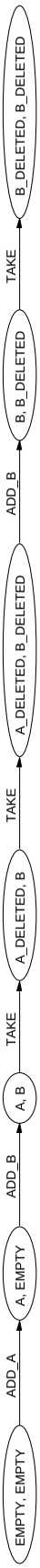


Figure 8.6: Meaning of evidential statement es_{ACME} with $infinitum = 6$

8.3.2 Example 2. Example of event time bounding

This section uses the developed formalisation of event reconstruction to analyse correctness of investigative reasoning of a published case study [9]. More specifically, this section analyses the proof that refutes the suspect's alibi. The proof is an example of event time bounding reasoning, which was described in Section 3.2.3.

The example is organised into four parts. First, a description of the case study is given. Second, event time bounding is formalised in terms of concepts developed in Chapter 6. Third, a model of the system is created. Finally, automatic event reconstruction followed by event time bounding are performed. The automated analysis was able to detect several implicit assumptions, whose validity is not justified in [9].

A blackmail investigation

The incident The following description, with some omissions, is taken from [9]. "The police in the UK received a complaint from a Mr. C, alleging that he was being blackmailed. The evidence was in the form of a floppy disk on which was a word processor data file which contained number of allegations, threats and demands. The floppy was known to be sent by a Mr. A, a computer consultant and friend of Mr. C. Police officers immediately went to interview Mr. A and found that he was on holiday abroad. However, his business premises were open and a computer found there was seized for examination.

When Mr. A returned from holidays, he was interviewed, and admitted sending the disk. He also admitted writing the letter found on his own machine but denied making the threats and demands. He suggested that Mr. C had added these himself in order to discredit Mr. A ... (skipped) ... Mr. A offered his full co-operation but suggested that care should be taken in the investigation since during his absence on holidays, his computer was available for Mr. C to use. It was therefore possible that Mr. C had used the computer

to introduce the threats and demands into the file on the floppy disk and this may have left traces which might be misinterpreted as suggesting that Mr. A had made them."

Forensic examination and analysis The contents of Mr. A computer's hard drive was examined. A total of 17 recognisable fragments of the letter located in various areas of the disk space were identified. One of the fragments was a "clean" letter, without threats, stored in an active file. Other fragments contained threats and were found in unallocated disk space.

It was concluded by the investigators that the fragments found in unallocated space were deleted versions of the letter. The conclusion follows from the fact that, when a file is deleted, FAT-based file systems do not erase the content of clusters previously used by the deleted file.

The textual contents of the fragments was compared and it "enabled the fragments to be placed in a unique sequence indicating precisely how the original document had been created and subsequently edited through a number of revisions [9]." The timestamps available in the file system indicated that all modifications happened before Mr. A went on holiday. The timestamps, however, were considered to be inconclusive. To fix the editing sequence in time, a form of event time bounding was used instead.

The time bounding relied on the properties of so-called *slack space*, which is unused space at the end of the last cluster of an active file. The formation of slack space is illustrated in Figure 8.7. One of the blackmail fragments was found in the slack space of another letter unconnected with the incident. When the police interviewed the person to whom that letter was addressed, he confirmed that he had received the letter on the day that Mr. A had gone abroad on holiday. It was concluded that

> "This fixed the whole sequence in time and showed Mr. A's story to be completely false. The threats and demands had been re-introduced into the letter at least two days before Mr. A went on

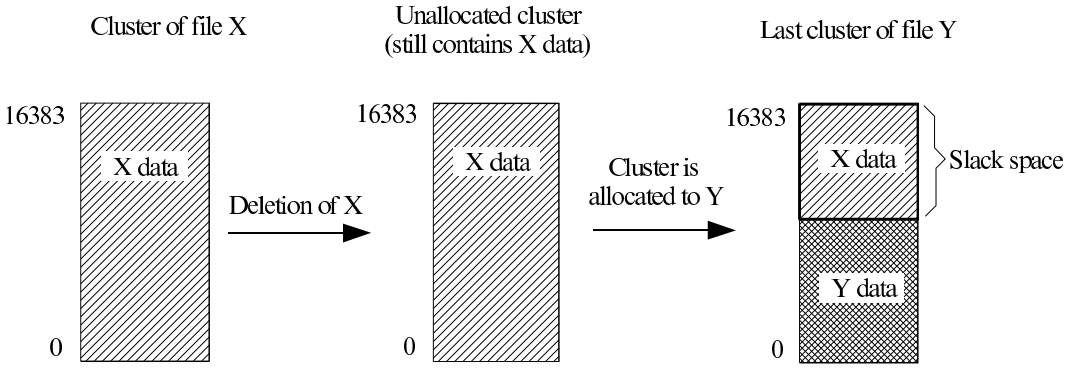


Figure 8.7: Formation of the slack space

holiday – Mr. C could not have been involved [9].

Mr. A has pleaded guilty to the charge of blackmail but there are many other complicating factors in this case and investigation are continuing."

The final piece of reasoning that stroke the final blow to the integrity of Mr. A's theory must have been that

1. The letter unconnected with the incident must have been written after the letter with threats and demands, because of the way the slack space is formed.

2. Since the letter unconnected with the incident was received on the day the Mr. A had gone on holidays, it must have been written and posted at least two days before (because of the way the postal service works).

3. Based on 1 and 2, the letter with threats and demands must have been written before Mr. A went on holiday.

Note that the first step in this reasoning is event reconstruction, while the second and the third steps are examples of event time bounding. Formalisation of this reasoning is the subject of the next two sections.

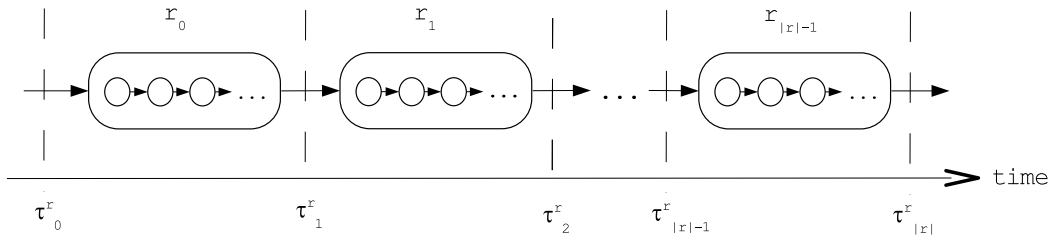


Figure 8.8: Times of transitions

Section 8.3.2 shows how formalisation of event reconstruction can be extended with real times of observations, and how event time bounding can be formalised in its context. Section 8.3.2 applies these results to the analysis of blackmail investigation. The formal analysis identified some of the implicit assumptions present in [9].

Formalisation of event time bounding

Assigning time to transitions and runs Real time can be introduced into state machine model of Chapter 6 by associating real times with transitions in style of [6].

Any run r is associated with $|r| + 1$ transitions. There are $|r|$ transitions *into* each computation of r and one transition *out* of the last computation of r.

Transition times of a run. The sequence of transition times of a run $r \in R$ is denoted τ^r. It consists of $|\tau^r|$ elements.

$$|\tau^r| = |r| + 1$$

All elements of τ^r are real valued numbers. The relationship between elements of τ^r and computations of r is shown in Figure 8.8. The first element τ^r_0 represents the time of transition into computation r_0, the first computation of run r. The last element $\tau^r_{|r|}$ represents the time of transition out of computation $r_{|r|-1}$, the last computation of run r. An intermediate element τ^r_i represents

the time of transition from computation r_{i-1} to computation r_i. Elements of τ^r are ordered in time. For all integer i, such that $0 \leq i < |r|$,

$$\tau_i < \tau_{i+1} \tag{8.1}$$

If r is empty, then r corresponds to a single moment, whose time is τ_0^r.

The following definition formalises what is meant by a run happening before another run.

Temporal precedence of runs A run ra precedes run rb in time, if the ending time of ra is less than or equal to the beginning time of rb:

$$\tau_{|ra|}^{ra} \leq \tau_0^{rb}$$

If ra and rb are sub-runs of some run rc, then positions of ra and rb in rc can be used to determine temporal precedence between ra and rb.

Let i be the index of the first computation of ra in rc, and let j be the index of the first computation of rb in rc, then

$$\tau_{|ra|}^{ra} = \tau_{i+|ra|}^{rc}$$

and

$$\tau_0^{rb} = \tau_j^{rc}$$

Run ra precedes run rb if

$$\tau_{|ra|}^{ra} \leq \tau_0^{rb}$$

or, equally,

$$\tau_{i+|ra|}^{rc} \leq \tau_j^{rc}$$

which by definition of τ^r is true if and only if

$$i + |ra| \leq j \tag{8.2}$$

Times of observations Witness observations regarding time of events are formalised as known times of observations.

Observation identifier. Observation identifier is a pair $id = (i, j)$. It denotes observation $e_{i,j}$ at the j-th position of the i-th observation sequence of evidential statement e.

Known time of observation. A known time of observation is a pair $t = (id, tim)$, where $id = (i, j)$ is an observation identifier and tim is a real valued number that represents time. The meaning of t is an assertion that for any run r explaining observation $e_{i,j}$, the following inequality holds

$$\tau_0^r \leq tim \leq \tau_{|r|}^r \tag{8.3}$$

A known time of observation corresponds to a witness statement that moment tim happened during the witness's observation. Such a statement may result from human looking at a clock during observation, or from an operating system appending clock reading to a log file entry.

Time bounding algorithm Time bounding algorithm uses known times of observations to determine time boundaries for any given observation $e_{i,j}$ within evidential statement. The idea of the algorithm is straightforward. An observation $e_{i,j}$ can happen only after the latest of observations preceding $e_{i,j}$ in time and only before the earliest of observations following $e_{i,j}$ in time.

The meaning of "preceding" and "following" observations is captured by the "happened-before" relation defined as follows. An observation $e_{i,j}$ happened before observation $e_{k,l}$ if and only if in *every* sequence of partitioned runs explaining e the run explaining $e_{i,j}$ precedes the run explaining $e_{k,l}$. The actual algorithm is based on the following two ideas.

1. Consider an sequence of partitioned runs spr that explains the evidential statement e

$$spr = (pr_0, pr_1, \ldots, pr_n)$$

By definition of explanation of evidential statement given in Chapter 6, all elements of *spr* are partitionings of the same run r. That is, any element of any pr_i is a sub-run of r. As a result, the precedence between the run that explains observation $e_{i,j}$ and the run that explains observation $e_{k,l}$ can be established by comparing their starting and ending positions within r.

2. The position of a sub-run that explains given observation $e_{i,j}$ can be calculated directly from the corresponding MSPR returned by $SolveES(e)$. Let (C, *lenlist*) be such an MSPR. In any sequence of partitioned runs represented by this MSPR, the indices of the first and last computation of the (non-empty) run that explains observation $e_{i,j}$ are

$$\sum_{l=0}^{j-1}(lenlist_i)j$$

and

$$\Big(\sum_{l=0}^{j}(lenlist_i)j\Big) - 1$$

respectively.

ACL2 code of the time bounding algorithm is given in Appendix C.5. It can be divided into three parts: (1) utility functions, (2) calculation of the earliest possible time for an observation, and (3) calculation of the latest possible time for an observation.

Calculation of the earliest time consists of two parts. First, the set of observations *bef* that happened before given observation is determined. Second, the maximal known time among observations in *bef* is found. Calculation of the latest time is similar. First, the set of observations *aft* that happened after given observation is determined. Second, the minimal known time among observations in *aft* is found.

The following paragraphs describes each part of the code in turn.

Utility functions. The algorithm uses four utility functions. Function `allobs` returns list of all observation identifiers for the given observation sequence.

```
(defun allobs (obs m n)
  (if (atom obs)
    nil
    (cons (list (nfix m) (nfix n))
          (allobs (cdr obs) (nfix m) (+ (nfix n) 1)))))
```

The caller must specify index `m` of the given observation sequence `obs` in the evidential statement. Counter `n` must be reset to 0.

Function `alles` returns list of all observation identifiers for the given evidential statement.

```
(defun alles (es n)
  (if (atom es)
    nil
    (append (allobs (car es) (nfix n) 0)
            (alles (cdr es) (+ (nfix n) 1)))))
```

The caller must reset counter `n` to 0.

Function `intersection-equal` takes two lists `x` and `y` and returns a list whose elements are members of both `x` and `y`.

```
(defun intersection-equal (x y)
  (declare (xargs :guard (and (true-listp x) (true-listp y))))
  (cond ((endp x) nil)
((member-equal (car x) y)
 (cons (car x) (intersection-equal (cdr x) y)))
(t (intersection-equal (cdr x) y))))
```

Finally, function `sumpref` adds first `n` elements of the given list `l`.

```
(defun sumpref (n l)
  (if (or (zp n) (atom l))
    0
    (+ (nfix (car l)) (sumpref (1- n) (cdr l)))))
```

If `n` is greater or equal to the length of `l`, function `sumpref` returns the sum of all elements of `l`.

