1. Introduction

Computers have been used for 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 achieve without computers. However, sometime they do not behave exactly as we expect them to do. In some cases, the consequence could be very serious such as errors in banking systems and airplanes. The error is obviously not caused by the machine itself, but it seems to originate from the program that controls the machine. The error is normally called bugs. The probability of existing bugs in complex software systems is not low, in particular concurrent systems. They usually have complicated input and involve many features, they have to handle many concurrent operations which makes them difficult to design and make them perfect by human effort. Detecting and fixing software bugs are important tasks in software process in software industry. Leave bugs undetected in any project might in fact lead to subsequent bugs and other problems. It will become too costly to solve them or too daunting to attempt to. Therefore, the quality phase takes a substantial amount of resources, both in terms of time and manpower.

Some bugs are less serious than others. We can still use software programs with these bugs. For example, bugs appearing in computer games and online news. However, in the case of critical systems such as banking or software systems in airplanes, safety is the most important aspect and we have to ensure that there no bugs in either the software nor the hardware. The predominant method to improve software quality is *testing*. It is a dynamic analysis where a program is run under specific conditions, so-called test cases, and checking whether the result with a given input matches the expected output. The test cases are carefully designed to cover all possible cases of program executions. Similarly, we can check for correctness of a program by using a *model* of the program. The model can be extracted by removing all parts that are irrelevant for the tests, and can be used to *simulate* the executions. However, there is no guarantee to cover all possible executions. Therefore, we have to find the way to achieve a full coverage of program executions. When the domain of the input parameters is large or if the program is complex, the method will suffer from the state-space explosion problem.

Formal Verification

Computer programs are written based on human intuition, which is probably leads to programming errors. Current practice is to test programs on various sample inputs in the hope of finding any possibility of incorrect program behavior. There exist many approaches like testing, simulation, static analysis and simple debugging techniques, such as inserting assertions and print statements in the source code, which show the presence of software errors. Formal verification uses mathematical methods to prove that a program is correct. Formally, formal verification is the process of checking whether a software satisfies its predefined properties. There is a wide variety of properties to be checked for software programs, these properties can be either safety or liveness properties. Liveness properties state that program execution eventually reaches several desirable states at some point of execution, for example liveness properties can be "the postman delivers the letter to the recipient", "A sent message is eventually received". In order to specify liveness properties, it is needed to describe traces of events by using temporal logics, statistics, and probabilities. Checking aliveness property is done by repeatedly checking reachability of good situations in program executions. In contract, verifying safety property of a program is satisfied is reduced to checking that something bad will never happen in the execution of the program [48]. There are three state-of-the-art approaches for formal verification namely model checking, theorem proving and equivalence checking. The first approach model checking exhaustively explorer all possible states of the model which can be finite or infinite models where infinite sets of states can be represented finitely by using abstraction techniques). Equivalence checking method decides whether system is equivalent to its specification with respect to some notation of behavioral equivalence. Theorem proving is a technique where both the system and its desired properties are expressed in mathematical logic. Then, theorem proving will try to prove these properties. In this thesis though, we consider programs where the specification describes the bad behaviors. We concentrate on safety properties and try to design abstraction techniques to verify that a program including both sequential and concurrent program respects its specifications.

Verification of Concurrent Data Structures

Concurrent data structures that can be accessed and manipulated concurrently by many parallel threads are a central component of many parallel software applications. They should allow a large degree of parallelism among accessing threads to minimize serialization bottlenecks, while maintaining the appearance of atomic operations. Many modern programming languages provide libraries of concurrent data structures (e.g., the java.util.concurrent package and Intel Threading Building Blocks library) that are widely used. One of the main properties of concurrent data structures is linearizability. Linearizability is generally accepted as the standard correctness criterion for such concurrent data structure implementations. 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. The linearizability guarantee relieves the programmer from complex reasoning about possible interference among data-structure methods and removes the need to add explicit synchronization. Concurrent implementations of abstract data structures (stacks, queues, sets, etc.) are becoming more and more complex as implementations that increase the degree of concurrency are identified. This in turn is making linearizability verification harder. Existing approaches lack generality as they are limited to specific classes of concurrent data structures so far no technique (manual or automatic) for proving linearizability has been proposed that is both sound and generic. In this thesis, we focus on verifying safety properties including linearizability of concurrent data structures.

