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.

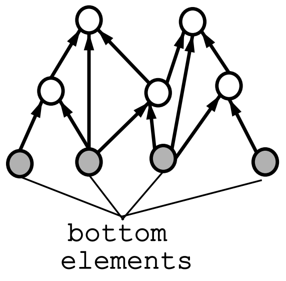

Figure 5.1: Well founded order

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 $f(a, b, c) = \mathit{body}(a, b, c)$ 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;

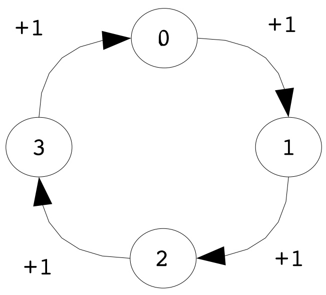

Figure 5.2: Counting state machine

- $\delta : Q \times I \rightarrow Q$ 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

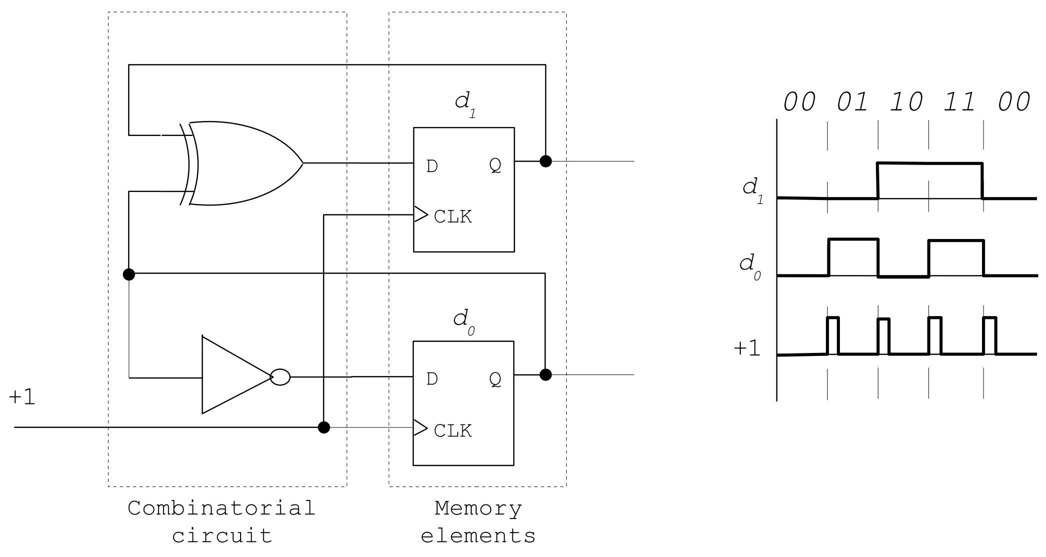

Figure 5.3: A 2-bit binary counter

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.

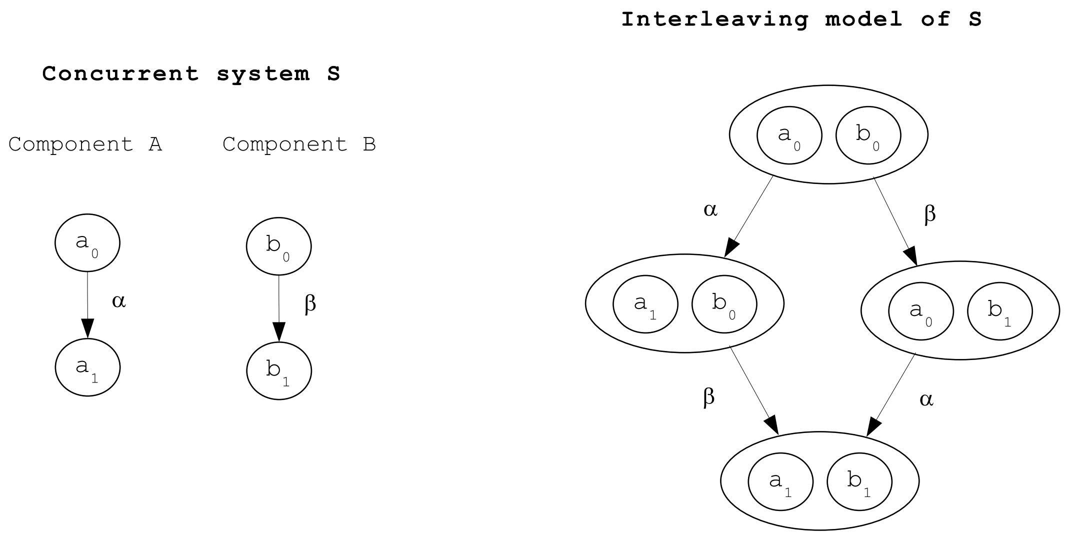

Figure 5.4: Interleaving model of concurrent system

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 $\kappa : Q \to O$ 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