Calculation of the earliest time. The set *bef* of observations that happened before observation with the given identifier is calculated using three functions shown in Figure 8.9.

```
(defun befpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= (+ cnt (car pm)) pos)
      (cons (list i j)
            (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

(defun befpml (pos pml i)
  (if (atom pml)
    nil
    (append (befpm (car pml) 0 pos i 0)
            (befpml pos (cdr pml) (+ i 1)))))

(defun findbef (v bef i j)
  (if (atom v)
    bef
    (findbef (cdr v)
             (intersection-equal
                bef
                (befpml (sumpref j (nth i (car (cdr (car v)))))
                        (car (cdr (car v)))
                        0))
             i j)))
```

Figure 8.9: Finding observations that happened before given observation

Function `befpm` takes an MPR `pm` and finds all runs whose last computation appears in the partitioned run before or at the position `pos`. A list of

observation identifiers corresponding to each of the runs is returned.

Function `befpml` applies `befpm` to every combination of $(C, lenlist_i$ in the given MSPR `pml`. The lists returned by `befpm` are concatenated.

Function `findbef` processes a list of MSPRs. For each MSPR it determines the set of observations that happened before observation with the identifier (i, j). The determined sets are intersected with each other and with parameter bef. The resulting set consists of observations that happened before observation with the identifier (i, j) in *all* MSPRs. In the initial call to `findbef`, parameter `bef` must contain the list of all observation identifiers in the evidential statement. Function `alles` is used for generating such a list. To process a list of MSPRs, function `findbef` determines the beginning position of the run explaining observation $e_{i,j}$ and uses function `befpml` to find identifiers of observations explained by runs that end before that position.

Once the set *bef* is calculated, the function `maxtime` shown in Figure 8.10 finds the latest known observation time among the elements of *bef*. It scans the list of known times and pick the latest time whose observation identifier is a member of *bef*.
Finally, the calculation of the set *bef* and finding the latest known time of its elements is combined in function `lbound`, which calls functions `findbef` and `maxtime`.

Calculation of the latest time. ACL2 code of this part of time bounding algorithm is shown in Figures 8.11 and 8.12. It is very similar to the code for calculating the earliest time. There are three differences with the code for calculating the earliest time.

1. Function `aftpm` returns a list of indices of observations whose starting positions in the given MPR `pm` are greater than or equal to the specified position `pos`;

```
(defun maxtime (max l tim)
  (if (atom tim)
    max
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (maxtime max l (cdr tim))
        (if (null max)
          (maxtime time l (cdr tim))
          (if (< max time)
            (maxtime time l (cdr tim))
            (maxtime max l (cdr tim))))))))

(defun lbound (i j es v tim)
  (maxtime nil
           (findbef v (alles es 0) i j)
           tim))
```

Figure 8.10: Calculating the earliest possible time of given observation

2. Function `findaft` calculates $sumpref(j+1)$ rather than $sumpref(j)$, because the result of $sumpref(j+1)$ is the position *after* the last computation of the j^{th} element of the i^{th} element of *listlen*.

3. Function `mintime` picks the minimal known time among observations whose elements are in the set *aft*.

Reliability of known times of observations Time bounding algorithm presented above assumes the truth of all known times of observations. This assumption simplifies reasoning by avoiding reasoning with uncertainty. This assumption is acceptable, because known times can be introduced into analysis gradually. First, time bounding can be performed with only the most reliable known times. If the results of time bounding are unsatisfactory, it can be repeated with less reliable known times included.

Reliability of time bounding results can be improved by checking consistency of known times. All known times must respect happened-before ordering imposed by the evidential statement. This can be checked by calculating the

```
(defun aftpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= pos cnt)
      (cons (list i j) (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

(defun aftpml (pos pml i)
  (if (atom pml) nil
    (append (aftpm (car pml) 0 pos i 0)
            (aftpml pos (cdr pml) (+ i 1)))))

(defun findaft (v aft i j)
  (if (atom v)
    aft
    (findaft (cdr v)
             (intersection-equal
                aft
                (aftpml (sumpref (+ j 1) (nth i (car (cdr (car v)))))
                        (car (cdr (car v)))
                        0))
             i j)))
```

Figure 8.11: Finding observations that happened after given observation

earliest $t_{min}^{e_{i,j}}$ and the latest $t_{max}^{e_{i,j}}$ times for every observation $e_{i,j}$ in the evidential statement. Every known time tim of observation $e_{i,j}$ must fall in between the two calculated times $t_{min}^{e_{i,j}} \leq tim \leq t_{max}^{e_{i,j}}$.

Automated analysis of the blackmail investigation

Formalisation of the system functionality The first step is to define a finite state machine that adequately describes the system under investigation. In the blackmail example, the functionality of the last cluster of a file was used to determine the sequence of events. Thus, the scope of the model can be restricted to the functionality of the last cluster in a file.

The last cluster in a file can be modeled as an array of bits augmented with a length (see Figure 8.13). The array of bits represents cluster data. The length

```
(defun mintime (min l tim)
  (if (atom tim)
    min
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (mintime min l (cdr tim))
        (if (null min)
          (mintime time l (cdr tim))
          (if (< time min)
            (mintime time l (cdr tim))
            (mintime min l (cdr tim))))))))

(defun ubound (i j es v tim)
  (mintime nil
           (findaft v (alles es 0) i j)
           tim))
```

Figure 8.12: Calculation of the latest possible time of given observation

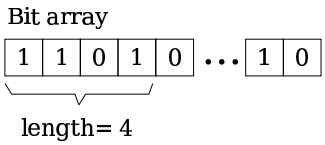


Figure 8.13: State machine model of the last cluster in a file

specifies how many bits from the beginning of the cluster are actually used by the file. Although real clusters do not have any length field, the number of data bits in the file's last cluster can be calculated from the file length and the known size of cluster in the file system. Zero length in the model would represent unallocated cluster.

Unfortunately, the event reconstruction program described in Chapter 7 was unable to work with that cluster model, because of the need to explicitly represent enormous number of possible states. It is noted in [11], that the size of cluster on the hard drive of Mr. A's computer was 16384 bytes. This results in 2^{131072} possible distinct states of the cluster model.

Despite inability to conduct analysis of the full-sized model, it was decided

to continue analysis with a simplified model. The hope was that, although properties of simplified model are not the same as the properties of the full-sized model, it may still indicate some flaws in the investigative reasoning.

To make the cluster model tractable, the size of cluster was reduced to two bits – the smallest cluster size in which slack space is possible. The state space of the simplified model is defined by

$$BIT = \{0, 1\}$$

$$LENGTH = \{0, 1, 2\}$$

$$Q = LENGTH \times BIT \times BIT$$

In FAT-based file systems, the state of the last cluster can be changed by three types of events: (a) direct writes into the cluster bypassing the file system, (b) writes into the file to which the cluster is allocated, and (c) deletion of the file. Each of these events is considered separately below.

Direct writes into the cluster. Alarmingly, there is no mentioning in [9] that cluster content can be modified directly, for example by using a low-level disk editor. It seems that an implicit assumption was made in [9] that Mr. C could not have performed low-level changes on Mr. A's computer. Reflecting this assumption, direct writes have also been excluded from the model.

Writes into the file. When cluster is modified as part of the file, the new data is written into consecutive locations starting from the beginning of the cluster. In the simplified cluster model, there are only six possible sequences that can be written into the two-bit cluster:

$$WRITE = \{(0), (1), (0, 0), (0, 1), (1, 0), (1, 1)\}$$

Apart from replacing one or two bits of data, every such event also modifies the length of active data in the cluster.

Deletion of the file. After a file is deleted, the information about the number of bits stored in the last cluster of the file sooner or later becomes unavailable. This happens when the deleted file's directory entry is reused by another file, or when the FAT chain of the deleted file is broken.

To model this eventual loss of length, the deletion event del is introduced. It sets the length of the model to zero. The set of all events in the simplified cluster model is defined by

$$I \;=\; \mathit{WRITE} \cup \{\mathrm{del}\}$$

The ACL2 / Common Lisp implementation of the simplified cluster model is given in the Appendix C.6. The set of states Q is defined by the recogniser function `statep`. The set of events I is defined by the recogniser function `eventp`. The transition function and its inverse are implemented by functions `st` and `rev-st` respectively.

Automated analysis of the simplified model The exact evidential data was not published in [9]. As a result, the specific cluster contents for the blackmail letter and for the unrelated letter had to be chosen arbitrarily. Sequence $(1, 1)$ is chosen to represent the contents of the blackmail letter. Sequence (0) is chosen to represent the contents of the unrelated letter. With these choices, state $(1, 0, 1)$ represent the final state discovered by investigators in the blackmail investigation. The state describes a non-empty cluster whose active content – the letter unrelated to investigation – is sequence (0), and whose slack space contains the end of the blackmail letter – the sequence (1). The observation of this state is captured by the following property:

$$P_{\mathit{final}} \;=\; \{\, c \,|\, c \in C_T,\, c_0^q = (1, 0, 1)\}$$

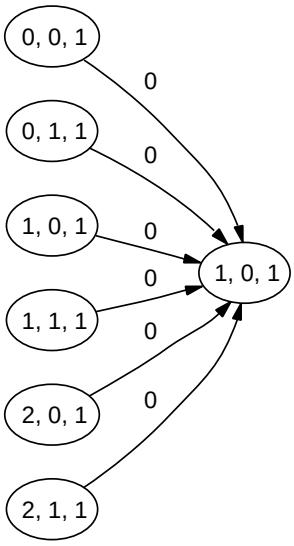


Figure 8.14: One step of event reconstruction of observation sequence os_{final}

The observations about the cluster content made by investigators in the blackmail investigation are formalised by the observation sequence os_{final}:

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

It states that nothing was observed about the cluster's content until forensic examination, which found that the cluster was in the state $(1, 0, 1)$.

Figure 8.14 shows the result of a single step of event reconstruction for the observation sequence os_{final}. The figure shows that, as expected, the current active content of the cluster – (0) – was produced by writing it into the cluster. However, there are two distinct situations, in which that writing could have taken place. First, the cluster could have been unallocated before (0) was written into it. This possibility is represented by transitions

$$(0, 1, 1) \xrightarrow{(0)} (1, 0, 1)$$

$$(0, 0, 1) \xrightarrow{(0)} (1, 0, 1)$$

Second possibility is that, the cluster was already allocated to the file, and

new value was written into it. That possibility is represented by transitions

$$(1,1,1) \xrightarrow{(0)} (1,0,1)$$

$$(1,0,1) \xrightarrow{(0)} (1,0,1)$$

$$(2,1,1) \xrightarrow{(0)} (1,0,1)$$

$$(2,0,1) \xrightarrow{(0)} (1,0,1)$$

Note the last two transitions. They suggest that the blackmail message in the slack space was *not* caused by overwriting deleted blackmail message, but by truncating the unrelated letter, which already contained the piece of the blackmail letter! This however, does not change the conclusions of the investigators.

To complete formalisation of evidence two more observation sequences need to be created. The first observation sequence $os_{blackmail}$ says that, at some point in time, the piece of the blackmail letter was written into the cluster:

$$P_{blackmail} = \{\, c \,|\, c \in C_T,\, c_0^\iota = (1,1) \,\}$$

$$\begin{aligned} &os_{blackmail} = \\ &(\,(C_T, 0, \mathit{infinitum}), \\ &\ (P_{blackmail}, 1, 0), \\ &\ (C_T, 1, \mathit{infinitum})\,) \end{aligned}$$

The minimal length of the last observation in $os_{blackmail}$ is set to 1 to exclude the possibility that the writing of the blackmail coincided with the observation of the final state.

The second observation sequence $os_{unrelated}$ says that the unrelated letter was created at some time in the past, and that later it was received by the person to whom it was addressed:

$$P_{unrelated} = \{\, c \,|\, c \in C_T,\, c_0^\iota = (0) \,\}$$

$$
\begin{aligned}
&os_{unrelated} = \\
&(\,(C_T, 0, \mathit{infinitum}), \\
&\ (P_{unrelated}, 1, 0), \\
&\ (C_T, 0, \mathit{infinitum}), \quad (C_T, 0, 0), \\
&\ (C_T, 1, \mathit{infinitum})\,)
\end{aligned}
$$

The zero-observation $(C_T, 0, 0)$ represents the reception of the letter by the addressee.