Research Challenges

In this thesis, we consider challenges in software verification. Our challenges are to automate its application to both sequential and concurrent programs that manipulate complex dynamic linked data structures. We have to deal with concurrent programs with an unbounded number of threads that concurrently access and manipulate a dynamically allocated shared heap where data stored in each heap cell can be in unbound domain. Such programs and algorithms are difficult to get correct and verify, since their shapes are complicated to represent and they typically employ fine-grained synchronization, replacing locks by atomic operations such as compare-and-swap, and are therefore notoriously difficult to get correct, witnessed. It is therefore important to develop efficient techniques for automatically verifying their correctness. This requires overcoming several challenges. This thesis presents simple and efficient techniques to verify that a concurrent implementation of a common data type abstraction, namely queue, stack, set, conforms to a simple abstract specification of its (sequential) functionality. The data structures we consider for these programs can be singly-linked lists, sets of linked lists or skip-lists. In order to deal with this problem, we have to deal with several combined challenges as follow.

Ⓐ Linearization points are not fixed, they depend on future executions of programs

Ⓑ The program is infinite-state in several dimensions:

- it consists of an unbounded number of concurrent threads,
- it uses unbounded dynamically allocated memory, and
- the domain of data values is unbounded.
- it consists of unbounded number of pointers

We present, in the next chapters, the general background about model checking, concurrent data structures. Thereafter, in the following chapter, we introduce in a stepwise manner how we cope with above challenges. In the last chapter, we summary and give future plans for our work.

2. Model Checking

The approach that we focus on this thesis is *model-checking*. This method will try to verify whether a model of the program satisfies its specification.

This approach was introduced by Emerson and Clarke [30] and by Queille and Sifakis [75]. The method requires a program and a property as input and then it extracts a model from the program. 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. 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. States and transitions together describe the *operational semantics*, that is, how every step of the system takes place in the model. The number of states and transitions can be finite or infinite Model-checking aims to explore the state-space entirely from some initial states. 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 deal with the state-space explosion problem. The choice of which transition to pick during the exploration

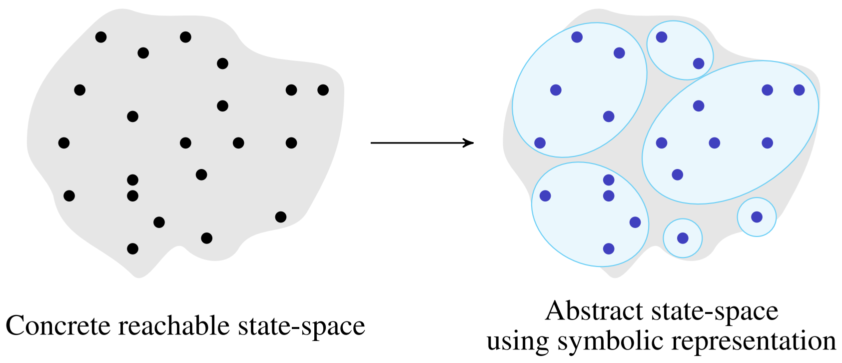


Figure 2.1. Symbolic Representation

can be crucial for the efficiency of the procedure. In some cases, exploring all orderings of events is not necessary because some states can be re-visited. *Partial order* techniques aim at detecting and avoiding redundant situations, while retaining important dependencies among actions. They however do not reduce the state-space. The main approach called *symbolic representation* to solve the state-space explosion problem is to avoid representing concretely all states of the system. the approach group several states together form sets of states and represent them by single states described in figure 2.1. The grouping process is performed by *dropping* irrelevant details (as opposed to disregarding them) based on properties that we want to verify. Figure 2.2 show the architecture of symbolic model checking in which it is sound to prove safety of the abstract model in order to imply safety for the original system, since all the behaviours of the latter are represented in the former. The challenge is to find over-approximations that do not 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 or from the concrete system itself.

