1. Introduction

Computers have been used for a variety of applications in business, science, education, engineering and so on. They help to solve real-world problems that would otherwise be slow, impossible or extremely difficult to address without computers and software. However, sometimes they do not behave exactly as we expect them. In many cases, the consequences could be very serious, for example when errors in banking or flight control software result in unexpected behaviours. Errors in computer systems are mostly not caused by the machine itself, but typically originate from the software that controls the computer systems, so-called bugs. Bugs are quite common in complex software systems since they typically have complicated input and involve many features, which makes them difficult to design and make them perfect by human effort. Detecting and fixing software bugs are important tasks in software development process. Remaining undetected bugs in any software project may lead huge problems. They can be very hard to detect and correct, especially if they are discovered after the software has been delivered. Therefore, it is very important to allocate sufficient resources, both in terms of time and manpower, to ensure that developed software is as free of bugs as possible.

Some bugs are less serious than others. Some types of software, e.g., in user interfaces or entertainment software, can be useable even if it contains a small number of bugs. However, in the case of critical systems and system components such as software in libraries of programming languages, bugs can have far-reaching consequences, and must be avoided as much as possible. Some of libraries provide standard data structures such as stacks, queues, containers. Such data structures provide ways of storing and retrieving data in a way that suits the application at hand. For example, a stack allows inserting and removing elements in a particular order. Every time an element is inserted, that element is removed in reverse order of insertion. The simplest application of a stack is to reverse a word. You insert a given word to stack - letter by letter - and then remove letters from the stack. By using data structures data can be easily, and efficiently exchanged; it allows portability, comprehensibility, and adaptability of information.

A data structure can be both sequential or concurrent which is tricky and difficult to get correct. Concurrent data structures can be accessed and manipulated concurrently by many parallel threads are a central component of many parallel software applications. A data structure should ideally provide a simple interface to the software that uses it. An interface provides the set of operations with specifications about their types of arguments and returned values. Data structures typically use heap-allocated memory to store their data. For example, the concurrent linked queue in java.util.concurrent uses a singly linked list to organize their data. The data structure may be quite complex like skiplists and binary trees which are used to implement sets.

The predominant method to ensure software quality is *testing*. It is a dynamic analysis where a program is executed under specific conditions, in so-called test cases, while checking whether the result for a given input matches the expected output. The test cases should be carefully designed to cover as many as possible cases of program executions. However, it is infeasible to cover all possible executions. Therefore, it is said by Edsger W. Dijkstra that testing can be used to show the presence of bugs, but never to show their absence. It would be nice to have techniques for checking that all executions conform to the interface of a data structure. A possibility is to use formal techniques which is the approach used in this thesis.

1.1 Formal Verification

Formal verification uses mathematical methods to check whether a program, or piece of software satisfies its specification. There are several approaches to formal verification, including equivalence checking, theorem proving, and model checking. Equivalence checking method decides whether a system is equivalent to its specification with respect to some notion of equivalence. In industry, this is mostly used for hardware designs. Theorem proving is a technique where both the behavior of the system and its desired properties are expressed in mathematical logic. Then, theorem proving, typically assisted by an interactive theorem prover, will try to prove that the system satisfies these properties.

Model checking takes as input a model of the program under consideration and a formal specification of the a property to be verified as inputs. The specification of a software component may consist of a number of such properties, each of which can be verified using model checking. The approach exhaustively explores all possible executions of the model. This is typically done exploring the set of reachable states of the model which can be finite or infinite. This works well if the set of reachable states is finite which typically happens for embedded controllers and hardware design. However, most softwares are infinite-state, e.g., a data structure may contain an unbounded amount of data. A common technique for handling this is to devise a symbolic representation of sets of states, such that a single symbolic representation represents an infinite set of states. However, it is difficult to find a suitable symbolic representation for data structures. In particular, complicated data structures such as trees and skip-lists where relationship between heap cells are complicated in both reachability and data aspects.

1.2 Research Challenges

Our challenge of this thesis is to develop techniques for automated verification of both sequential and concurrent data structures using dynamically heap-allocated memory. This requires to address several challenges in model checking:

- **Dynamically heap-allocated memory**: Data structures typically use dynamically heap allocated memory. In each cell of a heap, the domain of data values can be unbounded. In the area of formal verification, exist several approaches for heaps and for data, but not for combining them in suitable ways. In this thesis, we provide automate verification approach to sequential data structures where correctness depends on relationships between data values that are stored in the dynamically allocated structures. Such ordering relations on data are central for the operation of many data structures such as search trees, priority queues (based, e.g., on skip lists), key-value stores, or for the correctness of programs that perform sorting and searching. There exist many automated verification techniques dealing with these data structures, but only few of them can automatically reason about data properties. However, they are often limited to specific classes of structures mostly singly-linked lists (SLLs). Our approach is based on the notion of forest automata which has previously been developed for representing sets of reachable configurations of programs with complex dynamic linked data structures.
- **Unbounded number of threads**: For the case of concurrent data structures. We have to verify that the data structures are correct with any number of threads that access and manipulate the structures. We handle this challenge by extending the successful thread-modular approach which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread. By doing thread modular, we only verify each thread separately using an automatically inferred environment assumption that abstracts the possible steps of other threads.
- **Specification of correctness**: To ensure that a concurrent data structure is correct, we have to specify a correctness criterion that relates the concurrent interface to the interface of a corresponding sequential data structure. One way to capture such a correctness criterion is linearizability. Linearizability is generally accepted as the standard correctness criterion for such concurrent data structure implementations. Intuitively, it states that each operation on the concurrent data structure can be viewed as being performed atomically at some point (called linearization point (LP)) between its invocation and return. Existing approaches lack generality as they are limited to specific classes of concurrent data structures based on simple heaps such as singly linked lists, so far no technique (manual or automatic) for proving linearizability has been proposed that

is both sound and generic. In this thesis we provide a technique to specify linearizability of concurrent data structures.

- **Unbounded number of pointers**: In some data structures, the each cell can have unbound number of pointer fields, such as cells in skip-lists and arrays of lists. It is difficult to provide symbolic representation for these data structures. There are no techniques that have been applied to automatically verify concurrent algorithms that operate on such data structures. We propose a technique called *fragment abstraction* in which a heap is divided into small pieces called fragments. A fragment is an abstraction of a pair of heap cells that are connected by a pointer field. Our approach is general and precise enough to verify these complicated data structures.

The following sections are organized as follow: in Section 2, we present the general background about model checking, then in Section 3, we describe data structures. Thereafter, in Section 4, we describe how to specify linearizability. Our heap abstraction techniques are described in Section 5. Finally, in Section 6, we summarize and give future plans for our work.

2. Model Checking

The approach that we focus on this thesis is *model-checking*. This approach was introduced by Emerson and Clarke [30] and by Queille and Sifakis [75]. Model checking aims to check whether a model of a program satisfies its *[Better: "a given"]* specification. *[The following sentence is redundant, you can skip it.]* As input, the method requires a model of the system under consideration and a specification. *[Here,you can move the sentence "The method then ..."]* Models are typically transition systems consisting of states and transitions between states. *[The following sentence can be moved to later]* Whereas the specification contains safety requirements The method then computes and returns either "correct" when the specification is satisfied by the program, or "incorrect" when the program does not satisfy its specification. In the case of incorrect answer, the method can explain the reason by giving a counter-example. *[Here, you can start to explain how models look: "Models are typically ..."]* A state in the model contains relevant information about the program. Alongside all the states of the system, the model depicts the transitions, i.e. how to move from one state to another state. Every behaviour of the system is represented as a succession of transitions, starting from some initial states *[better "state"]*. *[THe following sentence is not really necessary]* States and transitions together describe the *operational semantics*, that is, how steps of the system take place in the model. The number of states and transitions can be finite or infinite. *[Now you can start to explain "specifications": You can say, e.g., that a specification consists of properties of behaviors (i.e., of sequences of states), that one usually distinguishes safety properties ("something bad must never happen") from liveness properties ("something good must eventually happen"), and that most properties are safety properties. Then say that a model cheking algorithm explores the set of states in a clever way, and explain the word "state-space"]* Model-checking aims to explore the state-space entirely from some initial states. *[The following sentence is incomplete. You can instead say that one of the main problems with model checking is that the state-space is typically very large]* However, when the state-space is of large size. It grows in-fact exponentially with the number of parameters or the size of their domain. Therefore, there have been several methods to address with *[delete "with"]* the state-space explosion problem.

