Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Formal System Analysis Group

Modeling and Verifying Concurrent Data Structures

Master Thesis

Roxana Paval / 0834605

Supervisors:
dr. S.P. (Bas) Luttik
dr.ir. T.A.C. (Tim) Willemse

Eindhoven, January 2018

Abstract

Concurrent data structures can be used to communicate between parallel processes in a system. The challenge in manipulating these objects arises from the many possible ways in which the processes can interleave. To ensure correct executions, the system should fulfill linearizability. Verifying linearizability consists of checking that every concurrent execution is equivalent to some sequential execution that respects the runtime ordering of methods. This work proposes building two process specifications of the object using the mCRL2 language. The concrete specification is built according to the implementation of the concurrent data structure, while the abstract specification is linearizable by construction. Then, linearizability can be tested by checking that their respective labeled transition systems, generated from the mCRL2 tool, are equivalent. This approach was applied on a number of concurrent data structures, and it detected both correct and faulty implementations.

Contents

Listings

Glossary and Acronyms

abstract specification A specification obtained from the concrete specification by adding an atomic layer; by this construction, the specification is linearizable. 2, 13

CAS Compare-And-Swap. 16

concrete specification A specification that formalizes the implementation of the concurrent data structure; this includes the execution histories. 1, 23

event An invocation or a response. 4

history A sequence of events. 5

LP linearization point. 1, 12

LTS labeled transition system. 2, 7

prime action An action that is either an invocation or response marked with prime. 23

1. Introduction

The advancements realized in multiprocessor architectures have made it desirable to build concurrent systems, where parallelism is employed. Specifically, the system consists of a number of threads running in parallel, where each thread is a sequential process. These processes can communicate either via message-passing or via shared-memory, where the second method of communication is the focus of this work.

In a shared-memory architecture, the data is captured by shared objects, which are data structures in the memory. The data structure has a type that determines the values that can be stored, and a set of methods that facilitate interaction with it. Since this data structure can be manipulated by parallel threads, it should support concurrent execution of methods. Data structures that have been designed to allow for concurrent executions are called *concurrent data structures*. The challenge that arises when designing for concurrency stems from the many possible ways in which processes can interleave. To ensure that the data accessed and written in an object is correct, synchronization mechanisms are employed. These mechanisms safeguard the data by allowing threads to manipulate it only under certain conditions, such as accessing the object in an exclusive manner or verifying that the object has not changed from the previous access. To guarantee that the concurrent executions are correct, the behavior of the concurrent data structure should be characterized through a correctness property.

All notions of correctness for concurrent data structures are based on some notion of equivalence with sequential behavior [14]. The reason behind this notion is quite straightforward: sequential data structures can be defined in terms of pre- and post-conditions, since the methods of the data structure are executed one by one. Thus, any intermediate states that may occur while a method is still executing may be safely ignored. However, this is not the case for a concurrent data structure, since its methods can be invoked by parallel processes. In this case, the intermediate states might influence the net effect of the methods, and cannot be ignored anymore.

One correctness property that applies to concurrent systems is *linearizability*. Informally, the observable behavior of a linearizable system illustrates that changes on the system occur instantaneously at a given point, where a change on the system represents a modification of one of its respective objects. Thus, it is desirable for a system to be linearizable when dealing with concurrent processing. In this context, the execution of a method starts with an *invocation*, which provides the required input, and concludes with a *response*, which provides some output. The method takes effect at some moment between its invocations and its response, where this moment is called a *linearization point (LP)*. Linearizability can apply to systems where multiple objects are shared, and it maintains the modularity of the system. The reason behind this is that a concurrent system is linearizable if and only if every object is linearizable **??**.

The most intuitive approach of proving linearizability is identifying the linearization point of each method, and checking that this point is the only moment where the object is modified. This approach would require fixed LPs, meaning that there is one statement that modifies the data structure in all possible executions. However, there are concurrent methods that contain non-fixed LPs ([12], [11], [9]), meaning that the LPs might depend on the current execution history or other characteristics of the system. As mentioned above, an object can be shown to be correct by establishing an equivalence between its concurrent behavior and its sequential behavior. Thus, proving that the object is linearizable boils down to verifying that every possible execution histories is consistent with some sequential execution histories. In this context, another approach to proving linearizability is using trace refinement checking [17], where a concrete specification

refines an abstract specification. The concrete specification represents the behavior of the system. The abstract specification restricts the execution histories of the object to those that are linearizable. Then, the set of execution traces of the concrete specification should be a subset of the set of execution traces of the abstract one in order to establish linearizability. This approach does not require identifying linearization points. The work in this report builds on this approach, by building concrete specifications of the concurrent data structures and verifying their equivalence with the abstract specifications.