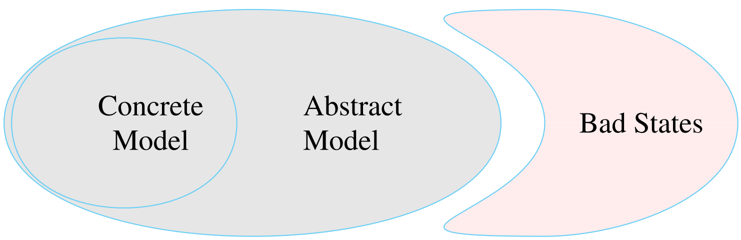


Figure 2.2. Symbolic model checking architecture

To palliate to the imprecision caused by a too coarse over-approximation, it is possible to analyze 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 following problem statement.

3. Concurrent Data Structures

A concurrent data structure is a way of storing and organizing data for access and manipulated by multiple computing threads (or processes) on a single computer. The implementation of a concurrent data structure usually requires writing a set of operations that access and manipulate instances of that structure concurrently. Processes are sequential, each process applies a sequence of its operations to a share structures. A process can halt or have various speed. Actually, we do not tell whether a process is halted or is running fast or slowly. Each data structure has a type which defines a set of possible values and a set of operations such as queue, stack or set. Each object has a sequential specification that defines how the object behaves when its operations are invoked one at a time by a single process. For example, the behavior of a queue object can be specified by requiring that enqueue insert an item in the queue, and that dequeue remove the oldest item present in the queue. In a concurrent system, however, an object's operations can be invoked by concurrent processes, and it is necessary to give a meaning to interleaved operation executions.

An object is linearizable if each operation appears to take effect instantaneously at some point between the operation's invocation and response. Linearizability implies that processes appear to be interleaved at the granularity of complete operations, and that the order of nonoverlapping operations is preserved. The notion of linearizability generalizes and uniles a number of ad hoc correctness conditions in the literature, and it is related to (but not identical with) correctness criteria such as sequential consistency and strict serializability.

There are four main techniques to construct concurrent data structures namely coarse-grained locking, fine-grained locking lazy, synchronization and lock-free programming. In the coarse-grained locking technique, a single lock is used to synchronize every access to an object. Coarse-grained synchronization is easy to reason about, however it works well when levels of concurrency are low, but if too many threads try to access the object at the same time, then the object becomes a sequential bottleneck, forcing threads to wait in line for access. In the second technique, they split the object into independently synchronized components, ensuring that method calls interfere only when trying to access the same component at the same time. In the third technique, the task of removing a component from a data structure can be split into two phases: the component is logically removed simply by setting a tag bit, and later, the component can be physically removed by unlinking it from the rest of the data structure. The lock-free technique help us to eliminate locks entirely, it relies on built-in atomic operations such as `compareAndSet()` for synchronization.

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.

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

```
ctn(e):
local c
c := Head;
while (c.val < e)
  c := c.next
b := c.mark
if (!b
     && c.val = e)
  return true;
else
  return false;
```

```
locate(e):
local p, c
while (true)
  p := Head;
  c := p.next;
  while (c.val < e)
    p := c;
    c := c.next
  lock(p); lock(c);
  if (! p.mark &&
      ! c.mark&&
      p.next=c)
    return (p,c);
  else
    unlock(p);
    unlock(c);
```

```
add(e):
local p, c, n, r
(p,c) := locate(e);
if (c.val <> e)
  n :=
  new Node(
       0,e,c,false);
  p.next := n;
  r := true;
else r := false;
unlock(p); unlock(c);
return r;
```

```
rmv(e):
local p, c, n, r
(p,c) := locate(e);
if (c.val = e)
  c.mark := true;
  n := c.next;
  p.next := n;
  r := true;
else r := false;
unlock(p); unlock(c);
return r;
```

Figure 3.1. Lazy Set Algorithm

As an example, Fig. 3.1 depicts a program `Lazy Set` [50] that implements a concurrent set containing integer elements. The set is implemented as an ordered singly linked list. The program contains three methods, namely `add`, `rmv`, and `ctn`, corresponding to operations that respectively add, remove, and check the existence of an element in the set. Each method takes an argument which is the value of the element, and returns a value which indicates whether the operation has been successful or not. For instance, the operation `add`(e) returns the value *true* if e is not already a member of the set. In such a case a new cell with data value e is added to its appropriate position in the list. If e is already present, then the list is not changed and the value `false` is returned. The program also contains the subroutine `locate` that is called by the three methods. A cell in the list has three fields `mark`, `lock`, and `val`. The `rmv` method first logically removes the node from the list by setting the `mark` field, before physically removing the node. The `ctn` method is wait-free and traverses the list ignoring the locks inside the cells. 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 all keys that may be inserted in the set.