There are several techniques addressing the state-space explosion problem. *[Next sentence becomes not clear. Better to start "One important approach is partial order techniques, which aim at ... They are based on the fact ..."]* Based on the fact that, in some cases, exploring all orderings of events is not necessary because some states can be re-visited. Therefore, *Partial order* techniques aim at detecting and avoiding redundant situations, while retaining important dependencies among actions. The main approach to solve the state-space explosion problem is called *[better "to use a"]* *symbolic representation*. It avoid *["avoids"]* representing concretely all states of the system. *[The following can better start "Its main idea is to design a symbolic ..."]* The approach is performed by designing a symbolic representation of sets of states. This designing process is done by *dropping* irrelevant details based on properties that we want to verify.

[For the following text, you can use a drawing of the transition system. I enclose my LATEX figure if you like to use it] As an example, consider the dining mathematicians which is a *[better: "is a protocol for implementing the classical .."]* classical mutual exclusion problem: There are two mathematicians living at the same place, whose life is focused on two activities, namely thinking (think01 and think02, respectively) and eating (eat01 and eat02, respectively). They do not want to eat at the same time. To ensure this, they agreed to have access to a common integer variable n. *[use math mode for the variable here and in the following]* If value of n is even, the first mathematician is allowed to start eating. When finished, the first mathematician sets (n/2) to n. A similar procedure holds for the second mathematician, where the check is if the value of the stored variable is odd, and the value set back after eating is (3n+1). *[The description of the protocol is not good.]* We want to verify that two mathematicians do not eat at the same time, *[Here, start a new sentence. Also say that this specification is stated by a safety property which says that the two mathematicians are not simultaneously in states eat 01 and eat 02]* if we keep all the information of the system, then, a state consists of the *[A state cannot consist of actions, rather it represents the control state of each mathematician]* actions of the mathematicians, and the value of the n. If `n` is large then we may end up to four million states. *[This depends on the type of n, make this clear]* However, it is obvious that n is not fully needed to verify the property, so we could present it symbolically where we use two boolean predicates `even(n)` and `odd(n)` to present *["represent"]* whether n is even or odd *[Skip the following word]* value. The dining mathematicians example is of course simple and does not reflect the complexity of today's software. Symbolic representations are of crucial help to combat the state-space explosion, accelerate the algorithms and get them to terminate in a reasonable amount of time. *[The last sentence should be put earlier (in the preceding paragraph), where you introduce symbolic techniques]*

Finding a right over-approximations *[Better "symbolic representation"]* is challenging, it might introduce behaviours that could turn out to be bad. Indeed, the method would return that the property is not satisfied and we would not know whether it comes from the approximation *["introduced by the symbolic representation"]* or from the concrete system itself. For example in the dining mathematicians problem. If we would start with an over-approximation that ignores the variable n, we get a false positive, and that one *[Better "We must then refine ..."]* needs to refine the symbolic representation to avoid it.

To deal with the imprecision caused by a too coarse over-approximation, it is possible to analyse the returned counter-example and find the origin of the problem. If it turns out to be a real concrete example, the method has in fact found a bug, and the property is surely not satisfied. Otherwise, the counter-example comes from the approximation, that is, there is a step in the sequence of events leading to that counter-example which is not performed by the original system but only by the abstract model. The approximation is be refined by discarding this step and the method should be run anew.

Nevertheless, finding suitable over-approximations is a challenge on its own. This thesis now revolves around the problems of unboundedness which are described in previous section. *[This last sentence is much too vague, and cannot be understood. You shoudl be concrete about which unboundedness you refer to, and be more precise about which text you refer to]*

3. Data Structures

In general, a data structure is any data representation and its associated operations. Even an integer or floating point number stored on the computer can be viewed as a simple data structure. Typically, a data structure is meant to be an organization or structuring for a collection of data items. Each data structure has an interface which defines a set of possible values and a set of operations. More precisely, the interface consists of a set of operations or methods, each having a number of input and output parameters, and a specification of the effect of each operation.

For example, a sequential set is an data structure for storing a collection of elements, with the three operations as following:

- `add(e)` adds element `e` into the set, returning `true` if, and only if `e` was not already there.
- `remove(e)` removes element `e` from the set, returning `true` if, and only if `e` was there.
- `contains(e)` checks the existence of element `e` in the set, returns `true` if, and only if the set contains `e`.

For each method, we say that a call is successful if it returns true, and unsuccessful otherwise. It is typical that in applications using sets, there are significantly more `contains()` calls than `add()` or `remove()` calls. A set is implemented as a singly linked list of cells. Each cell has two fields. The `val` field present value of a cell. Cells are sorted according to `val` order, providing an efficient way to detect when an item is absent. The `next` field is a reference to the next cell in the list. The list has two sentinel cells, called head and tail, which are first and last list elements. Sentinel nodes are never added, removed, or searched for, and their values are the minimum and maximum integer values.

The implementation of a data structure should provide an efficient way to store data in computer memory and perform its operations in an efficient way. Data structures typically used heap-allocated memory to store their data. Various schemes can be used to organize the heap-allocated memory, such as singly-linked lists, doubly-linked lists, skip-lists, trees. The implementation can be sequential or concurrent.

A concurrent data structure is a way of storing and organizing data for access and manipulation by multiple computing threads (or processes) on a shared-memory computer. Each operation is implemented as a sequential method that is executed by a thread. Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts. The primary source of this additional difficulty is concurrency: because threads are executed concurrently possibly on different processors, and are subject to operating system scheduling decisions, interrupts, etc., we must think of the interaction between threads as completely asynchronous, so that the steps of different threads can be interleaved arbitrarily.

There are several techniques to construct concurrent data structures including coarse-grained locking, fine-grained locking, and lock-free programming. The simplest technique is coarse-grained locking, where a single lock is used to synchronize every access to an object. Coarse-grained locking is easy to reason about, however it works well only when the level of concurrency is low. However, if too many threads try to access an object at the same time, then the object becomes a sequential bottleneck, forcing threads to wait in line for access. Therefore, Fine-grained synchronization techniques address this problem by splitting the object into independently synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time. Fine-grained locking requires very careful design of the data structure and its methods, since one must foresee what can happen when several threads access the same component in parallel. Fine-grained synchronization is often performed without locks, replacing them by less costly synchronization operations such as `compareAndSet()`. Each of these techniques can be applied (with appropriate customization) to a variety of common data structures (queues, stacks, sets) implemented by different linked data structures such as singly linked lists, skiplists, trees, or lists of lists.

As an example, Fig. 3.1 depicts a program `Lazy Set` [50] that implements a concurrent set containing integer elements with three operations `add`, `remove` and `contains`. It is just as the sequential version, but that each cell now has two additional fields fields `mark`, `lock`. The field `mark` is `true` if the node has been logically removed from the set. The `lock` field is a lock and the field `val` presents the data value which is integer in this case. The mechanism behind logically and physical removing is explained as following: it is impossible to atomically remove a cell from the list if other threads may concurrently access the adjacent cells. One reason is that one must both move a `next` pointer which reference to the cell and physically remove the cell. This cannot be done, e.g., if another thread currently is visiting the cell that is to be removed. Therefore, the task of removing a cell from the list can be split into two phases: the cell is logically removed simply by setting a $mark$ field to be `true`, and later, the cell can be physically deleted by unlinking it from the rest of the data structure. The removal "actually happens" when an entry is marked, and the physical removal is just a way to clean up The algorithm uses two global pointers, `head` that points to the first cell of the heap, and `tail` that points to the last cell. These two cells contain two values that are smaller and larger respectively than data

```
struct Node {
      bool lock;
      int val;
      Node* next;
      bool mark;
   }
```

```
  ctn(e):
  local c
1 c := Head;
2 while (c.val < e)
3   c := c.next
4 b := c.mark
5 if (!b
       && c.val = e)
6   return true;
7 else
8   return false;
```

```
 locate(e):
 local p, c
 1 while (true)
 2   p := Head;
 3   c := p.next;
 4   while (c.val < e)
 5     p := c;
 6     c := c.next
 7   lock(p); lock(c);
 8   if (! p.mark &&
        ! c.mark&&
        p.next=c)
 9     return (p,c);
10   else
11     unlock(p);
12     unlock(c);
```

```
add(e):
local p, c, n, r
1 (p,c) := locate(e);
2 if (c.val <> e)
3   n :=
    new Node(
        0,e,c,false);
4   p.next := n;
5   r := true;
6 else r := false;
7 unlock(p); unlock(c);
9 return r;
```

```
rmv(e):
local p, c, n, r
1 (p,c) := locate(e);
2 if (c.val = e)
3   c.mark := true;
4   n := c.next;
5   p.next := n;
6   r := true;
7 else r := false;
8 unlock(p); unlock(c);
9 return r;
```

Figure 3.1. Lazy Set Algorithm

values of all cells that may be inserted in the set. The algorithm also contains the subroutine `locate` that returns a structure containing the cells on either side of `e`. In more detail, the `locate` method traverses the list using two local variables `p` and `c`, starting at the head of the list and moves forward the list (line 2), comparing `c.val` to `e`. When `c` is set to the first cell whose the value of `val` is greater than or equal to `e`, the traversal stops, and the method locks cells pointed to by `p` and `c` (line 7) so that no other thread can update fields of `p` and `c`. Thereafter, if both `p.mark` and `c.mark` are `false` and `p.next` $=$ `c` meaning that there is no added cell from other thread between `p` and `c` then the method returns the pair $(\mathtt{p}, \mathtt{c})$ (line 9). Otherwise, it unlocks cells pointed to by `p` and `c` and tries traversing again from the head of the list.