The specifications in this work are built using mCRL2, which is a process algebra with an associated toolset [10]. A process algebra can be used to formally specify concurrent systems, by describing the behavior of the system through user-defined actions. Complex behavior can be specified using operators to compose actions, as well as processes. The mCRL2 language contains operators to define parallel behavior, and it can include data in the processes and actions. Based on a specification, mCRL2 can generate a labeled transition system (LTS), which serves as the underlying semantic model of the system. The equivalence between specifications is then expressed on the labeled transition systems. One limitation of this approach is verifying concurrent data structures with infinite domains or unbounded number of processes. While this type of data structures can be modeled in mCRL2, the corresponding LTS cannot be generated.

The methodology described here can be applied on different classes of concurrent data structures. The semantic models can be generated for objects that are used by a finite client, namely a client consisting of a fixed number n of threads running in parallel, where each thread calls non-deterministically a fixed number m of methods. Applying this methodology on a number of concurrent data structures ([14], [21], [20]) provides expected results, namely both linearizable and faulty implementations are detected.

Chapter 2 introduces the essential concepts of this work, namely defining and verifying linearizability, presenting the mCRL2 language and introducing the semantic model. Chapter 3 presents Treiber's stack, which serves as a motivating example. Chapter 4 formalizes the methodology of modeling and verifying concurrent data structures in mCRL2. Chapter 5 showcases the models of various data structures, while Chapter 6 presents the results of verifying them. Chapter 7 discusses related work, and Chapter 8 concludes the report.

2. Preliminaries

A system can consist of a number of threads running in parallel, where a thread is considered to be a sequential process. The parallelism of such a system arises from the possible interleavings occurring when operations are executed by the different threads. The communication amongst these threads can happen either via message-passing or via shared memory. This work focuses on communication via shared memory. This shared memory can be represented as a data structure. However, not all data structures allow correct operations when accessed in a parallel system. Data structures that have been designed to support concurrent access (i.e. reads and writes) are called *concurrent data structures* [19].

To define the correctness of a system, the notion of safety should be introduced. In general, safety of a program or a system states that some unwanted (i.e. bad) behavior should never happen. In this context, sequential correctness can be expressed by some safety property [14]. This can be achieved because the semantics of a sequential data structure can be expressed in a straightforward manner, by defining how the states change according to the methods. For example, assume a counter is provided as the sequential data structure. The only method allowed on the counter is *increase*. The transition function is quite simple: if the system is in a state where the counter has value x, then after executing *increase*, the counter will have value $x + 1$. This reasoning does not hold for a concurrent data structure, due to many possible interleavings that can happen on the data structure. An example is shown in Figure 2.1, where the method *increase* is executed twice, both in a sequential setting and a concurrent setting. While in the sequential processing, the internal actions of the method are processed one at a time, in the concurrent processing, it can happen that the internal actions are ordered in such a way that information gets overwritten. To mitigate for this kind of information loss, a synchronization mechanism should be used. Then, the data structure can be checked whether it satisfies a correctness property.

