FORMAL VERIFICATION OF A CONCURRENT BINARY SEARCH TREE

XIWEN CHEN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
AUGUST 2013
FORMAL VERIFICATION OF A CONCURRENT BINARY SEARCH TREE

by Xiwen Chen

a thesis submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE
© 2013

Permission has been granted to: a) YORK UNIVERSITY LIBRARIES to lend or sell copies of this dissertation in paper, microform or electronic formats, and b) LIBRARY AND ARCHIVES CANADA to reproduce, lend, distribute, or sell copies of this thesis anywhere in the world in microform, paper or electronic formats and to authorise or procure the reproduction, loan, distribution or sale of copies of this thesis anywhere in the world in microform, paper or electronic formats.

The author reserves other publication rights, and neither the thesis nor extensive extracts for it may be printed or otherwise reproduced without the author’s written permission.
FORMAL VERIFICATION OF A CONCURRENT BINARY SEARCH TREE

by Xiwen Chen

By virtue of submitting this document electronically, the author certifies that this is a true electronic equivalent of the copy of the thesis approved by York University for the award of the degree. No alteration of the content has occurred and if there are any minor variations in formatting, they are as a result of the coersion to Adobe Acrobat format (or similar software application).

Examination Committee Members:

1. Eric Ruppert
2. Franck van Breugel
3. Jonathan S. Ostroff
4. Scott P. Kelly
Abstract

In this thesis, we formally verify a simplified version of the non-blocking linearizable binary search tree of Ellen et al., which appeared in the Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing (pages 131-140), using the PVS specification and verification system. The algorithm and its specification are both modelled as I/O automata. In order to formally verify that the algorithm implements the specification, we show that the algorithm’s I/O automaton simulates the specification’s. An intermediate I/O automaton is constructed to simplify the simulation proof of linearizability. By showing there is a forward simulation from the algorithm’s I/O automaton to the intermediate automaton and there is a backward simulation from the intermediate automaton to the specification’s automaton, we formally verify that the algorithm implements its specification. While formalizing the proof, we found small errors in the original proof.
Acknowledgements

I would like to express my very great appreciation to Professor Eric Ruppert, my research supervisor, for his patient guidance, valuable and constructive suggestions for this research. Without his advice and help, this thesis would not have been completed.

I am particularly grateful for the assistance of Professor Franck van Breugel, who broadened my horizons and helped me to decide on my research area. I would like to thank all members of my examining committee for their helpful comments on my thesis.

Last but not least, I would like to acknowledge the support provided by my family for my thesis.
Table of Contents

Abstract iv

Acknowledgements v

Table of Contents vi

1 Introduction 1

1.1 Formal Verification 3

1.2 Previous Work 6

1.3 Overview of the Thesis 8

2 Proving Linearizability Using Simulations 10

2.1 Model of Computation 10

2.2 Data Types 11

2.3 Input/Output Automata 16

2.3.1 Concurrent Implementations and Linearizability 18

2.3.2 Canonical Automata 24
6 Conclusion 121

Bibliography 125
1 Introduction

With the arrival of the multi-core central processing unit (CPU) revolution, a considerable fraction of the applications developed today is concurrent. “But concurrency is hard. Not only are today’s languages and tools inadequate to transform applications into parallel programs, but also it is difficult to find parallelism in mainstream applications, and—worst of all—concurrency requires programmers to think in a way humans find difficult”, as Microsoft’s Herb Sutter and James Larus wrote in 2005 [1]. Unfortunately, almost a decade later, the above quote still reflects the current state of affairs in the field of software development.

Since concurrency is hard, libraries with concurrency primitives such as a concurrent array, a concurrent queue and a concurrent set are essential for today’s software developer. Such concurrency primitives provide objects that can be accessed concurrently by multiple processes. One popular way of specifying correctness of such a concurrent object is linearizability. This means that it behaves as if operations on it occur instantaneously. For example, to design a concurrent algorithm that stores a searchable set that can be updated by insert and delete operations, which is a very
common scenario, a concurrent implementation of binary search trees will be very helpful.

It has been a long time since researchers realized that traditional approaches, which are based on mutual exclusion, are often not efficient enough and have associated problems such as deadlock, livelock and convoying. To combat these problems, they developed concurrent objects that do not rely on mutual exclusion mechanisms such as locks and semaphores. Lock-free implementations, which avoid mutual exclusion using other techniques to coordinate access by different processes to shared data, provide high parallelism and good performance under a variety of different workloads. There are several different progress properties that a lock-free algorithm can satisfy. The wait-free property \cite{2} ensures that any process can complete any operation in a finite number of steps, regardless of the execution speeds of the other processes. The non-blocking property guarantees that some operation will complete in a finite number of steps. This weaker condition allows individual processes to starve but guarantees system-wide progress. All wait-free algorithms are non-blocking. The terms non-blocking and lock-free are often used interchangeably. We use non-blocking as a term to specify algorithms that satisfy this progress condition and use lock-free as a general term to describe algorithms that do not rely on mutual exclusion. Lock-free algorithms are generally more complex than lock-based algorithms, because intricate parallelism techniques are introduced to avoid locks.
Recently, Ellen et al. [3] gave a non-blocking linearizable implementation of a binary search tree. They provided a proof of correctness written in English which was quite lengthy and complex. The goal of our work is to provide a formal verification of this proof.

1.1 Formal Verification

Since concurrency is hard, it should come as no surprise that concurrent algorithms are prone to errors. Therefore, there are significant challenges to ensure the correctness of concurrent algorithms in general, and lock-free algorithms in particular. For example, Detlefs et al. [4] developed a lock-free double-ended queue, called the Snark algorithm, with a proof of correctness, but bugs were reported later by Doherty [5]. Shann et al. [6] published their non-blocking queue algorithm with safety proofs, but bugs were found when formal verification methods were applied by Colvin and Groves in [7].

Although it is hard to avoid bugs in an algorithm, there are a variety of methods to reduce the chance of making a mistake. Testing is one such method to detect bugs. However, testing all executions of a nontrivial concurrent algorithm is usually infeasible (or even impossible) because there are simply too many. Formal verification is another method of checking whether a design satisfies some properties. It is the act of proving or disproving the correctness of algorithms with respect to a
certain formal specification or property, using formal methods of mathematics. The
two main approaches are model checking and theorem proving.

Model checking builds a mathematical model of the algorithm as a collection of
states and actions that move from one state to another. Then, a model checker per-
forms a systematic, exhaustive exploration of the state space, for example to check
that an invariant is true in all reachable states. Lamport [8] described a way to ver-
ify concurrent algorithms using the +CAL algorithm language and the TCL model
checker and successfully detected the bugs in the Snark algorithm found by Do-
herty et al. [9]. He translated the algorithm into the +CAL algorithm language and
then to a TLA+ specification that can be model checked. Liu et al. [10] presented
their way to check linearizability based on refinement relations from abstract speci-
fications to concrete implementations using model checking methods. Their method
exploits model checking of finite state systems specified as concurrent processes with
shared variables, and partial order reduction is applied to reduce the search space.
The toolset they used can automatically check a variety of algorithms. However,
no tree implementation has been verified in their paper and the tool has not been
published. The approaches that use model checking techniques can help us quickly
discover bugs in some algorithms. However, for complicated concurrent algorithms,
the search spaces are usually too large to be explored in a reasonable amount of
time and using a reasonable amount of space.
Theorem proving (also called deductive reasoning), is the proving of mathematical theorems in a format so that the proof can be checked by a computer program such as PVS, Coq, HOL or Isabelle. Theorem proving is more powerful than model checking because it can deal with an infinite state space more easily. Because the binary search tree of Ellen et al. [3] is a data structure of unbounded keys, and processes, the state space is infinite. So, we use the PVS theorem prover to formalize the binary search tree’s proof of correctness.

PVS is a verification system that contains a specification language and a theorem prover. We used this verification system in our proofs. Compared with other theorem proving systems, such as HOL and Isabelle, PVS has several advantages. First, its specification language allows the user to define things in a way that is similar to programming languages. For instance, we can define a data type using square brackets and specify its fields’ names and types. We can also define data types by explicitly writing \textit{datatype}, and PVS then automatically generates basic axioms for it. Hence, having such a specification language, we can easily formalize pseudocode using PVS and focus on proving correctness. Second, unlike HOL, the PVS theorem prover allows users to define their axioms, which provides more freedom to construct proofs based on some facts without getting stuck on low level details. However, because introducing user defined axioms may introduce inconsistencies, axioms must be very carefully designed. Third, PVS also provides some automatic
reasoning procedures such as \textit{grind}, \textit{assert} and \textit{bddsimp} etc., which simplify the task of proving correctness.

1.2 Previous Work

Lynch and Tuttle \cite{11} introduced I/O automata (IOA), which can be used to model concurrent algorithms and specifications of correctness. To prove that an algorithm correctly implements a specification, one has to show that for every execution of the algorithm’s automaton, the externally observable behaviour is the same as the behaviour of the specification automaton. Lynch and Vaandrager \cite{12} introduced techniques for doing this, including forward simulations and backward simulations.

In Doherty’s Master’s thesis \cite{5}, he introduced a way to use the PVS theorem prover to check the forward simulation between the Snark algorithm \cite{4} and its specification, and detected bugs in the algorithm. A canonical automaton was introduced there to model the specification and capture the property of linearizability. Then, a forward simulation was used to show that the automaton of the concurrent implementation simulates the canonical automaton, thus showing that the implementation is linearizable. (We shall discuss this technique in more detail in Chapter 2.) While the author was trying to prove the correctness of the Snark algorithm using a forward simulation, he detected bugs. He also showed how to fix those bugs, and mentioned that a backward simulation might be needed to complete
the proof. The complete verification of this algorithm was done later in [9].

Doherty et al. [13] used a similar method to verify a queue that is slightly op-
timized from Michael and Scott’s non-blocking FIFO queue [14] using the PVS
theorem prover. In this paper, they formally described a way to verify the non-
blocking algorithm using simulation techniques. Their idea was to introduce three
IOAs: an abstract one which was the same as the canonical automaton, an interme-
diate one and a concrete one which represented the implementation. A backward
simulation was proved between the abstract IOA and the intermediate IOA, and a
forward simulation was verified between the intermediate IOA and the concrete IOA.
This technique was also used in [9] to complete the proof of the correctness of the
modified Snark algorithm. More details about the relationship between backward
simulations and non-blocking algorithms are also discussed in [15].

Colvin et al. [7] used this method (i.e., using three IOAs) to prove the correct-
ness of an array-based non-blocking implementation of a bounded queue by Shann
et al. [6] and also detected errors in the algorithm. After they fixed that algorithm,
they successfully formally verified the modified version using PVS.

In later work of Colvin et al. [16], they used this hybrid forward and backward
simulation technique to verify a lazy concurrent list-based set algorithm [17]. Al-
though this algorithm is based on locks, one of its operations (a contain operation)
has the wait-freedom property.
Although many concurrent algorithms manipulating arrays, queues and lists have been formally verified, no one has tried to formally verify non-blocking binary search tree algorithms so far. Hence, our goal is to formally verify the non-blocking binary search tree algorithm discovered by Ellen et al. [3]. Our idea of how to formally verify the algorithm was inspired by Colvin et al. [16].

Many people have formally verified concurrent algorithms using the PVS theorem prover. Gao used PVS to prove the correctness and progress properties of a lock-free dynamic hash table in his Ph.D. thesis [18]. In his thesis, he also proved the correctness of a lock-free parallel garbage collector using PVS. Archer et al. [19] described a general way to model concurrent algorithms as timed I/O automata using the Tempo toolkits [20] and then verify the properties of the I/O automata in PVS through the interface TAME [21] [22].

1.3 Overview of the Thesis

In this thesis, we first introduce our model of computation in Chapter 2. For a data type, we define its sequential executions, atomic executions and concurrent executions. Once, an implementation of a data type is modelled by means of an I/O automaton, simulations are used to prove the correctness of its executions.

Chapter 3 presents the non-blocking binary search tree algorithm of Ellen et al. [3]. We give an overview of how this algorithm works and then explain some key steps
of that algorithm. A simplified algorithm is given later in this chapter to make our proofs easier.

In Chapter 4, the details of modelling the binary search tree algorithm are presented. Then, we define a forward simulation and a backward simulation in order to prove the correctness of the algorithm. Chapter 5 discusses the proofs for the forward and backward simulation. The difficulties and bugs that we detected during the formal verification using PVS are also discussed there. Chapter 6 gives a summary and describes some of the future work we would like to pursue.
2 Proving Linearizability Using Simulations

2.1 Model of Computation

To model data types and algorithms in a shared-memory architecture, we use an asynchronous shared-memory model. In the asynchronous model, different processes take steps in an arbitrary order, at arbitrary relative speeds. Intuitively, in the asynchronous model, we assume there is a scheduler that determines which process will take the next step. Algorithms and data structures should behave correctly for all possible schedules made by that scheduler. Our model allows failures to model the fact that the systems in which the algorithms are running may not be completely reliable. Therefore, programs need to tolerate faulty behaviour. We consider crash failures: a process executing some code may stop without a warning. (These are also known as halting failures.)

In a shared-memory model, a collection of processes interact with one another via a collection of shared objects [23]. Such an assumption captures the way communication occurs in a multi-core CPU. We assume our system provides some atomic
shared objects, either implemented in hardware or by the operating system. Atomic objects can be accessed concurrently by several processes, but they ensure that each operation performed by the processes occurs atomically. For example, read/write registers are one of the most frequently used types of shared objects in concurrent systems. A read/write register stores a value. A write\( (v) \) operation changes the value to \( v \) and returns \textit{ack}. A read operation returns the value currently stored in the read/write register without changing it. Both write and read operations are atomic. Compare-and-swap is also a popular type of atomic object in multi-core systems. A compare-and-swap (CAS) object \( X \) stores a value from a universe \( U \), and provides only one operation, \( CAS(u, v) \), where \( u \) and \( v \) are in \( U \). A \( CAS(u, v) \) on object \( X \) successfully writes \( v \) into \( X \) if and only if the value previously stored in \( X \) is equal to \( u \). Otherwise, \( CAS(u, v) \) does nothing to the value stored in \( X \). Whether it succeeds or fails, \( CAS(u, v) \) returns the old value of \( X \). We present a formal way to describe the model we discussed here in Section 2.3.

2.2 Data Types

If we wish to use more complex data structures than those provided by the system, we must implement them in software. In order to describe the correctness of a concurrent implementation of a data type, Herlihy and Wing [24] defined a property called \textit{linearizability}. It ensures that every operation on the concurrent objects...
appears to take effect atomically at some point between its invocation and response.

Before we formally define linearizability, we introduce the definition of a sequential specification of a data type, adapted from [23, Chapter 9.4].

**Definition 2.1.** A data type is a tuple \( \langle V, v_0, I, R, f \rangle \) consisting of

- a set \( V \) of values,
- an initial value \( v_0 \in V \),
- a set \( I \) of invocations,
- a set \( R \) of responses, and
- a function \( f: V \times I \rightarrow R \times V \).

Intuitively, an object of type \( \langle V, v_0, I, R, f \rangle \) stores a value from \( V \) and starts with the initial value \( v_0 \). If an invocation \( inv \in I \) is performed when the object’s value is \( v \in V \), then \( f(v, inv) = (res, v') \) describes the outcome of the invocation: the object’s value is changed to \( v' \) and the object returns the response \( res \). For simplicity, we restrict data types to behave deterministically here. This covers most data types encountered in practice.

A data type can be manipulated by either one process or a set of processes. We define a sequential execution of a data type by only one process in Definition 2.2.
**Definition 2.2.** A *sequential execution* of a data type $⟨V,v_0,I,R,f⟩$ is a finite sequence $⟨v_0,inv_1,res_1,v_1,inv_2,res_2,\ldots,v_k⟩$, such that for all $0 \leq j < k$, $f(v_j,inv_{j+1}) = (res_{j+1},v_{j+1})$.

We denote the empty sequence by $\epsilon$ and use $\diamond$ to denote concatenation of sequences. Using the definition of sequential executions of a data type, we can define the notion of a *sequential trace* of a data type. A *trace* is also called a *history* by Herlihy and Wing [24].

**Definition 2.3.** The *trace* of a sequential execution is defined inductively as follows:

- $trace(\epsilon) = \epsilon$, and
- $trace(⟨v,inv,resp⟩\diamond E) = ⟨inv,resp⟩\diamond trace(E)$, for any execution $⟨v,inv,resp⟩\diamond E$.

A sequence is a sequential trace of a data type if it is the trace of some sequential execution of that data type. Here, we give an example of a *fetch and increment* ($fetch\&inc$) data type. A $fetch\&inc$ data type stores an integer value and provides only one type of operation. It atomically performs the following three small steps: (1) read the integer value of the object and store it to a local variable $val$; (2) increase the object’s value by 1; (3) return $val$. More formally, the specification of a $fetch\&inc$ data type is shown in Definition 2.4.

**Definition 2.4.** A *fetch\&inc* data type is defined as follows:
• $V = \mathbb{N}$,
• $v_0 = 0$,
• $I = \{fi\}$,
• $R = \{fi\text{Resp}(n) \mid n \in \mathbb{N}\}$, and
• $f(v, fi) = (fi\text{Resp}(v), v + 1)$, for any $v \in V$.

For instance, a *fetch&inc* data type may have a sequential execution:

$$E_s = \langle 0, fi, fi\text{Resp}(0), 1, fi, fi\text{Resp}(1), 2, fi, fi\text{Resp}(2), 3, fi, fi\text{Resp}(3), 4 \rangle.$$  

The trace $T_s$ corresponding to $E_s$ is:

$$T_s = \langle fi, fi\text{Resp}(0), fi, fi\text{Resp}(1), fi, fi\text{Resp}(2), fi, fi\text{Resp}(3) \rangle.$$

A data type can also be concurrently manipulated by a finite set of processes \textit{PROC}. Its executions, known as \textit{atomic} executions, are defined in Definition 2.5.

**Definition 2.5.** An \textit{atomic execution} of a data type $\langle V, v_0, I, R, f \rangle$ manipulated by a set of processes \textit{PROC} is a finite sequence $\langle v_0, (inv_1, p_1), (res_1, p_1), v_1, (inv_2, p_2), (res_2, p_2), \ldots, v_k \rangle$ such that for all $0 \leq j < k$, $f(v_j, inv_{j+1}) = (res_{j+1}, v_{j+1})$ and for all $1 \leq j \leq k$, $p_j \in \textit{PROC}$.

For instance, an atomic execution of a data type among processes $p, q$ and $r$ may look like:

$$E_a = \langle v_0, (inv, p), (res, p), v_1, (inv, r), (res, r), v_2, (inv, q), (res, q), v_3 \rangle.$$
Or, we can put processes as subscripts for a more compact representation.

\[ E_a = \langle v_0, inv_p, res_p, v_1, inv_r, res_r, v_2, inv_q, res_q, v_3 \rangle. \]

The definition of the atomic trace can also be defined by applying Definition 2.3. Hence, atomic traces are the trace of atomic executions of a data type. For instance, the trace of \( E_a \) is:

\[ T_a = \langle inv_p, res_p, inv_r, res_r, inv_q, res_q \rangle. \]

The invocation and the matching response by \( p \) compose a complete operation of \( p \). For each invocation by process \( p \), the matching response is the next response by \( p \). Intuitively, in an atomic trace, each response of a process \( p \) is immediately preceded by a matching invocation of \( p \).

An example of an atomic execution on a fetch\&inc object manipulated by processes \( p, q, \) and \( r \) is shown below.

\[ \langle 0, fi_p, fiResp_p(0), 1, fi_r, fiResp_r(1), 2, fi_p, fiResp_p(2), 3, fi_q, fiResp_q(3), 4 \rangle. \]

Its atomic trace is:

\[ \langle fi_p, fiResp_p(0), fi_r, fiResp_r(1), fi_p, fiResp_p(2), fi_q, fiResp_q(3) \rangle. \]

Besides sequential executions and atomic executions, there are concurrent executions of implementations of data types. We define concurrent executions and traces after introducing I/O automata.
2.3 Input/Output Automata

The input/output (I/O) automaton is a formal model for asynchronous computing \[23\]. It is a powerful and general model that is suitable for describing almost any type of asynchronous concurrent system. An I/O automaton is a simple type of state machine in which the transitions are associated with named actions, which are classified as external or internal. Following the definitions given by Lynch and Tuttle \[11\] and Doherty \[5\], external actions, which are visible to the outside world, are further classified according to whether they model invocation (input) or response (output) events. On the other hand, internal actions are visible only to the automaton itself. Given three disjoint sets $in$, $out$, and $int$ of input, output and internal actions, respectively, we use the triple $(in, out, int)$ as an action signature $S$. We denote the sets $in$, $out$, and $int$ of the action signature $S$ by $in(S)$, $out(S)$, and $int(S)$, respectively. Furthermore, we define the external actions, $ext(S)$, to be $in(S) \cup out(S)$; $acts(S)$ to be all the actions of $S$.

**Definition 2.6.** An input/output automaton $A$ consists of four components,

1. a set $\text{states}(A)$ of states,

2. a non-empty set $\text{start}(A) \subseteq \text{states}(A)$ of start states,

3. an action signature $\text{sig}(A)$, and

4. a transition relation $\text{trans}(A) \subseteq \text{states}(A) \times acts(\text{sig}(A)) \times \text{states}(A)$. 

16
We use acts(A) as shorthand for acts(sig(A)), and similarly for in(A), and so on. We call an element (s, π, s') of trans(A) a transition of A. The transition (s, π, s') is called an input transition, output transition or internal transition, based on whether the action π is an input action, output action or internal action. When (s, π, s') ∈ trans(A), we write $s \stackrel{\pi}{\rightarrow}_A s'$, or $s \rightarrow s'$ when no confusion is possible. If $s \rightarrow s'$ we refer to s as the pre-state of the transition, and s' as the post-state.

Now we present some definitions related to I/O automata, mainly adapted from Doherty [5].

**Definition 2.7.** For any I/O automaton A:

1. An execution fragment of A is a finite sequence $\alpha = \langle s_0, \pi_1, s_1, \pi_2, s_2, \cdots, \pi_k, s_k \rangle$ of alternating states and actions of A such that $(s_i, \pi_{i+1}, s_{i+1})$ is a transition of A for all $0 \leq i < k$. If such a finite execution fragment exists, we write $s_0 \xrightarrow{\alpha}_A s_k$, or $s_0 \xrightarrow{\alpha} s_k$ when no confusion is possible.