Linearizability

In a concurrent program, the methods of the different executing threads can overlap in time. Therefore, the order in which they take effect is ambiguous. The program statements in each method are totally ordered. Whereas, statements from different methods in different executing threads might form a partial order. This partial order raise the difficult of reasoning about program execution. One of the main correctness criterion of a concurrent program is linearizability, which 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 often a moment where the effect of the method becomes visible to other threads. 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.

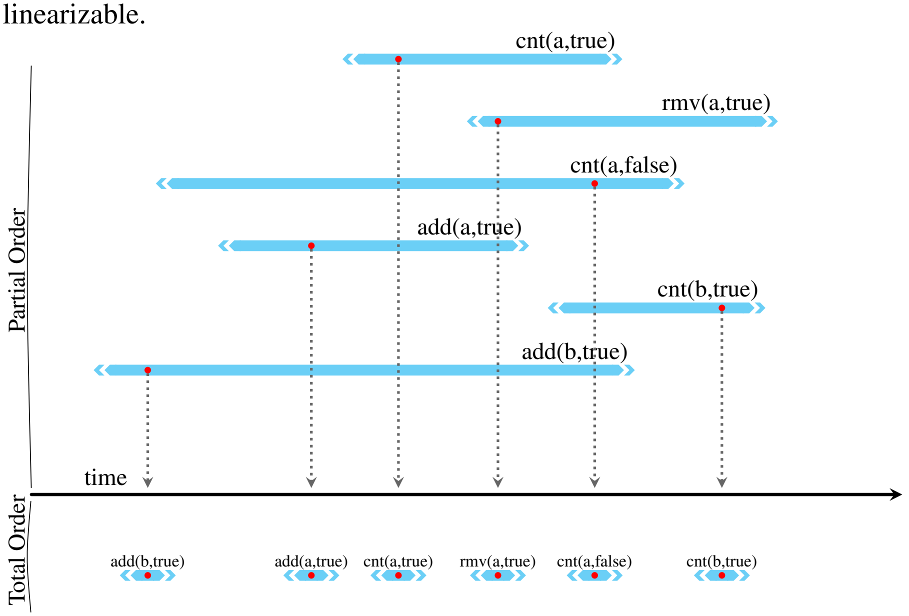


Figure 3.2. Linearizability, where the commit points are marked with •.

The 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 behavior of the set. A linearization point normally stays inside the code of the method. However, in some cases, it it is located in the code of another method depending on the execution path.

4. Verification of Linearization

In order to derive the totally ordered execution from a concurrent execution, each method is instrumented to generate a so-called abstract event whenever a linearization point is passed. Right after the program pass the linearization point, the abstract event is communicated to an external *observer*, which records the sequences of abstract events from the code execution. In the next section, we introduce the notion of observer, which essentially separates good traces of events from bad ones. Several observers shall be used to specify the safety property.

Observers

We specify the serial semantics of data structures by observers, as introduced in [2]. Observers are finite automata extended with a finite set of observer registers that assume values in $\mathbb{Z}$. At initialization, the registers are nondeterministically assigned arbitrary values, which never change during a run of the observer. Transitions are labeled by linearization events that may be parameterized

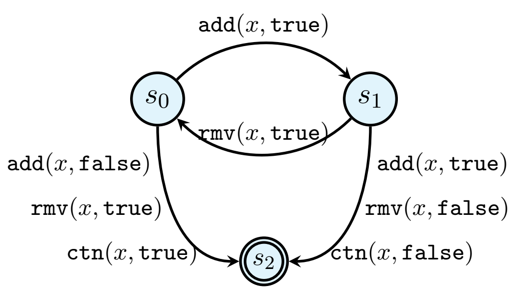


Figure 4.1. A stack observer

on registers. Observers are used as acceptors of sequences of linearization events. The observer processes such sequences one event at a time. If there is a transition, whose label, after replacing registers by their values, matches the event, 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 data structures such as sets, queues, and stacks. The observer is defined in such a way that it accepts precisely those sequences of abstract events 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 behavior of a set