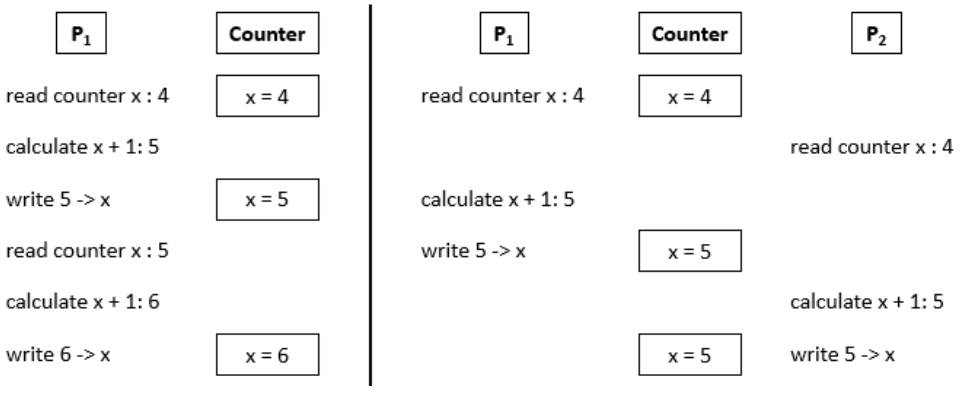

Figure 2.1: Left: Sequential processing. Right: Parallel processing

Linearizability is such a correctness property. In general, correctness properties are established by finding an equivalence between the concurrent behavior and a sequential behavior. Hence, reasoning about a concurrent data structure could be transformed into simpler reasoning about its sequential equivalent. Additionally, verifying that a concurrent data structure is linearizable ensures that all possible interleavings of the operations of this data structure will reach a correct state. The next section provides a formal definition of linearizability. Concurrent data structures should be abstracted into a model. Subsequent sections describe a way to represent and model processes. Finally, methods to verify linearizability are given.

2.1. Linearizability

Linearizability is a correctness property for concurrent data structures, which ensures that certain undesirable behavior cannot occur. Intuitively, when reasoning about this type of correctness property, two requirements seem to make sense. First, each operation should appear to take effect instantaneously, to ensure that changes on the system (i.e. data structure) are consistent. Second, the order of non-concurrent operations should be preserved. Linearizability meets these requirements, providing the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response [15].

To define linearizability, a few notions should be introduced. Firstly, a concurrent system consists of a collection of sequential threads of control called processes that communicate through shared data structures called objects. Each object has a type, which defines the set of possible values, and a set M of primitive methods that provide the only means to create and manipulate that object [15]. This idea is demonstrated by the following example.

Example 2.1. (Stack) A stack is a data structure, whose elements are processed in a last-in-first-out order. Implementations of the stack usually have a pointer referring to the latest element added, called *top*. The stack then allows for two methods: adding an element at the top of the stack (i.e. push), or removing and returning the element from the top (i.e. pop). The stack is stored in the shared memory as a linked list, and it can be accessed by many threads at the same time. However, without a manner to properly handle concurrent access and modification of the data structure, the stack might not behave correctly. An example of a stack, storing integer values, can be seen in Figure 2.2.

The stack object stores elements of type T, and the set M consists of following methods: {push(T a), pop(): T}. Additionally, the stack is a recurring example throughout the chapter.

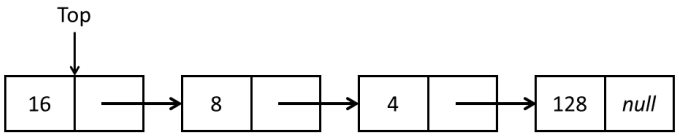

Figure 2.2: Example of a stack with integer elements

Linearizability was formally defined by Herlihy and Wing [15], and this definition concerns arbitrarily many objects. However, we assume only one shared object in the system. This assumption is based on the fact that objects do not interact with, and thus influence, each other. Hence, information about the name or the type of the object may be stripped from subsequent examples and concepts.

The only option for manipulating an object is through calling its methods. Each method call has an *invocation* and a *response* associated with it. An invocation denotes the moment when the method call is initiated. A response denotes the moment when the method finishes its execution. Let x be an object, and let M be the set of methods allowed by the object. Each method has associated with it an invocation event and a response event. The invocation event might have some arguments, while the response event might return some values. Let S denote the set of processes that can interact with the object.

Definition 2.2. (Invocation and response events) An *invocation* is formally represented as $\langle x, m_{inv}(args*), P \rangle$, where x is the object name, $m \in M$ is the method name, $args*$ is a sequence of arguments, and $P \in S$ is the process calling the method. A *response* is formally represented as $\langle x, m_{res}(res*), P \rangle$, where $res*$ may be a sequence of results. An *event* is either an invocation or a response [15].