2. An execution of A is a finite execution fragment whose first state $s_0$ is in start(A). We denote the set of executions of A by execs(A).

3. For an execution $\alpha$ of A, trace($\alpha$) is the sequence $\alpha$ restricted to external actions of the automaton A.

4. The set traces(A) is the set of traces of executions of A.
2.3.1 Concurrent Implementations and Linearizability

Intuitively, a concurrent implementation of a data type specifies a program that can be executed by each process $p \in PROC$ to perform an operation. This program will also specify the result of the operation to be returned to $p$. More formally, a concurrent implementation will be described as an I/O automaton, whose external actions are invocations and responses of the data type. Since the program that implements an operation may take many steps, those steps are all modelled as internal actions. When describing a concurrent implementation using an I/O automaton, we also put processes as subscripts of the actions in each transition to identify which process takes a step.

Thus, we can formally define concurrent executions and concurrent traces.

**Definition 2.8.** A concurrent execution of a concurrent implementation manipulated by a set of processes $PROC$ is an execution of its I/O automaton, in which each action has a process as its subscript.

**Definition 2.9.** A concurrent trace of a concurrent implementation manipulated by a set of processes $PROC$ is a trace of its I/O automaton.

Because the steps of processes are interleaved by the scheduler, an invocation of an operation by one process and its matching response are not guaranteed to be next to each other, in the trace of a concurrent implementation. Between an invocation
and the matching response made by one process, there may be some invocations and responses for other processes. For example, if the set of processes is \( \{p, q, r\} \), a concurrent trace of a data type \( \mathcal{D} \) may look like:

\[
T = \langle \text{inv}_p, \text{inv}_r, \text{res}_r, \text{inv}_q, \text{res}_p, \text{res}_q \rangle.
\]

![Diagram of concurrent processes](image)

Figure 2.1: The trace from a finite execution sequence on *fetch*\&*inc* object by three processes: \( p, q, \) and \( r \).

Figure 2.1 shows one possible trace \( T_1 \) of a concurrent implementation of a *fetch*\&*inc* data type executed by processes \( p, q \) and \( r \). The trace of a concurrent data type, like \( T_1 \), is complicated because of the interleaving of operations by different processes. Therefore, we need some properties to identify if a trace is “good” or not. Linearizability [24] is such a property which guarantees that each concurrent trace is equivalent to some legal atomic trace that satisfies its sequential specification. Before formally specifying linearizability, we need to introduce some definitions,
adapted from [24].

**Definition 2.10.** Given a trace $T$ and a process $p$, a process subtrace, $T|p$ ($T$ at $p$), is the subsequence of all invocations and responses in $T$ whose process names are $p$.

**Definition 2.11.** Two traces $T$ and $T'$ are equivalent if, for every process $p$, $T|p = T'|p$.

**Definition 2.12.** A trace $T$ of a data type $D$ is well-formed if, for each process $p$, its subtrace $T|p$ starts with an invocation, and alternates between invocations and responses.

We also define a partial order relation on operations of different processes. Recall that an operation $e$ consists of the invocation $\text{inv}(e)$ together with the matching response $\text{res}(e)$ (if it exists).

**Definition 2.13.** The irreflexive partial order $<_T$ on the operations in trace $T$, is defined by: $e_i <_T e_j$ if and only if $\text{res}(e_i)$ precedes $\text{inv}(e_j)$ in $T$.

Informally, the irreflexive partial order $<_T$ shows a “sequential” relation among some operations. Pairs of operations that are not ordered by $<_T$ are regarded as “concurrent” operations. Straightforwardly, if a trace is atomic, $<_T$ becomes a total order. Based on the preceding definitions, we define a trace $T$ to be linearizable as follows.
Definition 2.14. For a trace $T'$, let $\text{complete}(T')$ be the maximal subsequence of $T'$ consisting of all responses and their matching invocations. A trace $T$ is linearizable with respect to a data type $\mathcal{D}$, if it can be extended (by appending zero or more response events) to some trace $T'$ such that:

L1: $\text{complete}(T')$ is equivalent to some legal atomic trace $S$, and

L2: $<\text{complete}(T') \subseteq <S$.

A pending operation in an execution is an operation without a matching response. Intuitively, for each pending operation that takes effect but does not return a response, we add the response to obtain $T'$. For example, in Figure 2.1, the second operation made by process $p$ is pending, but it must take effect in trace $T_1$, since that is the only way that process $q$ can return $\text{fiResp}(3)$. On the other hand, $\text{complete}(T')$ excludes those pending operations that have not yet taken effect. As described by Herlihy [24], “L1 in Definition 2.14 states that processes act as if they were interleaved at the granularity of complete operations. L2 states that this sequential interleaving corresponds to the precedence ordering of operations.” $S$ is called a linearization of $T$. Note that L1 also implies that subtraces of each process in $T$ are well-formed.

To show linearizability of a trace, we can identify a time within each operation when the operation can be considered to take effect, namely the linearization point, and show that ordering the operations in the concurrent execution by their
linearization points gives an equivalent atomic trace. Each operation that has no response may or may not be assigned a linearization point. For example, we can assign linearization points to $T_1$ as shown in Figure 2.2 so that its corresponding legal atomic trace is $T_0$. Another example of assigning linearization points to a trace is illustrated in Figure 2.3(a). Since there is no way to assign linearization points to the trace shown in Figure 2.3(b), it is not linearizable. To see why, in any atomic trace that preserves the order $<_{\text{complete}(T)}$, there is a partial order relation between the first operations performed by process $q$ and $r$ (because $f_{i\text{Resp}_q}(2)$ precedes $f_{i_r}$ in $T_3$). However, their responses violate the specification of the $\text{fetch}&\text{inc}$ data type.

![Trace T1](image)

**Figure 2.2**: Trace $T_1$ of a $\text{fetch}&\text{inc}$ object with linearization points (shown as “stars”). Here, $p$’s second operation took effect but the operation was pending.

**Definition 2.15.** A *linearizable implementation* of a data type $\mathcal{D}$ is one whose concurrent traces are linearizable.
In other words, a concurrent data structure is linearizable if, for each concurrent trace of the data type, there is an atomic trace in which every operation returns the same result, and non-concurrent operations occur in the same order in the atomic trace as in the concurrent trace.

**Definition 2.16.** The trace inclusion relation $\subseteq_T$ is defined as follows: for any I/O automata $A$ and $B$, $A \subseteq_T B$ iff $\text{traces}(A) \subseteq \text{traces}(B)$.

An I/O automaton can be viewed as a “black box” from the point of view of a
user. What the user sees is just the trace of the automaton’s execution. If, for any automata $A$ and $B$, $A \subseteq_T B$, then any (external) behaviour exhibited by $A$ could also be exhibited by $B$. Trace inclusion allows us to identify if one I/O automaton specifies the desired external behaviour of another automaton. Section 2.4 describes a formal verification technique based on the trace inclusion property between a specification automaton and an implementation automaton.

### 2.3.2 Canonical Automata

Doherty [5] described a *canonical automaton* which is able to capture all linearizable traces of the data type $\mathcal{D}$. Doherty proved the traces of *any* automaton that is a linearizable implementation of $\mathcal{D}$ will be included in the traces of the canonical automaton [5].

Recall that an automaton consists of four key parts: *states, start states, actions* and a *transition relation*. In the canonical automaton $CA$ for a data type $\mathcal{D} = (V, v_0, I, R, f)$ and a set of processes $PROC$, the state consists of a value from $V$ and a value of the program counter for each process. Intuitively, a state records the current value of an instance of $\mathcal{D}$, and the value of the program counter for process $p$ indicates the next action that $p$ is allowed to perform. $Start(CA)$ contains a single state where the value is the initial value $v_0$ of $\mathcal{D}$ and program counter of each process $p$ is set to be *idle*, indicating that $p$ is ready to perform an invocation. As defined
in [5], the canonical automaton’s external input actions are invocations of the data
type coupled with processes. Similarly, each external output action is one of the
data type’s responses together with a process. Thus, we will have the following
external signature for CA.

- \( \text{in}(CA) = \{ \text{inv}_p \mid \text{inv} \in I, p \in PROC \} \)
- \( \text{out}(CA) = \{ \text{resp}_p \mid \text{resp} \in R, p \in PROC \} \)

We call \( \text{in}(CA) \) its invocations, and \( \text{out}(CA) \) its responses. Internal actions in
\( \text{int}(CA) \) which represent the linearization points, apply the update function \( f \) of
\( D \)’s specification to the value. The transition relation \( \text{trans}(CA) \) is constructed
straightforwardly based on states and actions.

Figure 2.4 shows how we construct \( \text{states}(CA), \text{start}(CA), \text{in}(CA), \text{out}(CA) \) and
\( \text{int}(CA) \), given a set of processes \( PROC \). To guarantee the well-formed property of
the traces of \( CA \), there are three types of values for the program counter: \( \text{pc}._{\text{idle}} \),
\( \text{pc}._{\text{inv}} \) and \( \text{pc}._{\text{resp}} \). Intuitively, \( \text{pc}._{\text{idle}} \) indicates that only invocations are allowed
to be performed, and \( \text{pc}._{\text{inv}} \) indicates that a process has performed the invocation
\( \text{inv} \) and is able to execute an internal action. Similarly, \( \text{pc}._{\text{resp}} \) indicates that the
response \( \text{resp} \) is allowed to be returned to a process.

Because a state of an automaton is usually represented using a Cartesian prod-
tect, we introduce “state variables” [3] to describe each component of it. For example,
we use a state variable “\( \text{pc}_p \)” for each \( p \in PROC \) and a state variable “\( \text{val} \)”, where
\( \text{pc}_p = \pi_p(\pi_2(s)) \) and \( \text{val} = \pi_1(s) \), for a state \( s \in \text{states}(CA) \). Thus, we have a more
\[ \text{states}(CA) = V \times \prod_{p \in \text{PROC}} \text{Pcval}, \text{ where } \text{Pcval} = \{ \text{pc} \text{-idle} \} \cup \{ \text{pc} \text{-inv} : \text{inv} \in I \} \cup \{ \text{pc} \text{-resp} : \text{resp} \in R \} \]

\[ \text{start}(CA) = \{ v_0 \} \times \prod_{p \in \text{PROC}} \{ \text{pc} \text{-idle} \}, \text{ where } v_0 \text{ is the initial value of } D \]

\[ \text{in}(CA) = \{ \text{inv}_p \mid \text{inv} \in I, p \in \text{PROC} \} \]

\[ \text{out}(CA) = \{ \text{resp}_p \mid \text{resp} \in R, p \in \text{PROC} \} \]

\[ \text{int}(CA) = \{ \text{do} \text{-inv}_p \mid \text{inv} \in I, p \in \text{PROC} \} \]

Figure 2.4: The states, start and actions of the canonical automaton CA with respect to a data type D and a set of processes PROC.

A convenient way to describe \( \text{trans}(CA) \).

Based on the construction of states(CA), start(CA) and acts(CA), the transition relation \( \text{trans}(CA) \subseteq \text{states}(CA) \times \text{acts}(CA) \times \text{states}(CA) \) of the canonical automaton is obvious. We describe \( \text{trans}(CA) \) as the set of all triples \( (s, \pi, s') \) such that the state \( s \) satisfies the some preconditions before executing the action \( \pi \) and the state \( s' \) is obtained from \( s \) by updating \( s \) according to \( \pi \). When a process’s program counter is \( \text{pc} \text{-idle} \), it is able to perform an invocation. After the invocation, it may perform an internal action and then a response action may be invoked eventually. Whenever an action is performed by \( p \), its program counter is updated to ensure the well-formed property. Additionally, when an internal action is executed, the value of D in the state of CA is also updated according to the update function \( f \) of D.

Table 2.1 illustrates how we use Pre and Eff to capture the pre-conditions before
executing an action and the *effects* of the action on the state, respectively. Note that the effects of each action form a set of parallel assignments. State variables that are not mentioned in $\textbf{Eff}$ remain the same.

<table>
<thead>
<tr>
<th>Action</th>
<th>Pre</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{inv}_p$</td>
<td>$pc_p = pc_{idle}$</td>
<td>$pc_p \leftarrow pc_{\text{inv}}$</td>
</tr>
</tbody>
</table>
| $\text{do}_{\text{inv}}_p$ | $pc_p = pc_{\text{inv}}$ | val $\leftarrow v'$, $pc_p \leftarrow pc_{\text{resp}}$,  
val $= v$ | where $(\text{resp},v') = f(v, \text{inv})$ |
| $\text{resp}_p$ | $pc_p = pc_{\text{resp}}$ | $pc_p \leftarrow pc_{\text{idle}}$ |

Table 2.1: Transitions of a Canonical Automaton.

In Table 2.1, $\text{inv}_p$, $\text{do}_{\text{inv}}_p$ and $\text{resp}_p$ all represent actions, where $\text{inv} \in I$, $\text{resp} \in R$ and $p \in \text{PROC}$. The construction of our canonical automaton follows Doherty [5]. A slightly different construction is given by [23, Section 13.2]. To show a concrete example of constructing a canonical automaton, recall the $\text{fetch} \& \text{inc}$ data type described in Definition 2.4. The canonical automaton $CF$ for the $\text{fetch} \& \text{inc}$ data type is shown in Figure 2.5 and Table 2.2.

**Linearizability of the Canonical Automaton**

This section proves that all traces of the canonical automaton are well-formed and linearizable. The results in this section are fairly straightforward and mainly follow
states(CF) = \mathbb{N} \times \prod_{p \in \text{PROC}} P_{\text{cval}}, \text{ where } P_{\text{cval}} = \{\text{pc\_idle}, \text{pc\_fi}\} \cup \\
\{\text{pc\_fiResp}(n) \mid n \in \mathbb{N}\}

start(CF) = \{0\} \times \prod_{p \in \text{PROC}} \{\text{pc\_idle}\}

in(CF) = \{fi_p \mid p \in \text{PROC}\}

out(CF) = \{fiResp(n) \mid n \in \mathbb{N}, p \in \text{PROC}\}

int(CF) = \{do\_fi_p \mid p \in \text{PROC}\}

Figure 2.5: states, start and actions of the canonical automaton CF for the fetch\&inc data type.

<table>
<thead>
<tr>
<th>Action</th>
<th>Pre</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>fi_p</td>
<td>pc_p = pc_idle</td>
<td>pc_p \leftarrow pc_fi</td>
</tr>
<tr>
<td>doFi_p</td>
<td>pc_p = pc_fi</td>
<td>pc_p \leftarrow pc_fiResp(v)</td>
</tr>
<tr>
<td></td>
<td>val = v</td>
<td>val \leftarrow v + 1</td>
</tr>
<tr>
<td>fiResp_p(n)</td>
<td>pc_p = pc_fiResp(n)</td>
<td>pc_p \leftarrow pc_idle</td>
</tr>
</tbody>
</table>

Table 2.2: Transitions for the canonical automaton CF of the fetch\&inc data type.

the proofs in [23] and [5]. We assume we have a data type \( D = (V, v_0, I, R, f) \) and the canonical automaton \( CA \) as described in the preceding construction.

Recall that a linearizable I/O automaton \( A \) should satisfy three properties: its external actions must match the invocations and responses of \( D \), its traces must be well-formed, and its traces must be linearizable. According to the construction, it is
trivial to show that \textit{in}(CA) and \textit{out}(CA) correspond to \textit{invocations} and \textit{responses} of \( D \).

\textbf{Lemma 2.17.} All the traces of CA are well-formed.

\textit{Proof.} Consider an execution of CA and a process \( p \).

- For the state \( s \in \text{start}(A) \), \( pc_p = pc_{idle} \). Because only \( p \)'s actions modify \( pc_p \), the first \( p \)-indexed action has to be an invocation according to the transitions in Table 2.1.

- Immediately after any \( p \)-indexed invocation \( \text{inv}_p \), \( pc_p = pc_{\text{inv}} \). According to the Table 2.1, the only admissible action for \( p \) is a \( \text{do}_{\text{inv}} p \), if it exists.

- The next \( p \)-indexed action after each \( \text{do}_{\text{inv}} p \) is a response, if it exists. The \( p \)-indexed response action also sets \( pc_p \) to \( pc_{\text{idle}} \).

- The next \( p \)-indexed action after each response \( \text{resp} \) is an invocation, if it exists, because \( pc_p = pc_{\text{idle}} \).

We next show that all traces of CA are linearizable.

\textbf{Lemma 2.18.} All traces of CA are linearizable.
Proof. Let $\alpha = \langle s_0, \pi_1, s_1, \pi_2, s_2, \ldots, s_{n-1}, \pi_n, s_n \rangle$ be an execution of CA and $T$ be the trace of $\alpha$. A complete operation in $\alpha$ consists of its invocation $\text{inv}_{\text{op}}$, the internal action $\text{do}_{\text{inv}_{\text{op}}}$, and the matching response $\text{resp}_{\text{op}}$.

Let $T'$ be an extension of $T$ obtained by appending the response for every pending operation in $T$ that has a $\text{do}_{\text{inv}}$ in $\alpha$. Let $\text{op}_1, \text{op}_2, \text{op}_3, \ldots, \text{op}_k$ ($k \leq n$) be the operations whose $\text{do}_{\text{inv}}$ actions are in $\alpha$, in the order of those $\text{do}$ actions. Let $S$ be the atomic trace $\langle \text{inv}_{\text{op}_1}, \text{resp}_{\text{op}_1}, \text{inv}_{\text{op}_2}, \text{resp}_{\text{op}_2}, \ldots, \text{inv}_{\text{op}_k}, \text{resp}_{\text{op}_k} \rangle$.

According to Definition 2.14, we must show that $S$ is a legal trace with respect to the data type $\mathcal{D} = (V, v_0, I, V, f)$; $\text{complete}(T') \subseteq S$; and $\text{complete}(T')$ is equivalent to $S$.

(a). Let the value of the state just prior to the $\text{do}_{\text{inv}}$ action of $\text{op}_i$ in $\alpha$ be $v_{i-1}$.

Since only $\text{do}_{\text{inv}}$ actions modify the value component of a state of CA, $v_i$ remains the same among all states between states after $\text{do}_{\text{inv}}_{i-1}$ and before $\text{do}_{\text{inv}}_i$. Moreover, $v_0$ is the same as the initial value in state $s_0$. Thus, by the definition of $\text{do}_{\text{inv}}$ action of $\text{op}_i$, $f(v_{i-1}, \text{inv}_{\text{op}_i}) = (\text{resp}_{\text{op}_i}, v_i)$, where in the state after $\text{do}_{\text{inv}_{\text{op}_i}}$, its $\text{pc}_p = \text{pc}_{\text{resp}}$. Thus, $\text{resp}_{\text{op}_i} = \text{resp}$. So, $\forall i : f(v_{i-1}, \text{inv}_{\text{op}_i}) = (\text{resp}_{\text{op}_i}, v_i)$ and $S$ is a trace of a legal execution.

(b). For all $i, j$, if $\text{resp}_{\text{op}_i}$ precedes $\text{inv}_{\text{op}_j}$ in $\text{complete}(T')$, $\text{op}_i$ precedes $\text{op}_j$ in $S$.

Because we have shown in the proof of Lemma 2.17 that the $\text{do}$ action of an operation is in between its invocation and matching response, $\text{do}_{\text{inv}}_i$ precedes
do\_inv_j in \alpha. Therefore, by the construction of \( S \), \texttt{resp\_op}_i precedes \texttt{inv\_op}_j in \( S \). So we have \(<_{\text{complete}(T')}\subseteq <_{S}\).

(c). We argue that \( S|p \) is the same as \texttt{complete}(T')|p for every process \( p \). As proved in part (b), we can easily obtain \(<_{\text{complete}(T')}|p \subseteq <_{S}|p \), where both \(<_{\text{complete}(T')}|p \) and \(<_{S}|p \) are total orders. Combined with the fact that for every operation, its invocation precedes the matching response, the order of invocations and responses in \texttt{complete}(T')|p and \( S|p \) must be the same. It remains to show that every operation in \texttt{complete}(T') if and only if it is in \( S \). This is true because any operation, either a complete operation or a pending one, which contains the \texttt{do} action is in \texttt{complete}(T'), and also is in \( S \) by the construction. Pending operations which do not have \texttt{do} actions are in neither \texttt{complete}(T') nor \( S \).

\[
\blacksquare
\]

Lynch [23] shows a proof of Lemma 2.18 for a slightly different canonical automaton, but the idea of that proof is similar to the one given here. The intuition in both proofs is that we can always linearize the traces according to the \texttt{do\_inv} actions of the canonical automaton. Since \texttt{do\_inv} actions directly follow the update function of the data type, this order of execution sequence forms a legal sequential execution.
2.4 Simulations

The *simulation* method [12] is an approach for proving one concurrent system $A$ implements $B$ by showing a trace inclusion relationship between them. In a simulation proof, each system is modelled as an $I/O$ automaton and we show that each transition in $A$ has a corresponding execution in $B$, such that their traces are the same. This technique has been frequently used for formal verification of linearizability of concurrent implementations. An implementation is linearizable if the traces of the automaton that models the implementation of data type $D$, are subsumed by the traces of the canonical automaton of $D$. To show the inclusion relationship, we mainly consider two types of simulations: forward and backward.

2.4.1 Forward Simulations

A *forward simulation* [12] from automaton $A$ to automaton $B$ is a relation $fsr$ from states of $A$ to states of $B$ such that every initial state of $A$ is related to an initial state of $B$, and every action of $A$ yields a corresponding sequence of actions of $B$.

**Definition 2.19.** A *forward simulation* from the I/O automaton $A$ to I/O automaton $B$ is a relation $fsr \subseteq states(A) \times states(B)$ that satisfies the following properties:

1. For every $s \in start(A)$, there exists a $u \in start(B)$, such that $(s, u) \in fsr$. 

32
2. If \( s \xrightarrow{\alpha} A s' \) and \((s,u) \in \text{fsr}\), then there exists \( u \xrightarrow{\hat{\alpha}} B u' \) for some \( u' \) such that \((s',u') \in \text{fsr}\), and

3. the external action in \( \hat{\alpha} \) is the same as the external action in \( \alpha \) (i.e., either equals \( \alpha \), if \( \alpha \) is an external action, or is empty otherwise).