The `add(e)` method calls `locate(e)` at line 1 to locate the position in the list where `e` is to be inserted. Its local variables `p` and `c` are assigned the first and second values of the pair return by `locate(e)` respectively. If `c.val` $=$ `e` meaning that a cell whose data value of `val` is equal to `e` is already in the list, the method unlocks `p` and `c` and returns `false` (line 7-8). Otherwise, a new cell `n` is created (line 3), and inserted into the list by linking it into the list between the `p` and `c` pointers returned by `locate` (line 3-4). Then, the method unlocks cells pointed to by `p` and `c` and returns `true`.

The `rmv(e)` method also calls `locate` at line 1 locate the position in the list where `e` is to be inserted. If `c.val` $\neq$ `e` meaning that a cell with val `e` is not already in the list, the method unlocks `p` and `c` and returns `false`(line

7-9). Otherwise, cell `c` is logically removed (line 3) where the `mark` field of `c` gets assigned `true`, and unlinked from the list (line 4-5). Then, the method unlocks cells pointed to by `p` and `c` and returns `true`.

The `ctn(e)` method traverses the list by using local variable `c`, ignoring whether nodes are marked or not, until `c` is set to the first cell with a value of `val` greater than or equal to `e`. It simply returns `true` if and only if the cell pointed to by `c` is unmarked with the desired value of `val` equal to `e`.

3.1 Linearizability

In a concurrent program, the methods of the different executing threads can overlap in time. Thus, a `rmv` method that executes in parallel with an `add` method for the same key may or may not find the element in the set, depending on how the individual method statements overlap in time. For a user of the data structure, it is important to know precisely what can happen when several methods access a data structure concurrently without inspecting the code of each method. Such a user would want to have a criterion for how operations take effect, which considers only the points in time of method calls and returns. The most widely accepted such condition is linearizability. Linearizability defines consistency for the history of call and response events generated by an execution of the program at hand [51]. Intuitively, linearizability requires every method to take effect at some point (*linearization point*) between it's call and return events. A linearization point is intuitively a moment where the effect of the method becomes visible to other threads. An execution of a (concurrent) system is modeled by a (concurrent) history, which is a finite sequence of method invocation and response events. A (concurrent) history is linearizable if and only if there is some order for the effects of the actions that corresponds to a valid sequential history. The valid sequence history can be generated by an execution of the sequential specification object. A concurrent object is linearizable iff each of its histories is linearizable.

Figure 3.2 provides a examples of trace of methods of concurrent program implementing sets. In the trace, each method takes effect instantaneously at its (called the *linearization point*) between call and return events [51]. When we order methods according to its linearization point, we get a total ordered sequence that respect the sequential specification of the set.

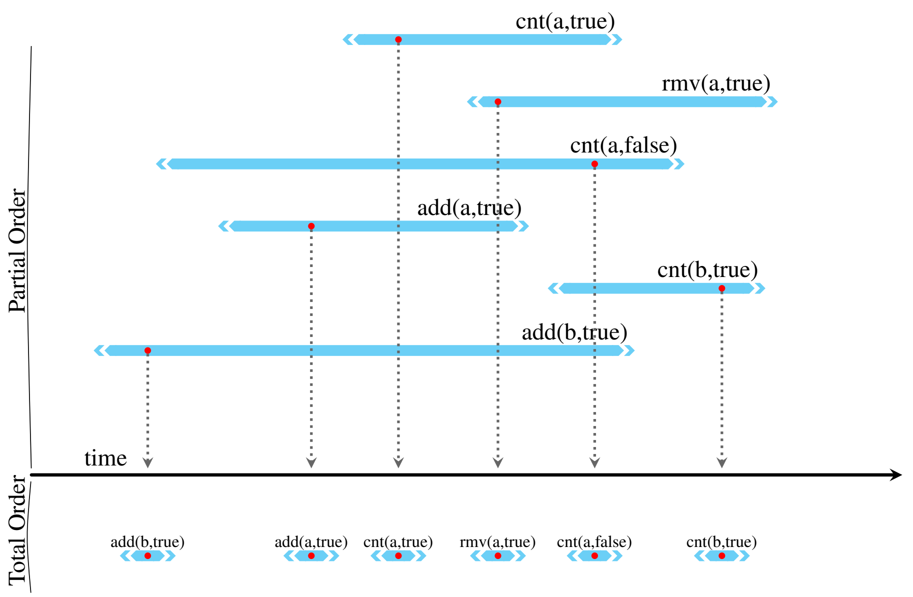

Figure 3.2. Linearizability, where the commit points are marked with •.

4. Specifying Linearizability

In the previous sections, we describe the correctness criterion of linearizability for concurrent data structures. The goal is to specify linearizability in a way that is suitable for automated verification. We separate the problem of specifying linearizability into several ones.

- To specify the sequential semantics of a data structure in a way that is suitable for automated verification.
- To specify placement of linearization points in executions of a concurrent data structure.

For the first, we use the techniques of observers [2]. For the second, we present a new technique, in which methods are equipped with controllers. Controllers specify so-called "linearization policies", which prescribe how LPs are placed in executions.

4.1 Observers

Data structures are, by nature, infinite-state objects, since they are intended to carry an unbounded number of data elements. For automated verification, it is desirable with specifications that are constructed without explicitly mentioning such infinite objects. This problem is addressed by observers [2]. Observers specify allowed sequences of operations by constraining their projection on a small number of data elements. Observers are finite automata extended with a finite set of observer *registers* that assume values in integer domain. At initialization, the registers are nondeterministically assigned arbitrary values, which never change during a run of the observer. Transitions are labeled by operations that may be parameterized on registers. Observers are used as acceptors of sequences of operations. The observer processes such sequences one operation at a time. If there is a transition, whose label, after replacing registers by their values, matches the operation, such a transition is performed. If there is no such transition, the observer remains in its current state. The observer accepts a sequence if it can be processed in such a way that an accepting state is reached. We use observers to give exact specifications of the behaviors of

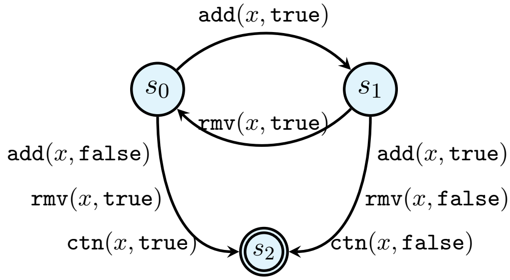

Figure 4.1. A stack observer

data structures such as sets, queues, and stacks. The observer is defined in such a way that it accepts precisely those sequences of abstract operations that are not allowed by the semantics of the data structure. This is best illustrated by an example. Fig. 3 depicts an observer that accepts the set of method invocations that are not allowed by sequential specification of a set. The observer have three states s_0, s_1 and s_2. The initial state s_0 corresponds to positions in the runs where the non-determnistically tracked value stored in the observer variable z is not present in the set (i.e. each time it has been inserted it got deleted afterwards). The state s_1 corresponds to positions in the runs where the tracked value is present in the set (i.e. it has not been deleted since it was last inserted). The accepting state s_2 corresponds to positions in the runs where the bad specification captured by the observer has been observed. The captured bad specification are those where a data value is deleted or found although it is not present in the set, or a data value is not found or cannot be deleted although it is already present in the set

4.2 Linearization Policies

In order to prove linearizability, the most intuitive approach is to find a linearization point (LP) in the code of the implementation, and show that it is the single point where the effect of the operation takes place [2, 18, 89]. However, for a large class of linearizable implementations, it is not possible to assign fixed LPs in the code of their methods, but depend on actions of other threads in each particular execution. For example, in the `Lazy Set` algorithm, a successful `rmv(e)` method has its LP at line 3, and an unsuccessful `rmv(e)` has its LP at line 2 when the test `c.val` $=$ `e` evaluates to `false`. A successful `add(e)` method has its LP at line 4 and an unsuccessful `add` has its LP at line 2 when the test `c.val` $<>$ `e` evaluates to `false` . The successful `ctn` is linearized at line 4 then the value of `b` is `true`. However it is not possible to assign a fixed LP in the code of the `ctn` method.