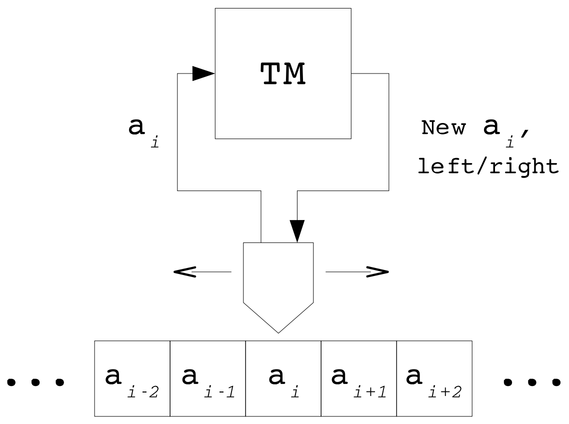

Figure 5.5: Turing machine

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:     $C_{next} \leftarrow \emptyset$
5:     for every computation $c \in C_{current}$ do
6:         $q \leftarrow$ the last state in $c$
7:         for every input symbol $\iota \in I$ do
8:             $p \leftarrow \delta(q, \iota)$
9:             Make new computation $c'$ by suffixing $c$ with a transition $q \xrightarrow{\iota} p$
10:            $C_{next} \leftarrow C_{next} \cup \{c'\}$
11:        end for
12:    end for
13:    $C_{A_k} \leftarrow C_{A_k} \cup C_{next}$
14:    $C_{current} \leftarrow C_{next}$
15: end for
16: for every computation $c \in C_{A_k}$ do
17:     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].

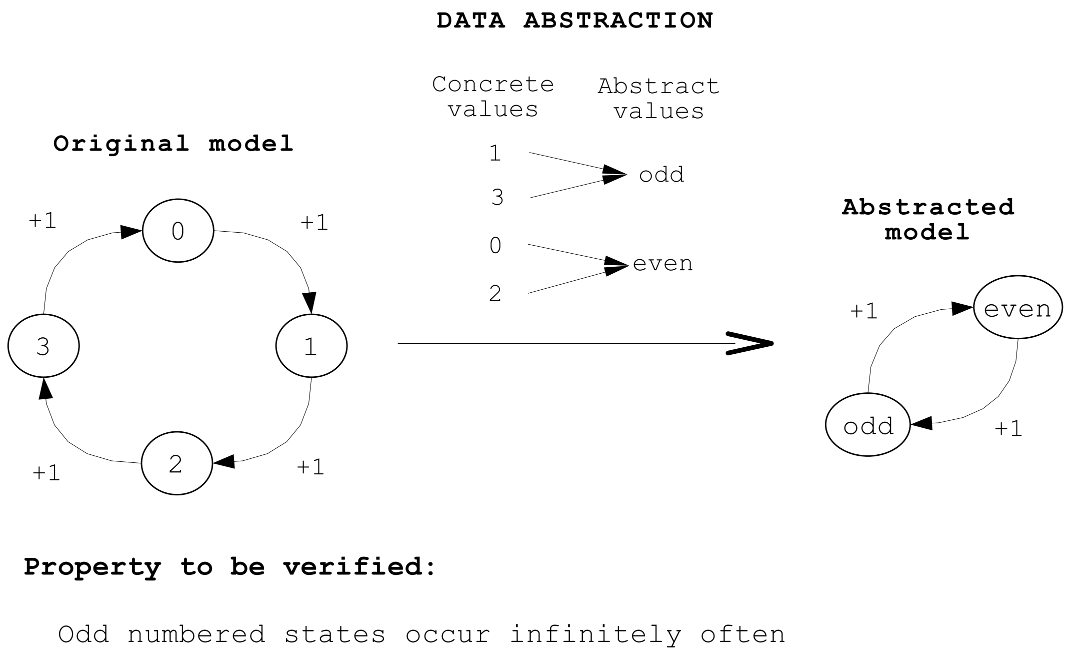

Figure 5.7: Data abstraction

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.