Recall that \( s \xrightarrow{\alpha} A s' \) denotes that by performing the action \( \alpha \), state \( s \) becomes the post state \( s' \) in \( A \). The notation \( u \xrightarrow{\hat{\alpha}} B u' \) means that in \( B \), the automaton moves from state \( u \) to \( u' \) by performing a sequence of actions \( \hat{\alpha} \). If the relation \( \text{fsr} \) over \( \text{states}(A) \) and \( \text{states}(B) \) in Definition 2.19 is a function, we call it a refinement \([12]\). A refinement is a simplified forward simulation that is often used in formally verifying the correctness of concurrent implementations \([3,7,9,10,13,16]\).

**Theorem 2.20.** If \( \text{fsr} \) is a forward simulation from \( A \) to \( B \), then \( \text{traces}(A) \subseteq \text{traces}(B) \) \([12]\).

**Proof.** Let \( E_A = \langle s_0, \pi_1, s_1, \pi_2, \ldots, s_{n-1}, \pi_n, s_n \rangle \) be an execution of \( A \) and \( T_A \) be the trace of \( E_A \). We argue that there exists an execution \( E_B \) of \( B \) that has the same trace as \( T_A \). Let \( c_0 \) be an initial state of \( B \), such that \((s_0,c_0) \in \text{fsr}\) (such a \( c_0 \) exists according to Definition 2.19). We do induction on the length of \( E_A \). If we know \( s_0 \xrightarrow{(\pi_1,\ldots,\pi_i)} A s_i \) and \( c_0 \xrightarrow{(\hat{\pi}_1,\ldots,\hat{\pi}_i)} B c_i \) such that \( c_0 \in \text{fsr}(s_0) \), \( c_i \in \text{fsr}(s_i) \) and \( \text{trace}(\langle \pi_1, \ldots, \pi_i \rangle) = \text{trace}(\langle \hat{\pi}_1, \ldots, \hat{\pi}_i \rangle) \), we can construct an execution sequence \( c_i \xrightarrow{\hat{\pi}_{i+1}} B c_{i+1} \) of \( B \) for \( s_i \xrightarrow{\pi_{i+1}} A s_{i+1} \) of \( A \), where \( \text{trace}(\pi_{i+1}) = \ldots \)
\(\text{trace}(\hat{\pi}_{i+1})\) and \(c_{i+1} \in fsr(s_{i+1})\). Then, \(c_0 \xrightarrow{\langle \hat{\pi}_1, \cdots, \hat{\pi}_{i+1} \rangle} B\ c_{i+1},\ c_{i+1} \in fsr(s_{i+1})\) and \(\text{trace}(\langle \hat{\pi}_1, \cdots, \hat{\pi}_{i+1} \rangle) = \text{trace}(\langle \pi_1, \cdots, \pi_{i+1} \rangle)\). The base case is trivial when \(i = 0\).

Finally, we have \(\text{trace}(\langle \hat{\pi}_1, \cdots, \hat{\pi}_n \rangle) = \text{trace}(\langle \pi_1, \cdots, \pi_n \rangle) = \text{trace}(E_A)\).

\[\square\]

An Example of a Forward Simulation

We shall show an example of how a forward simulation can be used to prove the correctness of an implementation. A simple implementation of the \(\text{fetch} \& \text{inc}\) using a \text{CAS} object is illustrated in Figure 2.6. Recall the specification \(D = (V, v_0, I, R, f)\) of a \(\text{fetch} \& \text{inc}\) data type in Definition 2.4 and the canonical automaton \(CF\) in Figure 2.5 and Table 2.2. It is fairly easy to see that this implementation is linearizable: the linearization point of each \(\text{fetch} \& \text{inc}\) operation is when it performs its successful \text{CAS}. We shall formalize this argument using a forward simulation.

As described in Section 2.1, a \text{CAS}(X,u,v)\) operation always returns the old value stored in \(X\). It successfully changes the value stored in \(X\) to \(v\) if and only if the old value of \(X\) is equal to \(u\). Otherwise, it does not change the value. In order to show the relation between the implementation and its specification, we first formalize the algorithm in Figure 2.6 as a concrete automaton \(C\). Let \(PROC\) be a set of process and let \(Pcval = \{\text{pc\_idle}, \text{pc\_Line1}, \text{pc\_Line2}\} \cup \{\text{pc\_fResp}(n) \mid n \in \mathbb{N}\}\). Consider the concrete automaton for the \(\text{fetch} \& \text{inc}\) implementation shown in Figure 2.7.

The state of the concrete automaton consists of a shared variable \(v\), a local vari-
Figure 2.6: An algorithm that uses CAS to implement the *fetch-and-increment* data type.

```
Fetch&Inc()

    while TRUE {
1       res ← v
2       if CAS(v, res, res + 1) = res
        return res
    }
```

Figure 2.7: The concrete automaton that models the implementation of the *fetch-and-increment* data type.

\[
\begin{align*}
states(C) &= \{v \mid v \in \mathbb{N}\} \times \prod_{p \in \text{PROC}} \{\text{res} \mid \text{res} \in \mathbb{N}\} \times \prod_{p \in \text{PROC}} \text{Pcval} \\
start(C) &= \{s \in states(C) \mid s.v = 0 \land \forall p \in \text{PROC} : s.\text{res}_p = 0 \land s.\text{pc}_p = \text{idle}\} \\
in(C) &= \{fi_p \mid p \in \text{PROC}\} \\
out(C) &= \{fiResp_p(n) \mid n \in \mathbb{N}, p \in \text{PROC}\} \\
int(C) &= \{\text{line1}_p, \text{line2T}_p, \text{line2F}_p\}
\end{align*}
\]

able res\(_p\) for each \(p\) and a program counter pc\(_p\) for each \(p\). Each action corresponds to executing a line of the code. For simplicity, res\(_p\) is initialized to 0 for all \(p\). We use multiple actions to model those lines of code which may subsequently execute
different lines depending on their pre-condition, such as if, until and while operations. For example, in this algorithm, Line 2 in Figure 2.6 is modelled as two internal actions: line2T_p and line2F_p. One indicates the if condition on line 2 evaluates to true and the other indicates it evaluates to false.

<table>
<thead>
<tr>
<th>Action</th>
<th>Pre</th>
<th>Eff</th>
</tr>
</thead>
<tbody>
<tr>
<td>fi_p</td>
<td>pc_p = pc_idle</td>
<td>pc_p ← pcLine1</td>
</tr>
<tr>
<td>line1_p</td>
<td>pc_p = pcLine1</td>
<td>pc_p ← pcLine2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>res_p ← v</td>
</tr>
<tr>
<td>line2F_p</td>
<td>pc_p = pcLine2</td>
<td>pc_p ← pcLine1</td>
</tr>
<tr>
<td></td>
<td>v ≠ res_p</td>
<td></td>
</tr>
<tr>
<td>line2T_p</td>
<td>pc_p = pcLine2</td>
<td>pc_p ← pc_fiResp(res_p)</td>
</tr>
<tr>
<td></td>
<td>v = res_p</td>
<td>v ← res_p + 1</td>
</tr>
<tr>
<td>fiResp_p(n)</td>
<td>pc_p = pc_fiResp(n)</td>
<td>pc_p ← pc_idle</td>
</tr>
</tbody>
</table>

Table 2.3: Transitions of the concrete automaton C of the fetch-and-increment algorithm.

The preconditions and effects of each action are shown in Table 2.3. The internal action line1_p formalizes the “read” action in Figure 2.6, and line2F_p captures a failed CAS (where the value it wants to change is not equal to its expected value). Action line2T_p corresponds to a successful CAS, where the value is increased by one. After
defining the concrete automaton of the algorithm, we need to show that there is a forward simulation from the concrete automaton $C$ to canonical automaton $CF$, to show that $C$ implements $CF$. To show that a forward simulation exists, we establish a relation $fsr$ over $states(C)$ and $states(CF)$ and an action correspondence between $acts(C)$ and $acts(CF)$. First of all, because the program counters are one of the most important components of the states for both automata, we present how they are changed and their relation in $C$ and $CF$ in Figure 2.8.

Figure 2.8: The state diagrams of process $p$’s program counter value for both concrete and canonical automata and the relation between them.

Intuitively, the connection of program counters between $C$ and $CF$ relates to the action correspondence of the two automata. A successful CAS action in $C$ corresponds to a $doFi_p$, which increases the value by one in $CF$. Any other internal
actions in $C$ corresponds to a null sequence of actions in $CF$. Additionally, all
external actions are performed in the same way in $C$ and $CF$ in order to have the
same traces in the forward simulation. This gives us the forward simulation relation
in terms of the values of the program counter of process $p$, illustrated by the red
dotted lines in Figure 2.8. Together with the requirement that the value of the
fetch&inc data type recorded in both automata is the same, we have the forward
simulation relation $fsr(s, s')$, where $s \in states(C)$ and $s' \in states(CF)$, defined as
follows:

$$fsr(s, s') = (s.v = s'.val) \land$$

$$(\forall p \in PROC : (s.pc_p = s'.pc_p)) \lor$$

$$((s'.pc_p = pc_{fi} \land (s.pc_p = pcLine1 \lor s.pc_p = pcLine2))).$$

Basically, (2.1) requires the value of the abstract data type to match the value
stored in CAS object $v$. (2.2) and (2.3) require that the values of the program
counters are also the same, except that the two values, $pcLine1$ and $pcLine2$, in $C$
both correspond to a single value $pc_{fi}$ in $CF$. Because local variables are invisible
from $CF$, they may be arbitrary. Intuitively, this mapping captures the fact that
internal actions: $line1$ and $line2F$ correspond to null actions in $CF$.

Lemma 2.21. There exists a forward simulation from $C$ to $CF$ using the relation
We proved this lemma using the PVS theorem prover.

Proof. We prove that (2.1)-(2.3) define a forward simulation, according to the Definition 2.19. Firstly, if \( s \in \text{start}(C) \) then there exists a \( u \in \text{start}(CF) \) such that \( u.v = s.val \) and \( \forall p : u.pc_p = s.pc_p = pc_{idle} \). Thus \((s, u) \in fsr\).

Secondly, if \( s \xrightarrow{\alpha} C s' \) and \((s, u) \in fsr\), we show there exists a state \( u' \) and a sequence of actions \( \hat{\alpha} \) such that \( u \xrightarrow{\hat{\alpha}}_{CF} u' \) and \((s', u') \in fsr\), and the external actions in \( \hat{\alpha} \) are the same as the external actions in \( \alpha \). Because there are five types of action: \( fi_p, line1_p, line2_{Tp}, line2_{Fp} \) and \( fi_{Resp_p}(n) \) in \( C \), we prove this property by distinguishing the following cases.

1. If \( \alpha = fi_p \), let \( u' \in \text{states}(CF) \), such that \( u'.v = u.v \) and \( \forall q \neq p : u'.pc_q = u.pc_q, u'.pc_p = pc_{fi} \). We then have \( u \xrightarrow{fi_p}_{CF} u' \) in \( CF \) according to Table 2.3.

Because \( s'.pc_p = pc_{fi} \) and \((s, u) \in fsr\), we know \((s', u') \in fsr\).

2. If \( \alpha = fi_{Resp_p}(n) \), let \( u' \in \text{states}(CF) \), such that \( u'.v = u.v \) and \( \forall q \neq p : u'.pc_q = u.pc_q, u'.pc_p = pc_{idle} \). We then have \( u \xrightarrow{fi_{Resp_p}(n)}_{CF} u' \) according to Table 2.3.

Because \( s'.pc_p = pc_{idle} \) and \((s, u) \in fsr\), we know \((s', u') \in fsr\).

3. If \( \alpha = line2_{Tp} \), let \( k = s.v = u.val \) and \( u' \in \text{states}(CF) \) such that \( \forall q \neq p : u'.pc_q = u.pc_q \) and \( u'.pc_p = pc_{fi_{Resp}(k)}, u'.val = k + 1 \). We then have \( u \xrightarrow{doFi_p}_{CF} u' \), since \( u.pc_p = pc_{fi} \) and \( u.val = k \) according to \((s, u) \in fsr\).

Because \( s'.pc_p = pc_{fi_{Resp}(k)}, u'.val = s'.v = k + 1 \) and \((s, u) \in fsr\), we know \((s', u') \in fsr\).
4. If $\alpha = \text{line1}_{p}$ or $\alpha = \text{line2F}_{p}$, let $u' \in \text{states}(CF)$ such that $u' = u$. We have $u \xrightarrow{\text{nil}_{CF}} u'$. We easily obtain $(s', u') \in \text{fsr}$, because for $\text{line1}_{p}$: $s'.pc_{p} = \text{pc.} \text{Line2}$ and $u'.pc_{p} = \text{pc.} \text{fi}$, or for $\text{line2F}_{p}$: $s'.pc_{p} = \text{pc.} \text{Line1}$ and $u'.pc_{p} = \text{pc.} \text{fi}$.

\[ \square \]

2.4.2 Backward Simulations

A backward simulation relation \[12\] $\text{bsr}$ is similar to a forward simulation relation, but the main difference is that in a forward simulation, we reason about execution sequences in a forward direction and in a backward simulation from $A$ to $B$, we start from the end of $A$'s execution and construct the corresponding execution of $B$ backwards.

Definition 2.22. A backward simulation from the I/O automaton $A$ to the I/O automaton $B$ is a total relation\[1] $\text{bsr} \subseteq \text{states}(A) \times \text{states}(B)$ that satisfies:

1. If $s \in \text{start}(A)$ and $u \in \text{states}(B)$, for all $(s, u) \in \text{bsr}$, $u \in \text{start}(B)$, and

2. if $s' \xrightarrow{\alpha} A s$ and $u \in \text{states}(B)$ such that $(s, u) \in \text{bsr}$, then there exists a state $(s', u') \in \text{bsr}$ such that $u' \xrightarrow{\hat{\alpha}} B u$, and

\[ ^{1}\text{A relation } R \text{ over } \text{states}(A) \text{ and } \text{states}(B) \text{ is total if, for every } a \in \text{states}(A), \text{ there exists } b \in \text{states}(B) \text{ such that } (a, b) \in R \text{ is true.} \]
3. the external action in $\hat{\alpha}$ is the same as the external action in $\alpha$ (i.e., either equals $\alpha$, if $\alpha$ is an external action, or is empty otherwise).

**Theorem 2.23.** If $bsr$ is a backward simulation from $A$ to $B$, then $traces(A) \subseteq traces(B)$.

**Proof sketch.** The idea of the proof here is similar to the proof of Theorem 2.20. It also can be found in [12]. The intuition is that given an execution $\alpha = \langle s_0, \pi_1, s_1, \cdots, \pi_n, s_n \rangle$ in $A$, we construct a corresponding execution $\hat{\alpha}$ in $B$ starting from the end to the beginning inductively. Because the states $u_{i+1} \in states(B)$ and $s_{i+1} \in states(A)$ are related by $bsr$, there is a $u_i \in states(B)$ such that $(u_i, s_i) \in bsr$ and $u_i \xrightarrow{\hat{\pi}_i} B u_{i+1}$ and $trace(\pi_i) = trace(\hat{\pi}_i)$ according to the definition of backward simulation. For $s_0 \in start(A)$, by property 1 of Definition 2.22, we have a $u_0 \in start(B)$ such that $u_0 \xrightarrow{\langle \hat{\pi}_1, \cdots, \hat{\pi}_n \rangle} B u_n$ and $trace(\langle \pi_1, \cdots, \pi_n \rangle) = trace(\langle \hat{\pi}_1, \cdots, \hat{\pi}_n \rangle)$.

Intuitively, backward simulations are similar to forward simulations, except that in a backward simulation, all states in the image of a state in $start(A)$ are in $start(B)$, whereas, in a forward simulation, some states in the image of $start(A)$ are in $start(B)$. This is because when we construct a related trace backwards, the first state of the trace should be an initial state.

Both forward and backward simulations can be used to show one automaton implements another. People use backward simulations because sometimes it is more intuitive to show a backward simulation relation between two automata [7, 13, 15].
We use a backward simulation in Chapter 4.

Sometimes, to show that an automaton $C$ implements another automaton $A$, we create an intermediate automaton $B$ and show a forward simulation from $C$ to $B$ and a backward simulation from $B$ to $A$. Together, these imply $\text{traces}(C) \subseteq \text{traces}(A)$.

This approach is called a hybrid forward and backward simulation. [12]
3 Non-blocking Binary Search Trees and a Simplified Algorithm

A binary search tree (BST) \[25\] is one of the most fundamental data structures used in the traditional sequential setting. It can be used to support sorting and searching algorithms and also to implement sets, multisets, priority queues and dictionaries.

A node in a BST with or without children is called an \textit{internal} node or a \textit{leaf}, respectively. The node without a parent is the \textit{root}. Each internal node can have two children \textit{left} and \textit{right}. Every node stores a \textit{key}. Node \textit{x} is a \textit{descendant} of node \textit{y}, if \textit{x} is the child of \textit{y} or \textit{x} is a descendant of \textit{y}’s child. Intuitively, if there is a path of child pointers from \textit{y} to \textit{x}, \textit{x} is a descendant of \textit{y}. Figure \[3.1\] illustrates an instance of a BST, whose nodes store integer keys. The node with key 10 is the \textit{root}. The node containing key 4 is one of the descendants of the node with key 6.

The subtree of a BST rooted at a given node is the tree containing that node and all of its descendants. For example, the nodes with keys 6, 3, 4 and 8 form a subtree of the BST shown in Figure \[3.1\]. A BST must also have an important property in
terms of its key values: for every internal node $x$, all keys in the left subtree of that node are less than $x.key$ and all keys in the right subtree are greater than or equal to $x.key$.

A BST can be used to implement a set data type, which stores a set of keys and provides find, insert and delete as basic operations.

**Definition 3.1.** A set data type $\text{SET}$ has the following sequential specification:

- a state set $S = P(\text{Key})$, where $\text{Key}$ is a totally ordered set of all possible keys,
- an initial value $s_0 = \emptyset$,
- a set $I = \{\text{FindInv}(k), \text{InsertInv}(k), \text{DeleteInv}(k) \mid k \in \text{Key}\}$ of invocations,
• a set $R = \{\text{FindResp}(r), \text{InsertResp}(r), \text{DeleteResp}(r) \mid r \in \text{boolean}\}$ of responses, and

• an update function $f : V \times I \rightarrow R \times V$, such that:

$$
\begin{align*}
    f(s,\text{findInv}(k)) &= (\text{findResp}(true), s), \text{if } k \in s, \\
    f(s,\text{findInv}(k)) &= (\text{findResp}(false), s), \text{if } k \notin s, \\
    f(s,\text{insertInv}(k)) &= (\text{insertResp}(false), s), \text{if } k \in s, \\
    f(s,\text{insertInv}(k)) &= (\text{insertResp}(true), s \cup \{k\}), \text{if } k \notin s, \\
    f(s,\text{deleteInv}(k)) &= (\text{deleteResp}(true), s - \{k\}), \text{if } k \in s, \\
    f(s,\text{deleteInv}(k)) &= (\text{deleteResp}(false), s), \text{if } k \notin s,
\end{align*}
$$

for all $k \in \text{Key}$ and $s \in S$.

Intuitively, find operations return true or false depending on whether the given key value is in the set or not. An insert operation inserts a new key into the set and returns true if the key was not already in it. The operation returns false and the set remains unchanged if the given key is already in the set. (We assume that the set data type does not allow duplicate keys.) A delete operation removes the given key from the set and returns true if the key is in the set. Otherwise, it return false and the set remains unchanged.
3.1 A Non-blocking Binary Search Tree Algorithm

Ellen et al. [3] developed the first efficient non-blocking implementation of a BST for an asynchronous shared-memory system. They provided a detailed proof of correctness, which was written in natural language. The BST algorithm they considered is leaf-oriented, meaning that all keys in the set are stored in leaf nodes and each internal node has exactly two children. Internal nodes only store auxiliary keys that are used to direct the searches towards the leaf containing a particular key.

Definition 3.2. Given a key $k$, the search path for $k$ in a leaf-oriented BST is the sequence of nodes $\langle n_0, n_1, n_2, \cdots, n_m \rangle$, such that $n_0$ is the root, $n_m$ is a leaf, and for $1 \leq i \leq m$, $n_i$ is the left child or right child of $n_{i-1}$ depending on whether $k < n_{i-1}.key$ or $k \geq n_{i-1}.key$, respectively.

3.1.1 Implementation Overview

To support the set data type, the non-blocking BST provides algorithms for three operations: find, insert and delete. All of them use a common sub-routine called search, which starts from root and searches toward a leaf that potentially contains the given key. The find operation returns true if the leaf node where the search terminates contains the given key. Otherwise, it returns false. Examples of a successful and unsuccessful find operation are shown in Figure 3.2. Square boxes and circles represent leaf nodes and internal nodes, respectively. Triangles represent subtrees.
Figure 3.2: An example of find operations in a leaf-oriented BST. (a). The find(2) operation ends with a node containing key 2, and returns true. (b) The find(1) operation ends up with a node containing key 2, and returns false.

Figure 3.3: An example of an insert operation in a leaf-oriented BST.

A typical successful insert and delete operation on a leaf-oriented BST are shown in Figure 3.3 and 3.4. The insert(1) operation locates a leaf node which potentially contains key 1 by using the search subroutine. If it successfully finds such a node, insert(1) returns false since no duplicated keys are allowed. In Figure 3.3, because the leaf does not contain key 1, the search tries to insert key 1 into the BST by replacing the leaf with a subtree containing three nodes. Two leaves containing the
Figure 3.4: An example of a delete operation in a leaf-oriented BST.

key of the replaced leaf and the inserted key are in that subtree and their parent node contains the maximum of the two keys. After such an update, insert(1) returns true.

A delete operation locates a leaf node that potentially contains the given key using the search routine as well. If such a node does not contain the given key, the delete operation returns false. Otherwise, the given key is detected, as in the example shown in Figure 3.4 and the child pointer of the leaf’s grandparent is changed from the leaf’s parent to the leaf’s sibling and true is returned. This ensures that the deleted node is no longer reachable through the child pointers of the BST.