To see why unsuccessful `ctn` method invocations may not have fixed LPs, note that the naive attempt of defining the LP at line 4 provided that the test at

add
ρ_1 : **when** • **provided** `pc=4` **emit** `add(e,true)` **broadcast** `add(e)`
ρ_2 : **when** • **provided** `pc=2 && (c.val = e)` **emit** `add(e,false)`
rmv
ρ_3 : **when** • **provided** `pc=3` **emit** `rmv(e,true)`
ρ_4 : **when** • **provided** `pc=2 && c.val <> e` **emit** `rmv(e,false)`
ctn
ρ_5 : **when** • **provided** `pc=4 && !b && c.val = e` **emit** `ctn(e,true)`
ρ_6 : **from** q_0 **when** • **provided** `pc=4&&!(!b&&c.val=e)` **emit** `ctn(e,false)` **goto** q_0
ρ_7 : **from** q_0 **when** `(add(e),b)` **provided** `1<=pc<=4` **emit** `ctn(e,false)` **goto** q_1

Figure 4.2. Reaction Rules for Controllers of `Lazy Set`.

line 5 fails will not work. Namely, the `ctn` method may traverse the list and arrive at line 4 in a situation where the element e is not in the list (either e is not in any cell, or the cell containing `e` is marked). However, before executing the command at line 4, another thread performs an add operation, inserting a new cell containing the element `e` into the list. The problem is now that the `ctn` method cannot "see" the new cell, since it is unreachable from the cell currently pointed to by the variable b of the `ctn` method. If the `ctn` method would now try to linearize an unsuccessful `ctn`, this would violate the semantics of a set, since the add method just linearized a successful insertion of `e`.

There have been several previous works dealing with the problems of non-fixed linearization points [97, 33, 32, 81, 37, 80, 90, 27]. However, they are either manual approaches without tool implementation or not strong general enough to cover various types of concurrent programs. In this thesis we handle non-fixed linearization points by providing semantic for specifying linearization policies by a mechanism for assigning LPs to executions, which we call *linearization policies*.

A linearization policy is expressed by defining for each method an associated controller, which is responsible for generating operations announcing the occurrence of LPs during each method invocation. The controller is occasionally activated, either by its thread or by another controller, and mediates the interaction of the thread with the observer as well as with other threads.

To add controllers, we first declare some statements in each method to be *triggering*: these are marked by the symbol • as in Figure 3.1. We specify the behavior of the controller, belonging to a method `m`, by a set of *reaction rules*. To define these rules, we first define different types of events that are used to specify their behaviors. Recall that an *operation* is of the form $\mathtt{m}(d^{in}, d^{out})$ where `m` is a method name and d^{in}, d^{out} are data values. Operations are emitted by the controller to the observer to notify that the thread executing the method performs a linearization of the corresponding method with the given input and output values. Next, we fix a set Σ of *broadcast messages*, each with a fixed arity, which are used for synchronization between controllers. A message is formed by supplying data values as parameters. In reaction rules, these data values are denoted by expressions over the variables of the method, which are evaluated in the current state when the rule is invoked. In an operation, the first parameter, denoting input, must be either a constant or the parameter of the method call.

- A *triggered* rule, of form **when** $\bullet$ **provided** cnd **emit** op **broadcast** se, specifies that whenever the method executes a triggering statement and the condition cnd evaluates to `true`, then the controller performs a *reaction* in which it emits the operation obtained by evaluating op to the observer, and broadcasts the message obtained by evaluating se to the controllers of other threads. The broadcast message se is optional.
- A *receiving* rule, of form **when** $\langle re, \texttt{ord}\rangle$ **provided** cnd **emit** op, specifies that whenever the observer of some other thread broadcasts the message obtained by evaluating re, and cnd evaluates to `true`, then the controller performs a reaction where it emits the operation obtained by evaluating op to the observer. Note that no further broadcasting is performed. The interaction of the thread with the observer may occur either *before* or *after* the sender thread, according to the flag `ord`.

A controller may also use a finite set of states, which restrict the possible sequences of reactions by a controller in the standard way. Whenever such states are used, the rule includes source and target states using kewords **from** and **goto**. In Figure 4.2, the rule ρ_7 changes the state from q_0 to q_1, meaning that no further applications of rules ρ_6 or ρ_7 are possible, since they both start from state q_0. Rules that do not mention states can be applied regardless of the controller state and leave it unchanged.

Let us illustrate how the reaction rules for controllers in Figure 4.2 specify LPs for the algorithm in Figure 3.1. Here, a successful `rmv` method has its LP at line `3`, and an unsuccessful `rmv` has its LP at line `2` when the test `c.val = e` evaluates to `false`. Therefore, both these statements are marked as triggering. The controller has a reaction rule for each of these cases: in Figure 4.2: rule ρ_3 corresponds to a successful `rmv`, whereas rule ρ_4 corresponds to an unsuccessful `rmv`. Rule ρ_4 states that whenever the `rmv` method executes a triggering statement, from a state where `pc=2` and `c.val <> e`, then the operation `rmv(e,false)` will be emitted to the observer.

A successful `add` method has its LP at line `4`. Therefore, the controller for `add` has the triggered rule ρ_1 which emits the operation `add(e,true)` to the observer. In addition, the controller also broadcasts the message `add(e)`, which is received by any controller for a `ctn` method which has not yet passed line `4`, thereby linearizing an unsuccessful `ctn(e)` method by emitting `ctn(e,false)` to the observer. The keyword **b** denotes that the operation `ctn(e,false)` will be presented before `add(e,true)` to the observer. Since the reception of `add(e)` is performed in the same atomic step as the triggering statement at line `4` of the `add` method, this describes a linearization pattern, where a `ctn` method, which has not yet reached line `4`, linearizes an unsuccessful ctn-invocation just before some other thread linearizes a successful add of the same element.

To see why unsuccessful ctn method invocations may not have fixed LPs, note that the naive attempt of defining the LP at line 4 provided that the test at line 5 fails will not work. Namely, the ctn method may traverse the list and arrive at line 4 in a situation where the element e is not in the list (either e is not in any cell, or the cell containing e is marked). However, before executing the command at line 4, another thread performs an add operation, inserting a new cell containing the element e into the list. The problem is now that the ctn method cannot "see" the new cell, since it is unreachable from the cell currently pointed to by the variable b of the ctn method. If the ctn method would now try to linearize an unsuccessful ctn(e), this would violate the semantics of a set, since the add method just linearized a successful insertion of e.

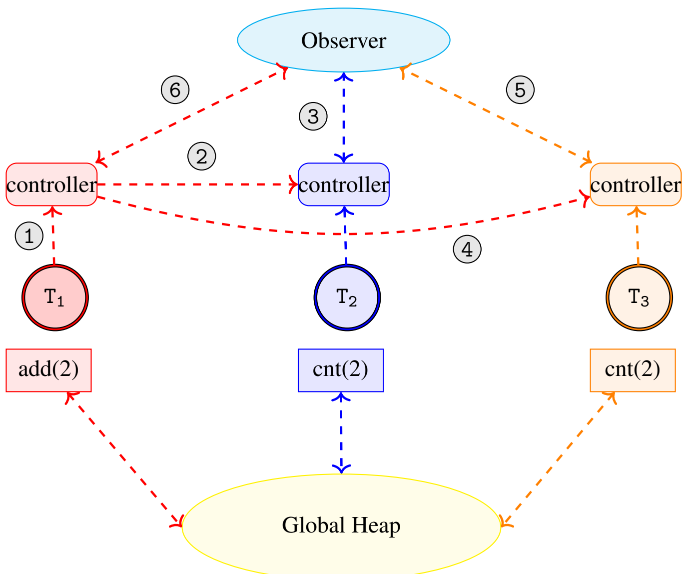

Figure 4.3. An example of linearization policies of Lazy Set algorithm

Let us describe an example of how the controller handles non-fixed linearization point of unsuccessful ctn(e) method. Figure 4.3 give an example of Lazy Set with three threads 1, 2, and 3. The thread 1 is executing the add(e) method to insert the cell whose value of val is equal to e into the set, while both 2 and 3 are executing ctn(e) to search for the cell whose value of val is equal to e in from set. When thread 1 reaches the triggering statement at line 4 of the add(e) method at the step 1. The controller rule ρ_1 is active. to informs the observer that an add operation with argument e has been performed, and that the outcome of the operation is true (the operation was successful) in the step 6. However, before that the controller will help other threads by to linearize. This is done by *broadcasting* a message `add(e)` to the threads 2, 3 which is executing `ctn(e)` in steps 2, 3. These threads get message then the rule ρ_7 is active to inform the observer that the element `e` is not in the list in 3, 4 respectively.