Example 2.3. (Events for the stack methods) Consider the stack x shown in Figure 2.2. Popping the top of this stack may give rise to the following events: $\langle pop_{inv}(), P\rangle$, $\langle pop_{res}(16), P\rangle$. This method call terminates successfully and provides as a result 16, i.e. the value from the top of the stack.
Pushing the element 64 may render the following events: $\langle push_{inv}(64), P\rangle$, $\langle push_{res}(), P\rangle$. This method call terminates successfully and returns no values, thus the sequence of results is empty.

Herlihy and Wing introduced the notion of a history. Formally, a single execution of a concurrent system can be modeled by a *history*, which is a finite sequence of events. In a history, a response matches an invocation if it is the first response whose object name and process name agree [15].

Definition 2.4. (Matching response) Let e_k be an invocation in a history $e_0, e_1, e_2, \cdots, e_n$, meaning that e_k is of the shape $\langle x, m_{inv}(args_1*), P\rangle$. The matching response is the event e_l, where $k < l$, e_l is of the shape $\langle x, m_{res}(res_1*), P\rangle$, and $\nexists i$ with $k < i < l$ such that $e_i = \langle x, m_{res}(res_2*), P\rangle$.

Given a history H, an invocation might not always have a matching response. When this happens, the invocation is said to be *pending*. Furthermore, $complete(H)$ is the maximal subsequence of H without pending invocations [15].

Example 2.5. (Complete history) Consider the following history on the shared stack x:
$H = \langle push_{inv}(4), P\rangle$, $\langle push_{inv}(2), Q\rangle$, $\langle pop_{inv}(), R\rangle$, $\langle push_{res}(), P\rangle$, $\langle pop_{res}(4), R\rangle$.
Then, $complete(H) = \langle push_{inv}(4), P\rangle$, $\langle pop_{inv}(), R\rangle$, $\langle push_{res}(), P\rangle$, $\langle pop_{res}(4), R\rangle$.

Definition 2.6. (Sequential history) A history H is sequential when the following three conditions are met: (1) the first event of H is an invocation; (2) each invocation, except possibly the last, is immediately followed by a matching response; (3) each response is immediately followed by an invocation.

Equivalently, a history is sequential when it is composed of alternating invocations and responses, where an invocation and its succeeding response are matching. In this context, a history that is not sequential is said to be *concurrent*.

The histories can be projected on a given process or on a given object, resulting in different subhistories. Thus, a *process subhistory*, denoted $H|P$, of a history H is the subsequence of all events in H executed by process P. An *object subhistory* $H|x$ is similarly defined for an object x. The notion of equivalence among histories uses process subhistories: two histories H and H' are *equivalent* if for every process P, $H|P = H'|P$ [15].

Furthermore, processes are assumed to be represented as sequential threads. Thus, each process calls a sequence of methods on objects, alternatingly issuing an invocation and then receiving a matching response. This is the underlying concept for the notion of well-formed histories. Thus, a history H is *well-formed* if each process subhistory $H|P$ of H is sequential. All histories considered in this work are assumed to be well-formed [15].

In a history H, an *operation* o is defined as a triple consisting of an invocation and its matching response, as well as an execution id, which is a natural number representing the position of the operation in the process subhistory. Let eId be the number of operations that have already finished on the process plus 1. A shorthand notation is used for denoting the events in an operation o, namely $inv(o)$ denotes the invocation of o, $res(o)$ its response. Thus, an operation $o = \langle inv(o), res(o), eId\rangle$. Furthermore, a process subhistory can be expressed as follows: $o_1, o_2, ..., o_k$, where o_i are operations, for $1 \leqslant i \leqslant k$. Hereafter, each operation is uniquely identified by the combination of the process id and the execution id.

Example 2.7. (Histories) Consider the following histories:

H_1	H_2
$\langle pop_{inv}(), P\rangle$,	$\langle push_{inv}(256), Q\rangle$,
$\langle push_{inv}(256), Q\rangle$,	$\langle push_{res}(), Q\rangle$,
$\langle push_{res}(), Q\rangle$,	$\langle pop_{inv}(), P\rangle$,
$\langle pop_{res}(256), P\rangle$,	$\langle pop_{res}(256), P\rangle$,
$\langle push_{inv}(2), P\rangle$,	$\langle push_{inv}(2), P\rangle$,
$\langle push_{res}(), P\rangle$	$\langle push_{res}(), P\rangle$

In the two histories H_1 and H_2, P and Q are the process names. The invocations include the method name and, possibly, arguments, e.g. $push_{inv}(256)$, and the responses may include the returned values, e.g. $pop_{res}(256)$. Since the operation $push(256)$ is the first operation executed by process Q, its execution id is 1. Thus, the operation $push(256)$ on process Q consists of the following triple: $\langle\langle push_{inv}(256), Q\rangle, \langle resPush(), Q\rangle, 1\rangle$. Then, $res(push(256))) = \langle resPush(), Q\rangle$.