Some coordination between processes is needed to avoid problems when more than one process wants to update the same part of the tree concurrently. Partly inspired by Fomitchev and Ruppert’s linked list implementation [26] and the coop-
erative technique of Barnes [27], the non-blocking BST algorithm uses a flagging system to indicate whether there is a process operating at a node. Intuitively, each internal node can be flagged and flags behave like a kind of lock. There are different types of flags used to represent different operations. When a node is flagged, only some particular steps can be applied to it to continue the operation that placed the flag. Other operations have to help this operation to complete before they can place their own flags. Every node has a field to indicate its current state. Initially, the state of a node is set to CLEAN. Before an insert or delete operation changes the child pointer of a node, the node’s state must be set to IFLAG or DFLAG, respectively. After the child pointer is changed, the state of the node is set to CLEAN again. The state field of a node is flagged using a CAS step which succeeds only if the state of that node is CLEAN and has not changed since the operation read the node’s child pointer. This guarantees that during the whole operation of a process, no other operations modify those flagged nodes.

However, these flag states are not sufficient for a delete operation. Figure 3.5 illustrates a problem when two simultaneous delete operations happen using flags only to “lock” the grandparents. In Figure 3.5(a), delete(5) and delete(1) occur concurrently. They set the states of the internal nodes with keys 6 and 4 to DFLAG at the same time before changing their child pointers. Initially the set contains keys \{1, 2, 5, 7\}. Then, both operations modify the child pointers of their flagged nodes.
Figure 3.5: A problem caused by two delete operations if we only use the DFLAG state. (a). delete(5) and delete(1) are being executed and the nodes whose keys are 6 and 4 are set to DFLAG before changing their child pointers. (b). The BST after delete(5) and delete(1) were completed.

and the resulting subtree is shown in Figure 3.5(b), where the leaves contain keys \{1, 2, 7\}. This is because only leaf nodes containing 1, 2 and 7 are reachable from the root of the BST. However, according to the specification of the set (Definition 3.1), the BST should contain only \{3, 7\} after those two deletes.

To solve this kind of problem, Ellen et al. introduce another MARK state. A delete operation must set the state of the leaf’s parent to MARK before changing the grandparent’s child pointer to remove the parent node from the tree. The state of a node can be set to MARK only if it is CLEAN, and once a node is marked, it remains so forever. Intuitively, the MARK state guarantees that a node cannot be
set to MARK and DFLAG/IFLAG at the same time. Thus, when a `delete` operation removes a marked node from the BST, no operation can subsequently modify the marked node.

Because the flagging system intuitively behaves like locks, it may prevent progress. Figure 3.6 illustrates an example where no more operations can be done on the nodes whose keys are 3 and 4 due to the crash of `delete(1)`. The operation `delete(5)` gets blocked because it attempts to MARK the node whose key is 4. However, that node is not in its CLEAN state. In order to guarantee the progress property of this algorithm, Ellen et al. [3] used helping mechanisms in the `insert` and `delete` operations. Basically, besides setting the `states` of a node, every operation also stores some essential information about itself in that node. Thus, if an operation is blocked by an unfinished operation, it uses this information to try to help complete the unfinished one before restarting its own operation. To ensure that only one helper of an

Figure 3.6: If `delete(1)` dies, it blocks `delete(5)`.
operation performs the required change to the tree, child pointers are also updated using CAS steps.

Figure 3.7 illustrates the big picture of how the state of a node changes during different steps of an insert or delete operation. Its right part, included in the blue box, describes steps of an insert operation. Refer to pseudocode in Figure 3.10 and 3.11. An insert operation tries to set a node’s state from CLEAN to IFLAG by an iflag CAS (Line 31). After that, the insert operation changes its child pointer to a new subtree containing three nodes by an ichild CAS (Line 41) while the state of the node remains IFLAG. Subsequently, the operation changes the node with IFLAG state to a CLEAN node by an iunflag CAS (Line 43).

The rest of Figure 3.7 describes steps of a delete operation. A delete operation first flags a CLEAN grandparent node, changing its state to DFLAG by a dflag CAS (Line 54). Then, such a delete operation may continue or backtrack depending on whether it successfully marks the parent node by a mark CAS (Line 62) or not. If the mark CAS succeeds, the grandparent node’s state remains unchanged and the parent node’s state is changed from CLEAN to MARK. Subsequently, the delete operation changes the child pointer of the grandparent node and then sets it back to a CLEAN node by a dchild CAS (Line 85) and a dunflag CAS (Line 87), respectively. If the mark CAS fails, the delete operation backtracks and changes the grandparent node’s state from DFLAG to CLEAN through a backtrack CAS (Line 80) and restarts the
The non-blocking BST uses objects that support read, write and CAS operations. The key set $U$ is totally ordered. To avoid special cases that would require changing the root, the tree is initialized as shown in Figure 3.8. We assume there are two special values $\infty_1$ and $\infty_2$, such that every value in $U$ is less than $\infty_1$ and $\infty_2$, and $\infty_1 < \infty_2$. Hence, every insert or delete operation only modifies the left subtree.
of root. The types of objects we use to represent the data structure are defined in Figure 3.9. Internal nodes and leaf nodes are distinguished by the truth value of the

| type Node{                     | type Info{                      |
| Key ∪ {∞₁, ∞₂} key           | {CLEAN, DFLAG, IFLAG, MARK} infotype |
| Node left, right             | Node gpn, pn, ln, nIntern,      |
| Info info                    | Info pinfo, dinfo               |
| Bool isinternal              | }                               |

Figure 3.9: Data types defined in the non-blocking BST algorithm.

isinternal field of Node objects. For simplicity, both internal nodes and leaf nodes have left and right fields. However, for the leaf nodes, they all point to a special NIL Node. Every node has an info field, which points to an Info object. There are four types of Info objects, CLEAN, DFLAG, IFLAG and MARK, distinguished by the value of the infotype field. An Info object can also record essential information about an insert or delete operation. This information is stored in its gpn, pn, ln, nIntern, pinfo and dinfo fields when an Info object is created. A CLEAN Info object does not need to store any further information in those fields. An IFLAG Info object, which is created by an insert operation, usually stores the leaf node to be replaced.
in its \( ln \) field, the parent of that leaf node in \( pn \), and the newly created internal node in \( nIntern \). A DFLAG Info object, which is created by a \textit{delete} operation, stores the leaf node to be removed, the parent of the leaf node and the grandparent of the leaf node in \( ln, pn \) and \( gpn \), respectively. It also stores an Info object that was read from the parent in \( pinfo \). (This is used by other processes helping the \textit{delete} as the old value for the mark \textit{CAS}.) A MARK Info object, which is also created by a \textit{delete} operation after the creation of a DFLAG Info object, just has a pointer to the DFLAG Info object created by the deletion.

The detailed implementations of the non-blocking algorithms are shown in Figure 3.10 and Figure 3.11, where comments are preceded by \( \triangleright \). Basically, all three operations call the sub-routine \( \text{Search}(k) \) to traverse nodes until reaching a leaf. The \( \text{Search}(k) \) routine takes a key \( k \) as its input parameter and returns five objects. At Line 2, the search starts from the root. The search goes down to the left or right child depending on whether the key field of the current internal node is less or greater than the given key \( k \). It stops when it hits a leaf node (Line 4). During the while loop, it stores the last three visited nodes as \( gpn, pn \) and \( ln \) (grandparent, parent and leaf node). It also stores the \textit{info} field of \( gpn \) and \( pn \). A \( \text{Find}(k) \) operation calls \( \text{Search}(k) \) and gets the returned leaf node. If the \textit{key} field of the leaf node is equal to \( k \), it returns \textit{true}, otherwise it returns \textit{false}.

\textbf{Definition 3.3.} The sequence of visited nodes by an invocation of \textit{search} is the
Search(Key k) : ⟨Node, Node, Node, Info, Info⟩ {  
▷ Used by Insert, Delete and Find to traverse a branch of the BST; satisfies following postconditions:  
▷ (1) ln points to a Leaf node and pn points to an Internal node  
▷ (2) Either pn.left has contained ln (if k < pn.key) or pn.right has contained ln (if k ≥ pn.key)  
▷ (3) pn.info has contained pinfo  
▷ (4) if ln.key ≠ ∞, then the following three statements hold:  
▷ (4a) gpn points to an Internal node  
▷ (4b) either gpn.left has contained pn (if k < gpn.key) or gpn.right has contained pn (if k ≥ gpn.key)  
▷ (4c) gpn.info has contained gpinfo  
  
Node gpn, pn, ln := Root  
Info gpinfo, pinfo  
▷ Each stores a copy of an info field  
  
while ln points to an internal node {  
  gpn := pn  
  pn := ln  
  gpinfo := pinfo  
  pinfo := pn.info  
  if k < ln.key then ln := ln.left else ln := ln.right  
}  
return ⟨gpn, pn, ln, pinfo, gpinfo⟩  
}

Find(Key k) : boolean {  
Node ln  
⟨−, −, ln, −, −⟩ := Search(k)  
if ln.key = k then return true  
else return false  
}

Insert(Key k) : boolean {  
Node ln, pn, nIntern, nSib  
Node nNode := a new leaf node whose key field is k  
Info pinfo, result, op  
while True {  
  ⟨−, pn, ln, pinfo, −⟩ := Search(k)  
  if ln.key = k then return false  
  if pinfo.infotype ≠ Clean then Help(pinfo)  
  else {  
    nSib := a new leaf whose key is ln.key  
    nIntern := a new internal node with key field max(k, ln.key),  
        info field ⟨CLEAN, ⊥, ⊥⟩, and with two child fields equal to nNode and nSib  
        (the one with the smaller key is the left child)  
    op := a new Info object containing ⟨IFlag, pn, ln, nIntern⟩  
    result := CAS(pn.info, pinfo, op)  
    if result = pinfo then {  
      HelpInsert(op)  
      return True  
    }  
    else Help(result)  
  }  
}

HelpInsert(Info op) {  
▷ Precondition: op is to an IFlag Info object (i.e., it is not ⊥)  
CAS-Child(op.pn, op.ln, op.nIntern)  
▷ ichild CAS  
clean := a new CLEAN Info object  
CAS(op.pn.info, op, clean)  
▷ iunflag CAS
}

Figure 3.10: Pseudocode for Search, Find and Insert [3].
Delete(Key k) : boolean {
    Node gpn, pn, ln; Info pinfo, gpinfo, result, op;

    while TRUE {
        ⟨gpn, pn, ln, pinfo, gpinfo⟩ := Search(k) ▷ Key k is not in the tree
        if ln.key ≠ k then return false ▷ Try to flag gpn
        if gpinfo.infotype ≠ CLEAN then Help(gpinfo) ▷ dflag CAS
        else if pinfo.infotype ≠ CLEAN then Help(pinfo) ▷ Either finish deletion or unflag
        else {
            ⊲ Try to flag gpn
            op := a new DFlag Info object containing ⟨gpn, pn, ln, pinfo⟩
            result := CAS(gpn.info, gpinfo, gpinfo, op) ▷ dflag CAS
            if result = gpinfo then {
                ⊲ CAS successful
                if HelpDelete(op) then return true ▷ Either finish deletion or unflag
            }
            else Help(result) ▷ The dflag CAS failed; help the operation that caused the failure
        }
    }
}

HelpDelete(Info op) : boolean {
    ▷ Precondition: op points to a DFlag Info object (i.e., it is not ⊥)
    Info result, result2, op2, op3, clean
    op2 := a new Mark Info object ⟨Mark, dinfo := op⟩ ▷ mark CAS
    result := CAS(op.pn.info, op.pinfo, op2)
    if result = op.pinfo or [ result.infotype = Mark, result.dinfo = op ] then {
        ⊲ op.pn is successfully marked
        HelpMarked(op) ▷ Complete the deletion
        return true ▷ Tell Delete routine it is done
    }
    else {
        ⊲ The mark CAS failed
        if result.infotype = IFlag then HelpInsert(result) ▷ op.pn is an IFlag node
        if result.infotype = Mark then HelpMarked(result.dinfo) ▷ op.pn is a Mark node
        if result.infotype = DFlag then {
            ⊲ op.pn is a DFlag node
            op3 := a new Mark Info object ⟨Mark, dinfo := result⟩
            ▷ Non-recursively help the DFlag node
            result2 := CAS(result.pn.info, result.pinfo, result.dinfo, op3)
            if result2 = result.pinfo or [ result2.infotype = Mark, result2.dinfo = result ]
            then HelpMarked(result)
            else {
                clean := a new Clean Info object ▷ The non-recursive mark help fails
                CAS(result.gpn.info, result, clean) ▷ Help op.pn backtrack
            }
            clean := a new Clean Info object ▷ backtrack CAS
            CAS(op.gpn.info, op, clean)
            return false ▷ Tell Delete routine to try again
        }
    }
}

HelpMarked(Info op) {
    ▷ Precondition: op points to a DFlag Info object (i.e., it is not ⊥)
    Node other; Info clean;
    ▷ Set other to point to the sibling of the node to which op.ln points
    if op.pn.right = op.ln then other := op.pn.left else other := op.pn.right ▷ dchild CAS
    ▷ Splice the node to which op.pn points out of the tree, replacing it by other
    CAS-Child(op.gpn, op.pn, other)
    clean := a new Clean Info object ▷ dunflag CAS
    CAS(op.gpn.info, op, clean)
}

Figure 3.11: Pseudocode for Delete and some auxiliary routines.[3]
Help(Info u) {
  ▷ General-purpose helping routine
  ▷ Precondition: u has been stored in the info field of some internal node
  if u.infotype = IFlag then HelpInsert(u)
  else if u.infotype = Mark then HelpMarked(u)
  else if u.infotype = DFlag then HelpDelete(u)
}

CAS-Child(Node parent, Node old, Node new) {
  ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is ⊥)
  ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
  if new.key < parent.key then
    CAS(parent.left, old, new)
  else
    CAS(parent.right, old, new)
}

Figure 3.12: Pseudocode for Delete and some auxiliary routines [3].

sequence of nodes \(⟨n_0, n_1, n_2, \cdots, n_m⟩\) that \(ln\) points to. More specifically, \(n_0\) is root since \(ln\) is first set to root on Line 2. For \(1 \leq i \leq m\), \(n_i\) is the node that \(ln\) points to immediately after \(ln\) gets updated by the \(i\)th iteration of Line 8.

An \(Insert(k)\) operation first creates a new leaf node containing \(k\) at Line 21. Then, it tries to insert this leaf until it succeeds. In a single iteration of the loop, if the leaf node returned by a \(Search(k)\) sub-routine does not contain \(k\), and no other operation was changing \(pn\), it creates a subtree containing three nodes (Line 28-29). After that, the operation creates an Info object that stores the information about the operation (Line 30) and tries to flag \(pn\). If the flagging succeeds, the operation changes a child pointer of \(pn\) from \(ln\) to the newly created subtree and unflags the node with IFLAG state to CLEAN (Line 42-43). If the flagging was blocked by another unfinished operation, the \(search\) tries to help the other operation and then starts its own work again (Line 36).
A \textit{Delete}(k) operation returns \textit{false} if the \textit{Search}(k) returns a leaf node which does not contain \( k \). Otherwise, the \textit{Delete}(k) operation tries to remove the leaf returned by the \textit{Search}(k) from the BST. It consists of three main steps: flag the grandparent node \( gpn \) using a dflag \textit{CAS} (Line 53-54), mark the parent node \( pn \) using a mark \textit{CAS} (Line 61-62), and change the child pointer using a dchild \textit{CAS} (Line 87). If the dflag \textit{CAS} on \( gpn \) is blocked by another unfinished operation, the delete helps the unfinished operation (Line 58). After setting the \textit{state} of \( gpn \) to DFLAG, it attempts a mark \textit{CAS} on \( pn \). If this mark \textit{CAS} is blocked by another unfinished operation, the delete helps the unfinished one (Line 67-78) and backtracks (\textit{i.e.}, performs Line 79-80 to set \( gpn \)'s state to CLEAN) and starts a new iteration of \textit{Delete}(k). Otherwise, the mark \textit{CAS} succeeds and the delete operation continues by performing a dchild \textit{CAS} (Line 85) to change the child pointer and then resetting the state of \( gpn \) to CLEAN using a dunflag \textit{CAS} (Line 87).

### 3.2 A Simplified Algorithm

To make our verification of the proof of correctness easier, we introduce a simplified version of the non-blocking BST algorithms without helping mechanisms and prove this new version correct in PVS using simulations. Once this proof is complete, we believe it will be possible to extend it to prove the correctness of the original algorithm. The ideas behind the simplified algorithm are the same as the original...
one, except that if an operation is blocked by other unfinished operations, it tries again and until the unfinished one gets finished. This technique is called busy waiting, and does not guarantee the progress property. The pseudocode for the simplified algorithms is shown in Figure 3.13 and 3.14.

We use $S_i$, $F_i$, $I_i$ and $D_i$ to represent the $i$th Line in the Search, Find, Insert and Delete pseudocode in Figure 3.13 and 3.14, respectively. In the simplified version of the BST algorithm, we have made a few changes. In the original paper, a bit in the word of the node’s pointer to an Info object represents the type of Info object. But we use the $infotype$ (CLEAN/IFLAG/DFLAG/MARK) field inside the Info object to distinguish them. This makes it more clear and straightforward when we implement the algorithm. As a consequence, we always create new CLEAN objects to avoid the ABA problem.

The main steps of Find, Insert and Delete operations are the same as in the original algorithms. The subroutine $Search(k)$ remains the same as before and is used by all $Find(k)$, $Insert(k)$ and $Delete(k)$ operations. The $Find(k)$ operation is exactly the same. The $Insert(k)$ operation inserts a node containing $k$ (created at I1) if there is no such leaf node containing $k$ found by $Search(k)$. First, it calls the subroutine $Search(k)$ to determine if there is a leaf node that potentially contains $k$ (I2). If such a leaf does not exist, the operation attempts to insert the key into the BST. From I6 to I9, a new subtree containing three nodes is created. The operation
then attempts to set the state of $pn$ to IF $LAG$ by a iflag $CAS$ (I10-I11). If this iflag $CAS$ is blocked by an other unfinished operation, it loops and tries again. Otherwise, after a successful iflag $CAS$, it changes the child pointer of $pn$ from $ln$ to the newly created subtree by an ichild $CAS$ (I13-I15). The operation changes the state of $pn$ to CLEAN by an iunflag $CAS$ (I16-I17).

A $Delete(k)$ operation searches the BST to check if there is a node potentially containing $k$ (D1). It returns $false$ if the leaf node returned by $Search(k)$ does not contain $k$. Otherwise, the operation sets $gpn$’s state to DFLAG by a dflag $CAS$ (D6-D7). If the dflag $CAS$ is blocked by some other unfinished operations, the current $Delete(k)$ loops and attempts again. After a successful dflag $CAS$ (D8), the $delete$ operation tries to set the state of $pn$ to MARK by a mark $CAS$ (D9-D10). If the mark $CAS$ is blocked by some other unfinished operation, it backtracks (D20) and starts $Delete(k)$ again. If the mark $CAS$ succeeds (D12), the operation then changes the child pointer of $gpn$ from $pn$ to the sibling of $ln$ using a dchild $CAS$ (D17-D18), thereby deleting $ln$ from the BST. After the dchild $CAS$, a dunflag $CAS$ (D19) resets the state of $gpn$ to CLEAN.
Search(Key k) : <Node, Node, Node, Info, Info>
1 ln ← Root
2 while ln is not a leaf {
3 gpn ← pn
4 pn ← ln
5 gpinfo ← pinfo
6 pinfo ← pn.info
7 if k < ln.key
8 ln ← pn.left
9 else ln ← pn.right
}

Find(Key k) : Node
1 <-, -, ln, -, -> ← Search(k)
2 if ln.key = k
3 return true
4 else return false

Insert(Key k) : boolean
1 nNode ← newNode(key← k, isleaf← true, isinternal← false)
2 while TRUE {
3 <-, pm, ln, pinfo, -> ← Search(k)
4 ink ← ln.key
5 if ink = k
6 return false
7 if pinfo.infotype = CLEAN {
8 nSib ← newSib(key← ink, isleaf← true, isinternal← false)
9 if k > ink
10 nIntern ← newIntern(key← k, left← nSib, right← nNode,
11 isleaf← false, isinternal← true)
12 else
13 nIntern ← newIntern(key ← ink, left← nNode, right← nSib,
14 isleaf← false, isinternal← true)
15 op ← newIInfo(IFLAG, pm, ln, nIntern)
16 result ← CAS(pm.info, pinfo, op)
17 if result = pinfo {
18 if op.nIntern.key < op.pm.key
19 CAS(op.pm.left, op.ln, op.nIntern)
20 else CAS(op.pm.right, op.ln, op.nIntern)
21 clean ← newCInfo(CLEAN, -, -, -)
22 CAS(op.pm.info, op, clean)
23 return true
24 }
25 }

Figure 3.13: Pseudocode for Search and Find operations.
Delete(Key k) : boolean

    while TRUE {
        1      <gpn, pn, ln, pinfo, gpinfo> ← SEARCH(k)
        2      lnk ← ln.key
        3      if lnk ≠ k
            return false
        4      if gpinfo.infotype = CLEAN {
            5        if pinfo.infotype = CLEAN {
                6        op1 ← newDInfo(DFLAG, gpn, pn, ln, pinfo)
                7        result ← CAS(gpn.info, gpinfo, op1)
                8        if result = gpinfo {
                    9        op2 ← newMInfo(MARK, dinfo ← op1)
                    10       result ← CAS(op1.pn.info, op1.pinfo, op2)
                    11       clean ← newCInfo(CLEAN, −, −, −, −)
                    12       if result = op1.pinfo
                        13          if op1.pn.right = op1.ln
                            14              other ← op1.pn.left
                        15          else other ← op1.pn.right
                        16        if other.key < op1.gpn.key
                            17                      CAS(op1.gpn.left, op1.pn, other)
                        18        else CAS(op1.gpn.right, op1.pn, other)
                        19              CAS(op1.gpn.info, op1, clean)
                        return true }
                    20              } } } }
                9        result = gpinfo 
            10    } } }
        11 } } }
    20 CAS(op1.gpn.info, op1, clean)
    
Figure 3.14: Pseudocode for Delete operations.
4 Modelling the Algorithms

In order to prove the correctness of the simplified BST algorithm using PVS, we model the implementation and the specification as automata which are called the concrete automaton and canonical automaton, respectively. To make the proof easier, we introduce an intermediate automaton and use a hybrid forward and backward simulation to prove correctness. We show that the concrete automaton implements the intermediate one via a forward simulation and the intermediate automaton implements the canonical one via a backward simulation.