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. However, for a large class of linearizable implementations, the LPs are not fixed in the code of their methods, but depend on actions of other threads in each particular execution. This happens, e.g., for algorithms that employ various forms of helping mechanisms, in which the execution of a particular statement in one thread defines the LP for one or several other threads. For example, in the the `Lazy Set` algorithm, the linearization point of unsuccessful `ctn` method is not fixed in the code of the method. It stays in the code of the `add` method. There have been several previous works dealing with the problems of non-fixed linearization points [96, 33, 32, 81, 37, 80, 89, 27]. However, they are either manual approaches without tool implementation or not strong enough to cover various types of concurrent programs. In this thesis we handle non-fixed linearization points by providing semantic for specifying linearization policies. The linearization point of a thread may be defined in two ways: (i) The thread may define its own linearization point, and in that case may also help other threads define their own linearization points. (ii) The thread may be helped by other threads. The helping mechanism may contain complicated patterns. For instance, the helping thread may broadcast a message to the other threads (e.g., the `Lazy Set` algorithm). Both the helping and the helped threads will then interact with the observer to communicate their parameters and return values. In such a case, the helping thread may be able to help an unbounded number of threads (all those who can be helped in the current configuration). In other cases, the helping thread may explicitly linearize for the helped thread, which means that the helped thread itself need not communicate with the observer. Furthermore, a given algorithm may use several of these patterns to define its linearization points.

To specify the linearization patterns, we equip each method with a *controller* whose behavior is defined by a set of rules which are described detail in paper II. The controller is occasionally activated by the thread, and helps organize the interaction of the thread with other threads as well as with the observer. More precisely, some statements in a method are declared to be *triggering*. If

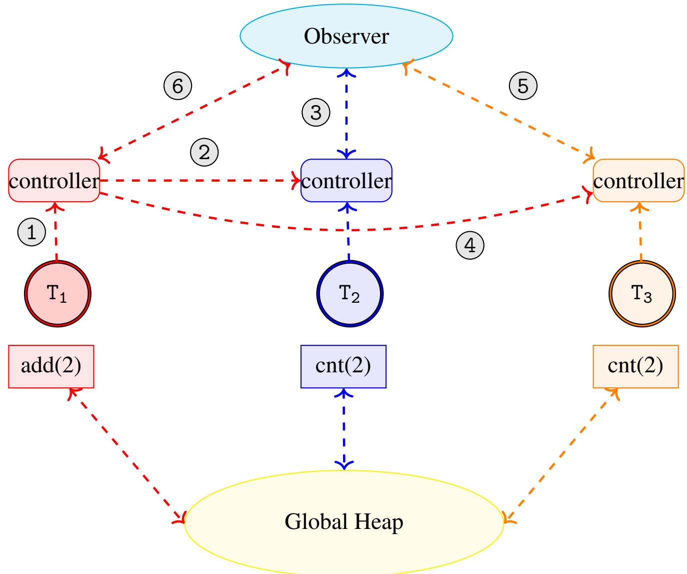


Figure 4.2. An example of linearization policies of `Lazy Set` algorithm

a triggering statement is executed then the controller of the thread will also be executed simultaneously.

Let us describe an example of how the `controller` works in the `Lazy Set` algorithm, the linearization point of the successful `add` is statically defined at the line 4 in the code of the method. When the thread T_1 reaches the triggering statement at line 4 at the step 1. Before a successful `add` operation is communicated to the observer 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. The controller will help other threads by to linearize. This is done by *broadcasting* a message to the threads T_2, T_3 which is executing `ctn` in steps 2, 3. These threads then inform the observer that the element e is not in the list in 3, 5 respectively. Note that T_2, T_3 can get message in any order. The detail of controller is described in paper II.