Verifying Linearization Policies

By using an observer to specify the sequential semantics of the data structure, and defining controllers that specify the linearization policy, the verification of linearizability is reduced to establishing four conditions: (i) each method invocation generates a non-empty sequence of operations, (ii) the last operation of a method conforms to its parameters and return value, (iii) only the last operation of a method may change the state of the observer, and (iv) the sequence of all operations cannot drive the observer to an accepting state. Our verification framework automatically reduces the establishment of these conditions to a problem of checking control state reachability. This is done by augmenting the observer by a *monitor*. The monitor is automatically generated. It keeps track of the state of the observer, and records the sequence of operations and call and return actions generated by the threads. For each thread, it keeps track of whether it has linearized, whether it has caused a state change in the observer, and the parameters used in its last linearization. Using this information, it goes to an error state whenever any of the above four conditions is violated.

5. Shape Analysis

Pointers and heap-allocated storage are features of all modern imperative programming languages. They are among the most complicated features of imperative programming language: updating pointer variables (or pointer-fields of records) may have large side effects. For example, dereferencing a pointer that has been freed will lead to segmentation fault in a C or C++ program. Such side effects also make program analysis harder, because they make it difficult to compute aliasing relationships among different pointers in a program. Aliasing arises when the same memory location can be accessed using different names. For instance, consider the instruction statement `x.f` := `y` in an imperative language, where `x` and `y` are pointer variables. Its effect is to assign the value of the pointer `y` to the cell pointed to by `x.f`. In order to update aliasing information of `y`. We have to require information about all the cell pointed by `x.f` which is not an easy task.

In verification and program analysis, it is a problem to deduce and describe how the heap-allocated memory is organized. E.g., program invariants must often describe how the heap-allocated memory is structured in order to infer the effects of statements that dereference pointer fields. This is the topic of “shape analysis”.

5.1 Previous Approaches

Shape analysis is a generic term denoting static program-analysis techniques that attempt to discover and verify properties of the heap contents in (usually imperative) programs. The shape analysis problem becomes more challenging in concurrent programs that manipulate pointers and dynamically allocated objects, which are usually complicated. In concurrent programs, different threads interact in complex ways, which are difficult to foresee in the analysis. Several approaches for representing the possible structures of the heap have been proposed. TVLA (Three-Valued Logic Analysis) [78] is one of the first and one of the most popular shape analysis methods. It is based on a three-valued first-order predicate logic with transitive closure. Intuitively, concrete heap structure is represented by a finite set of abstract summary cells, each of them representing a set of concrete cells. Summary cells are obtained by merging several heap cells that agree on the values of a chosen set of unary abstraction predicates. A unique important aspect of TVLA is that it automatically generates the abstract transformers from the concrete semantics; these transformers

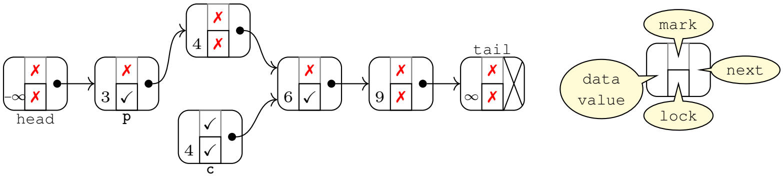

Figure 5.1. A heap configuration of the `Lazy Set` Algorithm where `head` and `tail` are global variable, and `p` and `c` are two local variables. The symbols ✓ and ✗ represent the Boolean values `true` and `false`.

are guaranteed to be sound, and precise enough to verify wide ranges of applications. However, it cannot fully automatically handle all programs, and that one may have to extend it by appropriate predicates etc. Its the synthesis of appropriate predicates that are able to express the invariants in the program. This problem is even more difficult with complicated heap structures such as skip-lists, trees, or arrays of lists.

There are several other approaches based on the use of logics to present heap configurations. The logics can be separation logic [76, 64, 19, 95, 38, 28, 60, 72, 41], monadic second-order logic [69, 55, 63] and others [77, 96]. Among these works, the works based on separation logic are more efficient than the others. The reason for that is that their approaches effectively decompose the heap into disjoint components and treat them independently. However, most of the techniques based on separation logic are either specialised for some particular data structure, or they need to be provided inductive definitions of the data structures. In addition, their entailment checking procedures are either for specific class of data structures or based on folding/unfolding inductive predicates in the formulae and trying to obtain a syntactic proof of the entailment.

This issue can be fixed addressed by automata-based techniques using the generality of the automata-based representation such as techniques using tree automata. Finite tree automata, for instance, have been shown to provide a good balance between efficiency and expressiveness. The work [26] uses a finite tree automaton to describe a set of tree parts and represent non-tree edges of heaps by using regular "routing" expressions. Finite tree transducers are used to compute set of reachable configurations, and symbolic configuration is abstracted collapsing certain states of the automata. The refinement technique called counterexample-guided abstraction refinement (CEGAR) technique is used during the run of the analysis. This technique is fully automatically and can handle complex data structures such as binary trees with linked leaves. However, it suffers from the inefficiency and it also can not handle concurrent programs.

The problem of inefficiently of the previous technique can be solved by the approach based on forest automata [82]. In such representation, a heap is split in a canonical way into several tree components whose roots are the so-called cut-points. Cut-points are cells which are pointed to by either program variables

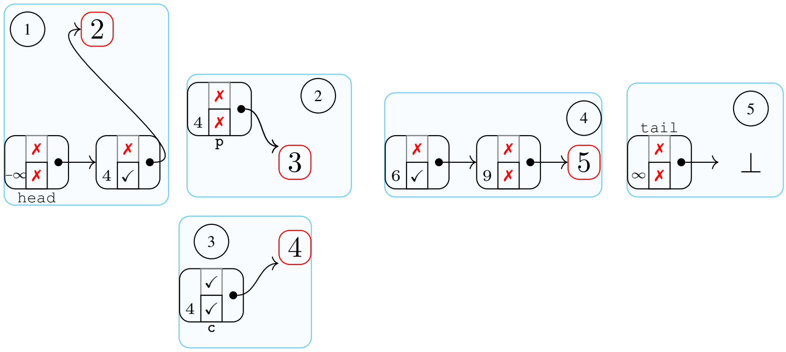

Figure 5.2. The forest representation of the heap configuration in Fig. 5.1.

or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [76]. Using this decomposition, sets of heaps with a bounded number of cut-points can then be represented by so called forest automata (FA). Each of the tree automata within the tuple accepts trees whose leaves may refer back to the roots of any of these trees. A forest automaton then represents exactly the set of heaps that may be obtained by taking a single tree from the language of each of the component tree automata and by gluing the roots of the trees with the leaves referring to them. Moreover, they allow alphabets of FA to contain nested FA, leading to a hierarchical encoding of heaps, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL, skiplist).

Let us take an example of how to split a heap into small tree components. Figure 5.2 shows five tree components obtained by splitting the heap in figure 5.1. These components are named as 1,2,3,4 and 5 from left to right. Each root of a tree component is a cut-point in the heap in figure 5.1. These cut-points are cells pointed to by variables `head`, `tail`, `p`, `c` and the cell which has two incoming pointers. In each tree component, the red node show which tree component it refers. For instance, the tree component 1 refers to tree component 2, and both tree components 2 and 3 refer to tree component 4.

This forest automata approach is fully automatic and able to verify various classes of data structures, including complicated structures such as trees and skip-lists with bounded number of levels. However, the approach can not verify properties related to data values of heap cells such as sortedness in the lazy set algorithm. Therefore, in this thesis, we extend their work to verify data properties.