4.1 The Canonical Automaton

The canonical automaton models the abstract specifications of the \texttt{SET} data type defined in Definition 3.1. By using the method introduced in Section 2.3.2, we can build the canonical automaton easily.

As mentioned in Section 3.1.2, let \( U \) be a totally ordered set and \( UPlus = U \cup \{\infty_1, \infty_2\} \) such that every value in \( U \) is less than \( \infty_1 \) and \( \infty_2 \) and \( \infty_1 < \infty_2 \). Intuitively, \( U \) contains all possible keys that can be inserted into the data structure.
Let $PROC$ be a finite set of processes. Let $Pcval$ be the set of all possible values for the program counter of a process. More precisely, we define $Pcval$ as follows.

$$Pcval = \{ \text{idle},\ pcDoFind(k), pcFindResp(true), pcFindResp(false),$$

$$pcDoInsert(k), pcInsertResp(true), pcInsertResp(false),$$

$$pcDoDelete(k), pcDeleteResp(true), pcDeleteResp(false) \mid k \in U \}.$$  

The state of the canonical automaton $AbsAut$ is a pair: $(\text{keys}, \text{pc})$, where $\text{keys} \subseteq U$ and $\text{pc} : PROC \rightarrow Pcval$. The initial state $\text{start}$ in the canonical automaton $AbsAut$ has $\text{start}.\text{keys} = \emptyset$ and $\text{start}.\text{pc}(p) = \text{pc}.\text{idle}$ for all $p \in PROC$. In PVS, we model a state of the $AbsAut$ as follows:

$$\text{state} : \text{TYPE} = \{ \text{keys} : \text{setof}[U],$$

$$\text{pc} : [PROC \rightarrow Pcval] \}.\)  

$PROC$ is modelled as subset of the natural numbers from 0 to some $n \geq 1$ in PVS. We use $\text{setof}[U]$ to model a set whose elements are all in $U$. Thus, $\text{state}.\text{keys}$ records the set of keys the BST currently contains, and $\text{state}.\text{pc}$ records the program counter of each process.

Figure 4.1 shows all external and internal actions for $AbsAut$. For each kind of operation, two different internal actions are used to capture the linearization points.
in(AbsAut) = \{findInv_p(k), insertInv_p(k), deleteInv_p(k) \mid k \in U, p \in PROC\}

out(AbsAut) = \{findResp_p(r), insertResp_p(r), deleteResp_p(r) \mid r \in boolean, p \in PROC\}

int(AbsAut) = \{doFindT_p(k), doFindF_p(k), doInsertT_p(k), doInsertF_p(k),
\quad doDeleteT_p(k), doDeleteF_p(k) \mid k \in U, p \in PROC\}

Figure 4.1: Actions of the canonical automaton AbsAut for a SET data type.

of operations that return true or false. All transitions for the AbsAut are defined in Table 4.1. To make the description similar to our formalization in PVS, we use keys.add(k) or keys.remove(k) to represent adding or removing an element k from a set keys.

4.2 The Concrete Automaton

The concrete automaton ConcAut is used to represent the implementation. This automaton models the pseudocode we described in Figure 3.13 and 3.14. More details of modelling the ConcAut in PVS can be found in our PVS scripts. We only discuss some key parts of the modelling here.

A state of ConcAut contains four parts: program counters, local variables, shared objects in shared memory and auxiliary variables. The program counter of a process records which line of code the process will next execute. We define a set Pcval
<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>$findInv(k, p)$</td>
<td>$s.pc(p) = idle$</td>
<td>$s.pc(p) \leftarrow pcDoFind(k)$</td>
</tr>
<tr>
<td>$doFindT(k, p)$</td>
<td>$s.pc(p) = pcDoFind(k)$ ( k \in s.keys )</td>
<td>$s.pc(p) \leftarrow pcFindResp(true)$</td>
</tr>
<tr>
<td>$doFindF(k, p)$</td>
<td>$s.pc(p) = pcDoFind(k)$ ( k \notin s.keys )</td>
<td>$s.pc(p) \leftarrow pcFindResp(false)$</td>
</tr>
<tr>
<td>$findResp(r, p)$</td>
<td>$s.pc(p) = pcFindResp(r)$</td>
<td>$s.pc(p) \leftarrow idle$</td>
</tr>
<tr>
<td>$insertInv(k, p)$</td>
<td>$s.pc(p) = idle$</td>
<td>$s.pc(p) \leftarrow pcDoInsert(k)$</td>
</tr>
<tr>
<td>$doInsertT(k, p)$</td>
<td>$s.pc(p) = pcDoInsert(k)$ ( k \notin s.keys )</td>
<td>$s.pc(p) \leftarrow pcInsertResp(true)$ ( s.keys.add(k) )</td>
</tr>
<tr>
<td>$doInsertF(k, p)$</td>
<td>$s.pc(p) = pcDoInsert(k)$ ( k \in s.keys )</td>
<td>$s.pc(p) \leftarrow pcInsertResp(false)$</td>
</tr>
<tr>
<td>$insertResp(r, p)$</td>
<td>$s.pc(p) = pcInsertResp(r)$</td>
<td>$s.pc(p) \leftarrow idle$</td>
</tr>
<tr>
<td>$deleteInv(k, p)$</td>
<td>$s.pc(p) = idle$</td>
<td>$s.pc(p) \leftarrow pcDoDelete(k)$</td>
</tr>
<tr>
<td>$doDeleteT(k, p)$</td>
<td>$s.pc(p) = pcDoDelete(k)$ ( k \in s.keys )</td>
<td>$s.pc(p) \leftarrow pcDeleteResp(true)$ ( s.keys.remove(k) )</td>
</tr>
<tr>
<td>$doDeleteF(k, p)$</td>
<td>$s.pc(p) = pcDoDelete(k)$ ( k \notin s.keys )</td>
<td>$s.pc(p) \leftarrow pcDeleteResp(false)$</td>
</tr>
<tr>
<td>$deleteResp(r, p)$</td>
<td>$s.pc(p) = pcDeleteResp(r)$</td>
<td>$s.pc(p) \leftarrow idle$</td>
</tr>
</tbody>
</table>

Table 4.1: Transitions of the canonical automaton $AbsAut$, where $s$ is a variable of TYPE $state$, $k$ is an element of $U$ and $P$ is an element of $PROC$.

of possible values for a process’s program counter. Intuitively, each line of the pseudocode is modelled as an element in $Pcval$.
\[ \text{Pcval} = \left\{ \text{idle}, \right. \]
\[ pcSearch1, pcSearch2, \ldots, pcSearch9, \]
\[ pcFind1, pcFind2, pcFindResp(r), \]
\[ pcInsert1, pcInsert2, \ldots, pcInsert17, pcInsertResp(r), \]
\[ pcDelete1, pcDelete2, \ldots, pcDelete20, pcDeleteResp(r) \mid r \in \text{boolean} \left\} \right. \]

Then, the component \( \text{pc} \) of the state of ConcAut is a function \( \text{pc} : \text{PROC} \rightarrow \text{Pcval} \).

The way to model shared objects in ConcAut is a bit tricky. Node and Info objects which are defined in Figure 3.9 are modelled as two abstract types in PVS called Node and Info. Their fields, such as the child pointers of a node, the key field of a node or the leaf field of an Info object are modelled as functions from Node (Info) to the desired type. For clarity, the name of each field has a “f” as suffix. Thus, shared variables are modelled by the functions described in Table 4.2.

One can easily construct the types in Table 4.2 from Figure 3.9. The Flag type is defined by: Flag TYPE = \{CLEAN, DFLAG, IFLAG, MARK\}, as described in Section 3.1.2.

In order to record the local information of each process, each local variable is modelled by a component of the state in ConcAut. Because these variables are local, they are modelled as functions from processes to the appropriate type, as listed in Table 4.3.
Table 4.2: Representing fields of shared objects in the state of \textit{ConcAut}.

<table>
<thead>
<tr>
<th>Node object</th>
<th>Info object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shared variable</strong></td>
<td><strong>function</strong></td>
</tr>
<tr>
<td>keyf</td>
<td>Node $\to$ UPlus</td>
</tr>
<tr>
<td>leftf</td>
<td>Node $\to$ Node</td>
</tr>
<tr>
<td>rightf</td>
<td>Node $\to$ Node</td>
</tr>
<tr>
<td>infof</td>
<td>Node $\to$ Info</td>
</tr>
<tr>
<td>isinternf</td>
<td>Node $\to$ boolean</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Local variables of a state in \textit{ConcAut}.

<table>
<thead>
<tr>
<th>local variable</th>
<th>function</th>
<th>local variable</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret_addr</td>
<td>PROC $\to$ pc_return</td>
<td>k</td>
<td>PROC $\to$ U</td>
</tr>
<tr>
<td>lnk</td>
<td>PROC $\to$ UPlus</td>
<td>pn</td>
<td>PROC $\to$ Node</td>
</tr>
<tr>
<td>gpn</td>
<td>PROC $\to$ Node</td>
<td>ln</td>
<td>PROC $\to$ Node</td>
</tr>
<tr>
<td>other</td>
<td>PROC $\to$ Node</td>
<td>result</td>
<td>PROC $\to$ Info</td>
</tr>
<tr>
<td>gpinfo</td>
<td>PROC $\to$ Info</td>
<td>pinof</td>
<td>PROC $\to$ Info</td>
</tr>
<tr>
<td>op</td>
<td>PROC $\to$ Info</td>
<td>op1</td>
<td>PROC $\to$ Info</td>
</tr>
<tr>
<td>op2</td>
<td>PROC $\to$ Info</td>
<td>clean</td>
<td>PROC $\to$ Info</td>
</tr>
<tr>
<td>nSib</td>
<td>PROC $\to$ Node</td>
<td>nIntern</td>
<td>PROC $\to$ Node</td>
</tr>
</tbody>
</table>

All local variables in Table 4.3 are straightforward to obtain from the simplified algorithm, except for ret_addr. This local variable is used when the \textit{search} subroutine is invoked and it records where to continue from if the subroutine completes. Hence, $pc\_return = \{pc\text{\textunderscore}Find2, pc\text{\textunderscore}Insert3, pc\text{\textunderscore}Delete2\} \subseteq P\text{\textunderscore}val$. The states of \textit{ConcAut} also include auxiliary variables: $aux\_keys \subseteq U$, $aux\_seen\_in$, $aux\_seen\_out : PROC \to boolean$. They do not model anything in the pseudocode, but are used
to simplify our proofs. They are discussed in Section 4.4.

The initial state of ConcAut is defined as follows:

* Most local variables are initialized to NIL, except that for all $p : pc(p)=idle$, $lnk(p)=\infty$.

* The value of some fields of shared objects, namely the keyf and isinternf fields of a Node object and the infotypef field of an Info object, is not specified. Their initial value are irrelevant to some lemmas we need to prove later.

* Most fields of shared objects are initialized to NIL, except three shared Node objects listed in Table 4.4 and three Info objects listed in Table 4.5.

* The initial values of auxiliary variables are: $aux\_keys = \emptyset$, $aux\_seen\_in(p) = false$, $aux\_seen\_out(p) = false$ for all $p \in PROC$. (More details are discussed in Section 4.4.)

There are three allocated Nodes: root and its two children ($nInf_1$ and $nInf_2$) in an initial state, as well as the Info objects that belong to them ($CL_1$, $CL_2$ and $CL_3$).

As discussed in Section 2.4.1, the idea of building the ConcAut is straightforward: each line of the code, which contains at most one shared memory access, is modelled by a single internal action except for an if statement, a test of the exit condition of a while loop or a CAS operation. Each of those three types of lines are modelled by
two actions: a successful one and a failed one. However, there are some actions in our concrete automaton consisting of several shared memory access. That is allowed, because only one of them accesses a changeable field, and the others are reading from unchangeable fields. Hence, it does not matter if we collapse steps that read from unchangeable fields into one action. For instance, when an Info object is created, the values of its fields remain unchanged. Hence, each of Lines I14, I15, I17, D10, D17, D18, D19 and D20 can be regarded as an atomic action in our concrete automaton, thereby simplifying our model of the concrete automaton. In addition to the internal

<table>
<thead>
<tr>
<th>field $f$</th>
<th>value of $f(root)$</th>
<th>value of $f(nInf1)$</th>
<th>value of $f(nInf2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyf</td>
<td>$\infty_2$</td>
<td>$\infty_1$</td>
<td>$\infty_2$</td>
</tr>
<tr>
<td>leftf</td>
<td>nInf1</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>rightf</td>
<td>nInf2</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>infof</td>
<td>CL1</td>
<td>CL2</td>
<td>CL3</td>
</tr>
<tr>
<td>isinternf</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 4.4: Initial state of root and its two children.

<table>
<thead>
<tr>
<th>field $f$</th>
<th>value of $f(CL1)$</th>
<th>value of $f(CL2)$</th>
<th>value of $f(CL3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>infotypef</td>
<td>CLEAN</td>
<td>CLEAN</td>
<td>CLEAN</td>
</tr>
<tr>
<td>gpnf</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>pnf</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>lnf</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>nInternf</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>pinfof</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
<tr>
<td>dinfo</td>
<td>NIL</td>
<td>NIL</td>
<td>NIL</td>
</tr>
</tbody>
</table>

Table 4.5: Initial state of Info objects belong to root and its two children.
actions, for each kind of operation, we define two external actions: an invocation and a response action. In the same way we modelled the implementation of the *fetch-and-increment* object in Table 2.3, we model the BST algorithm as follows. Most of steps in the pseudocode can be trivially translated into a transition in ConcAut, except for a few cases. All three operations (*find*, *insert* and *delete*) invoke the *search* subroutine. When the invocation of process *p* occurs, we set ret_addr(*p*) to the appropriate return address, while changing pc(*p*) to pcSearch1. Another interesting case is to model an allocation step in the pseudocode. We introduce two new variables *allocatedNode* and *allocatedInfo* in the state of ConcAut, which maintain a set of used Nodes and Info objects, respectively. Hence, whenever an allocation step for a Node is performed by a process *p*, we pick a node that is not in *allocatedNode* and return the node to *p*, add the node to *allocatedNode*, and then assign appropriate values to its fields. This can be done by assuming an axiom that there are always infinitely many unallocated nodes to pick. Allocation of an Info object is done in the same way.

Table 4.6 shows some examples of modelling lines of the pseudocode. Whenever an invocation is performed at state *s* by a process *p*, the key *k* is saved into the local variable *s.k(p)*. The function newNode shown in Figure 4.2 behaves exactly as we discussed above. More precisely, when *p* creates a new *nNode* at Line I1, it picks an unused Node object *n* and Info object *x* and adds them into *allocatedNode* and
allocatedInfo set, sets the type of \( x \) to CLEAN, points \( infof(n) \) to \( x \), assigns the node a key value and sets \( isinternf(n) \) to \( false \). If \( p \) creates a MARK Info object through the newMInfo function that has three parameters, \( p \) picks an unused Info object \( x \) and adds it into the allocatedInfo set, sets \( infotypef(x) \) to MARK, points \( dinfof(x) \) to an Info object and returns this newly allocated MARK Info object to \( op2(p) \).

<table>
<thead>
<tr>
<th>newNode( (c: state, p: PROC, k: U) : )</th>
<th>newMInfo( (c: state, p: PROC, dinfo: Info) : )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET ( n = ) getNode( (c) ), ( x = ) getInfo( (c) ).</td>
<td>LET ( x = ) getInfo( (c) ).</td>
</tr>
<tr>
<td>( c.allocatedNode.add(n) )</td>
<td>( c.allocatedInfo.add(x) )</td>
</tr>
<tr>
<td>( c.allocatedInfo.add(x) )</td>
<td>( c.infotypef(x) \leftarrow ) MARK</td>
</tr>
<tr>
<td>( c.infotypef(x) \leftarrow ) CLEAN</td>
<td>( c.dinfof(x) \leftarrow ) ( dinfo )</td>
</tr>
<tr>
<td>( c.infof(n) \leftarrow x )</td>
<td>( c.op2(p) \leftarrow x )</td>
</tr>
<tr>
<td>( c.isinternf(n) \leftarrow ) false</td>
<td></td>
</tr>
<tr>
<td>( c.keyf(n) \leftarrow k )</td>
<td></td>
</tr>
<tr>
<td>( c.nNode(p) \leftarrow n )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Definition of the newNode and newMInfo functions. Function getNode\( (c) \) picks a new Node object that is not in \( c.allocatedNode \). Symmetrically for getInfo\( (c) \).
4.3 An Intermediate Automaton and Backward Simulation

In the concurrent BST algorithm [3], some operations, such as a $find(k)$ operation by a process $p$ that returns $true$, may not actually “take effect” at the time when $p$ determines at Line F2 that $find(k)$ should return $true$ by comparing $k$ to the key of the leaf reached by the search. Consider the example shown in Figure 4.3 (a), which involves three processes. Processes $q_0$ and $q_1$ insert and then delete key 3, while $p$ is concurrently executing $find(3)$. The other diagrams in Figure 4.3 describe the shape of the BST, as a result of those operations. Note that, when $p$ successfully finds the desired key in the leaf, that leaf is no longer in the BST. Namely, when process $p$ executes Line F2 and decides to return $true$ for the $find(3)$ operation, that key is not in the BST. So, Line F2 can not be used as a linearization point of the Find operation. Similarly, I4 and D3 are not the linearization points for some failed Insert and Delete operations, respectively. For those operations, it is not obvious how to define the linearization points explicitly. The proofs by Ellen et al. [3] show that there exists a time during the search when the leaf eventually reached is in the BST. However, at that time, it is not known where the search will eventually end up. So, it is difficult to recognize the linearization point of a search when it happens, without knowing the future actions of that search. Hence, it is difficult to come up with a forward simulation from $ConcAut$ to $AbsAut$ such that some actions in $ConcAut$ are directly mapped to the internal actions of $AbsAut$ that represent
linearization points. Therefore, we use a hybrid forward and backward simulation by building an intermediate automaton $\text{IntAut}$, such that this intermediate automaton simulates the canonical one via a backward simulation and the concrete automaton simulates the intermediate one via a forward simulation.

Since a proof using backward simulation is usually conceptually harder than a proof using a forward simulation, we chose to make $\text{IntAut}$ as similar to $\text{AbsAut}$ as possible to make the backward simulation proof easier [16]. Thus, most of the components of the intermediate automaton will be similar to those in $\text{AbsAut}$. Each process has two additional local boolean variables $\text{seen\_in}(p)$ and $\text{seen\_out}(p)$, which are inspired by the work of Colvin et al. [16]. If $p$ is performing $\text{find}(k)$ or $\text{insert}(k)$, the variable $\text{seen\_in}(p)$ is set to be true if $k$ is in the key set of $\text{IntAut}$ either at the invocation $p$'s operation, or $k$ was not in the key set at the beginning but during $p$’s operation some other $\text{insert}(k)$ operation by $q$ successfully inserts $k$ into the key set of $\text{IntAut}$. Symmetrically, $\text{seen\_out}(p)$ is set to be true if $k$ is not in the key set at the invocation of $p$’s $\text{find}(k)$ or $\text{delete}(k)$, or some $\text{delete}(k)$ by another process successfully deletes $k$ from the key set of $\text{IntAut}$ during $p$’s $\text{find}(k)$ or $\text{delete}(k)$. Intuitively, these two variables record whether the desired key has been in the BST at any time since the beginning of the present operation, with the aim of helping process $p$ to determine the return value of its operation. For instance, if $\text{seen\_in}(p)$ is true when $p$ is performing a $\text{find}(k)$ operation, it means that the key $k$ has been
Figure 4.3: (a) Interleavings of Proc $p$, $q_0$ and $q_1$ which execute $\text{find}(3)$, $\text{insert}(3)$ and $\text{delete}(3)$, respectively (all three operations succeed). (b), (c), (d) illustrate how the three operations modify the binary search tree. (b) Proc $p$ invoked $\text{find}(3)$ and it has set its local variable $ln$ to the internal node with key 5. (c) Proc $q_0$ runs very quickly and successfully inserts key 3. Subsequently, $p$ continues to search for key 3 and has set its local variable $ln$ to (arrives at) internal node with key 3 on Line S9. (d) Proc $q_1$ executes a complete deletion of key 3, but after that $p$ is still able to get to the external node with key 3 and subsequently return $true$. 

\[ p: \text{find}(3) = \text{true} \]
\[ q_0: \text{insert}(3) = \text{true} \]
\[ q_1: \text{delete}(3) = \text{true} \]
in the key set at some time since the invocation of $find(k)$. This, indeed, enables the $find$ operation to return $true$ even if when executing F2, the key $k$ is actually not in the set any more.

A state of the $IntAut$ is the tuple

$$(keys, pc, seen\_in, seen\_out), \text{ where}$$

$$\begin{align*}
\text{keys} & \subseteq U, \\
pc & : \text{PROC} \rightarrow \text{Pcval}, \\
seen\_in & : \text{PROC} \rightarrow \text{boolean}, \\
seen\_out & : \text{PROC} \rightarrow \text{boolean},
\end{align*}$$

and $U$, PROC and Pcval are defined as in the definition of AbsAut. The possible values for $seen\_in$ and $seen\_out$ are $true$ and $false$. The initial states start and actions for the $IntAut$ are shown in Figure 4.4.