The evidential statement for the blackmail example combines os_{final}, $os_{blackmail}$, and $os_{unrelated}$:

$$es_{blackmail} = (\, os_{final},\ os_{unrelated},\ os_{final})$$

Once $es_{blackmail}$ was defined, *infinitum* was arbitrarily chosen to be 4, and the reconstruction was performed. The code given in the Appendix C.6 saves the result of reconstruction of $es_{blackmail}$ with $\mathit{infinitum} = 4$ to the constant `*SOL-4*`.

The result of event reconstruction was then used to perform time bounding of the blackmail writing. Since the exact time of reception of the unrelated letter is not specified in [9], and since there is no other timed event in $es_{blackmail}$ except the unrelated letter reception, the time of the reception was arbitrarily chosen to be 5. The code that performs event time bounding is shown below:

```
(defconst *tim* '(((1 3) 5)))
(defconst *l* (lbound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))
(defconst *u* (ubound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))
```

The outcome of this computation was that both variables `*u*` and `*l*` were equal to `NIL`, which means that the algorithm was unable to determine nether upper nor lower time bound for the blackmail writing. To investigate this problem, the reconstruction results contained in *SOL-4* were examined. This revealed that one of the possible explanations of $es_{blackmail}$ was the sequence of transitions

$$\ldots \overset{(0)}{\rightarrow} (1,0,1) \overset{(11)}{\rightarrow} (2,1,1) \overset{(0)}{\rightarrow} (1,0,1)$$

This sequence of events suggests that someone could have framed Mr. A by

1. finding an unrelated letter, which was written by Mr. A earlier,
2. writing the last piece of the blackmail letter into the last cluster of the unrelated letter,
3. writing the last piece of the unrelated letter back into the cluster.

This could have been easily accomplished using some low-level disk editor. However, as was noted earlier, the possibility of using such tools seems to be excluded from [9]. If ordinary text editing tools were used instead, this result is unlikely, because text editors tend to save modified document in a new file rather than modify the original[6]. Unfortunately, there is not enough information in [9] about the system software to make any reliable assumptions.

To replicate the investigative reasoning, an assumption had to be forced that the unrelated letter was written into the cluster only once. The modified observation sequence and evidential statement are

$$P_{no\_unrelated} = \{\, c \,|\, c \in C_T,\, c_0^\iota \neq (0) \,\}$$

$$\begin{aligned} os'_{unrelated} = \\ (\,&(P_{no\_unrelated}, 0, \mathit{infinitum}), \\ &(P_{unrelated}, 1, 0), \\ &(P_{no\_unrelated}, 0, \mathit{infinitum}), \\ &(P_{no\_unrelated}, 0, 0), \\ &(P_{no\_unrelated}, 1, \mathit{infinitum})\,) \end{aligned}$$

$$es'_{blackmail} = (\, os_{final},\, os'_{unrelated},\, os_{final})$$

[6] Microsoft Word, for example, does not modify the existing file. The changed document is first written into a new file, then the original document is deleted [1].

The automated analysis of $es'_{blackmail}$ with *infinitum* = 4 yields the expected result – that the upper time bound for the writing of the blackmail letter is the time of reception of the unrelated letter.

8.4 Summary

The discussion presented in this chapter has demonstrated that formal approach to event reconstruction can be useful at least in some cases of digital forensic investigations. Perhaps the most important benefit of formality is its ability to focus attention of the analyst on the obscure detail, which in turn, reduces the possibility of analytical error.

In the beginning of the chapter, three criteria for a useful forensic analysis technique have been put forward. They are efficiency, effectiveness, and conformance to legal admissibility requirements. The approach to event reconstruction developed in this dissertation has been evaluated against these criteria. It has been shown that the event reconstruction approach proposed in this dissertation has better effectiveness than existing semi-formal event reconstruction techniques, because it reduces error rate and provides comprehensive reconstruction of possible incident scenarios. At the same time, it has potentially lower efficiency due to higher formalisation effort and exponential complexity of the event reconstruction algorithm. In addition to this analysis, two examples of formal event reconstruction have been performed.

The *ACME investigation* example has shown that devices with well defined and relatively simple functionality can be successfully analysed using developed approach. The most likely examples of such systems are controllers embedded in consumer electronics and appliances. The continuing integration of computing technology into human habitat is making forensic analyses of such devices increasingly likely.

The *blackmail example* has shown that even incomplete model of the system may be useful in the investigation. The need to formalise system functional-

ity combined with probing explorations of the state space forces the analyst to consider many aspects of the system and the evidence, thus facilitating a better understanding of the investigation. This, in turn, may suggest missed assumptions or omissions in expert reasoning. The blackmail example was able to detect two implicit, obscure assumptions in a published case study. It suggests that formal analysis might be particularly useful for analysing expert reports produced by the opposing party in legal proceedings.

In addition to exponential complexity of event reconstruction algorithm, complexity of real world systems is also likely to be a major challenge for formal event reconstruction. The event reconstruction approach presented in this dissertation relies on automatic exploration of a finite state machine's state space to perform event reconstruction. The state machines considered in this chapter are very simple. The ACME investigation, for example, required creation of a finite state machine with only 25 states and 75 possible transitions. Although similar cases are possible in real life, the majority of investigations is likely to encounter systems, whose exact finite state machine models are much more complex.

The relative success of model checking suggests that the complexity problem can be dealt with using model reduction techniques and symbolic representation of state sets. The investigation of the applicability of these techniques in the domain of digital investigations is an important direction for future research.

Chapter 9

Conclusions and future work

. . . Chance has put in our way a most singular and whimsical problem, and its solution is its own reward. . . .

Arthur Conan Doyle

This dissertation investigated the theory and practice of event reconstruction in digital investigations. The main outcome of this work is a formalisation of event reconstruction in terms of state machine model of computation. This formalisation has been validated through the development of an event reconstruction algorithm and using it to perform sample event reconstructions.

This chapter summarises the main points from this study and concludes the work.

9.1 Problem

Digital evidence is commonly encountered in many types of criminal and civil investigations. It has been argued, however (see for example [76], and [4]), that currently widespread *ad-hoc* analysis of digital evidence is inappropriate from forensic point of view, because it is error-prone and because its findings

are hard to explain and defend in court. More rigorous methods of analysis have been called for.

To answer this call, this dissertation explored the problem of event reconstruction in digital investigations, whose aim is to determine the sequence of events that happened in a given computer system during the incident. The need for such reconstructions arises in a variety of cases ranging from investigations of technically advanced network intrusions [38] to seemingly straightforward blackmailing cases [9].

To clarify the problem, a study of digital forensics and related disciplines has been performed. The study has shown that event reconstruction is expected to be

- efficient,
- effective (i.e. reliable and precise), and
- based on a scientifically valid methodology.

The study has also shown that current practices of event reconstruction are essentially manual, and their reasoning is based on common sense and investigator's experience rather than on any scientific theory.

The idea of this project was that, to improve this situation, event reconstruction should be defined as a computer science problem and solved using methods of computer science. More specifically, the objectives of this project were

- to formalise event reconstruction in a general setting, that is, assuming nothing specific about the digital system under investigation or about the purpose of event reconstruction, and
- to show that this formalisation can be used to describe and automate selected examples of digital forensic analysis.

9.2 Solution

To formalise the event reconstruction problem, this project used the state machine model of computation. This model of computation is convenient from the forensic point of view, because

- the operation of state machine closely resembles the operation of actual computing devices, which is appealing to the fact finder;
- state machine models are widely used in practice to specify and verify computing systems.

The idea behind the solution was to model the system under investigation as a finite state machine, and to define event reconstruction as the process of finding all possible computations of the machine that agree with the evidence of the incident.

To make this idea completely formal, the evidence about the incident had to be formalised. For this purpose, the notion of *evidential statement* has been developed. Evidential statement represents the evidence as a system of *observations* about the properties and change of the system state during the incident. By doing so, it restricts possible computations of the finite state machine. The formalisation of an incident, therefore, consists of two parts: a finite state machine model of the system under investigation, and an evidential statement that represents the evidence.

A precise definition of the event reconstruction problem was then given in Section 6.2.5. Informally speaking, event reconstruction is defined as the process of (1) finding all computations of the state machine that agree with the evidential statement, and (2) identifying how parts of these computations match individual observations within evidential statement.

To show that the developed formalisation can be used to automate event reconstruction, an event reconstruction algorithm has been designed and implemented. To fulfil the second aim of this project, the implementation of

the event reconstruction algorithm was then used to perform two examples of forensic event reconstruction. The results of this practical application, as well as other results of this project are summarised in the next section.

9.3 Lessons of the project

This research has produced a number of original and innovative ideas and results, as well as encountered some problems. They are summarised below.

9.3.1 Achievements

The key achievements of this research are as follows.

- For the first time, a precise mathematical definition of event reconstruction in digital investigations has been given. Apart from providing a basis for automation of event reconstruction, it contributes to the development of digital forensics theory as a discipline.

- A generic event reconstruction algorithm has been designed and implemented. Unlike many digital forensic tools, it has solid theoretical foundation, which can be used to defend its admissibility in legal proceedings.

- As example applications of the developed formalisation, two instances of digital forensic analysis have been formalised and automated. They demonstrate why and how formal event reconstruction can be used in practical investigations.

- It has been demonstrated on a practical example, that formal approach to event reconstruction can discover weak points and hidden assumptions in informal event reconstructions.

- The formal approach to event reconstruction has been compared with existing semi-formal event reconstruction techniques. It has been shown

that formal event reconstruction is likely to have lower error rate and provide more complete event reconstruction that the existing techniques.

- A study of legal and practical aspects of forensic event reconstruction identified the deficiency of existing semi-formal techniques in the context of digital investigations.

On the personal side, this was a very interesting and delightful, albeit very long and sometimes difficult project. The author acquired knowledge and skills in many areas, which include (but are not limited to) the following.

- Legal concepts surrounding the use of digital evidence in litigation.
- The investigative process as well as the techniques for collection, examination, and analysis techniques of digital evidence.
- Existing semi-formal techniques for forensic event reconstruction.
- Formal specification of computing systems.

9.3.2 Problems encountered

In addition to many interesting and innovative results, a number of problems associated with the formal approach to event reconstruction have been discovered. They are summarised below.

- *Computational complexity of the event reconstruction algorithm.* The running time of the event reconstruction algorithm has been estimated in Chapter 7. An upper bound on the running time of the algorithm has been derived analytically. It turned out to be exponential with respect to the parameters of the evidential statement, but polynomial with respect to the parameters of the finite state machine. The exponential complexity suggests that the algorithm may not be able to handle large evidential statements with many observation sequences.

Although this is a serious limitation on the applicability of the developed algorithm, it has been demonstrated in Section 8.3.2 that even small and incomplete models can be useful in practical investigations.

- *Complexity of real world systems.* The formalisation of event reconstruction developed in this dissertation relies on state space exploration to perform event reconstruction. The state machines considered in this dissertation are very simple. For example, the ACME investigation described in Section 8.3.1 required a finite state machine with only 25 states and 75 possible transitions. The majority of investigations are likely to encounter systems, whose finite state machine models are much more complex. The experience of formal methods suggests that for many real-world systems the brute-force exploration of their exact finite state machine models is infeasible due to limitations of modern computers.

  The experience of model checking is also instructive in another respect. At the beginning of model checking era, the complexity of exact finite state machine models of almost all systems was beyond capabilities of model checking programs. However, as the model checking algorithms improved, and various model reduction techniques were developed, model checking of industrial scale systems became possible. The same kind of success may be possible with formal event reconstruction in digital investigations.

- *Formality of the approach.* Despite providing more effective event reconstruction, the formal approach developed in this dissertation requires considerable effort for formalising the incident. This, together with the need to learn formal notation is likely to be a deterrent for its use in practical applications. Nevertheless, the developed formal approach will probably find its application in cases, such as [38] where the success of the legal action depends on the comprehensiveness and reliability of event reconstruction.

9.4 Future work

The work performed in this project provides basis for future research in several areas. At least four such areas can be identified. These areas include:

- extending formalisation of event reconstruction,
- developing more efficient event reconstruction algorithm,
- investigating new ways of constructing system models, and
- developing practical applications of the results of this work.

The following sections discuss each of these areas in more detail.