5. Shape Analysis

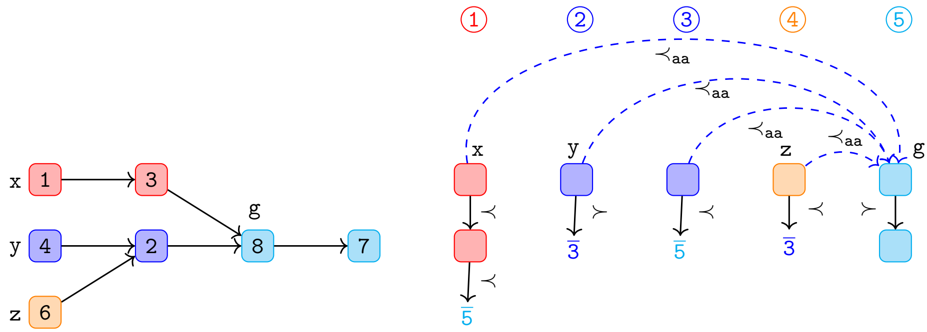
Pointers and heap-allocated storage are features of all modern imperative programming languages. However, they are ignored in most formal treatments of the semantics of imperative programming languages because their inclusion complicates the semantics of assignment statements: an assignment through a pointer variable (or through a pointer-valued component of a record) may have far-reaching side effects. Works that have treated the semantics of pointers include [5, 42, 43, 45]. These far-reaching side effects also make program dependence analysis harder, because they make it difficult to compute the aliasing relationships among different pointer expressions in a program. Having less precise program dependence information decreases the opportunities for automatic parallelization and for instruction scheduling. The usage of pointers is error prone. Dereferencing NULL pointers and accessing previously deallocated storage are two common programming mistakes. The usage of pointers in programs is thus an obstacle for program understanding, debugging, and optimization. These activities need answers to many questions about the structure of the heap contents and the pointer variables pointing into the heap. By shapes, we mean descriptors of heap contents. Shape analysis is a generic term denoting static program-analysis techniques that attempt to determine properties of the heap contents relevant for the applications mentioned above A difficult challenge in verifying heap manipulation programs is to handle infinite sets of reachable heap configurations. The area of verifying programs with dynamic linked data structures has been a subject of intense research for quite some time. Currently, there are several competing approaches for symbolic heap abstraction. The first approach is based on the use of logics to present heap configurations. The logics can be separation logic [76, 64, 19, 94, 38, 28, 60, 72, 41], 3-valued logic [78], monadic second- order logic [69, 55, 63] or other [77, 95]. Another approach is based on the use of automata. In this approach, elements of languages of the automata describe configurations of the heap [26, 25]. The last approach that we will mention is based on graph grammars describing heap graphs [49, 48]. 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).

Among the works based on separation logic, the work, such as [19, 94, 60] proposed more efficient approaches. The reason for that is that their approaches effectively decomposes the heap into disjoint components and process 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 by automata 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 heaps on a tree structure, and represent non-tree edges by using regular "routing" expressions. These expressions describe how the target can be reached from the source using tree edges. Finite tree transducers are used to compute set of reachable configurations, and symbolic configuration is abstracted collapsing certain states of the automat. 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. TVLA (Three-Valued Logic Analyzer) [78] is the first and one of the most popular shape analysis method. 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 nodes, each of them representing a set of concrete nodes. The shape of the heap is characterized by a set of usersupplied predicates. The method is not fully automatic, 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 skiplist, trees, or lists of lists.

Contribution

In this thesis, we proposed three approaches for heap abstractions. In paper I, we proposed a novel approach of representing sets of heaps via tree automata (TA). In our representation, a heap is split in a canonical way into several tree components whose roots are the so-called cut-points. Cut-points are nodes pointed to by program variables 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 [15]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by the so called forest automata (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we 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).



(a) A graph

(b) Forest Presentation

Figure 5.1. A graph and its forest representation

In addition, we express relationships between data elements associated with nodes of the heap graph by two classes of constraints. Local data constraints are associated with transitions of TA and capture relationships between data of neighboring nodes in a heap graph; they can be used, e.g., to represent ordering internal to some structure such as a binary search tree. Global data constraints are associated with states of TA and capture relationships between data in distant parts of the heap. This approach was applied to verification of sequential heap manipulation programs. This approach is general and fully automatic, it can handle many types of sequential programs without any manual step. However, due to the complexity of tree automata operations, this approach is not suitable to handle concurrent programs where a large number of states and computation are needed. Figure 5.1 shows an example of how to represent a heap by a set of tree automata. Figure 5.1(a) shows an example of a heap where nodes whose values are $\boxed{1}$, $\boxed{4}$, $\boxed{6}$, $\boxed{2}$, $\boxed{8}$ are cut-points, and x, y, z are local pointer variables and g is global pointer variable. Figure 5.1(b) shows its forest representation. In the forest representation, there are five TAs in which the TAs 1 and 3 refer to the root of the last TA 5, and both TAs 2 and 4 refer to the root of TA 3. The local data constraints are located along the solid arrows between nodes, whereas global constraints are located along the dashed arrows. In this figure, the global constraints $\prec_{aa}$ means that all nodes in the left hand side are smaller than all nodes in the right hand side. We just show here small examples of data constraints, the detail about different types of constraints can be found in paper I.