We define the states of $IntAut$ in PVS as:

$$\text{state} : \text{TYPE} = \{ pc : [\text{PROC} \rightarrow \text{Pcval}],$$

$$\text{keys} : \text{setof}[U],$$

$$\text{seen\_in} : [\text{PROC} \rightarrow \text{bool}],$$

$$\text{seen\_out} : [\text{PROC} \rightarrow \text{bool}] \}.$$

The actions for $IntAut$ are shown in Table 4.7. Intuitively, $seen\_in(p)$ and $seen\_out(p)$ are initialized during the invocation of each $find, insert$ and $delete$ operation of process $p$. The response of a $find(k)$ operation of process $p$ now depends on the value of $seen\_in(p)$ and $seen\_out(p)$. An $insert_p(k)$ can decide to
Figure 4.4: Initial states and actions of the intermediate automaton \( \text{IntAut} \) for a \( \mathcal{S} \mathcal{E} \mathcal{T} \) data type.

return \text{false} if \( \text{seen.in}(p) \) is \text{true}. Such an insert can be linearized at the time \( k \) was in \text{keys}. Similarly, a \text{delete}_p(k) \) can decide to return \text{false} if \( \text{seen.out}(p) \) is \text{true}. It can be linearized at the time \( k \) was not in \text{keys}. An \text{insert}(k) \) of process \( p \) that returns \text{true} not only adds the value into the abstract key set, but also sets the value of \( \text{seen.in}(q) \) to be true for any process \( q \) that is performing either \text{find}(k) \) or \text{insert}(k). Even if the key \( k \) is deleted later by some operation, by applying these changes, such a \text{find}(k) \) or \text{insert}(k) \) is allowed to return \text{true} or \text{false}, respectively. Similarly, a \text{delete}(k) \) of process \( p \) which returns \text{true} removes the value from the
\[ \text{bsr}(i, a) \equiv (i.\text{keys} = a.\text{keys}) \]

\[
\begin{align*}
&\text{AND } \forall p : [ i.\text{pc}(p) = a.\text{pc}(p) \\
&\quad \text{OR } (i.\text{pc}(p) = \text{pcDoFind}(k) \text{ AND } a.\text{pc}(p) = \text{pcFindResp}(\text{false}) \text{ AND } i.\text{seen_out}(p) = \text{true}) \\
&\quad \text{OR } (i.\text{pc}(p) = \text{pcDoFind}(k) \text{ AND } a.\text{pc}(p) = \text{pcFindResp}(\text{true}) \text{ AND } i.\text{seen_in}(p) = \text{true}) \\
&\quad \text{OR } (i.\text{pc}(p) = \text{pcDoInsert}(k) \text{ AND } a.\text{pc}(p) = \text{pcInsertResp}(\text{false}) \text{ AND } i.\text{seen_in}(p) = \text{true}) \\
&\quad \text{OR } (i.\text{pc}(p) = \text{pcDoDelete}(k) \text{ AND } a.\text{pc}(p) = \text{pcDeleteResp}(\text{false}) \text{ AND } i.\text{seen_out}(p) = \text{true}) \]
\]

Figure 4.5: The backward simulation relation \textit{bsr} between \textit{IntAut} and \textit{AbsAut}.

abstract key set, and sets the value of \textit{seen_out}(q) to be true for any process \(q\) that is performing either \textit{find}(\textit{k}) or \textit{delete}(\textit{k}), thereby allowing such a \textit{find}(\textit{k}) or \textit{delete}(\textit{k}) to return \textit{false}, even if key \textit{k} is inserted into the BST later by some operation.

After defining the intermediate automaton \textit{IntAut}, we construct a backward simulation relation between them as follows. For \(i \in \text{state}(\textit{IntAut})\) and \(a \in \text{state}(\textit{AbsAut})\) we define \textit{bsr} shown in Figure 4.5.

As we can see from the definition, \textit{bsr} contains two parts. The first part (Line 101) requires that the \textit{data} (i.e., the keys set) of the related states of \textit{IntAut}
and AbsAut should be identical, and the second part (Line 102-110) requires that the Pcval of each process p in IntAut stays “in step” with process p in AbsAut, with four exceptions. For example, Line 103 and 104 say that in AbsAut, p may already have executed doFindF, indicating that p’s find operation will subsequently return false, whereas in IntAut, p is still processing the find operation and has not yet decided to return false. This is allowed only if seen_out(p) is true, which means either k is not in the key set at the invocation of the find operation, or is present at the invocation but is subsequently successfully deleted by some other process q before the doFindF is performed. The other three cases are similar.

When we construct the execution sequence of AbsAut in the backward simulation, for each action of IntAut, we choose the same action for the AbsAut, with the following exceptions. Intuitively, a find\(_p\)(k) operation that returns true is linearized either at the time when the search begins (if key k is in the BST at the beginning of find\(_p\)(k)) or at the time immediately after some other operation successfully inserts k (if key k is not in the BST at the beginning of find\(_p\)(k)). At least one of those situations must be applicable, because seen_in(p) must be true before performing a doFindT\(_p\)(k) in IntAut. Hence, key k is either in the BST at the beginning of a find\(_p\)(k) or k is inserted by some other operations during the find\(_p\)(k). Accordingly, when a findInv(k, p) action is performed in IntAut, we choose a sequence of actions containing the same findInv(k, p) action in AbsAut. This sequence in
AbsAut may also contain a doFindT(k, p) action immediately after the invocation, if seen_in(p) is true in IntAut and the find_p(k) operation subsequently returns true. We know the future behaviour of an operation, because it is a backward simulation. Figure 4.6 shows an example.

In the other case, when the find_p(k) operation subsequently returns true in the future, but after the invocation of the find_p(k) seen_in(p) is false, we linearize doFindT(k, p) immediately after a doInsertT(k, q) by some process q. Therefore, when a doInsertT(k, p) action that successfully adds k into the key set in IntAut occurs, we may choose a sequence of actions not only containing the same doInsertT(k, p) action in AbsAut, but also followed by one doFindT(k, q) action for each q that is executing a find(k) operation that subsequently returns true in the post state of AbsAut. Figure 4.7 shows an example of this case.

Figure 4.6 and 4.7 illustrate examples of how we construct states and actions in AbsAut step by step starting from the end of the execution. In Figure 4.6, the doFindT(3, p) action in IntAut is linearized immediately after its invocation, because key 3 is in the BST at the invocation. However, in Figure 4.7, we cannot do the same thing, because the post state of findInv(3, p) in AbsAut indicates that find(3) will not subsequently return true. Note that when doInsertT(3, q) occurs in IntAut, we choose a sequence of actions containing the same doInsertT(3, q) action in AbsAut, followed by one doFindT(3, p) action for process p because it is
Figure 4.6: A simple example of how the backward simulation bsr works between IntAut and AbsAut. Circles are known states and the dashed circle is the state constructed backwardly according to actions taken in AbsAut.

executing a $find(k)$ operation that subsequently returns true according to the post state of $doInsertT(3,q)$ in AbsAut. It is also important to see that states paired by the green dotted lines satisfy bsr.

Similarly, a $find_p(k)$ operation that returns false is linearized either at the time when the search begins (if key $k$ is not in the BST at the beginning of $find_p(k)$) or at the time immediately after some other operation successfully deletes $k$ (if key $k$ is in the BST at the beginning of $find_p(k)$). At least one of those situations must be applicable, because $seen_out(p)$ must be true before performing a $doFindF_p(k)$
Figure 4.7: Another example of how the backward simulation \( bs_r \) works between \( IntAut \) and \( AbsAut \).

In \( IntAut \). Hence, key \( k \) is either not in the BST at the beginning of the \( find_p(k) \) or \( k \) is deleted by some other operation during the \( find_p(k) \). Therefore, when a \( findInv(k,p) \) action is performed in \( IntAut \), we choose a sequence of actions containing the same \( findInv(k,p) \) action in \( AbsAut \). In addition, the sequence contains a \( doFindF(k,p) \) action immediately after the invocation, if \( seen_out(p) \) is \( true \) in \( IntAut \) and the \( find_p(k) \) subsequently returns \( false \). Otherwise, if \( find_p(k) \) operation subsequently returns \( true \), but \( seen_out(p) \) is \( false \) when the find is invoked, we linearize \( doFindT(k,p) \) immediately after a \( doDeleteT(k,q) \) by some process \( q \). Consequently, when a successful \( doDeleteT(k,p) \) action in \( IntAut \) occurs, we may
choose a sequence of actions not only containing the same \( \text{doDeleteT}(k, p) \) action in \( AbsAut \), but also followed by one \( \text{doFindF}(k, q) \) action for each \( q \) that is executing a \( \text{find}(k) \) operation that subsequently returns \( false \) according to the post state of \( AbsAut \).

Similarly, an \( \text{insert}_p(k) \) (or a \( \text{delete}_p(k) \)) that returns \( false \) may be linearized at the time immediately after its invocation or immediately after some other successful \( \text{doInsertT}(k, q) \) (or \( \text{doDeleteT}(k, q) \)), depending on the value of \( \text{seen} \_\text{in}(p) \) (\( \text{seen} \_\text{out}(p) \)).

To summarize: when a \( \text{doInsertT}(k, p) \) action in \( IntAut \) occurs, we may choose a sequence of actions not only containing the same \( \text{doInsertT}(k, p) \) action in \( AbsAut \), but also followed by one \( \text{doFindT}(k, q) \) action for each \( q \) that is executing a \( \text{find}(k) \) operation that subsequently returns \( true \) and one \( \text{doInsertF}(k, q) \) action for each \( q \) that is executing an \( \text{insert}(k) \) that subsequently returns \( false \) according to the post state of \( AbsAut \). When a successful \( \text{doDeleteT}(k, p) \) action in \( IntAut \) occurs, we may choose a sequence of actions not only containing the same \( \text{doDeleteT}(k, p) \) action in \( AbsAut \), but also followed by one \( \text{doFindF}(k, q) \) action for each \( q \) that is executing a \( \text{find}(k) \) operation that subsequently returns \( true \) according to the post state of \( AbsAut \), and one \( \text{doDeleteF}(k, q) \) action for each \( q \) that is executing a \( \text{delete}(k) \) operation that subsequently returns \( false \) according to the post state of \( AbsAut \). Because we already linearized \( \text{doFindT}(k, p) \), \( \text{doInsertF}(k, p) \),
doFindF\( (k, p) \) and doDeleteF\( (k, p) \) actions of AbsAut, when any of these actions are performed in IntAut, they are ignored (i.e., we do not choose any action in AbsAut for those four types of actions in IntAut). This is the action correspondence between IntAut and AbsAut.

By using the bsr relation and our explicit construction of the action correspondence, we were able to show that a backward simulation exists between IntAut and AbsAut. We have formalized the proof of this backward simulation using PVS.

### 4.4 The Forward Simulation

We also construct a forward simulation \( fsr \) from ConcAut to IntAut. Firstly, we describe the action correspondence of the forward simulation. Most internal actions in ConcAut correspond to the empty sequence \( \epsilon \) of IntAut, except for some key actions shown in Table 4.8.

Intuitively, successful ick Child CASs (insert14\( T(p) \) and insert15\( T(p) \)) starting from a state \( c \) in ConcAut are mapped to doInsertT\( (c.k(p), p) \) in IntAut, because both of these actions insert key \( c.k(p) \) into the BST. Successful dchild CASs (delete17\( T(p) \) and delete18\( T(p) \)) starting from \( c \) in ConcAut are mapped to doDeleteT\( (c.k(p), p) \) in IntAut, since these actions delete key \( c.k(p) \) from the BST. If an insert4\( T(p) \) or find2\( T(p) \) starting from \( c \) is performed in ConcAut, we shall prove that the given key \( c.k(p) \) has been in the BST at some time since the be-
ginning of the invocation, which is similar to the pre-condition for performing an 
\textit{doInsertF}(c.k(p), p) or \textit{doFindT}(c.k(p), p) in \textit{IntAut}. If a \textit{delete3T}(p) or \textit{find2F}(p) 
is performed, we shall prove there was a time since the beginning of the invocation 
when the given key \(c.k(p)\) was not in the BST. Hence, these two actions can be 
mapped to \textit{doDeleteF}(c.k(p), p) or \textit{doFindF}(c.k(p), p), respectively. Each external 
action of \textit{ConcAut} is mapped to its counterpart in \textit{IntAut}.

Once again, \textit{fsr} consists of a \textit{data} relationship and a \textit{Pcval} relationship. However, 
since it is not convenient for us to relate the concrete data structure of \textit{ConcAut} 
to the abstract set in \textit{IntAut} directly, we add an auxiliary variable \textit{aux\_keys} to the 
state of \textit{ConcAut} to represent all current keys in the BST. Therefore, the data rela-
tion part in \textit{fsr} can simply require that \textit{aux\_keys} of \textit{ConcAut} is the same as \textit{keys} 
of \textit{IntAut} if we establish as an invariant that \textit{aux\_keys} matches the set of all keys 
in the leaves of the BST. More specifically, \textit{aux\_keys} is updated as follows.

\textit{aux\_keys}: Intuitively, this variable denotes all keys in the reachable leaves of 
the BST in \textit{ConcAut}. Initially, \textit{aux\_keys} = \emptyset. The new key \(k\) is added if 
a successful \textit{ichild CAS} of \textit{insert}_p(k) operation is performed. The key \(k\) is 
removed if a successful \textit{dchild CAS} of \textit{delete}_p(k) operation is performed.

Similarly, if there is a transition \(c \xrightarrow{\alpha} \textit{ConcAut} c'\) and a state \(i\) in \textit{IntAut}, such that 
\((c, i) \in \textit{fsr}\), it is not convenient to reason about the value of \textit{i\_seen\_in} or \textit{i\_seen\_out} 
directly from the given state of \textit{ConcAut}. We thus introduce \textit{aux\_seen\_in} and
The auxiliary variables aux_seen_in and aux_seen_out are updated as follows.

1. aux_seen_out: [PROC → bool]. The variable aux_seen_out(p) is set to true or false according to whether k ∉ aux_keys or not when a findInv(k, p) or deleteInv(k, p) action is performed. When a successful dchild CAS (delete17T(p) or delete18T(p)) is performed at a state c and c.k(p) = k, then for each process q such that aux_out_affected(q, c) (see Figure 4.8) is true and the given key of q is equal to k, aux_seen_out(q) is set to true.

2. aux_seen_in: [PROC → bool]. The variable aux_seen_in(p) is set to true or false according to whether k ∈ aux_keys or not when a findInv(k, p) or insertInv(k, p) action is performed. When a successful ichild CAS (insert14T(p) or insert15T(p)) is performed at state c and c.k(p) = k, then for each process q such that aux_in_affected(q, c) (see Figure 4.9) is true and the given key of q is equal to k, aux_seen_in(q) is set to true.

The function aux_in_affected(p, c) is evaluated to be true, if process p is either in Line F1-F2 or Line I1-I15 of the simplified algorithm or is performing a search(v) subroutine which is not invoked by a delete operation. Note that, p is considered to have completed an insert operation, if it is performing Line I17 or I18. Hence, aux_seen_in(p) cannot be affected. Likewise, aux_out_affected(p, c) is evaluated to be true, if process p is either in Line F1-F2 or Line D1-D18 or Line D20 of the simplified
aux_in_affected(p, c) \equiv c.pc(p) \in \{ pcFind1, pcFind2, \\
    pcInsert1, pcInsert2, pcInsert3, pcInsert4, pcInsert5, \\
    pcInsert6, pcInsert7, pcInsert8, pcInsert9, pcInsert10, \\
    pcInsert11, pcInsert12, pcInsert13, pcInsert14, pcInsert15 \} \\
    \text{OR} \left( c.pc(p) \in \{ pcSearch1, pcSearch2, pcSearch3, pcSearch4, pcSearch5, \\
    pcSearch6, pcSearch7, pcSearch8, pcSearch9 \} \right) \\
    \text{AND} c.ret_addr \neq pcDelete2 \right)
aux_in_affected(p, c) \equiv c.pc(p) \in \{ pcFind1, pcFind2, pcDelete1, pcDelete2, pcDelete3, pcDelete4, pcDelete5, pcDelete6, pcDelete7, pcDelete8, pcDelete9, pcDelete10, pcDelete11, pcDelete12, pcDelete13, pcDelete14, pcDelete15, pcDelete16, pcDelete17, pcDelete18, pcDelete20 \} \\
OR \left( c.pc(p) \in \{ pcSearch1, pcSearch2, pcSearch3, pcSearch4, pcSearch5, pcSearch6, pcSearch7, pcSearch8, pcSearch9 \} \right) \\
\text{AND } c.ret_addr \neq pcInsert3)

Figure 4.9: If an action starting from c, is a dchild CAS, aux_seen_out(p) may be set to true for all p if aux_out_affected(p, c) is true.

Thus, for \( c \in \text{states}(ConcAut) \) and \( i \in \text{states}(IntAut) \), the data relationship between ConcAut and IntAut is stated as follows.

\[
fsr_{\text{data}_rel}(c, i) \equiv (c.aux_keys = i.keys) \land (c.aux_seen_in = i.seen_in) \\
\land (c.aux_seen_out = i.seen_out). \quad (4.1)
\]

In addition to the data relation in \( fsr \), there is also a program counter relation. For \( c \in \text{state}(ConcAut) \) and \( i \in \text{state}(IntAut) \), the program counter relation is defined in Figure 4.10. Relation \( fsr_{pc.rel.find}(c, i, p) \) describes that \( p \) is performing a find operation in ConcAut and \( p \) is performing a corresponding
\[ \text{fsr}_{pc}_{rel}(c,i) \equiv \forall p: \text{fsr}_{pc}_{rel}\text{find}(c,i,p) \]

OR \( \text{fsr}_{pc}_{rel}\text{insert}(c,i,p) \)

OR \( \text{fsr}_{pc}_{rel}\text{delete}(c,i,p) \)

OR ( inSearch(c,p) AND c.ret_addr = pcFind AND i.pc = pcDoFind(c.kp) )

OR ( inSearch(c,p) AND c.ret_addr = pcInsert AND i.pc = pcDoInsert(c.kp) )

OR ( inSearch(c,p) AND c.ret_addr = pcDelete AND i.pc = pcDoDelete(c.kp) )

Figure 4.10: Program counter relation of \text{fsr}.

\textit{find} operation in \textit{IntAut}. This relation is also shown in Figure 4.11. Relations \( \text{fsr}_{pc}_{rel}\text{insert}(c,i,p) \) and \( \text{fsr}_{pc}_{rel}\text{delete}(c,i,p) \) describing the program counter relations for an \textit{insert} and \textit{delete} operation are defined in a similar way. They are shown in Figure 4.13 and Figure 4.14, respectively. Details of the three relations \( \text{fsr}_{pc}_{rel}\text{find}(c,i,p) \), \( \text{fsr}_{pc}_{rel}\text{insert}(c,i,p) \) and \( \text{fsr}_{pc}_{rel}\text{delete}(c,i,p) \) are shown in Figure 4.12.

Intuitively, if \( (c,i) \in \text{fsr}_{pc}_{rel} \), then for every process \( p \), \( \text{fsr}_{pc}_{rel}\text{find}(c,i,p) \) is satisfied, or \( \text{fsr}_{pc}_{rel}\text{insert}(c,i,p) \) is satisfied, or \( \text{fsr}_{pc}_{rel}\text{delete}(c,i,p) \) is satisfied. Otherwise, it is in the case that process \( p \) is performing a \textit{search} subroutine as shown in Figure 4.15 if the \textit{search} is invoked by a \textit{find} operation in \textit{ConcAut} then \( i.pc = pcDoFind(c.kp) \) is \textit{true} in \textit{IntAut}, or if the \textit{search} is invoked by an \textit{insert} operation in \textit{ConcAut} then \( i.pc = pcDoInsert(c.kp) \) is \textit{true} in \textit{IntAut},
otherwise (the search is invoked by a delete operation), \( i.p_c = pcDoDelete(c.k_p) \) should be true in IntAut. Overall, a forward simulation relation \( fsr \) is defined as follows.

\[
fsr(c, i) \equiv fsr_{data\_rel}(c, i) \land fsr_{pc\_rel}(c, i).
\]
<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>findInv((k,p))</td>
<td>(s.\text{pc}(p) = \text{idle})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcFind1} \ s.k(p) &amp; \gets k \ \text{aux}<em>{\text{seen}}</em>{\text{in}}(p) &amp; \gets (k \in \text{s.aux}_\text{keys}) \ \text{aux}<em>{\text{seen}}</em>{\text{out}}(p) &amp; \gets (k \notin \text{s.aux}_\text{keys}) \end{align*} )</td>
</tr>
<tr>
<td>find(1(p))</td>
<td>(s.\text{pc}(p) = \text{pcFind1})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcSearch1} \ \text{s.}\text{ret_addr}(p) &amp; \gets \text{pcFind2} \end{align*} )</td>
</tr>
<tr>
<td>find(2T(p))</td>
<td>(s.\text{pc}(p) = \text{pcFind2}) AND (s.\text{keyf}(s.\text{ln}(p)) = s.k(p))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcFindResp}(true) \end{align*} )</td>
</tr>
<tr>
<td>find(2F(p))</td>
<td>(s.\text{pc}(p) = \text{pcFind2}) AND (s.\text{keyf}(s.\text{ln}(p)) \neq s.k(p))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcFindResp}(false) \end{align*} )</td>
</tr>
<tr>
<td>findResp((r,p))</td>
<td>(s.\text{pc}(p) = \text{pcFindResp}(r))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{idle} \end{align*} )</td>
</tr>
<tr>
<td>insertInv((k,p))</td>
<td>(s.\text{pc}(p) = \text{idle})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcInsert1} \ \text{aux}<em>{\text{seen}}</em>{\text{in}}(p) &amp; \gets (k \in \text{s.aux}_\text{keys}) \ s.k(p) &amp; \gets k \end{align*} )</td>
</tr>
<tr>
<td>insert(1(p))</td>
<td>(s.\text{pc}(p) = \text{pcInsert1})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcInsert2} \ \text{s.nNode}(p) &amp; \gets \text{newNode}(s,p,s.k(p)) \end{align*} )</td>
</tr>
<tr>
<td>insert(2(p))</td>
<td>(s.\text{pc}(p) = \text{pcInsert2})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcSearch1} \ \text{s.}\text{ret_addr}(p) &amp; \gets \text{pcInsert3} \end{align*} )</td>
</tr>
<tr>
<td>insert(14T(p))</td>
<td>(s.\text{pc}(p) = \text{pcInsert14}) AND (s.\text{lnf}(s.\text{op}(p)) = s.\text{leftf}(s.\text{pnf}(s.\text{op}(p))))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcInsert16} \ \text{s.aux}_\text{keys}\text{add}(s.k(p)) \ \text{FOR EACH } q \in \text{PROC}: \ \quad \text{IF } (s.k(q) = s.k(p) \ \quad \quad \text{AND aux}<em>{\text{in}}_\text{affected}(q,s)) \ \quad \quad \text{THEN s.aux}</em>{\text{seen}}_{\text{in}}(q) \gets true \end{align*} )</td>
</tr>
<tr>
<td>insertResp((r,p))</td>
<td>(s.\text{pc}(p) = \text{pcInsertResp}(r))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{idle} \end{align*} )</td>
</tr>
<tr>
<td>deleteInv((k,p))</td>
<td>(s.\text{pc}(p) = \text{idle})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcDelete1} \ \text{aux}<em>{\text{seen}}</em>{\text{out}}(p) &amp; \gets (k \notin \text{s.aux}_\text{keys}) \ s.k(p) &amp; \gets k \end{align*} )</td>
</tr>
<tr>
<td>delete(9(p))</td>
<td>(s.\text{pc}(p) = \text{pcDelete9})</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcDelete10} \ \text{s.op1}(p) &amp; \gets \text{newMinfo}(s,p,s.op1(p)) \end{align*} )</td>
</tr>
<tr>
<td>delete(17T(p))</td>
<td>(s.\text{pc}(p) = \text{pcDelete17}) AND (s.\text{pnf}(s.\text{op1}(p)) = s.\text{leftf}(s.\text{gpnf}(s.\text{op1}(p))))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{pcDelete19} \ \text{s.aux}_\text{keys}\text{remove}(s.k(p)) \ \text{FOR EACH } q \in \text{PROC}: \ \quad \text{IF } (s.k(q) = s.k(p) \ \quad \quad \text{AND aux}<em>{\text{out}}_\text{affected}(q,s)) \ \quad \quad \text{THEN s.aux}</em>{\text{seen}}_{\text{out}}(q) \gets true \end{align*} )</td>
</tr>
<tr>
<td>deleteResp((r,p))</td>
<td>(s.\text{pc}(p) = \text{pcDeleteResp}(r))</td>
<td>(\begin{align*} \text{s.}\text{pc}(p) &amp; \gets \text{idle} \end{align*} )</td>
</tr>
</tbody>
</table>