9.4.1 Extending formalisation of event reconstruction

The formalisation of event reconstruction developed in this dissertation can be extended in several ways. One possible extension is to provide support for uncertain reasoning. Uncertainty is deeply ingrained in forensics. The blackmail example from Chapter 8 is a vivid demonstration of this fact. It shows that many assumptions in actual investigations are being made on the basis of the investigator's experience of what is and is not *probable* in the specific circumstances of the case. Although the complete formalisation of this uncertain knowledge is problematic (consider, for example, measuring uncertainty of an eyewitness statement), there are statistical measures that can and should be incorporated into event reconstruction process. In addition, there is a body of applied mathematics [5] that has been specifically developed for calculating the impact of known statistical properties of the world — such as probability of finding bloodstains on clothes in the general population — on the probability of investigative hypotheses.

Formalisation of event reconstruction given in this dissertation ignores the issue of uncertainty. Although, as explained in Chapter 6, this assumption has some merit, the formalisation of event reconstruction can be enhanced by

adding explicit measures of uncertainty to formalisation of event reconstruction problem. The possibility and the impact of such additions on the process of event reconstruction needs to be investigated.

Another possible extension is to enrich the expressiveness of evidential statements. For example, the introduction of variables would allow the analyst to specify statements such as "some (unknown) phenomenon X has been observed twice":

$$os = (\,(x, 1, 1), (C_T, 0, \textit{infinitum}), (x, 1, 1)\,)$$

Note, however, that any such extension may increase complexity of event reconstruction. The analysis of the impact of such extensions on the semantics and complexity of event reconstruction problem is a topic for separate research project.

9.4.2 Developing more efficient event reconstruction algorithm

The event reconstruction algorithm presented in Chapter 7 is quite inefficient despite its ability to handle examples described in Chapter 8. Development of a more efficient algorithm is an important area for future research.

One possible direction of research is to investigate more efficient representations of computation sets. It can be shown that, if a single step of backtracing was *adding* a fixed amount of elements to the representation of its input computation set (rather than *multiplying* the size of the representation a fixed number of times), the time required for SolveFOS algorithm would become polynomial in the number of backtracing steps. Symbolic representation techniques used in model checking [28] represent one possibility that should be investigated in this respect.

In addition, the applicability of model reduction techniques, such as data abstraction and partial order reduction described in Chapter 5, should be

researched in the context of digital investigations.

9.4.3 Investigating new ways of constructing system models

As noted in Chapter 5, the initial formalisation of the problem is a critical and laborious part of any formal analysis. Simplification or automation of this process is an important direction for future research.

Some of the existing approaches to the development of finite state machine models of systems have been described in Chapter 5. However, there may be other approaches, which are more suited for the purposes of digital investigations. One interesting question is whether the model of the system under investigation can be derived directly from observations of the system behaviour without consulting the source code or user manuals. The applicability of machine learning techniques should be investigated in this respect.

9.4.4 Developing practical applications

The results of this work can be applied to practical investigations in several ways. Some of these ways are discussed below.

The most straightforward application is the development of a general-purpose event reconstruction tool similar to model checkers used for systems verification. When using such a tool, the human investigator would provide a formal description of evidence and a model of the system under investigation. The tool would calculate and visualise possible incident scenarios consistent with the given formal description. Such a tool would be welcome in investigations such as [38] where success of the legal action depends on the comprehensiveness and reliability of event reconstruction.

Another possible application of the developed formalisation of event reconstruction is proving correctness of forensic analysis tools. As discussed in Chapter 3, some of the tools used in digital forensic analysis can be viewed as performing specialised form of event reconstruction. For example, the recov-

ery of deleted files can be viewed as reconstruction of events in the file system back to the moment when the given file was deleted. Such specialised event reconstruction can be defined (with respect to the file system model) by the evidential statement

$$es_x = (a_0, \ldots, a_n, ((C_T, 0, \textit{infinitum}), (x, 1, 0)))$$

where $(x, 1, 0)$ formalises the knowledge of the final state of the system, and observation sequences $a_0, \ldots, a_n$ formalise assumptions made by the designers of the analysis tool. To prove correctness of the tool one should prove that for all possible inputs x, the meaning SPR_{es_x} of the evidential statement es_x is linked to the tool's output out_x according to some well defined interpretation relation $\overset{R}{\sim}$:

$$\text{for all possible } x, \quad SPR_{es_x} \overset{R}{\sim} out_x$$

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.

In addition to proving correctness of forensic *analysis* tools, the formalisation of event reconstruction can also be used to check correctness of evidence *collection*. To achieve this, the process of the evidence collection itself can be reconstructed. The results of such reconstruction can be tested to see if they contains possible scenarios that involve modification of the collected evidence.

9.5 Summary

Overall, this project was a success. This dissertation extended the theory of digital forensic science by formalising and solving event reconstruction problem using methods of computer science. The formalisation of event reconstruction has been validated through the development of an event reconstruction algorithm and using it to perform sample event reconstructions. A number of novel results as well as problems associated with the developed formalisation have

been discovered.

Several possible directions for future research have been proposed. They include: extending formalisation of event reconstruction, developing more efficient event reconstruction algorithm, investigating new ways of constructing system models, and developing practical applications of the results of this work.

Yet, every doctoral dissertation should have an end. So I rest my case!

Bibliography

[1] Wd97: How word for windows uses temporary files. Microsoft Knowledge Base Article 89274, 1997. See url `http://support.microsoft.com/support/kb/articles/Q89/2/47.ASP` (downloaded 17 Oct 2004).

[2] *Evidence.* Cavendish Publishing Limited, London, UK, 1998. ISBN 1–85941–428–1.

[3] Digital evidence: Standards and principles. *Forensic Science Communications 2*, 2 (April 2000).

[4] A roadmap for digital forensic research. Tech. rep., Digital Forensic Research Workshop, 2001. `http://www.dfrws.org/dfrws-rm-final.pdf` (downloaded 15 Oct 2002).

[5] Aitken, C. G. G. *Statistics and the Evaluation of Evidence for Forensic Scientists.* Statistics in Practice. John Wiley & Sons, Chichester, UK, 1995. ISBN 0–471–95532–9.

[6] Alur, R., and Dill, D. L. A theory of timed automata. *Theor. Comput. Sci. 126*, 2 (1994), 183–235.

[7] Anderson, M. R. Method and apparatus for identifying names in ambient computer data. United States Patent No. 6,263,349, July 2001.

[8] Association of Chief Police Officers of England, Wales and Northern Ireland. Good Practice Guide for Computer Based Evidence, Version 2. ACPO Crime Committee, London, UK, June 1999.

[9] Bates, J. Blackmail: Case study. *International Journal of Forensic Computing*, 2 (1997).

[10] Bates, J. Cluster analysis. *International Journal of Forensic Computing*, 6 (1997), 11–12.

[11] Bates, J. The fundamentals of computer forensics. *International Journal of Forensic Computing*, 1 (1997), 4–5.

[12] Bates, J. DIVA computer evidence. *International Journal of Forensic Computing*, 18 (June 1998), 19–22.

[13] Benner, L. Accident investigation: Multilinear event sequencing methods. *Journal of Safety Research 7*, 2 (1975), 67–73.

[14] Benner, L. Task guidance for bridging mes worksheet gaps with mes trees. Tech. rep., Starline Software, 2000.

[15] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without bdds. In *Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)* (London, UK, 1999), no. 1633 in Lecture Notes in Computer Science, Springer-Verlag, pp. 193–207.

[16] Boigelot, P. W. B. On the construction of automata from linear arithmetic constraints. In *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000* (2000), vol. LNCS 1785, Springer-Verlag Heidelberg, pp. 1–20. ISSN: 0302-9743.

[17] Brezinski, D., and Killalea, T. Guidelines for evidence collection and archiving. Internet Engineering Task Force, 2002. Request For Comments 3227.

[18] Bryant, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (August 1986), 677–691.

[19] Bryson, C., and Stevens, S. Tool testing and analytical methodology. In *Handbook of Computer Crime Investigation: Forensic Tools and Technology* (San Diego, CA, USA, 2002), E. Casey, Ed., Academic Press, pp. 53–71. ISBN 0–12–163103–6.

[20] Bultan, T., Gerber, R., and League, C. Verifying systems with integer constraints and boolean predicates: A composite approach. In *International Symposium on Software Testing and Analysis* (1998), pp. 113–123.

[21] Burstall, R. M. Proving properties of programs by structural induction. *Computer Journal 12*, 1 (1969), 41–48.

[22] Carrier, B. Defining digital forensic examination and analysis tools using abstraction layers. *International Journal of Digital Evidence 1*, 4 (2002). Journal website `http://www.ijde.org/`.

[23] Carroll, L. *Alice's Adventures in Wonderland and Through the Looking Glass*. Signet (Penguin group company), 375 Hudson St., New York, NY 10014, 2000. ISBN: 0451527747.

[24] Casey, E. Error, uncertainty, and loss in digital evidence. *International Journal of Digital Evidence 1*, 2 (2002). Journal website `http://www.ijde.org/`.

[25] Casey, E. *Digital Evidence and Computer Crime. Second edition.* Academic Press, San Diego, California, USA, 2004. ISBN 0–12–162885–X.

[26] Choi, Y., Rayadurgam, S., and Heimdahl, M. P. Automatic abstraction for model checking software systems with interrelated numeric constraints. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (2001), ACM Press, pp. 164–174.

[27] Clarke, E. M., Grumberg, O., Minea, M., and Peled, D. State space reduction using partial order techniques. *Software Tools for Technology Transfer 3*, 1 (1999), 279–287.

[28] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0–262–03270–8.

[29] Clugston, M. J., Ed. *The New Penguin Dictionary of Science*. Penguin Books, Ltd, London, UK, 1998. ISBN 0-14-051271-3.

[30] Daubert v. Merrell Dow Pharmaceuticals Inc., 1993. See url `http://supct.law.cornell.edu/supct/html/92-102.ZO.html` (downloaded 17 Oct 2004).

[31] de Vel, O. File classification using byte sub-stream kernels. *Digital Investigation 1*, 2 (2004), 150–157.

[32] Dwyer, M. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering* (2001), IEEE, pp. 177–187. ISBN: 0769510507.

[33] Emerson, E. A., and Halpern, J. Y. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM 33*, 1 (1986), 151–178.

[34] Farmer, D. What are MACtimes? Powerful tools for digital databases. *Dr. Dobb's Journal*, 10 (2000).

[35] Farmer, D. Bring out your dead. the ins and outs of data recovery. *Dr. Dobb's Journal*, 1 (2001).

[36] Gannon, S., Gladychev, P., and Patel, A. Extendible search utility for forensic computing. In *Proceedings of the World Multiconference on Systemics, Cybernetics and Informatics, and International Conference on Information Systems, Analysis and Synthesis (SCI 2001/ISAS 2001)* (Sheraton World, Orlando, FL, USA, 2001).

[37] Gansner, E., and North, S. C. An open graph visualization system and its applications to software engineering. *Software Practice and Experience* (1999).

[38] George, E. UK computer misuse act — the trojan virus defence regina v aaron caffrey, southwark crown court, 17 october 2003. *Digital Investigation 1*, 2 (2004), 89–89.

[39] Gladyshev, P., and Patel, A. Finite state machine approach to digital event reconstruction. *Digital Investigation 1*, 2 (2004), 130–149.

[40] Godefroid, P., and Wolper, P. A partial approach to model checking. In *Papers presented at the IEEE symposium on Logic in computer science* (1994), Academic Press, Inc., pp. 305–326.

[41] Hachtel, G. D., and Somenzi, F. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Boston, MA, USA, 1996. ISBN: 0792397460.

[42] Honderich, T., Ed. *The Oxford Companion to Philosophy*. Oxford University Press, Oxford, UK, 1995. ISBN 0198661320.

[43] Hopcroft, J., and Ullman, J. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[44] Huffman, D. A. The synthesis of sequential switching sircuits. *Journal of the Franklin Institurte 257*, 3–4 (1954), 161–190 and 275–303.

[45] Hume, D. *An Enquiry Concerning Human Understanding*. Oxford University Press, 1999 (originally published in 1748).

[46] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (Washington, D.C., 1990), IEEE Computer Society Press, pp. 1–33.

[47] Kaufmann, M., Manolios, P., and Moore, J. S. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, Boston, MA, USA, 2000. ISBN 0–7923–7849–0.

[48] Kaufmann, M., Manolios, P., and Moore, J. S. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, MA, USA, 2000. ISBN 0–7923–7744–3.

[49] Kaufmann, M., and Moore, J. A precise description of the ACL2 logic. Tech. rep., Department of Computer Sceince, University of Texas at Austin, 1997.

[50] Kesten, Y., Maler, O., Marcus, M., Pnueli, A., and Shahar, E. Symbolic model checking with rich assertional languages. *Theoretical Computer Science 256*, 1–2 (2001), 93–112.