The last approach that we will mention is based on graph grammars describing heap graphs [49, 48]. The approach is to model heap states hypergraphs, and represent both pointer operations and abstraction mappings by hypergraph transformations. The presented approaches differ in their degree of specialisation for a particular class of data structures, their efficiency, and their level of dependence on user assistance (such as definition of loop invariants or inductive predicates for the considered data structures).

5.2 Our Approaches

In this thesis, we propose three approaches for heap abstractions. In paper I, we propose a novel approach of extending the forest automata approach [82] by expressing relationships between data elements associated with cells of the heap by two classes of constraints.

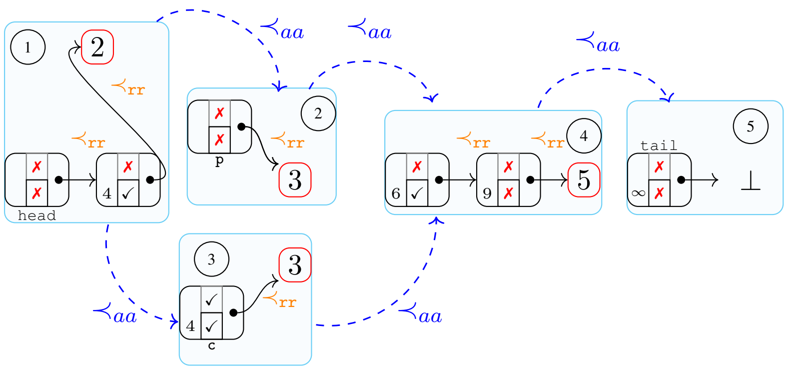

Figure 5.3. A graph and its forest representation

- Local data constraints are associated with transitions of each TA and capture relationships between data of neighboring cells in a heap; they can be used, e.g., to represent ordering internal to data structures such as sorted linked lists and binary search trees.
- Global data constraints are associated with states of TA and capture relationships between data in distant parts of the heap. Intuitively, a global data constraint between two TAs captures the data relationship between cells of two heaps accepted by these two TAs.

Figure 5.3 shows an example of how to represent a heap in 5.1 by a set of tree automata with added data constraints. In the figure, the local data constraints are located along the solid arrows between cells, whereas global constraints are located along the blue arrows. The global constraint $\prec_{\mathtt{aa}}$ means that data values of all cells in the left hand side are smaller than data values of all cells in the right hand side. The local constraint $\prec_{\mathtt{rr}}$ means that left hand side cell is smaller than data value of the cell in the right hand side. For instance, in tree component 1, the data value of the cell pointed to by `head` is smaller than its successor whose data value is 4, and data values of all cells in tree component 1 are smaller than data values of all cells in tree component 2. We just show here small examples of data constraints, the detail about different types of constraints can be found in paper I.

This approach was applied to verification of sequential heap manipulation programs. This approach is general and able to verify properties of many types of sequential programs without any manual step. However, due to the complexity of tree automata operations, this approach is not optimal to handle concurrent programs where a large number of states and computation are needed. In order to verify concurrent data structures with unbounded number of threads, thread-modular is a promising approach for this challenge. Its high efficiency is achieved by abstracting the interaction between threads. The approach verifies a concurrent data structure by generating an invariant that correlates the global state with the local state of an arbitrary thread. In other words, it only keep track of the shape viewed by one thread, while abstracting away all the other threads. The thread-modular approach includes a step where it takes the information about one thread and intersect it with the information from another thread, in order to take into account the interference of all the other threads on the first thread. Forest automata approach is not suitable for this thread-modular. The reason is that computing intersection between FAs is not efficient in concurrent systems where the number of FAs is huge.

Therefore, in paper II, we adapt FA to the new setting by providing a symbolic encoding of the heap structure, that is less precise than the forest automata approach in paper I. However it is precise enough to allow the verification of the concurrent data structures, and efficient enough to make the verification procedure feasible in practice. The main idea of the abstraction is to have a more precise description of the parts of the heap that are visible (reachable) from global variables, and to make a succinct representation of the parts that are local to the threads. Intuitively, a heap segment can be characterized by a TA with data constraints. More concretely, we will extract a set of heap segments between two cut-points which are same as cut-points in the forest automata approach. For each segment, we will store a summary of the content of the heap along the segment. This summary consists of three parts, each part contains different pieces of information, including

- the values of data fields of cells along the segment which have finite values. Note that, these values do not include values of cut-points, and
- the ordering among data values of fields of cells along the segment which have integer values.
- The sequence of observer registers that appear in the segment.

Figure 5.4 gives our symbolic abstraction of the heap in figure 5.1 where the observer register `z` is equal to 3. In this figure, there are four segments obtained from five cut-points. In each segment, the red box contains ordering information between data values of cells along the segment, the green box contains information about the values of fields `mark`, and `lock` of cells along the segment. Finally, the blue box contains the information about the sequence of observer registers. In this example, in the first segment between the two

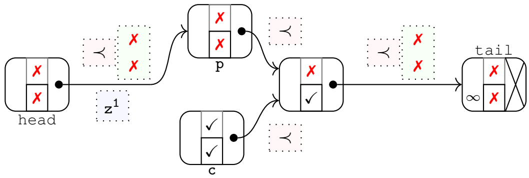

Figure 5.4. A symbolic representation of the configuration of `lazy set` in Figure 5.1

cells pointed to by `head` and `p`. The sequence of observer registers is $\mathtt{z}^1$, it means that between two cut-points there is exactly one cell whose data value of `val` field is equal to the observer register `z`.

This approach is very efficient but it is not optimal for complicated concurrent data structures like trees, lists of lists or skiplists. The approach is now specialized for SLLs.

In paper III, we present an approach which can handle concurrent programs implemented from simple to complex data structures. More precise, we can handle well-known data structures like singly-linked lists, skiplists, and lists of lists.

Heap abstraction for singly-linked list

The main idea of the approach is to represent a set of heap states by a set of fragments. A fragment represents two heap cells that are connected by a pointer field. For each of its cells, the fragment represents the contents of its non-pointer fields, together with information about how the cell can be reached from the program's pointer variables, under the applied data abstraction. The fragment contains both

- *local* information about the cell's fields and variables that point to it, as well as
- *global* information, representing how each cell in the pair can reach to and be reached from (by following a chain of pointers) a small set of globally significant heap cells.

A set of fragments represents the set of heap structures in which each pair of pointer-connected nodes is represented by some fragment in the set. Put differently, a set of fragments describes the set of heaps that can be formed by "piecing together" pairs of pointer-connected nodes that are represented by some fragment in the set. This "piecing together" must be both locally consistent (appending only fragments that agree on their common node), and globally consistent (respecting the global reachability information). Figure **??** shows a set of fragments that is sufficient to represents the configuration in

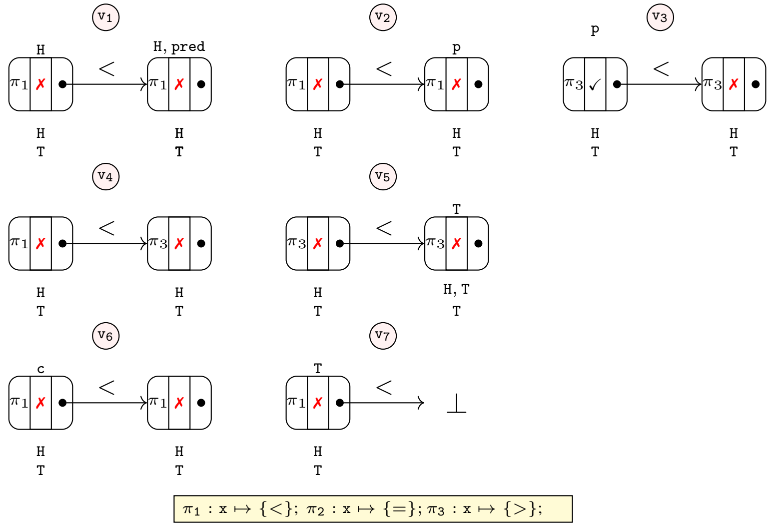

Figure 5.5. Fragment abstraction of the configuration in 5.1

Figure 5.1. There are 8 fragments, named $\mathtt{v}_1$, ..., $\mathtt{v}_8$. Fragment of $\mathtt{v}_8$) consists of a tag that points to $\perp$. All other fragments consist of a pair of pointer-connected tags. The two left-most cells in Figure **??** are represent by the level 1-fragment $\mathtt{v}_1$ in Figure 5.7. Here, the variable `preds`[3] is represented by `preds`[`higher`]. The mapping π_1 represents the data abstraction of the `key` field, here saying that it is smaller than the value 9 of the observer register. Note that, in our approach, the data abstraction is a mapping function from data value to a set of observer registers. The two left-most cells are also represented by a higher-level fragment, viz. v_6. The pair consisting of the two sentinel cells (with keys $-\infty$ and $+\infty$) are represented by the higher-level fragment v_7. In each fragment, the abstraction values of non-pointer fields are shown represented inside each tag of the fragment. The data relation is shown as a label on the arrow between two tags. Above each tag is pointer variables. The first row under each tag is reach from information, whereas the second row is reach to information.