Table 4.6: Some transitions for the concrete automaton ConcAut, where \( s \) is a state of ConcAut.
<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>findInv(k,p)</code></td>
<td><code>s.pc(p) = idle</code></td>
<td><code>s.pc(p) ← pcDoFind(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>seen_in(p) ← (k ∈ s.keys)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>seen_out(p) ← (k ∉ s.keys)</code></td>
</tr>
<tr>
<td><code>doFindT(k,p)</code></td>
<td><code>s.pc(p) = pcDoFind(k)</code></td>
<td><code>s.pc(p) ← pcFindResp(true)</code></td>
</tr>
<tr>
<td><code>doFindF(k,p)</code></td>
<td><code>s.pc(p) = pcDoFind(k)</code></td>
<td><code>s.pc(p) ← pcFindResp(false)</code></td>
</tr>
<tr>
<td><code>findResp(r,p)</code></td>
<td><code>s.pc(p) = pcFindResp(r)</code></td>
<td><code>s.pc(p) ← idle</code></td>
</tr>
<tr>
<td><code>insertInv(k,p)</code></td>
<td><code>s.pc(p) = idle</code></td>
<td><code>s.pc(p) ← pcDoInsert(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>seen_in(p) ← (k ∈ s.keys)</code></td>
</tr>
<tr>
<td><code>doInsertT(k,p)</code></td>
<td><code>s.pc(p) = pcDoInsert(k)</code></td>
<td><code>s.pc(p) ← pcInsertResp(true)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s.keys.add(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>FOR EACH q ∈ PROC:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>IF (s.pc(q) = pcDoFind(k))</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>OR s.pc(q) = pcDoInsert(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>THEN s.seen_in(q) ← true</code></td>
</tr>
<tr>
<td><code>doInsertF(k,p)</code></td>
<td><code>s.pc(p) = pcDoInsert(k)</code></td>
<td><code>s.pc(p) ← pcInsertResp(false)</code></td>
</tr>
<tr>
<td><code>insertResp(r,p)</code></td>
<td><code>s.pc(p) = pcInsertResp(r)</code></td>
<td><code>s.pc(p) ← idle</code></td>
</tr>
<tr>
<td><code>deleteInv(k,p)</code></td>
<td><code>s.pc(p) = idle</code></td>
<td><code>s.pc(p) ← pcDoDelete(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>seen_out(p) ← (k ∉ s.keys)</code></td>
</tr>
<tr>
<td><code>doDeleteT(k,p)</code></td>
<td><code>s.pc(p) = pcDoDelete(k)</code></td>
<td><code>s.pc(p) ← pcDeleteResp(true)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>s.keys.remove(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>FOR EACH q:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>IF (s.pc(q) = pcDoFind(k))</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>OR s.pc(q) = pcDoDelete(k)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>THEN s.seen_out(q) ← true</code></td>
</tr>
<tr>
<td><code>doDeleteF(k,p)</code></td>
<td><code>s.pc(p) = pcDoDelete(k)</code></td>
<td><code>s.pc(p) ← pcDeleteResp(false)</code></td>
</tr>
<tr>
<td><code>deleteResp(r,p)</code></td>
<td><code>s.pc(p) = pcDeleteResp(r)</code></td>
<td><code>s.pc(p) ← idle</code></td>
</tr>
</tbody>
</table>

Table 4.7: Transitions for the intermediate automaton `IntAut`, where `s` is the variable of TYPE `state` of `IntAut`.  

93
### Table 4.8: Internal action correspondence between ConcAut and IntAut.

<table>
<thead>
<tr>
<th>Actions in ConcAut starting from $c$</th>
<th>Actions in IntAut</th>
</tr>
</thead>
<tbody>
<tr>
<td>$find2T(p)$</td>
<td>$doFindT(c.k(p), p)$</td>
</tr>
<tr>
<td>$find2F(p)$</td>
<td>$doFindF(c.k(p), p)$</td>
</tr>
<tr>
<td>$insert4T(p)$</td>
<td>$doInsertF(c.k(p), p)$</td>
</tr>
<tr>
<td>$insert14T(p)$</td>
<td>$doInsertT(c.k(p), p)$</td>
</tr>
<tr>
<td>$insert15T(p)$</td>
<td></td>
</tr>
<tr>
<td>$delete3T(p)$</td>
<td>$doDeleteF(c.k(p), p)$</td>
</tr>
<tr>
<td>$delete17T(p)$</td>
<td>$doDeleteT(c.k(p), p)$</td>
</tr>
<tr>
<td>$delete18T(p)$</td>
<td></td>
</tr>
<tr>
<td>other actions</td>
<td>$\epsilon$</td>
</tr>
</tbody>
</table>
fsr_{pc\_rel\_find}(c, i, p) \equiv \left( \begin{array}{l}
i.pc(p) = pcDoFind(c.k_p) \\
\quad \text{AND } c.pc(p) \in \{pcFind1, pcFind2\} \\
\text{OR } \left( \begin{array}{l}
i.pc(p) = pcFindResp(r) \\
\quad \text{AND } c.pc(p) = pcFindResp(r) \end{array} \right) \end{array} \right)

fsr_{pc\_rel\_insert}(c, i, p) \equiv \left( \begin{array}{l}
i.pc(p) = pcDoInsert(c.k_p) \\
\quad \text{AND } c.pc(p) \in \{pcInsert1, \ldots, pcInsert15\} \\
\text{OR } \left( \begin{array}{l}
i.pc(p) = pcInsertResp(true) \\
\quad \text{AND } c.pc(p) \in \{pcInsert16, pcInsert17\} \end{array} \right) \\
\text{OR } \left( \begin{array}{l}
i.pc(p) = pcInsertResp(r) \\
\quad \text{AND } c.pc(p) = pcInsertResp(r) \end{array} \right) \right)

fsr_{pc\_rel\_delete}(c, i, p) \equiv \left( \begin{array}{l}
i.pc(p) = pcDoDelete(c.k_p) \\
\quad \text{AND } c.pc(p) \in \{pcDelete1, \ldots, pcDelete18, pcDelete20\} \\
\text{OR } \left( \begin{array}{l}
i.pc(p) = pcDeleteResp(true) \\
\quad \text{AND } c.pc(p) = pcDelete19 \end{array} \right) \\
\text{OR } \left( \begin{array}{l}
i.pc(p) = pcDeleteResp(r) \\
\quad \text{AND } c.pc(p) = pcDeleteResp(r) \end{array} \right) \right)

Figure 4.12: Program counter relation of \textit{fsr} when \textit{p} is performing a \textit{find}, \textit{insert} or \textit{delete} operation.
Figure 4.13: Program counter relation during the $insert_p(k)$ operation in the forward simulation relation.

Figure 4.14: Program counter relation during the $delete_p(k)$ operation in the forward simulation relation.
Figure 4.15: Program counter relation during the $search_p(k)$ subroutine in the forward simulation relation.
5 Invariants and Proofs

5.1 An Overview of the Proof

In Chapter 4, we described the way to model the specification and the simplified algorithm using a canonical automaton \( AbsAut \) and a concrete automaton \( ConcAut \), respectively. Because it is complicated to construct a forward simulation directly from \( ConcAut \) to \( AbsAut \), we introduced the intermediate automaton \( IntAut \) and proved that \( ConcAut \) implements \( IntAut \) and \( IntAut \) implements \( AbsAut \) through forward and backward simulations, respectively. This hybrid forward and backward simulation implies that \( ConcAut \) implements \( AbsAut \), and hence our simplified algorithm satisfies its specification.

We have already defined the forward simulation \( fsr \) and the backward simulation \( bsr \) in Chapter 4. For the forward simulation (Definition 2.19), we have to prove three main properties.

1. For every \( c \in \text{start}(ConcAut) \), there exists an \( i \in \text{start}(IntAut) \), such that \( (c, i) \in fsr \).
2. If $c \xrightarrow{\alpha_{ConcAut}} c'$ and $(c, i) \in fsr$, then there exists $\hat{\alpha}$ and $i'$ such that $i \xrightarrow{\hat{\alpha}_{IntAut}} i'$ and $(c', i') \in fsr$, and

3. the external action in $\hat{\alpha}$ is the same as the external action in $\alpha$.

The first condition requires that every initial state (in Section 4.2) in $ConcAut$ has a matching initial state in $IntAut$. Because $IntAut$ has a unique initial state (defined in Section 4.3), it was trivial to check that every initial state in $ConcAut$ is related with the initial state in $IntAut$ by $fsr$ (defined in Section 4.4). The second property was proved by case analysis of all actions in $ConcAut$, using the action correspondence shown in Table 4.8. In most cases proving the second property was not complicated. The exceptions were the actions in $ConcAut$ that map to non-nil actions of $IntAut$. These are, by far, the bulkiest part of the proof because they required proving many auxiliary lemmas about how the concrete automaton behaves. The last condition was straightforward to verify according to the action correspondence defined in Table 4.8.

Likewise, using Definition 2.22, which defines a backward simulation between $IntAut$ and $AbsAut$, we proved that the following four properties of the backward simulation hold.

1. For all $i \in states(IntAut)$, there exists $a \in AbsAut$ such that $(i, a) \in bsr$.

2. If $i \in start(IntAut)$ and there exists $a \in states(AbsAut)$ such that $(i, a) \in bsr$, then $a \in start(AbsAut)$.
3. If \( i' \xrightarrow{\alpha} \text{IntAut} i \) and \( a \in \text{states}(\text{AbsAut}) \) such that \((i, a) \in \text{bsr}\), then there exist \\
a' and \( \hat{\alpha} \) such that \((i', a') \in \text{bsr}\) and \( a' \xrightarrow{\hat{\alpha}} \text{AbsAut} a \), and \\
4. the external action in \( \hat{\alpha} \) is the same as the external action in \( \alpha \).

The first property was proved by explicitly constructing a state of \( \text{AbsAut} \) from the 
state of \( \text{IntAut} \). We proved the second property for the unique initial states for both 
\( \text{IntAut} \) and \( \text{AbsAut} \). Once again, the third property was proved by enumerating 
all actions in \( \text{IntAut} \) using the action correspondence defined in the last part of 
Section 4.3. The difficult cases are \( \text{doInsertT}(k, p) \) and \( \text{doDeleteT}(k, p) \), whose 
corresponding actions in \( \text{AbsAut} \) may consist of several internal actions of other 
processes. We shall discuss that part of backward simulation proof in Section 5.5. 
After showing the third property, the last one is easy to prove using the action 
correspondence.

5.2 Proofs in the Forward Simulation

To complete the proof of the forward simulation defined in Chapter 4 between 
\( \text{ConcAut} \) and \( \text{IntAut} \), we must show the key actions in Table 4.8 satisfy the following 
property.

If \( c \xrightarrow{\alpha} \text{ConcAut} c' \) and \((c, i) \in \text{fsr}\), then there exist \( \hat{\alpha} \) and \( i' \) such that \( i \xrightarrow{\hat{\alpha}} \text{IntAut} \\
i' \) and \((c', i') \in \text{fsr}\)
As we described in the overview section, given states $c, c' \in \text{states(ConcAut)}$ and $i' \in \text{states(IntAut)}$ and an action $\alpha$ such that $c \xrightarrow{\alpha}_{\text{ConcAut}} c'$ and $(c, i) \in \text{fsr}$, we can explicitly construct $\hat{\alpha}$ using the action correspondence between $\text{ConcAut}$ and $\text{IntAut}$ defined in Table 4.8. Hence, it remains to construct an $i' \in \text{states(IntAut)}$ which satisfies $i \xrightarrow{\alpha}_{\text{IntAut}} i'$ and $(c', i') \in \text{fsr}$.

Firstly, we prove that $i \xrightarrow{\alpha}_{\text{IntAut}} i'$ after constructing $i'$ using $\hat{\alpha}$. For each action in $\text{ConcAut}$, we need to prove pre-state $i$ enables $\hat{\alpha}$. A state $i$ enables an action if the value of the program counter and the data values of $i$ satisfy the precondition of the action defined in Table 4.7. Hence, we mainly focus on the cases which map to non-trivial actions in Table 4.8. If the action is a $\text{find2T}(p)$, to show there exists a $\text{doFindT}(c.k(p), p)$ action starting from $i$ in $\text{IntAut}$, we need to argue $i.\text{seen_in}(p)$ is true. This can be proved by showing $c.\text{auxseen_in}(p)$ is true since $i.\text{seen_in}(p) = c.\text{auxseen_in}(p)$ from $(c, i) \in \text{fsr}$. By the program counter relation part of $\text{fsr}$, because $c.\text{pc}(p) = \text{pcFind2}$, we have $i.\text{pc}(p) = \text{pcDoFind}(k)$. Therefore, preconditions of a $\text{doFindT}(k, p)$ action are satisfied in $i$. Symmetrically, we can prove that $i$ satisfies the preconditions of a $\text{doFindF}(c.k(p), p)$ action in $\text{IntAut}$ when a $\text{find2F}(p)$ is performed in $\text{ConcAut}$. Hence, Lemma 5.1 is needed when a $\text{find2T}(p)$ or $\text{find2F}(p)$ action occurs in $\text{ConcAut}$.

**Lemma 5.1.** Let $c$ be any reachable state of $\text{ConcAut}$. If $c \xrightarrow{\text{find2T}(p)} c'$, then the value of $\text{auxseen_in}_p$ is true at the state $c$. If $c \xrightarrow{\text{find2F}(p)} c'$ then the value of $\text{auxseen_out}_p$
is true at the state $c$.

A similar argument is applied to $insert 4T(p)$ and $delete 3T(p)$, as well. When a successful $ichild$ CAS is performed, by an $insert 14T(p)$ or $insert 15T(p)$ action, let $k = c.k(p)$. We have to argue that $k \notin i.keys$ and $i.pc(p) = pcDoInsert(k)$ before the action $doInsertT(k,p)$ is taken in $IntAut$ according to Table 4.7. Since $i.keys = c.aux.keys$ follows from $(c,i) \in fsr$, we just have to prove $k \notin c.aux.keys$. Once again, because $c.pc(p) = pcInsert14$ or $c.pc(p) = pcInsert15$ before $insert 14T(p)$ or $insert 15T(p)$, respectively, $i.pc(p) = pcDoInsert(k)$ according to the program counter relation of $(c,i) \in fsr$. Similarly, when a successful $dchild$ CAS is performed by a $delete 17T(p)$ or $delete 18T(p)$ action, let $k = c.k(p)$. We have to argue that $k \in c.aux.keys$ and $i.pc(p) = pcDoDelete(k)$. Hence, we need to prove Lemma 5.2 and 5.3 for the $ichild$ and $dchild$ steps in $ConcAut$, respectively.

**Lemma 5.2.** Let $c$ be any reachable state of $ConcAut$. If $c \xrightarrow{insert 14T(p)} c'$ then the value of $aux.seen.in_p$ is true at the state $c$. If $c \xrightarrow{insert 14T(p)} c'$ or $c \xrightarrow{insert 15T(p)} c'$, then $c.k(p)$ is not in $aux.keys$ at the state $c$.

**Lemma 5.3.** Let $c$ be any reachable state of $ConcAut$. If $c \xrightarrow{delete 3T(p)} c'$, then the value of $aux.seen.out_p$ is true at the state $c$. If $c \xrightarrow{delete 17T(p)} c'$ or $c \xrightarrow{delete 18T(p)} c'$, then $c.k(p)$ is in $aux.keys$ at the state $c$.

Secondly, we need to show that after taking $\hat{\alpha}$ from $i$, the resulting state $i'$ satisfies $(c',i') \in fsr$. Thus, for each action in $ConcAut$, we need to prove the data
relation is satisfied between $c'$ and $i'$, as well as the program counter relation part of $fsr$. We again focus on the cases in Table 4.8, which are the most complicated ones.

With respect to the data relation between $c'.aux\_keys$ and $i'.keys$, we know that $aux\_keys$ changes only if a successful child CAS is performed. The key $k$ is added to $aux\_keys$ when a successful ichild CAS by process $p$ that inserts key $k$ occurs. Hence, in $IntAut$, a $doInsertT(k, p)$ action that adds $k$ to $i.\text{keys}$ is performed. Because $c.aux\_keys = i.\text{keys}$, we have $c'.aux\_keys = i'.\text{keys}$. Symmetrically, $c'.aux\_keys = i'.\text{keys}$ holds if a successful dchild CAS that deletes $k$ occurs.

To prove the set stored in the BST is the same as the set of keys in $IntAut$, we have to ensure that $aux\_keys$ is equal to the set of keys in the BST’s reachable leaves in $ConcAut$. This is Invariant 1, which encapsulates the connection between the key set and its representation in shared memory as a BST. Hence, the complex structure of the BST in the concrete automaton is hidden by this auxiliary key set variable.

**Invariant 1.** The set $aux\_keys$ in $ConcAut$ always contains the same keys as the current reachable leaves in the tree starting from the Root node.

With respect to the data relation between $c'.aux\_seen\_in$ and $i'.\text{seen\_in}$, and between $c'.aux\_seen\_out$ and $i'.\text{seen\_out}$, these parts of the states are initialized at each invocation of each operation and modified only during a successful child CAS. If $\alpha$ is an invocation by $p$, because $c.aux\_keys = i.\text{keys}$, $c'.aux\_seen\_in(p)$ is initialized to
the same value as $i'.\text{seen\_in}(p)$, as are $c'.\text{aux\_seen\_out}(p)$ and $i'.\text{seen\_out}(p)$. When a successful $ichild$ CAS by process $p$ for key $k$ is performed, $\hat{\alpha} = \text{doInsertT}(k, p)$, and $\text{aux\_seen\_in}(q)$ of every process $q$ that is performing a $find(k)$ or an $insert(k)$ operation but has not yet decided to return $true$ will be set to $true$ in the post state $c'$. Hence, we also need to show that $i'.\text{seen\_in}(q)$ is $true$ in order to prove $c'.\text{aux\_seen\_in} = i'.\text{seen\_in}$, which is required for showing $(c',i') \in \text{fsr}$. Since that process $q$ is performing a $find(k)$ or an $insert(k)$ in $\text{ConcAut}$, it follows that $q$'s program counter value is $\text{pcDoFind}(k, q)$ or $\text{pcDoInsert}(k, q)$ at state $i$ in $\text{IntAut}$ due to the program counter relation of $(c, i) \in \text{fsr}$. Thus, according to the way that $\text{IntAut}$ updates variables in Table 4.7, $i'.\text{seen\_in}(q)$ is set to $true$ by $\hat{\alpha}$ as well. In a symmetric way, when a successful $dchild$ CAS occurs, we can show $\text{aux\_seen\_out}$ in $\text{ConcAut}$ is also related to $\text{seen\_out}$ in $\text{IntAut}$.

To prove the program counter relation holds between $c'$ and $i'$, we expand the effects of $\hat{\alpha}$ and show that after $\hat{\alpha}$, the program counter values of $c'$ and $i'$ are still related. There is one type of special case in proving the program counter relation between $c'$ and $i'$. For example, when a failed $dchild$ CAS ($\text{delete17F}(p)$) is performed, we know that $\hat{\alpha} = \epsilon$, and $(c, i) \in \text{fsr}$. However, we cannot relate $c'$ to $i'$, because $c.pc(p) = \text{pcDelete19}$ which is about to return, however, $i'.pc(p) = i.pc(p) = \text{pcDoDelete}(k)$. Therefore, we have to show $\text{delete17F}(p)$ cannot occur. This is because no helping mechanism is implemented, so it is impossible for a delete
operation that successfully marks a parent node to fail on the dchild CAS or for an insert operation that successfully flags a parent node to fail on the ichild CAS. This is formalized in the following Lemma 5.4.

**Lemma 5.4.** For any execution of ConcAut, if a process $p$ successfully performs a mark CAS ($\text{delete}10T(p)$), it cannot perform an unsuccessful dchild CAS ($\text{delete}17F(p)$ or $\text{delete}18F(p)$ in the same iteration of the loop). Similarly, if $p$ successfully performs an iflag CAS ($\text{insert}11T(p)$), it cannot perform an unsuccessful ichild CAS ($\text{insert}14F(p)$ or $\text{insert}15F(p)$ in the same iteration of the loop).

Combined with the auxiliary variables $\text{aux}_\text{seen}_\text{in}(p)$ and $\text{aux}_\text{seen}_\text{out}(p)$, Lemma 5.4 corresponds to the Lemma 5.5 and 5.6 and 5.7 reproduced below, in the original English proof in the tech report [3].

**Lemma 5.5.** If a Find($k$) operation returns true, then the BST contains a leaf with key $k$ at some point between the beginning and end of the operation. If it returns false, there exist a time between the beginning and end of the operation such that the BST does not contain a leaf with key $k$.

**Lemma 5.6.** An Insert($k$) operation returns true if and only if the BST does not contain a leaf with key $k$ just before it performs the ichild CAS. If the operation returns false, there exist a time between the beginning and end of the operation such that the BST contains a leaf with key $k$. 


Lemma 5.7. A Delete($k$) operation returns true if and only if the BST does contain a leaf with key $k$ just before it performs the dchild CAS. If the operation returns false, there exist a time between the beginning and end of the operation such that the BST does not contain a leaf with key $k$.