[51] Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata studies* (Princeton, New Jersey, U.S.A., 1956), Princeton University Press, pp. 3–42.

[52] Kumho Tire Co. v. Carmichael, 1999. See url `http://supct.law.cornell.edu/supct/html/97-1709.ZS.html` (downloaded 17 Oct 2004).

[53] Ladkin, P. Analysis of a technical description of the airbus A320 braking system. *High Integrity Systems 1*, 4 (1996).

[54] Lamport, L. *Specifying systems: The TLA+ language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, U.S.A., 2002. ISBN 0–321–14308–X.

[55] Lewis, D. Causation. *Journal of Philosophy 70*, 17 (1973), 556–557.

[56] Lewis, D. *Counterfactuals*. Basil Blackwell, Oxford, UK, 1973.

[57] Loer, K. *Towards "Why...Because" Analysis of Failures*. PhD thesis, Universität Bielefeld, Germany, 1998.

[58] Loer, K., and Ladkin, P. *Why-Because Analysis: Formal Reasoning About Incidents*. Draft manuscript. See `http://www.rvs.uni-bielefeld`

`.de/publications/books/WBAbook/` (downloaded 9 Oct 2002).

[59] Luque, M. E. Logical level analysis of unix systems. In *Handbook of Computer Crime Investigation: Forensic Tools and Technology* (San Diego, CA, USA, 2002), E. Casey, Ed., Academic Press, pp. 182–195. ISBN 0–12–163103–6.

[60] Lylle, J. J. Nist cftt: Testing disk imaging tools. *International Journal of Digital Evidence 1*, 4 (2002). Journal website `http://www.ijde.org/`.

[61] Matthew B. Gerber, and John J. Leeson. Shrinking the ocean: Formalizing i/o methods in modern operating systems. *International Journal of Digital Evidence 1*, 2 (2002). Journal website `http://www.ijde.org/`.

[62] Mealy, G. H. A method for synthesizing sequential circuits. *Bell System Technical Journal 34*, 5 (1955), 1045–1079.

[63] Meyer, J.-J. C., Wieringa, R., and Dignum, F. The role of deontic logic in the specification of information systems. In *Logics for Databases and Information Systems* (1998), pp. 71–115.

[64] Moore, E. F. Gedanken experiments on sequential machines. In *Automata studies* (Princeton, New Jersey, U.S.A., 1956), Princeton University Press, pp. 129–153.

[65] Morris, J. *Crime Analysis Charting – An Introduction To Visual Investigative Analysis.* Palmer Enterprises, Orangevale, CA, USA, 1988. ISBN 0–912–47901–9.

[66] NASA Office of Safety and Mission Assurance. *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volumes I and II.* Washington, DC, May 1997.

[67] Overman, W. T. *Verification of concurrent systems: function and timing.* PhD thesis, Univeristy College of Los Angeles, 1981.

[68] Patzakis, J. The encase process. In *Handbook of Computer Crime Investigation: Forensic Tools and Technology* (San Diego, CA, USA, 2002), E. Casey, Ed., Academic Press, pp. 53–71. ISBN 0–12–163103–6.

[69] Peled, D. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification* (1993), Springer-Verlag, pp. 409–423.

[70] Rabin, M. O., and Scott, D. Finite automata and their decision problem. *IBM Journal of Research and Development 3*, 2 (1959), 114–125.

[71] Reith, M., Carr, C., and Gunsch, G. An examination of digital forensic models. *International Journal of Digital Evidence 1*, 3 (2002). Journal website `http://www.ijde.org/`.

[72] Romig, S. Computer forensics investigations class handouts, 2000. See url `http://www.net.ohio-state.edu/security/talks/ /2000/2000-12-05_forensic-computer-investigations_lisa /forensics-6up.pdf` (downloaded 25 Oct 2002).

[73] Schneier, B. Attack trees. modelling security threats. *Dr. Dobb's Journal*, 12 (1999).

[74] Sheldon, B. Forensic analysis of windows systems. In *Handbook of Computer Crime Investigation: Forensic Tools and Technology* (San Diego, CA, USA, 2002), E. Casey, Ed., Academic Press, pp. 53–71. ISBN 0–12–163103–6.

[75] Stephenson, P. Modeling of post-incident root cause analysis. *International Journal of Digital Evidence 2*, 2 (2003). Journal website `http://www.ijde.org/`.

[76] Stephenson, P. Putting the horse back in front of the cart. In *Proceedings of the Third Digital Forensic Research Workshop* (Portlend, U.S.A, August 2003).

[77] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of Proceedings of the London Mathematical Society* (London, U.K., 1936–1937), vol. 42, pp. 230–265.

[78] Valmari, A. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification* (1991), Springer-Verlag, pp. 156–165.

[79] Venema, W. File recovery techniques. wanted, dead or alive. *Dr. Dobb's Journal*, 12 (2000).

[80] Weil, M. C. Dynamic time & date stamp analysis. *International Journal of Digital Evidence 1*, 2 (2002). Journal website `http://www.ijde.org/`.

[81] Whitcomb, C. M. An historical perspective of digital evidence: A forensic scientist's view. *International Journal of Digital Evidence 1*, 1 (2002). Journal website `http://www.ijde.org/`.

[82] Wilding, E. *Computer Evidence: a Forensic Investigations Handbook*. Sweet & Maxwell, 1997. ISBN 0–421–57990–0.

[83] Wu, S., and Manber, U. Agrep - a fast approximate pattern-matching tool. In *Proceedings of the Winter 1992 USENIX Conference* (San Francisco, USA, January 1991), pp. 153–162.

Appendix A

Selected ACL2 functions and macros

Given below are logical definitions of some of the primitive ACL2 functions which are used in the Appendix C.1 to formalise event reconstruction. Please refer to [49] for more detail.

A.1 Functions

A.1.1 Logical functions

$$\texttt{(equal x y)} = \begin{cases} \texttt{t} & \text{if } \texttt{x} = \texttt{y} \\ \texttt{nil} & \text{if } \texttt{x} \neq \texttt{y} \end{cases}$$

$$\texttt{(if x y z)} = \begin{cases} \texttt{y} & \text{if } \texttt{x} \neq \texttt{nil} \\ \texttt{z} & \text{if } \texttt{x} = \texttt{nil} \end{cases}$$

$$\texttt{(not x)} = \texttt{(if x nil t)}$$

$$\texttt{(implies x y)} = \texttt{(if x (if y t nil) t))}$$

$$\texttt{(iff x y)} = \texttt{(if x (if y t nil) (if y nil t)))}$$

A.1.2 Integer functions

$$\texttt{(integerp x)} = \begin{cases} \texttt{t} & \text{if } \texttt{x} \text{ is an integer} \\ \texttt{nil} & \text{if } \texttt{x} \text{ is not an integer} \end{cases}$$

$$\texttt{(ifix x)} = \begin{cases} \texttt{x} & \text{if } \texttt{x} \text{ is an integer} \\ 0 & \text{if } \texttt{x} \text{ is not an integer} \end{cases}$$

$$\texttt{(nfix x)} = \begin{cases} \texttt{x} & \text{if } \textit{integerp}(\texttt{x}) \text{ and } \texttt{x} > 0 \\ 0 & \text{if } \neg\textit{integerp}(\texttt{x}) \text{ or } \texttt{x} <= 0 \end{cases}$$

$$\texttt{(1+ x)} = \texttt{(ifix x)} + 1$$

$$\texttt{(1- x)} = \texttt{(ifix x)} - 1$$

$$\texttt{(binary-+ x y)} = \texttt{(ifix x)} + \texttt{(ifix y)}$$

$$\texttt{(binary-* x y)} = \texttt{(ifix x)} * \texttt{(ifix y)}$$

$$\texttt{(unary-- x)} = -\texttt{(ifix x)}$$

$$\texttt{(< x y)} = \begin{cases} \texttt{t} & \text{if } \texttt{(ifix x)} < \texttt{(ifix y)} \\ \texttt{nil} & \text{if } \texttt{(ifix x)} \geq \texttt{(ifix y)} \end{cases}$$

A.1.3 Functions for manipulating ordered pairs

$$\texttt{(consp x)} = \begin{cases} \texttt{t} & \text{if } \texttt{x} \text{ is an ordered pair} \\ \texttt{nil} & \text{if } \texttt{x} \text{ is not an ordered pair} \end{cases}$$

$$\texttt{(atom x)} = \texttt{(not (consp x))}$$

$\texttt{(cons x y)}$ = ordered pair whose first element is $\texttt{x}$ and second element is $\texttt{y}$

$$\texttt{(car x)} = \begin{cases} \text{first element of x} & \text{if } \texttt{x} \text{ is an ordered pair} \\ \texttt{nil} & \text{if } \texttt{x} \text{ is not an ordered pair} \end{cases}$$

$$\texttt{(cdr x)} = \begin{cases} \text{second element of x} & \text{if } \texttt{x} \text{ is an ordered pair} \\ \texttt{nil} & \text{if } \texttt{x} \text{ is not an ordered pair} \end{cases}$$

A.1.4 Functions for manipulating lists

```
(defun len (x) (if (consp x) (+ 1 (len (cdr x))) 0)
```

```
(defun binary-append (x y)
   (if (consp x)
     (cons (car x) (binary-append (cdr x) y))
     y))
```

```
(defun con (x)
   (if (consp x)
     (binary-append (car x) (con (cdr x)))
     nil))
```

A.2 Macros

- `(and a1 a2 ... an)`. This expands to

  ```
  (if a1 (if a2 ... (if an t nil) ... nil) nil)
  ```

  Logically this is equivalent to $\texttt{a1} \wedge \texttt{a1} \wedge \ldots \wedge \texttt{an}$.

- `(or a1 a2 ... an)`.

  ```
  (if a1 t (if a1 t ... (if an t nil) ...))
  ```

  Logically this is equivalent to $\texttt{a1} \vee \texttt{a1} \vee \ldots \vee \texttt{an}$.

- `(+ a1 a2 ... an-1 an)`. This expands to

  ```
  (binary-+ a1 (binary-+ a2 ... (binary-+ an-1 an) ... ))
  ```

  If $\texttt{a1}, \texttt{a2}, \ldots, \texttt{an}$ are numbers, this is equivalent to $\texttt{a1} + \texttt{a2} + \ldots + \texttt{an}$

- `(- a b)` expands to `(binary-+ a (unary-- b))`

- `(let bindings body)`, is equivalent to the result of substituting terms from `bindings` for corresponding variables in the `body`.

  For example, `(let ((a (+ y z))) (equal a b))` expands to

  `(equal (+ y z) b)`.

Appendix B

Prefix based representation of computation sets

This appendix gives formal definition of prefix based representation of computation sets, discusses its properties, and gives algorithms for computing set intersection and backtracing of computation sets represented as lists of prefixes.

B.1 Prefix based representation of computation sets

A prefix based representation of a computation set is a list

$$L_X = (x_0, \ldots, x_{|L_X|-1}) \tag{B.1}$$

whose every element is a non-empty computation: $x_i \in C_T$, and $|x_i| > 0$.
The elements of L_X are called *prefixes*.
The meaning of L_X is the set of all computations whose prefixes are

contained in L_X:

$$X = \bigcup_{i=0}^{|L_X|-1} \{\, c \,|\, c \in C_T,\ |x_i| \leq |c|,\ \text{and for all integer } 0 \leq j < |x_i| \ :\ c_j = (x_i)_j \,\} \tag{B.2}$$

B.2 Basic properties of prefix lists

The following properties follow directly form the equation B.2 and properties of lists

- $c \in X$ if and only if there exists x_i such that for all integer $0 \leq j < |x_i|$, $c_j = (x_i)_j$
- $X = \emptyset$ if and only if $L_X = \epsilon$. As a result, set emptiness of X can be checked in constant time by checking emptiness of L_X.
- Let X and Y be computation sets represented by lists L_X and L_Y respectively, then concatenation $L_X \cdot L_Y$ represents $X \cup Y$.
- Let $a \in C_T$ and $b \in C_T$ be two prefixes, and let A and B be the sets of computations represented by (a) and (b) respectively. Observe that
  - $A \subseteq B$ if b is a prefix of a
  - $b \subseteq A$ if a is a prefix of b
  - A and B have no common elements if neither b is a prefix of a, nor a is a prefix of b.

As a result,

$$A \cap B = \begin{cases} A & \text{if } b \text{ is a prefix of } a \\ B & \text{if } a \text{ is a prefix of } b \\ \emptyset & \text{otherwise} \end{cases}$$

Representation of C_T With prefix-based representation, the set C_T can be represented by the list

$$L_{C_T} = (((q_0, \iota_0)) \ldots ((q_{|Q\times I|}, \iota_{|Q\times I|})))$$

where $q_i \in Q$ is a state, $\iota_i \in I$ is an event, and there is an element $((q_i, \iota_i))$ for every possible combination of state and event. Since every computation begins with some state–event pair, it has to be represented by one of the elements in L_{C_T}.