Heap abstraction for skiplist

Let us illustrate how pairs of heap cells can be represented by fragments. Before going detail to the example, let us describe in short the definition of skiplist. A skiplist consists of a collection of sorted linked lists, each of which is located at a *level*, ranging from 1 up to a maximum value. Each skiplist node has a key value and participates in the lists at levels 1 up to its *height*. The skiplist has sentinel head and tail nodes with maximum heights and key values $-\infty$ and $+\infty$, respectively. The lowest-level list (at level 1) constitutes an ordered list of all nodes in the skiplist. Higher-level lists are increasingly sparse sublists of the lowest-level list, and serve as shortcuts into lower-level lists.

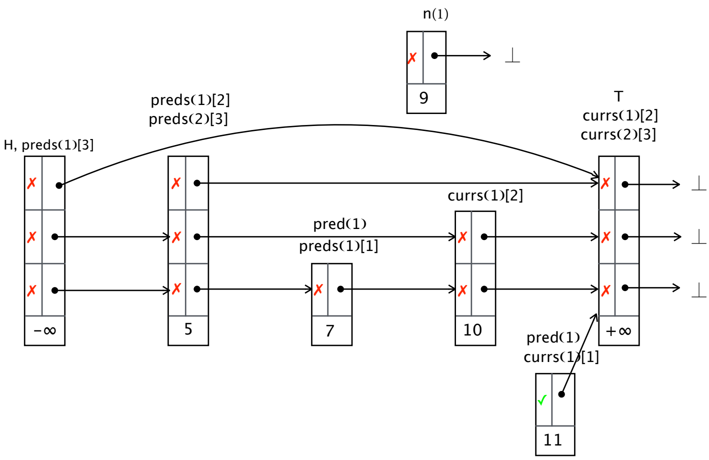

Figure 5.6. A concrete shape of 3-level skiplist

Figure 5.6 shows an example heap state of the skiplist algorithm with three levels. Each heap cell is shown with the values of its `mark` and `key` fields. Let us illustrate how pairs of heap nodes can be represented by fragments. Figure 5.7 shows a set of fragments that is sufficient to represents the configuration in Figure 5.6. There are 11 fragments, named $\mathtt{v}_1$, ..., $\mathtt{v}_{11}$. Three of these ($\mathtt{v}_9$, $\mathtt{v}_{10}$ and $\mathtt{v}_{11}$) consist of a tag that points to $\perp$. All other fragments consist of a pair of pointer-connected tags. The fragments $\mathtt{v}_1$, $\mathtt{v}_2$,$\mathtt{v}_3$,$\mathtt{v}_4$,$\mathtt{v}_5$,$\mathtt{v}_9$ and $\mathtt{v}_{10}$ are level-1-fragments, whereas the other fragments are higher level-fragments. The `private` field of the input tag of $\mathtt{v}_7$ is `true`, whereas the `private` field of other tags of other fragments are `false`. The two left-most cells in Figure 5.6 are represent by the level 1-fragment $\mathtt{v}_1$ in Figure 5.7. Here, the variable `preds`[3] is represented by `preds`[`higher`]. The mapping π_1 represents the data abstraction of the `key` field, here saying that it is smaller than the value 9 of the observer register. Note that, in our approach, the data abstraction is a mapping function from data value to a set of observer registers. The two left-most cells are also represented by a higher-level fragment, viz. v_6. The pair consisting of the two sentinel cells (with keys $-\infty$ and $+\infty$) are represented by the higher-level fragment v_7. In each fragment, the abstraction values of non-pointer fields are shown represented inside each tag of the fragment. The data relation is shown as a label on the arrow between two tags. Above each tag is pointer variables. The first row under each tag is reach from information, whereas the second row is reach to information.

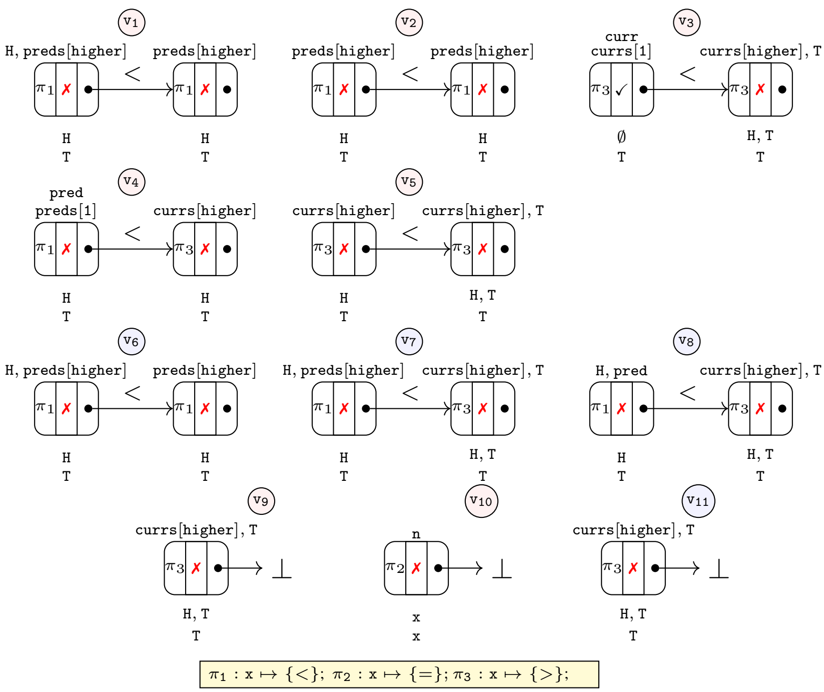

Figure 5.7. Fragment abstraction of skiplist algorithm

6. Conclusion and Future Work

We have presented, in this thesis, approaches to verify the complex problem of both sequential and concurrent heap manipulating programs.Such programs induce an infinite-state space in several dimensions: they (i) consist of an unbounded number of concurrent threads, (ii) use unbounded dynamically allocated memory, and (iii) the domain of data values is unbounded. (iv) consist of un unbounded number of pointers. In addition, the linearization points of some programs are not fixed. They are depended on the future executions of these programs. In this thesis, we focus on proving both safety properties, and linearization properties for the system, regardless of the value of this parameter. In order to prove safety properties, we defines an abstract model of the program, and we employ approximation techniques to that ignore irrelevant information so that we can reduce the problem into a finite-state model. In fact, we use an over-approximation, such that the abstract model cover all the behaviors of the original system. However, it might cover other behaviors which are not in the original system. If the bad states are not reached during the computation of reachable states of the abstract model, then the abstract model is considered safe, and so is the original system is also safe. If the bad state is reachble, we have to refine the abstraction. In order to verity linearization properties of a program, we add a specification which expresses its a data structure, using the technique of observers. In our approaches, the user have to provide linearization policies which specify how the program is linearized. We use a technique call `controller` to specify linearization policies. We then verify that in any concurrent execution of a collection of method calls, the sequence of announced operations satisfies the semantics of the data structure. This check is performed by an observer, which monitors the sequence of announced operations. This reduces the problem of checking linearizability to the problem of checking that in this cross-product, the observer cannot reach a state where the semantics of the set data structure has been violated. To verify that that the observer cannot reach a state where a violation is reported, we compute a symbolic representation of an invariant that is satisfied by all reachable configuration of the cross-product of a program and an observer.

There are two main possible lines of future work we would like to work on from this thesis. The first line is to extend the type programs we consider, by allowing more complicated data structures such as trees and design methods that allow the automatic synthesis of the controllers. Another possible line of work is to extend the view abstraction to multi-threaded programs running on machines with different memory models. Such hardware systems employ store buffers and cache systems that could be modeled using views. This is an interesting challenge since it would help programmers to write their code under a given memory model that is simpler to reason around, and verify that the behaviour of the program is the same under another less-restricted memory model.