These lemmas, which require another 25 technical lemmas, were all proved in English in the original paper [3]. Thus, we formalized their proofs in PVS to complete the forward simulation. For example, we needed to prove one of the most important lemmas which claims that in any reachable state, the data structure maintained by the implementation is a BST, shown as Lemma 5.8.

Lemma 5.8. In every reachable state, the tree of child pointers is a BST [3].

This is also one of the key lemmas proved in the original paper ([3], Lemma 22).

We encountered some difficulties in formalizing lemmas written in [3] and formally proving them using PVS. An important difference between proofs written in natural language and machine checkable proofs is that a small step in the natural language proof in a human’s mind is often not a straightforward automatic step in PVS. PVS provides some proof commands, such as grind, to automatically reason towards a goal. However those procedures, which try repeated skolemization, instantiation, and if-lifting, are not intelligent enough to prove complex goals, especially when some complicated data structures are involved. In proving the lemmas and invariants of the BST algorithms in PVS, one must be very careful of using such
commands: if one uses the *grind* command carelessly, PVS automatically expands the definition of *ConcAut* into many cases and complicated expressions, which are very hard to work with. To avoid this, we have to explicitly state those “small” steps for humans as lemmas such that we can apply them when proving a higher level statement. Therefore, our PVS script to state the invariants of *ConcAut* contains many lemmas which may seem fairly trivial.

For instance, Lemma 4 in [3] states that for each internal node $v$, no CAS ever changes $v$.info to a value that was previously stored there. In the proof, the authors state “Each successful flag CAS on $v$.info subfield sets the filed to point to a newly created Info object, so that this object could never have appeared in $v$.info before.” This is easy to verify for a human: a successful iflag CAS (Line I11) is always preceded by a creation of a new Info object (Line I10). We may use another implicit fact that no other processes can write or modify the object between Line I10 and I11 because the only pointer to it is in a local variable of the process that created it. Therefore, it has never been visible to other processes before the successful iflag CAS. However, to verify that sentence using PVS, we have to split it into three small lemmas as follows.

First, we show that an Info object newly created on Line I10 has never appeared in $v$.info before, as claimed by Lemma [5.9] The definition of executions in PVS are discussed in Section [5.3].
**Lemma 5.9.** For any execution, if an Info object $f$ is newly created by I10 at step $i$, any Info object accessed by any process before step $i$ is not the same as $f$.

This lemma can be proved by a contradiction. Assume that before step $i$, a process accessed an Info object that is the same as $f$. By applying Lemma 5.10 and Lemma 5.11 which claim that in any previous state, if $x$ was an allocated Info object, it is still in the set $allocatedInfo$, we know that $f$ is in $allocatedInfo$. By applying the axiom that describes the creation a new Info object, the object that I10 allocated must not be in $allocatedInfo$. Thus, it completes the proof.

**Lemma 5.10.** For any execution, if a process accesses an Info object, this object has been allocated.

**Lemma 5.11.** For any execution, if $f$ was in $allocatedInfo$, it is still in $allocatedInfo$.

Second, Lemma 5.12 says that if process $p$ creates an Info object $f$ at I10, before $p$ successfully performs its iflag CAS (I11) that writes a pointer to $f$ into a node $v$, no other process can access $f$. This lemma can be proved by induction on the length of the execution.

**Lemma 5.12.** For any execution and any node $v$, if an Info object $f$ is newly created by I10 by process $p$ at step $j$, and $p$ points $v.info$ to $f$ by a successful iflag CAS at step $i$, then no other process can access $f$ between steps $j$ and $i.$
Third, we use Lemma \ref{lemma:5.13} to state that for any successful iflag \texttt{CAS} performed by process $p$, there exists a previous execution of Line I10 by $p$ such that no step is taken by $p$ in between. Lemma \ref{lemma:5.13} can be observed from the pseudocode. Hence, no process other than $p$ can access $f$ before the successful iflag \texttt{CAS}. It follows that no $v.info$ can be set by a \texttt{CAS} operation to point to $f$ by any process other than $p$. Combined with Lemma \ref{lemma:5.9} we can prove the single sentence written by the authors.

\textbf{Lemma 5.13.} \textit{For any execution and any node $v$, if process $p$ successfully changes $v.info$ by an iflag \texttt{CAS} to $f$ at step $i$, there exists a step $j$ such that $j < i$ and $f$ is created by I10 performed by $p$ and no step belongs to $p$ in between $j$ and $i$.}

Another kind of difficulty that appears when proving invariants of \textit{ConcAut} using PVS is what we call code structure problems. One may get a flavour of this problem by considering Lemma \ref{lemma:5.13}. Since we model the pseudocode as an I/O automaton, each line of the code becomes an independent action. In the pseudocode, one can easily observe how the code is executed line by line. But when we model it as an I/O automaton, proving something that relies on properties of the code structure, requires reasoning about many independent actions, which are tied together via their effects on the program counter. More specifically, if we know that process $p$ is executing Line I13, we can easily conclude that the IF condition executed by $p$ at Line I5 returns true, by looking at the pseudocode. However, in PVS, as the steps are modelled as independent actions, to prove the same statement, we have to...
state it as a lemma and infer step by step from Line I13 back to Line I5 and then conclude that if \( p \) executes Line I13, there exists an earlier step when \( p \) executed I5 and returned true. More generally, whenever one of this kind of situation occurs, we have to come up with an individual lemma to state it. This is quite inefficient and thus we want to build the kind of general tools for proving this kind of facts about an automaton that models code.

5.3 Some Definitions in PVS

Suppose \( c \xrightarrow{\alpha} c' \), where \( c \) and \( c' \) are states and \( \alpha \) is an action in ConcAut. As we can see, besides describing properties of \( c \) and \( c' \), we also want to reason about the previous actions before \( \alpha \) or some possible actions after \( \alpha \) in an execution. Many proofs of lemmas in the original paper use that kind of reasoning ([3], Lemma 4, 5, 6, 7, 9, 10, 11, 12, 13, 14, etc.). Our first attempt to formalize a reachable state by an execution sequence starting from an initial state, adapted from [16], was inductive:

\[
\text{reachable?}(c): \text{INDUCTIVE bool =}
\]

\[
\text{initial.config}(c) \lor (\exists s, \alpha: \text{reachable?}(s) \land \text{transition}(s, \alpha, c))
\]

This says that a state \( c \) is reachable if it is an initial state or there is another state \( s \) which is reachable and there is a transition from \( s \) to \( c \) by performing an action \( \alpha \). This definition, although clear and simple, makes it awkward to reason about the actions before a given action or the actions that occur after it. Inspired
by [28], we use a more natural way to define states and executions. We define a finite execution to be a sequence of \( n + 1 \) states and \( n \) actions alternating with each other.

We define a \textit{FiniteStepSeq} in PVS to be a type that consists of two finite sequences (finseq), one containing the actions and the other containing the states, where the length of the state-sequence is one larger than the length of the action-sequence. In PVS, the pair “[#” and “#]” represents a definition of a record type with several attributes. Its attributes, which consist of a name followed by a type, are separated by “,”. The finseq type, provided by PVS, is a function that maps each natural number that is smaller than the length to an element of a generic type \( T \).

\[
\text{finseq: TYPE = [# length: nat, seq: [ below[length] \to T ] #]}
\]

\[
\text{FiniteStepSeq: TYPE = [# actions: finseq[action],}
\]

\[
\text{states:{ss: finseq[state] \mid ss.length = actions.length+1 } #]}
\]

If \textit{stepseq} is a variable of type \textit{FiniteStepSeq}, we define the function \textit{steps}, which takes a \textit{FiniteStepSeq} \textit{stepseq} as its argument and returns a finite sequence of \( \langle \text{state, action, state} \rangle \) tuples, \textit{i.e.}, transitions.

111
stepseq: VAR FiniteStepSeq

steps(stepseq): finseq[ [state, action, state] ] =

(# length := stepseq.actions.length,

  seq := λ(n:below[stepseq.actions.length]):

    (stepseq.states(n), stepseq.actions(n), stepseq.states(n+1))
#

)

A stepseq is a finiteExecFrag of an I/O automaton if and only if every tuple in steps(stepseq) is a legal transition of the I/O automaton. Furthermore, a finite-ExecFrag stepseq is a finiteExecution if and only if the first state of stepseq is an initial state.

finiteExecFrag(stepseq): bool =

  ∀ (n: below[stepseq.actions.length]): transition(steps(stepseq)(n))

finiteExecution(stepseq): bool =

  finiteExecFrag(stepseq) ∧ initial_config(stepseq.state(0))

Hence, if c α→ c' is the ith transition in a finiteExecution stepseq, we can easily reason about properties of any actions or states in the execution before ith step or after ith by referring to their indices. For example, Lemma 5.15 (3, Lemma 9) can now be formalized using our definitions about finiteExecutions as shown in Figure 5.1, given the definition of “belong to” as follows.
**Definition 5.14.** A successful flag $CAS$ belongs to an Info object $f$, if the flag $CAS$ stores a pointer to an Info object $f$. A mark $CAS$ belongs to an Info object $f$, if the $dinfof$ field of the Info object used by the $CAS$ points to $f$.

**Lemma 5.15.** Each mark $CAS$ that belongs to an Info object $f$ is preceded by a successful dflag $CAS$ that belongs to $f$.

Let $c \xrightarrow{\alpha} c'$. A frequently used lemma that requires proof in PVS is that every local variable of process $p$ remains unchanged between $c$ and $c'$ if $\alpha$ belongs to a process $q \neq p$. When we prove an invariant, we always have to prove it is preserved by all possible actions $\alpha$. There are 79 possible actions in $ConcAut$. However, except for a few important actions, most of the actions can be proved to preserve an invariant by using the same proof steps. Therefore, we usually construct PVS proof strategies, which consist of a batch of PVS proof commands, to automate the proofs when we need to enumerate all the actions.

### 5.4 Errors Found

Although proving the correctness of invariants and lemmas using PVS took a long time, we did detect some errors in the original proof. An author of [3] detected that the proof of Lemma 5.16 ([3], Lemma 2(10)) has a small error.

**Lemma 5.16.** The top part of the tree is always as shown in Figure 5.2. More precisely:
\text{lemma_dflag_before_mark}: \text{LEMMA } \forall \text{ stepseq}: \text{finiteExecution}(\text{stepseq}) \Rightarrow \\
\left( \forall i \leq \text{stepseq}.\text{actions}.\text{length} : \text{LET } \alpha = \text{stepseq}.\text{actions}(i), \right. \\
p = \text{process}(\text{stepseq}.\text{actions}(i)), \\
c = \text{stepseq}.\text{states}(i), \\
f = c.\text{dinfof}(c.\text{op2}(p)) \text{ IN} \\
\left( \text{markCAS}(\alpha) \Rightarrow \\
\exists j: \text{nat} : j \leq i \land \text{success_dflagCAS_belong_f}(\text{stepseq}, j, f) \right. \\
\right) \\
\right)

\text{markCAS}(\alpha): \text{bool} = (\alpha = \text{delete10T OR } \alpha = \text{delete10F})

\text{success_dflagCAS_belong_f}(\text{stepseq}, i, f): \text{bool} = \text{LET } s = \text{stepseq}.\text{states}(i), \\
b = \text{LET } b = \text{stepseq}.\text{actions}(i) \text{ IN} \\
(b = \text{delete7T AND } s.\text{op1}(\text{process}(b)) = f)

Figure 5.1: Using definition of finiteExecution to formalize a lemma.
(a) \( \text{Root.left.key} = \infty_1 \), and
(b) if \( \text{Root.left} \) is an internal node, then \( \text{Root.left.right} \) is a leaf with key \( \infty_1 \).

Their proof was done by induction on states in an execution. It is trivial that the lemma holds for the base case, where the state is the initial state. However, for the induction step, let \( c \xrightarrow{\alpha} c' \) and assume the claim holds in \( c \). For the case where \( \alpha \) is a \( \text{dchild CAS} \) that changes the node \( \text{root.left} \) using some \( \text{Info} \) object \( f \), they argued that after the \( \text{dchild CAS} \), \( \text{root.left} \) is a leaf with key \( \infty_1 \), because \( f.pn.right \) is a leaf with key \( \infty_1 \). Because the \( \text{dchild CAS} \) is successful, \( \text{root.left} \) points to \( f.pn \) in state \( c \). They also have a lemma that proves that for any \( \text{DInfo} \) object \( f \), \( f.pn \) is an internal node. They claimed that, it follows from the induction hypothesis that \( f.pn.right \) is a leaf with key \( \infty_1 \).

![Trees showing leaves when the set is (a) empty and (b) non-empty.](image)

This is incorrect. We agree that \( \text{root.left} \) is an internal node at \( c \), and thus we can apply the hypothesis to show \( \text{root.left.right} \) is a leaf with key \( \infty_1 \) in state...
c. However, the dchild CAS writes the value stored in other to root.left, and the value of other was assigned at a step β before α. In the state immediately before β, we do not know if f.pn.right at this point is a leaf with ∞₁, since we do not know that f.pn is the left child of the root at that time. One way to fix the proof of this lemma is to use lemmas proved subsequently, which say that if a node is flagged or marked then no other process can modify its child pointers, and before a successful dchild CAS, the node is marked. However, in order to do so, all those lemmas have to be composed into a big induction lemma, which is a bit complicated. Instead, we fixed this lemma by making it a bit weaker as stated in Lemma 5.17.

**Lemma 5.17.** The node root.left is always a node with key ∞₁.

This weaker lemma turns out to be sufficient to be used in the later proofs, since we can still conclude that root.left.right has a key greater than or equal to ∞₁ by combining a few lemmas.

In the process of formalizing the proof, I discovered one flaw in the original proof in Lemma 5.18 ([3], Lemma 14(7)). Note that Lemma 14 in the original proof is a big induction lemma which has many parts and we mainly discuss the seventh part of it.

**Lemma 5.18.** A child (either an ichild or dchild) CAS writes a value into a node v’s child pointer that has never been stored there before.

We can focus on the case of a dchild CAS dcasₖ that changes the left child of
$x$ from a node $z$ to $y$. In the original proof, to derive a contradiction, the authors assume that $y$ was the left child of $x$ at some earlier time. Figure 5.3 illustrates the execution for the proof. In the state just before $dcas_k$, $z$’s child is $y$. Because we know $y \neq z$, there must be an earlier child CAS $ccas_j$ that caused $y$ to stop being the left child of $x$. They proved that just after $ccas_j$, $y$ is not a descendant of $x$. The case where $ccas_j$ is an ichild CAS is fine. So, we only consider the case where $ccas_j$ is a $dchild$ CAS as shown in Figure 5.3. According to another part of the induction hypothesis “before a $dchild$ CAS, the child pointers of the parent node $f.pn$ do not change between the last read in search belong to $f$ and the $dchild$ CAS”, so $ccas_j$ replaces a pointer to $y$ by a pointer to $y$’s child, and $y$ is no longer a descendant of $x$ (since $y$ cannot be a descendant of its own child by “the binary tree property” (another part of the induction hypothesis)). This is incorrect, because “the binary tree property” can only be applied here when $y$ is reachable. However, there is no proof to show node $x$ or $y$ is reachable before the $dchild$ CAS.

We fix this lemma by adding more auxiliary claims into the original Lemma 14.
to compose a bigger induction lemma. The added claims are stated in Lemma 5.19. Some of these claims were proved in [3], but were not wrapped up in the induction proof used to prove Lemma 14 in [3].

**Lemma 5.19.**  1. After a successful dchild CAS by a process $p$, the node that was marked by $p$ before the dchild CAS and was reachable right before the CAS, becomes unreachable and will never become reachable again.

2. If a successful child (either an ichild or dchild) CAS occurs on a node $v$, $v$ is reachable in the state right before the CAS.

3. During a search subroutine of process $p$, each visited node was reachable at a time before it is visited by $p$.

4. The node which is unreachable and becomes reachable by an ichild CAS was never reachable before.

5. If a node is reachable after any action other than a successful ichild CAS then it was reachable before the action as well.

Intuitively, Lemma 5.19(1) can be proved by two cases. In case 1, if a node is added by a successful ichild CAS we prove this node is not the marked node by contradiction by applying Lemma 5.19(4). In the other case, when a node becomes the new child of its grandparent by a dchild CAS we prove this node is not the marked node by contradiction by applying Lemma 5.19(5). Lemma 5.19(2) and
Lemma 5.19(3) are proved by using the induction hypothesis of each other. Furthermore, Lemma 5.19(4) can be proved using the fact that an ichild CAS always changes a pointer to a newly allocated node. Lemma 5.19(5) can be proved by applying Lemma 5.19(2) plus the fact that a dchild CAS changes the child pointer of $f.gpn$ from $f.pn$ to $f.other$. Thus, we can use Lemma 5.19(1) to show the contradiction that an unreachable node $y$ becomes reachable in the execution shown in Figure 5.3, thereby correcting the flaw in the original proof of Lemma 14(7).

### 5.5 Proofs in the Backward Simulation

It is easier to prove the correctness of the backward simulation compared with the forward one, because our intention was to design the $IntAut$ to be as similar to the $AbsAut$ as possible. As discussed earlier, in Section 5.1, we were required to show that for each type of $\alpha$ such that $i' \xrightarrow{\alpha} IntAut i$ and each $a \in states(AbsAut)$ such that $(i, a) \in bsr$, there exists a state $a'$ of $AbsAut$ and a sequence of actions $\hat{\alpha}$ such that $(i', a') \in bsr$ and $a' \xrightarrow{\hat{\alpha}} AbsAut a$, and the external action in $\hat{\alpha}$ is the same as the external action in $\alpha$.

Recall the backward simulation relation $bsr$ and backward action correspondence defined in Section 4.3. It is trivial to prove that external actions, which are invocations and responses, satisfy the above properties. It is also not hard to prove that internal actions, except for $doInsertT(k, p)$ and $doDeleteT(k, p)$, satisfy this prop-
erty, because they never modify shared objects which appear in the data relation part of $bsr$. A $doInsertT(k, p)$ action has a more complicated behaviour. It adds $k$ into $i'.keys$, sets $i'.seen\_in(q)$ to be $true$ for any process $q$ which is performing a $find(k)$ or an insert($k$) operation but has not decided to return a value, which allows us to linearize all $find(k)$ operations that subsequently return $true$ and all insert($k$) operations that subsequently return $false$ immediately after $doInsertT(k, p)$. For this action $\alpha$, we need to construct the pre-state $a'$ from $a$ by removing $k$ from $a.keys$. The value of program counter $a'.pc(q)$ for a process $q$ is retrieved by setting its values to the precondition of $q$‘s action in $\hat{\alpha}$. Then, we show that $(i', a') \in bsr$ and $a' \xrightarrow{AbsAut} a$ and the external action in $\hat{\alpha}$ and $\alpha$ is the same. Because a $doDeleteT(k, p)$ action behaves in a symmetric way as $doInsertT(k, p)$, we used a similar method to construct $\hat{\alpha}$ and $a'$ to complete the proof. Those proofs can all be found in our PVS scripts.
6 Conclusion

We believe that forward simulations are highly related to a concept called strong linearizability recently defined by Golab et al. [29]. We conjecture that an implementation is strongly linearizable if and only if there exists a forward simulation between the implementation and its sequential specification. Because we believe the BST algorithm is actually strongly linearizable, we believe that a forward simulation exists between the implementation and its sequential specification. Therefore, one may be tempted to try to prove that the concrete automaton (the implementation) implements the canonical automaton (specification) directly by a forward simulation. However, that relation is much more complicated to formalize. Even when we split the proof into a backward and forward simulation, the forward simulation is still complicated to be proved using PVS, because the pseudocode of the algorithm is far from trivial, the concrete automaton is complicated, and the program counter relationship defined in Figure 4.10 consists of a lot of possibilities. When we were proving a lemma about the concrete automaton, it was almost impossible to use PVS’s built-in automated reasoning procedures such as “grind” to save time, be-
cause PVS “got lost” in those automatic generated subgoals when auto-rewriting
the concrete automaton.

It seems more reasonable to use a forward simulation to prove the correctness
of an implementation, when its pseudocode is short and simple, as Colvin et al. [16]
and Doherty et al. [13] did. Another reasonable approach is to develop tools to au-
tomatically generate the I/O automaton model from the implementations, such as
its program counter values and actions. The Tempo toolkits developed by Lynch
et al. [20] can translate specifications described in an I/O automata like language
into I/O automata and help with the verification. It would be easier to handle com-
plicated implementations if we were to have a tool that automatically proves some
easy facts about the pseudocode. For instance, local variables are not nil when they
are used, and process p’s local variables remains the same if p did not modify them.
It may also be useful to have a tool to generate lemmas on the pseudocode structure,
such as when a process is executing inside an IF block, the IF condition held at some
earlier time. This would really save proofs designers’ time and let them focus on the
more important and difficult lemmas required to prove correctness. Lesani et al. [28]
tried to construct a general framework for formally verifying software transactional
memory algorithms. Their framework provides templates which make it easier to
construct I/O automata and forward or backward relations.

We have formally verified the correctness of the simplified algorithm using PVS
by showing that a forward simulation exists between the concrete automaton, which models the implementation, and the intermediate automaton, and a backward simulation exists between the intermediate automaton and the canonical one which models the sequential specification. Thus, the algorithm without the helping mechanism is linearizable. Our future work is to verify that the original algorithm with the helping mechanism is also linearizable. This can be done by building another concrete automaton and showing that there is a simulation between this newly built one and ConcAut. This may be applicable since the automata for the original algorithm and the simplified version are quite similar. An alternative way to verify the original implementation would be to model the algorithm as a concrete automaton and redo the backward and forward simulation proof again. In this approach, a lot of lemmas and proofs we have proved in the old automaton can be directly reused and that will save a great deal of time. When modelling the pseudocode as ConcAut, we considered that accessing unchangeable fields of a shared object can be considered as 0 step of accessing the shared memory. We planned to explore such property in the future, which may simplify the way of modelling of general concurrent pseudo codes, thereby simplifying a formal verification. The BST algorithm is a non-blocking algorithm, which means that it guarantees that in any infinite execution some operation completes. We are also interested in formalizing the proof of this progress property of the BST using PVS, which may be quite different from
verifying the correctness property. In particular, it will require us to reason about infinite executions.
Bibliography


of Distributed Computing, ACM (August 1987) 137–151