Clearly, C_T has more than one possible representation. Instead of using the list of all singleton prefixes $((q, \iota))$ it is possible to use a list that contains all prefixes of length 2, 3, or any fixed length. The number of elements in such lists will increase exponentially with the length of prefixes. Observe that the list of all possible computation prefixes of length m consists of $|Q||I|^m$ elements, because transition function identifies $|I|$ possible next states for every possible state of the machine.

Representation of a set of computations with restricted prefixes Consider a set of computations P that restricts only the first m elements of its member computations:

$$P_m = \{\, c \,|\, c \in C_T \text{ where } c_0, \ldots, c_{m-1} \text{ satisfy condition } f(c_0, \ldots, c_{m-1})\}$$

A prefix-based representation L_{P_m} of the set P_m can be constructed by listing all prefixes $(c_0, \ldots, c_{m-1})$ that satisfy $f(c_0, \ldots, c_{m-1})$. Since $P_m \subseteq C_T$, the number of elements in L_{P_m} is less or equal than the number of elements in the representation of C_T with prefixes of length m:

$$|L_{P_m}| \leq |Q||I|^m$$

```
1: function IntersectPrefixes(x, y)
2:     for i ← 0 to min(|x|, |y|) step 1 do
3:         if x_i ≠ y_i then
4:             return ϵ
5:         end if
6:     end for
7:     if |x| > |y| then
8:         return x                                  ▷ because x ⊂ y
9:     else
10:        return y                                  ▷ because y ⊆ x
11:    end if
12: end function
```

Figure B.1: Algorithm for computing *IntersectPrefixes*(x, y)

Set intersection. Observe that, by distributivity of $\cap$ over $\cup$ and by the equation B.2, the intersection of two sets X and Y represented by L_X and L_Y can be computed as a union of pair-wise intersections of sets represented by the elements of L_X and L_Y.

Function *IntersectPrefixes*(x, y) computes a prefix that represents the intersection of sets represented by its argument prefixes x and y. The algorithm for computing it is given in Figure B.1. Assuming that all operations in *IntersectPrefixes*(x, y) are constant time operations, the worst case running time of *IntersectPrefixes*(x, y) algorithm is $O(min(|x|, |y|)$, because the loop in Figure B.1 iterates at most $min(|x|, |y|)$ times.

The algorithm for computing intersection of sets represented by lists L_X and L_Y is given in Figure B.2. The running time of this algorithm is proportional to the lengths of both lists L_X and L_Y and the running time of *IntersectPrefixes*(x, y). Since the running time of *IntersectPrefixes*(x, y) is bounded by $O(min(x, y))$, the running time of $\bigcap(L_X, L_Y)$ is bounded by $O(|L_X||L_Y|m)$, where $m = max(min(x_i, y_j)))$ for all $0 \leq i < |L_X|$, and $0 \leq j < |L_Y|$. Clearly, m is less or equal than the length of the longest prefix in both L_X and L_Y.

```
 1: function ⋂(L_X, L_Y)
 2:     result ← ∅
 3:     for each prefix x in L_X do
 4:         for each prefix y in L_Y do
 5:             z ← IntersectPrefixes(x, y)
 6:             if z is not empty then
 7:                 result ← result · (z)
 8:             end if
 9:         end for
10:     end for
11: end function
```

Figure B.2: Algorithm for computing $X \cap Y$

```
 1: function Ψ^{-1}(L_X)
 2:     result ← ∅
 3:     for each pattern x in L_X do
 4:         q ← the first state in x
 5:         for each state p and event ι, such that δ(p, ι) = q do
 6:             Create new prefix x' ← ( (p, ι) ) · x
 7:             result ← result · (x')
 8:         end for
 9:     end for
10:     return result
11: end function
```

Figure B.3: Algorithm for computing $\Psi^{-1}(X)$

Computing $\Psi^{-1}(X)$. The algorithms for computing $\Psi^{-1}(X)$ using prefix-based representation of computation sets is given in Figure B.3. It is a direct implementation of definition of $\Psi^{-1}(X)$ given in Chapter 6. Assuming that all operations in the algorithm take constant time[1], both the running time and the number of created prefixes are $O(|Q||I||L_X|)$, because the outer loop iterates $|L_X|$ times, the inner loop iterates $|Q||I|$ times, and each iteration of the inner loop produces at most one element of the result.

[1] Observe that $\delta(q, \iota)$ can be implemented as a table lookup.

Appendix C

Source code

This appendix contains ACL2 scripts and Common Lisp programs encoding concepts and algorithms described in this dissertation.

C.1 fd.lisp

```
;*****************************************************************
;*       BASIC DEFINITIONS OF EVENT RECONSTRUCTION THEORY
;*****************************************************************

(in-package "ACL2")

; ======================== SYSTEM MODEL ============================

(encapsulate (((cp *) => *)
              ((psi *) => *)
              ((wc *) => *)
              ((dp *) => *)
              ((sid * *) => *)
              ((emp *) => *)
              ((int * *) => *)
              ((uni * *) => *)
              ((rev *) => *)
              ((fwd *) => *)
              (($) => *))

; System model is defined by three functions: cp(), psi() and wc()
;
; cp(c) => {t/nil}  is true for "proper" computations, and false for all
;                     other objects. The set of all proper computations
;                     is closed under psi(x), i.e. (cp x) <=> (cp (psi x))
;
; psi(c0) => c1       is a transition function that defines exactly one
;                     successor computation for every computation
;                     (either proper or not) c0. If input sequence of
;                     c0 is empty, c1 == c0.
;
; wc(c1) => c0        is a witness function that returns one of
;                     predecestor computations for the given
;                     computation c1.
```

```
;                    psi(wc(c1)) = c1.

(local (defun cp (c) (if (equal c 'comp-universe) nil t)))
(local (defun psi (c0) c0))
(local (defun wc (c1) c1))

(defthm cp-is-boolean
  (booleanp (cp c))
  :rule-classes :type-prescription)

(defthm psi-wc-cancel
  (equal (psi (wc c)) c))

; set of proper computations is closed under psi

(defthm cp-psi-cp-x
  (equal  (cp (psi x)) (cp x)))

; =============== DESCRIPTIONS OF COMPUTATION SETS ==================

; There are eight functions for manipulating descriptions of
; computation sets.
;
; dp (d) => {t/nil}    is true for all proper descriptions and
;                      false otherwise. Sets of proper descriptions and
;                      proper computations are disjoint:
;                      (dp x) => (not (cp x))
;
; $()                  this constant function denotes description
;                      of all proper computations C_T. Note that
;                      (dp ($)) and (cp x) <=> (sid x ($))
;
; sid (c d) =>{t/nil}  is true iff given computation
;                      satisfies given description.
;                      Only proper computations can be described:
;                      (sid c d) => (comp c)
;
; emp (d) => {t/nil}   is true iff there are
;                      no computations satisfying given
;                      description
;
; int (d1 d2) => d3    returns description of intersection
;                      of computation sets described by d1 and d2
;
; uni (d1 d2) => d3    returns description of union
;                      of computation sets described by d1 and d2
;
; fwd (d0) => d1       returns description of
;                      all immediate successors
;                      of computations in d0.
;
; rev (d1) => d0       returns description of all
;                      immediate predecestors
;                      of computaitons in d1.

; ----- base function witnesses -----

(local (defun dp (d) (if (equal d 'comp-universe) t nil)))

(local (defun $ () 'comp-universe))

(local (defun sid (c d)
         (if (cp c) (if (equal d 'comp-universe) t nil) nil)))

(defthm dp-is-boolean
  (booleanp (dp d))
  :rule-classes :type-prescription)
```

```
(defthm sid-is-boolean
  (booleanp (sid c d))
  :rule-classes :type-prescription)

(defthm $-is-description
  (dp ($)))

(defthm $-contains-all-and-only-comps
  (equal (sid c ($)) (cp c)))

(defthm descr-contain-only-comps
  (implies (sid c d) (cp c)))

(defthm descs-arent-comps
  (implies (dp x) (not (cp x))))

; ----- description manipulation function witnesses -----

(local (defun emp (d) (declare (ignore d)) nil))

(defthm emp-is-boolean
  (booleanp (emp d))
  :rule-classes :type-prescription)

(defthm emp-is-vacuous
  (implies (emp d)
           (not (sid c d))))

(defthm $-is-not-empty
  (not (emp ($))))

(local (defun int (d1 d2)
         (if (equal d1 ($)) d2 (if (equal d2 ($)) d1 nil))))

(defthm int-semantics
  (equal (sid c (int d1 d2))
         (and (sid c d1)
              (sid c d2))))

(in-theory (disable int-semantics))

(local (defun uni (d1 d2)
         (if (or (equal d1 ($)) (equal d2 ($))) ($) nil)))

(defthm uni-semantics
  (equal (sid c (uni d1 d2))
         (or (sid c d1)
             (sid c d2))))

(in-theory (disable uni-semantics))

(local (defun fwd (d0) d0))

(defthm fwd-semantics
  (equal (sid c (fwd d))
         (sid (wc c) d)))

(local (defun rev (d1) d1))

(defthm rev-semantics
  (equal (sid c (rev d))
         (sid (psi c) d)))
)

; ================== FORMALISATION OF EVIDENCE ====================
```

```
; 1. RUN
;
; A sequence of computations is a run if every computation in it
; (except the first computation) is the successor of the
; immediately preceding computation
;
; Runs are represented by a lists. This representation is
; appropriate, because event reconstruction only deals with
; finite runs.

(defun runp-tail (ctail c0)
   (if (atom ctail)
     t
     (and (equal (car ctail) (psi c0))
          (runp-tail (cdr ctail) (car ctail)))))

(defun runp (c) (runp-tail (cdr c) (car c)))


; 2. OBSERVATION

; An observation is a triple (p m o), where p is
; a description of observed property (set of computations
; possessing that property), m and o are natural
; numbers.
;
; To simplify formalisation, p, m and o are allowed to be
; arbitrary objects in ACL2 universe, assuming that
; non-descriptions represent empty sets, and that non-naturals
; represent 0.

; Thus, a proper observation is just a triple

(defun observp (ob)
    (= 3 (len ob)))

; A run satisfies an observation if
;
; 1) all its elements satisfy p

(defun list-in-desc (l p)
  (if (atom l)
    t
    (and (sid (car l) p)
         (list-in-desc (cdr l) p))))

; 2) if length L of the run us such that  m <= L <= (m+o)

(defun rio (r observ)
  (let ((p (car observ))
        (m (car (cdr observ)))
        (o (car (cdr (cdr observ)))))

     (and (runp r)
          (observp observ)
          (list-in-desc r p)
          (<= (nfix m) (len r))
          (<= (len r) (+ (nfix o) (nfix m))))))


; 3. PARTITIONED RUN

; To define a partitoined run we need a funciton that
; concatenates elements of list in the order of listing

(defun con (l)
  (if (atom l)
```

```
    nil
    (append (car l) (con (cdr l)))))

; A partitioned run is a list such that concatenation of
; its elements gives a run

(defun partitionp (r)
  (runp (con r)))

; Every element of a partitioned run is a run

(defun list-of-runs (l)
  (if (atom l)
    t
    (and (runp (car l))
         (list-of-runs (cdr l)))))

(defthm partition-list-of-runs
  (implies (partitionp r)
           (list-of-runs r)))


; 4. OBSERVATION SEQUENCE

; An observation sequence is a list of observations.

(defun obsp (s)
  (if (atom s)
      t
      (and (observp (car s))
           (obsp (cdr s)))))

; Observations in observation sequence are listed in temporal order.
; Thus, a partitioned run (rl) explains observation sequence (s) if their
; lengths are equal and every element of partitioned run explains corresponding
; element of observation sequence

(defun runlst-in-obs (rl s)
  (cond ((atom rl)(atom s))
        ((atom s)(atom rl))
        (t (and (rio (car rl) (car s))
                (runlst-in-obs (cdr rl) (cdr s))))))

(defun pio (r s)
  (and (partitionp r)
       (runlst-in-obs r s)))


; 6. SEQUENCE OF PARTITIONED RUNS

; A sequence of partitioned runs is list whose elements partition the same run

(defun plistp-t (r pl)
  (if (atom pl)
    t
    (and (partitionp (car pl))
         (equal r (con (car pl)))
         (plistp-t r (cdr pl)))))

(defun plistp (pl)
  (plistp-t (con (car pl)) pl))

; 5. EVIDENTIAL STATEMENT

; An evidential statement is a list of observation sequences.

(defun esp (e)
```

```
  (if (atom e)
      t
      (and (obsp (car e))
           (esp (cdr e)))))

; A sequence of partitioned runs (v) explains evidential statement (e) if
; their lengths are equal and
; 1) all elements of the sequence of partitioned runs partition the same run,
;    i.e. (con v_0) = (con v_1) = ... = (con v_n) = u
; 2) every partitioned run explains corresponding
;    observation sequence in the evidential statement

(defun vie-tail (u v e)
  (cond ((atom v)(atom e))
        ((atom e)(atom v))
        (t (and (equal (con (car v)) u)
                (pio (car v) (car e))
                (vie-tail u (cdr v) (cdr e))))))

; function vie(v e) calculates u = (con v_0) and uses
; vie-tail(u v e) to check that v explains e

(defun vie (v e)
  (vie-tail (con (car v)) v e))
```