In paper II, we provide a symbolic encoding of the heap structure, that is less precise than the approach in paper I. However it is precise enough to allow the verification of the concurrent algorithms, 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. More concretely, we will extract a set of heap segments, where the end points of a segment is pointed to by a cut-point which is reachable from global variables. A cut-point in this approach is a reachable node from global variable, and pointed by a global variables or having more than two incoming pointers. For each segment, we will store a summary of the content of the heap along the segment. This summary consists of two parts, each part contains different pieces of information, including the values of the cell variables if they have finite values, and the ordering among them if they are integer variables. The first part summaries information between the end point and its predecessor, whereas the second part summaries information between the start point and the predecessor node. For each given program, the set of possible abstract shapes insight and hence the verification procedure is guaranteed to terminate. This approach is very efficient but it is not optimal for complicated concurrent data structures like trees, lists of lists or skiplists. Figure 5.2 gives our summary abstraction of the heap in figure 5.1(a). In this approach, $\boxed{1}$, $\boxed{4}$, $\boxed{6}$, $\boxed{8}$ are cut-points. The node $\boxed{2}$ is not a cut-point like the approach in paper I because its not reachable from the global variable g. In each heap segment, the fist part is described by the white box, and the second part is described by the gray box.

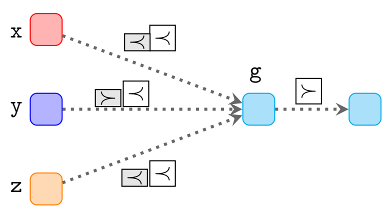


Figure 5.2. A summary representation

In paper III, we present an approach which can handle concurrent programs implemented from simple to complex data structures. In our fragment abstraction, we represent the part of the heap that is accessible to a thread by a set of fragments. A fragment represents a pair of heap cells (accessible to `th`) that are connected by a pointer field, under the applied data abstraction. The fragment contains both (i) *local* information about the cell's fields and variables that point to it, as well as (ii) *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).

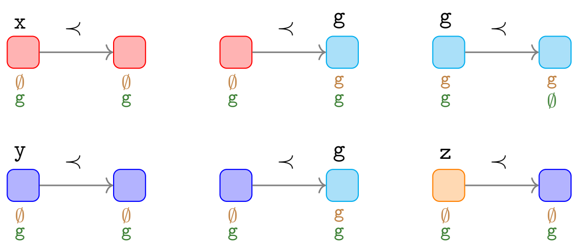


Figure 5.3. Fragment abstraction

Let us illustrate how pairs of heap nodes can be represented by fragments. Figure 5.3 shows the set of fragments abstracted from the heap in 5.1(a). In each fragment, the ordering between two keys of two nodes is shown as a label on the arrow between two tags. Above each tag is pointer variables. The first brown row under each tag is `reachfrom` information, whereas the second green row is `reachto` information.

Summaries of Papers

My Contributions

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.

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] IEEE Computer Society. Ieee standard for a high performance serial bus. std 1394-1995, August 1996.
[83] 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.
[84] 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.
[85] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. of VEPAS'01.
[86] R.K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.
[87] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2008.
[88] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI09*, VMCAI '09, pages 335–348. SV, 2009.
[89] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *LNCS*, pages 450–464, 2010.
[90] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS86*, pages 332–344, 1986.
[91] T. Vojnar. private communication, June 1993.
[92] Wikipedia. Aba problem — Wikipedia, the free encyclopedia, 2008. [Online; accessed 22-June-2015].
[93] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL86*, POPL '86, pages 184–193. ACM, 1986.
[94] 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.
[95] 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.
[96] 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.

Index