My Contributions

References

[1] Parosh A. Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.

[2] Parosh A. Abdulla, Frédéric Haziza, Bengt Jonsson, and Ahmed Rezine. An integrated specification and verification technique for highly concurrent data structures. Technical Report ???, Brno ot Uppsala, 2013.

[3] Parosh Aziz Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 16(4):457–515, 2010.

[4] Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS96*, pages 313–321, Jul 1996.

[5] Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In *CONCUR10*, pages 86–101, 2010.

[6] Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Monotonic abstraction: on efficient verification of parameterized systems. *Int. J. Found. Comput. Sci.*, 20(5):779–801, 2009.

[7] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *CAV07*, volume 4590 of *LNCS*, pages 145–157. Springer, 2007.

[8] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Approximated context-sensitive analysis for parameterized verification. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *FORTE'09*, volume 5522 of *Lecture Notes in Computer Science*, pages 41–56. Springer, 2009.

[9] Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. All for the price of few (parameterized verification through view abstraction). In *VMCAI13*, volume 7737 of *Lecture Notes in Computer Science*, pages 476–495, 2013.

[10] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In *TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer, 2007.

[11] Parosh Aziz Abdulla, Noomene Ben Henda, Giorgio Delzanno, and Ahmed Rezine. Handling parameterized systems with non-atomic global conditions. In *VMCAI08*, volume 4905 of *LNCS*, pages 22–36. Springer, 2008.

[12] Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In *Proc. CONCUR '02, 13th International Conference on Concurrency Theory*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2002.

[13] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Information and Computation*, 160(1-2):109–127, July 2000.

[14] ParoshAziz Abdulla, Lukáš Holík, Bengt Jonsson, Ondřej Lengál, CongQuy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *ATVA13*, volume 8172 of *LNCS*, pages 224–239. Springer, 2013.

[15] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2001.

[16] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. *STTT*, 5(1):49–58, 2003.

[17] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *VMCAI02*, volume 2294 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2002.

[18] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Thread quantification for concurrent shape analysis. In *CAV08*, CAV '08, pages 399–413. SV, 2008.

[19] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 178–192, 2007.

[20] Jesse Bingham and Zvonimir Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI06*, volume 3855 of *Lecture Notes in Computer Science*, pages 207–221. Springer, 2006.

[21] Jesse D. Bingham and Alan J. Hu. Empirically efficient verification for a class of infinite-state systems. In *TACAS'05*, volume 3440 of *LNCS*, pages 77–92. Springer, 2005.

[22] Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large. In *CAV'03*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2003.

[23] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *ICALP*, volume 9135 of *LNCS*, pages 95–107, 2015.

[24] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV '04, 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.

[25] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 517–531, 2006.

[26] Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomás Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, pages 52–70, 2006.

[27] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. *CoRR*, abs/1502.07639, 2015.

[28] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. volume 77, pages 1006–1036, 2012.

[29] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI'06*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2006.

[30] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[31] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In *VMCAI06*, volume 3855 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2006.

[32] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 475–488. Springer, 2006.

[33] Robert Colvin, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, volume 4144 of *LNCS*, pages 475–488, 2006.

[34] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *CAV'01*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[35] G. Delzanno. Automatic verification of cache coherence protocols. In Emerson and Sistla, editors, *CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2000.

[36] Giorgio Delzanno. Verification of consistency protocols via infinite-state symbolic model checking. In *FORTE'00*, volume 183 of *IFIP Conference Proceedings*, pages 171–186. Kluwer, 2000.

[37] J. Derrick, B. Dongol, G. Schellhorn, B. Tofan, O. Travkin, and H. Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In *FM*, volume 8442 of *LNCS*, pages 200–214. Springer, 2014.

[38] Kamil Dudka, Petr Peringer, and Tomás Vojnar. Byte-precise verification of low-level list manipulation. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 215–237, 2013.

[39] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *CADE'00*, volume 1831 of *LNCS*, pages 236–254. Springer, 2000.

[40] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *POPL95*, pages 85–94, 1995.

[41] Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. On automated lemma generation for separation logic with inductive definitions. In *Automated Technology for Verification and Analysis - 13th International Symposium, ATVA 2015, Shanghai, China, October 12-15, 2015, Proceedings*, pages 80–96, 2015.

[42] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *LICS99*. IEEE Computer Society, 1999.

[43] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *SPIN'03*, volume 2648 of *LNCS*, pages 213–224. Springer, 2003.

[44] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. In *VMCAI06*, volume 3855 of *LNCS*, pages 49–64. Springer, 2006.

[45] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge and check... made efficient. In *CAV'05*, volume 3576 of *LNCS*, pages

394–407. Springer, 2005.
[46] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *J. Comput. Syst. Sci.*, 72(1):180–203, 2006.
[47] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.
[48] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Verifying pointer programs using graph grammars. *Sci. Comput. Program.*, 97:157–162, 2015.
[49] Jonathan Heinen, Thomas Noll, and Stefan Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. *Electr. Notes Theor. Comput. Sci.*, 266:93–107, 2010.
[50] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2005.
[51] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, July 1990.
[52] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
[53] IBM. System/370 principles of operation, 1983.
[54] IEEE Computer Society. IEEE standard for a high performance serial bus. Std 1394-1995, August 1996.
[55] Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, and Michael I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, June 15-18, 1997*, pages 226–236, 1997.
[56] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *CAV10*, volume 6174 of *LNCS*, pages 645–659. Springer, 2010.
[57] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.
[58] J. B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95(2):pp. 210–225, 1960.
[59] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.
[60] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 52–68, 2014.
[61] Nancy A. Lynch and Boaz Patt-Shamir. Distrbuted algorithms, lecture notes for 6.852, fall 1992, 1992.
[62] Nancy A. Lynch and Boaz-Path Shamir. Distributed algorithms, lecture notes for 6.852, fall 1992. Technical Report MIT/LCS/RSS-20, MIT, 1993.
[63] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics

combining heap structures and data. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 611–622, 2011.

[64] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 211–222, 2010.

[65] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC'06*, volume 4281 of *LNCS*, pages 183–197. Springer, 2006.

[66] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In *SAS'07*, volume 4634 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2007.

[67] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[68] M.M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC96*, pages 267–275, 1996.

[69] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 221–231, 2001.

[70] Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI07*, volume 4349 of *LNCS*, pages 299–313. Springer, 2007.

[71] Marcus Nilsson. *Regular Model Checking*. PhD thesis, Uppsala UniversityUppsala University, Division of Computer Systems, Computer Systems, 2005.

[72] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 711–728, 2014.

[73] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *CAV'02*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.

[74] Amir Pnueli, Sitvanit Ruah, and Lenore D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2001.

[75] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[76] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74, 2002.

[77] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San*

Antonio, TX, USA, January 20-22, 1999, pages 105–118, 1999.

[78] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[79] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[80] G. Schellhorn, J. Derrick, and H. Wehrheim. A sound and complete proof technique for linearizability of concurrent data structures. *ACM Trans. Comput. Log.*, 15(4):31:1–31:37, 2014.

[81] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In *CAV*, volume 7358 of *LNCS*, pages 243–259. Springer, 2012.

[82] Natasha Sharygina and Helmut Veith, editors. *Computer Aided Verification - 25th International Conference*, volume 8044 of *Lecture Notes in Computer Science*. Springer, 2013.

[83] IEEE Computer Society. Ieee standard for a high performance serial bus. std 1394-1995, August 1996.

[84] Boleslaw K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.

[85] Boleslaw K. Szymanski. Mutual exclusion revisited. In *Proc. Fifth Jerusalem Conference on Information Technology, IEEE Computer Society Press, Los Alamitos, CA*, pages 110–117. IEEE Computer Society Press, 1990.

[86] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. of VEPAS'01.

[87] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.

[88] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.

[89] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI09*, VMCAI '09, pages 335–348. SV, 2009.

[90] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *LNCS*, pages 450–464, 2010.

[91] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS86*, pages 332–344, 1986.

[92] T. Vojnar. private communication, June 1993.

[93] Wikipedia. Aba problem — Wikipedia, the free encyclopedia, 2008. [Online; accessed 22-June-2015].

[94] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL86*, POPL '86, pages 184–193. ACM, 1986.

[95] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 385–398, 2008.

[96] Karen Zee, Viktor Kuncak, and Martin C. Rinard. Full functional verification of linked data structures. In *Proceedings of the ACM SIGPLAN 2008 Conference*

on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pages 349–361, 2008.

[97] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *CAV*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer-Verlag, 2015.