C.2 util.lisp

```
;********************
;* Helper functions
;********************

; append suff to the end of each element in lst
(defun combine (lst suff)
  (if (atom lst)
    nil
    (cons (cons (car lst) suff)
  (combine (cdr lst) suff))))

; make list of all possible pairwise
; combinations of the  elements of two lists
(defun product (l1 l2)
  (if (atom l2)
    nil
    (append (combine l1 (car l2)) (product l1 (cdr l2)))))

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

; similar to combine, but uses append instead of cons-ing
; and listifies the result
(defun combine-append (lst suff)
  (if (atom lst)
    nil
    (cons (cons (append (car lst) suff) nil)
  (combine-append (cdr lst) suff))))

; prepends each element in the given gist with the given prefix
(defun prepend (pref lst)
   (if (atom lst)
     nil
     (cons (cons pref (cons (car lst) nil)) (prepend pref (cdr lst)))))

(defun zp (x)
  (if (integerp x) (eq x 0) t))

(defun take (n lst)
  (if (zp n)
    nil
    (cons (car lst) (take (- n 1) (cdr lst)))))

; defconst macro for use in ordinary Common Lisp environment
(defmacro defconst (x y) `(defvar ,x ,y))
```

C.3 rec.lisp

```
;*****************************
;* Event reconsturciton algorithm
;*****************************

; proper computation predicate

(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))))))


(defvar *ALL-SINGLE-STEP-PATT*
          (listify-elements *ALL-EVENT-STATE-PAIRS*))


;
; partial list description language
;
; "true" lists denote themselves
; "untrue" lists denote patterns -- sets of lists

; proper description -- a list of patterns

(defun dp (d)
  (if (atom d)
    t
    (and (cp (car d))
 (dp (cdr d)))))

; a pattern is a special kind of computation. It is assumed
; that a pattern denotes all computations that begin with it.
; the following function 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))))))

; a computation matches description if it matches one of the
; patterns

(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)
    p1
    (if (matches p2 p1)
```

```
      p2
      nil)))

(defun intpd (p d acc)
  (if (atom d)
    acc
    (let ((i (intpp p (car d))))
      (if (null i)
        (intpd p (cdr d) acc)
        (intpd p (cdr d) (cons i acc))))))

(defun intdd-helper (d1 d2 acc)
  (if (atom d1)
    acc
    (intdd-helper (cdr d1) d2 (intpd (car d1) d2 acc))))

(defun intdd (d1 d2)
  (intdd-helper d1 d2 nil))

; union of two descriptions

(defun uindd (d1 d2)
  (append d1 d2))

; test for emptiness of a description

(defun emp (d) (atom d))

; single-step reverser

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

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

; multi-step reverser

(defun revers (os d)
  (if (emp d)
    nil
    (if (atom os)
      d
      (revers (cdr os) (intdd (car os) (rev d))))))


; convert observation into list of single-step observations

(defun single-step-obs (p n)
  (if (zp n)
    nil
    (cons p (single-step-obs p (1- n)))))

(defun single-step-os (fos)
  (if (atom fos)
    nil
    (append (single-step-obs (first (car fos))
                             (second (car fos)))
            (single-step-os (cdr fos)))))


; convert generic observation sequence into equivalent
```

```
; list of fixed-length observation sequences

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

(defun fix-os (os)
  (if (atom os)
    (cons nil nil)
    (product (fix-obs (first (car os))
                      (second (car os))
                      (third (car os)))
             (fix-os (cdr os)))))


; solving observation sequence

(defun fos-lengths (fos)
  (if (atom fos)
    nil
    (cons (second (car fos)) (fos-lengths (cdr fos)))))

(defun fos-total-len (fos-length-list)
   (if (atom fos-length-list)
     0
     (+ (car fos-length-list)
        (fos-total-len (cdr fos-length-list)))))

(defun solve-fix-os (fos)
  (cons (cons (fos-lengths fos)
         (cons (revers (reverse (single-step-os fos)) '(nil)) nil)) nil))

(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)))))

(defun solve-os (os)
  (solve-fix-os-list (fix-os os)))

(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))))

;(comp 'select-comps)

(defvar *ALL-TWO-STEP-PATT*
  (select-comps
    (product *ALL-EVENT-STATE-PAIRS*
             *ALL-SINGLE-STEP-PATT*) nil))


(defmacro defpatt1-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))))

(defmacro defpatt1 (name vars body)
  (let
    ((tester-name
      (intern
       (concatenate 'string (symbol-name name) "-TESTER"))))
    `(defpatt1-helper ,name ,tester-name ,vars ,body)))

(defmacro defpatt2-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))))

(defmacro defpatt2 (name vars body)
  (let
    ((tester-name
      (intern
       (concatenate 'string (symbol-name name) "-TESTER"))))
    `(defpatt2-helper ,name ,tester-name ,vars ,body)))


; intersecting solutions of two observation sequences


(defun singleton-es-chunk (os-chunk)
   (cons (cons (car os-chunk) nil)
      (cons (cadr os-chunk) nil)))

(defun singleton-es-sol (os-sol)
   (if (atom os-sol)
     nil
     (cons (singleton-es-chunk (car os-sol))
           (singleton-es-sol (cdr os-sol)))))

(defun add-chunk (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
         (cons
          (cons (cons (car os-chunk) (car es-chunk))
                (cons intersection nil))
          nil)))
     nil))

(defun add-chunk-to-sol (es-sol os-chunk)
   (if (atom es-sol)
     nil
     (append (add-chunk os-chunk (car es-sol))
             (add-chunk-to-sol (cdr es-sol) os-chunk))))
```

```
(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))))

(defun solve-es (es)
  (if (atom es)
    nil
    (if (atom (cdr es))
      (singleton-es-sol (solve-os (car es)))
      (add-sol (solve-os (car es)) (solve-es (cdr es))))))


; --- debugging tools ---

; stepper function
(defun stn (cl s)
  (if (atom cl)
    s
    (stn (cdr cl) (st (car cl) s))))
```

C.4 acme.lisp

```
;**************************
;* Printer analysis example
;**************************

(load "util")

; ======== Finite State Machine =======


; Proper state predicate

(defun valuep (v)
  (or (equal v 'empty)
      (equal v 'A)
      (equal v 'B)
      (equal v 'B_deleted)
      (equal v 'A_deleted)))

(defun statep (s)
  (and (equal (cdr (cdr s)) nil)
       (valuep (first s))
       (valuep (second s))))

; Proper event predicate

(defun eventp (e)
  (or (equal e 'add_A)
      (equal e 'add_B)
      (equal e 'take)))


; Set of all possible event-state pairs

(defconst *ALL-EVENT-STATE-PAIRS*
  (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))))))

; Transition funciton

(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))))


; 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))))


; ---- Loading generic reconsturciton algorithm ------

(load "rec")

; ---- Loading drawing utility -----

(load "draw")

; ===== Formalisation of evidence =====

 ; The value of infinitum

 (defconst *infinitum* 6)

 ; The set of all computations

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

 ; Carl's story

 (defpatt1 *B-DELETED*
        (x) (and (equal (first (second x)) 'B_deleted)
                 (equal (second (second x)) 'B_deleted)))

 (defconst *OS-CARL*  `((,*C_T* 0 ,*infinitum*) (,*B-DELETED* 1 0)))

 ; Manufacturer's story

 (defpatt1 *EMPTY*
        (x) (and (equal (first (second x)) 'empty)
                 (equal (second (second x)) 'empty)))


 (defconst *OS-MANU* `((,*EMPTY* 1 0) (,*C_T* 0 ,*infinitum*)))

 (defconst *ES-ACME* `(,*OS-MANU* ,*OS-CARL*))

 ; ============ Investigative hypothesis =================

 ; Alice's claim restricted to exclude speculative transitions

 (defpatt2 *ALICE-PRIME* (x y)
        (and (not (or (equal (first (second x)) 'A)
                      (equal (second (second x)) 'A)))
             (not (equal (second x) (second y)))
             (not  (and (equal (first (second x)) 'B_deleted)
                        (equal (second (second x)) 'B_deleted)
                        (equal (first x) 'Add_B)
```

```
                      (equal (first (second y)) 'B)
                      (equal (second (second y)) 'B_deleted)
                      (equal (first y) 'take)))))

(defconst *OS-PRIME-ALICE* `((,*ALICE-PRIME* 0 ,*infinitum*)
                          (,*B-DELETED* 1 0)))

(defconst *ES-PRIME-PRIME-ACME* (cons *OS-PRIME-ALICE* *ES-ACME*))

; ========== Computing meanings of evidential statements =========

(defconst *ES-ACME-SOL* (solve-es *ES-ACME*))
(defconst *ES-PRIME-PRIME-ACME-SOL* (solve-es *ES-PRIME-PRIME-ACME*))

(print "Is the meaning of es_ACME empty ?")
(print (null *ES-ACME-SOL*))

(print "Is the meaning of es''_ACME empty ?")
(print (null *ES-PRIME-PRIME-ACME-SOL*))
```

C.5 ft.lisp

```
;*********************************************************
;*         TEMPORAL BOUNDING OF EVENTS
;*********************************************************

;(in-package "ACL2")

; ACL2 functions not defined in CMU CL

(DEFUN MEMBER-EQUAL (X LST)
  (COND ((ENDP LST) NIL)
        ((EQUAL X (CAR LST)) LST)
        (T (MEMBER-EQUAL X (CDR LST)))))


(DEFUN NFIX (X)
  (IF (AND (INTEGERP X) (>= X 0)) X 0))

;(include-book "fd-tst")

; make list of all observation id's for given observation sequence obs.
; m is the index of obs in the evidential statement;
; n is the index of the first element of obs.

(defun allobs (obs m n)
  (if (atom obs)
    nil
    (cons (list (nfix m) (nfix n))
          (allobs (cdr obs) (nfix m) (+ (nfix n) 1)))))

; make list of observation id's for given evidential statement es.
; n is the index of the first observation sequence in es.

(defun alles (es n)
  (if (atom es)
    nil
    (append (allobs (car es) (nfix n) 0)
            (alles (cdr es) (+ (nfix n) 1)))))

; intersection-equal (taken from books/data-structures/set-defuns.lisp)
; returns list whose elements are members of both x and y

(defun intersection-equal (x y)
  ;(declare (xargs :guard (and (true-listp x) (true-listp y))))
  (cond ((endp x) nil)
((member-equal (car x) y)
 (cons (car x) (intersection-equal (cdr x) y)))
(t (intersection-equal (cdr x) y))))

; sum first n elements of the list l

(defun sumpref (n l)
  (if (or (zp n) (atom l))
    0
    (+ (nfix (car l)) (sumpref (1- n) (cdr l)))))

; ==== Calculating the lower (earliest) boundary ====


; Find id's of all runs in the given partition map pm,
; whose last computation appears in the partitioned run at a position
; less than or equal to pos.
;
; pm is the partition map;
; i is the index of the partition map in the partition map list;
; j is the index of the first run in the partition;
```

```
; cnt is the position of the first run of pm in the partitioned run.

(defun befpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= (+ cnt (car pm)) pos)
      (cons (list i j)
            (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (befpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

; find id's of all runs in the given partition map list pml,
; whose last computation appears in the partitioned run at a position
; less than or equal to pos.
;
; i is the index of the first partition map in pml.

(defun befpml (pos pml i)
  (if (atom pml)
    nil
    (append (befpm (car pml) 0 pos i 0)
            (befpml pos (cdr pml) (+ i 1)))))

; For the given list of partition list chunks v,
; find id's of all runs that precede run with ID=(i j) in
; all partition chunks of v.
;
; bef is the set of id's that passed test so far. Initially
;       bef is set to the set of all observation ids.
;       At each step, it is intersected with the set of
;       id'satisfying test for the current partition list chunk.

(defun findbef (v bef i j)
  (if (atom v)
    bef
    (findbef (cdr v)
             (intersection-equal
                bef
                (befpml (sumpref j (nth i (car (car v))))
                        (car (car v))
                        0))
             i j)))

; Find in the associative list tim of known observation times
; the maximal time whose id belongs to the set l.

(defun maxtime (max l tim)
  (if (atom tim)
    max
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (maxtime max l (cdr tim))
        (if (null max)
          (maxtime time l (cdr tim))
          (if (< max time)
            (maxtime time l (cdr tim))
            (maxtime max l (cdr tim)))))))))


; Find the earliest time boundary for observation
; with ID=(i j) in evidential statement es, given list v of
; partition list chunks and list of known observation times tim.

(defun lbound (i j es v tim)
  (maxtime nil
           (findbef v (alles es 0) i j)
           tim))
```

```
#|
(lbound 2 1
        '((((1 2 3) 1 25) ((3 4) 1 25))
          (((1 2 3 4) 3 25))
          ((universe 0 25) ((2) 1 25) (universe 0 25)))

        '(((4) ((1 3) (4) (2 1 1)))
          ((4) ((2 2) (4) (2 1 1)))
          ((3) ((1 2) (3) (1 1 1)))
          ((4) ((1 2) (3) (2 1 0)))
          ((4) ((2 1) (3) (2 1 0))))

        '(((0 0) 6) ((2 2) 7) ((0 0) 8)))

Answer: 8
|#

; ==== Calculating the upper (latest) boundary ====

; Find id's of all runs in the given partition map pm,
; whose first computation appears in the partitioned run at a position
; greater than or equal to pos.
;
; pm is the partition map;
; i is the index of the partition map in the partition map list;
; j is the index of the first run in the partition;
; cnt is the position of the first run of pm in the partitioned run.

(defun aftpm (pm cnt pos i j)
  (if (atom pm)
    nil
    (if (<= pos cnt)
      (cons (list i j)
            (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))
      (aftpm (cdr pm) (+ cnt (car pm)) pos i (+ j 1)))))

; find id's of all runs in the given partition map list pml,
; whose first computation appears in the partitioned run at a position
; greater than or equal to pos.
;
; i is the index of the first partition map in pml.

(defun aftpml (pos pml i)
  (if (atom pml)
    nil
    (append (aftpm (car pml) 0 pos i 0)
            (aftpml pos (cdr pml) (+ i 1)))))

; For the given list of partition list chunks v,
; find id's of all runs that precede run with ID=(i j) in
; all partition chunks of v.
;
; aft is the set of id's that passed test so far. Initially
;       aft is set to the set of all observation ids.
;       At each step, it is intersected with the set of
;       id'satisfying test for the current partition list chunk.

(defun findaft (v aft i j)
  (if (atom v)
    aft
    (findaft (cdr v)
             (intersection-equal
                aft
                (aftpml (sumpref (+ j 1) ; include length of run into sum
                                 (nth i (car (car v))))
                        (car (car v))
```

```
                        0))
             i j)))

; Find in the associative list tim of known observation times
; the minimal time whose id belongs to the set l.

(defun mintime (min l tim)
  (if (atom tim)
    min
    (let ((time (car (cdr (car tim))))
          (id (car (car tim))))
      (if (not (member-equal id l))
        (mintime min l (cdr tim))
        (if (null min)
          (mintime time l (cdr tim))
          (if (< time min)
            (mintime time l (cdr tim))
            (mintime min l (cdr tim))))))))

; Find the latest time boundary for observation
; with ID=(i j) in evidential statement es, given list v of
; partition list chunks and list of known observation times tim.

(defun ubound (i j es v tim)
  (mintime nil
           (findaft v (alles es 0) i j)
           tim))
```

C.6 slack.lisp

```
;*************************************
;* Formalisation of slack space analysis
;*************************************

(load "util")

; ======== Finite State Machine =======


; Proper state predicate

(defun lenvaluep (l)
  (or (equal l 0)
      (equal l 1)
      (equal l 2)))

(defun datavaluep (v)
  (or (equal v 1)
      (equal v 0)))

; The first element of state is the length of
; The rest of the elements is the data store
; e.g. (1 0 0)

(defun statep (s)
  (and (lenvaluep (car s))
       (datavaluep (cadr s))
       (datavaluep (caddr s))
       (null (cdddr s))))

; Proper event predicate

; Event is a sequence of 1s and 0s with at least
; 1 element and at most 2 elements


(defun eventp (e)
  (or (equal e 'del)
      (and (datavaluep (car e))
           (null (cdr e)))
      (and (datavaluep (car e))
           (datavaluep (cadr e))
           (null (cddr e)))))

; Set of all possible event-state pairs

(defconst *LENGTH*      '(0 1 2))

(defconst *ALL-STATES*  '((0 0 0) (0 0 1) (0 1 0) (0 1 1)
                         (1 0 0) (1 0 1) (1 1 0) (1 1 1)
                         (2 0 0) (2 0 1) (2 1 0) (2 1 1)))

(defconst *ALL-EVENTS*  '(del (0) (1) (0 0) (0 1) (1 0) (1 1)))

(defconst *ALL-EVENT-STATE-PAIRS*
  (product *ALL-EVENTS* (listify-elements *ALL-STATES*)))

; Transition funciton


(defun st (c s)
  (if (atom c)
    (if (equal c 'del)      (list 0  (cadr s) (caddr s))
                            s)
    (let ((l (length c)))
```

```
      (if (equal l 1)       (list 1 (car c) (caddr s))
      (if (equal l 2)       (list 2 (car c) (cadr c))
                            s)))))

; Inverse transition function

(defun rev-st (s)
  (let* ((l (car s))
         (data (cdr s)))
    (cond
      ((equal l 0)
         (list (list 'del (list* 0 data))
               (list 'del (list* 1 data))
               (list 'del (list* 2 data))))
      ((equal l 1)
         (list (list (list (car data)) (list* 0 0 (cdr data)))
               (list (list (car data)) (list* 0 1 (cdr data)))
               (list (list (car data)) (list* 1 0 (cdr data)))
               (list (list (car data)) (list* 1 1 (cdr data)))
               (list (list (car data)) (list* 2 0 (cdr data)))
               (list (list (car data)) (list* 2 1 (cdr data)))))
      ((equal l 2)
         (list (list data (list 0 0 0))
               (list data (list 0 0 1))
               (list data (list 0 1 0))
               (list data (list 0 1 1))
               (list data (list 1 0 0))
               (list data (list 1 0 1))
               (list data (list 1 1 0))
               (list data (list 1 1 1))
               (list data (list 2 0 0))
               (list data (list 2 0 1))
               (list data (list 2 1 0))
               (list data (list 2 1 1))))
      (t nil))))

; ---- Loading generic reconsturciton algorithm ------

(load "rec")

; ---- Loading drawing utility -----

(load "draw")

; ---- Loading event time bounding algorithms ----

(load "ft")

; ===== Formalisation of evidence =====

 ; The value of infinitum

 (defconst *infinitum* 4)

 ; The set of all computations

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

 ; Observed properties

 (defpatt1 *FINAL*
        (x) (equal (second x) '(1 0 1)))

 (defpatt1 *BLACKMAIL-WRITE*
        (x) (equal (first x) '(1 1)))
```

```
(defpatt1 *UNRELATED-WRITE*
       (x) (equal (first x) '(0)))

(defpatt1 *NO-UNRELATED-WRITE*
       (x) (not (equal (first x) '(0))))

(defconst *OS-FINAL*           `((,*C_T* 0 ,*infinitum*) (,*FINAL* 1 0)))

(defconst *OS-UNRELATED*       `((,*C_T* 0 ,*infinitum*)
                               (,*UNRELATED-WRITE* 1 0)
                               (,*C_T* 0 ,*infinitum*)
                               (,*C_T* 0 0)
                               (,*C_T* 1 ,*infinitum*)))

(defconst *OS-BLACKMAIL*       `((,*C_T* 0 ,*infinitum*)
                               (,*BLACKMAIL-WRITE* 1 0)
                               (,*C_T* 1 ,*infinitum*)))

(defconst *OS-PRIME-UNRELATED* `((,*NO-UNRELATED-WRITE* 0 ,*infinitum*)
                               (,*UNRELATED-WRITE* 1 0)
                               (,*NO-UNRELATED-WRITE* 0 ,*infinitum*)
                               (,*NO-UNRELATED-WRITE* 0 0)
                               (,*NO-UNRELATED-WRITE* 1 ,*infinitum*)))

(defconst *ES-BLACKMAIL* `( ,*OS-FINAL*
                          ,*OS-UNRELATED*
                          ,*OS-BLACKMAIL-OBS* ))

(defconst *ES-PRIME-BLACKMAIL* `( ,*OS-FINAL*
                                  ,*OS-PRIME-UNRELATED*
                                  ,*OS-BLACKMAIL* ))


; Compute the meanings of the evidential statements

(defconst *SOL-4* (solve-es *ES-BLACKMAIL*))

(defconst *SOL-PRIME-4* (silve-es *ES-PRIME-BLACKMAIL*))

; Run the time bounding algorithm

; 5 is the time of the reception of unrelated event
; the times of all other events are unknown.
(defconst *tim* '(((1 3) 5)))

; time bouding of blackmail-write in es_blackmail

(defconst *l*
  (lbound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))

(defconst *u*
  (ubound 2 1 *ES-BLACKMAIL* *SOL-4* *tim*))

; time bounding of blackmail-write in es'_blackmail

(defconst *l-prime*
  (lbound 2 1 *ES-PRIME-BLACKMAIL* *SOL-PRIME-4* *tim*))

(defconst *u-prime*
  (ubound 2 1 *ES-PRIME-BLACKMAIL* *SOL-PRIME-4* *tim*))
```

C.7 draw.lisp

```
;*****************************************
;* Drawing reconstruction results using DOT
;*****************************************

(defun digit-name (digit)
  (case digit
    (0 "0")
    (1 "1")
    (2 "2")
    (3 "3")
    (4 "4")
    (5 "5")
    (6 "6")
    (7 "7")
    (8 "8")
    (9 "9")
    (10 "A")
    (11 "B")
    (12 "C")
    (13 "D")
    (14 "E")
    (15 "F")
    (otherwise "?")))

(defun number-name-helper (n radix)
  (if (equal n 0)
    ""
    (concatenate 'string
                 (number-name-helper (floor n radix) radix)
                 (digit-name (rem n radix)))))
(defun number-name (n radix)
  (if (equal n 0) "0" (number-name-helper n radix)))

(defun pretty (l)
  (if (null l)
    ""
    (if (atom l)
         (if (numberp l) (number-name l 10) (symbol-name l))
      (let ((s (car l)))
        (concatenate
           'string
           (if (symbolp s) (symbol-name s)
             (if (consp s) (concatenate 'string "(" (pretty s) ")")
               (if (numberp s)
                 (number-name s 10)
                 " ??? ")))
           (if (not (null (cdr l))) ", " "") (pretty (cdr l)))))))

(defun stringify (l acc)
   (if (atom l)
     acc
     (stringify (cdr l) (concatenate 'string acc "_"
       (if (numberp (car l))
         (number-name (car l) 10)
         (symbol-name (car l)))))))

(defun flatten (l)
  (if (atom l)
    l
    (if (consp (car l))
      (append (flatten (car l)) (flatten (cdr l)))
      (cons (car l) (flatten (cdr l))))))

(defun draw-comp (file comp name1)
   (if (atom comp)
```

```
     nil
      (progn
        (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
                 file "n~A [label=\"~A\"];~%" name2 (pretty (cadadr comp)))
              (format
                 file "n~A -> n~A [label=\"~A\"];~%"
                      name2 name1
                        (pretty (car (cadr comp))))
              (draw-comp file (cdr comp) name2)))))))

(defun draw-comps (file comps)
   (if (atom comps)
     nil
     (let ((rv (reverse (car comps))))
       (progn
         (draw-comp file rv (stringify (flatten (cdar rv)) ""))
         (draw-comps file (cdr comps))))))


(defun draw-sol (file es-sol)
   (if (atom es-sol)
     nil
     (progn
       (draw-comps file (cadar es-sol))
       (draw-sol file (cdr es-sol)))))

(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 "}~%")
   ))
```

Appendix D

Evidence of publication

The following pages contain a photocopy of [39], in which the results of this dissertation have been published. Please turn the page.