The history H_1 is concurrent, while history H_2 is sequential. When projecting on the two processes, the following subhistories are obtained:
$H_1|P : \langle pop_{inv}(), P\rangle,\ \langle pop_{res}(256), P\rangle,\ \langle push_{inv}(2), P\rangle,\ \langle push_{res}(), P\rangle,$
$H_1|Q : \langle push_{inv}(256), Q\rangle,\ \langle push_{res}(), Q\rangle,$
$H_2|P : \langle pop_{inv}(), P\rangle,\ \langle pop_{res}(256), P\rangle,\ \langle push_{inv}(2), P\rangle,\ \langle push_{res}(), P\rangle,$
$H_2|Q : \langle push_{inv}(256), Q\rangle,\ \langle push_{res}(), Q\rangle,$
The histories H_1 and H_2 are equivalent, since their subhistories are equivalent for all processes.

The execution of operation o consists of the interval between its invocation and its response events. Two executions can overlap, meaning that both operations have been invoked before either of them returned. In contrast, two executions can also be ordered sequentially, meaning that one operation has returned before the other operation has been invoked. Then, a history H induces an irreflexive partial order $<_H$ on operations: $o_1 <_H o_2$ if $res(o_1)$ precedes $inv(o_2)$ in H. Operations unrelated by $<_H$ are said to be concurrent. If H is sequential, then $<_H$ is a total order [15].

The history H_1 in the above example induces the following partial order among operations: $pop() <_{H_1} push(2)$ and $push(256) <_{H_1} push(2)$. The operation $pop()$ consists of the following triple: $\langle\langle pop_{inv}(), P\rangle, \langle pop_{res}(), P\rangle, 1\rangle$; similar definitions can be given for the other two operations. Considering there is no ordering between the operations $pop()$ and $push(256)$, these two operations are concurrent. On the other hand, the history H_2 induces a total order, meaning that any two distinct operations are comparable: $push(256) <_{H_2} pop() <_{H_2} push(2)$.

In general, correctness for concurrent data structures is based on some type of equivalence with sequential behavior. To formally define such a sequential specification, a few notions should be introduced. A set K of histories is *prefix-closed* if, whenever H is in K, every prefix of H is also in K. A *single-object* history is one in which all events are associated with the same object [15]. Both histories given in Example 2.6 are single-object, since all the events (i.e. invocations and responses) are executed on the same stack.

Definition 2.8. (Sequential specification) A sequential specification for an object is a prefix-closed set of single-object sequential histories for that object. A sequential history H is *legal* if each object subhistory $H|x$ belongs to the sequential specification for object x.

Using all the notions introduced thus far, linearizability can be formally defined, encompassing the idea that concurrent histories may be equivalent to some sequential histories.

Definition 2.9. (Linearizability) A history H is linearizable if it can be extended (by appending zero or more response events) to some history H' such that: (1) $complete(H')$ is equivalent to some legal sequential history L_S; (2) $<_H \subseteq <_{L_S}$ [15].

Thus, when a history contains pending invocations, two options are available: append a matching response or remove the invocation altogether. This decision is based on whether the invoked operation has had effect on the state of the system. If an operation has changed the state, but it has not returned a response yet, then a matching responses is added to the history. Otherwise, since the operation had no effect on the system, it can be removed safely.

Example 2.10. (Removing or appending)
Given the history: $\langle push_{inv}(32), P\rangle$, $\langle pop_{inv}(), Q\rangle$, $\langle pop_{res}(32), Q\rangle$, the invocation of push is pending. Since the pop operation succeeding this invocation terminates successfully and returns 32, then the push operation has had effect over the state of the system, and a matching response $\langle push_{res}(), P\rangle$, is appended.
Given the history: $\langle push_{inv}(32), P\rangle$, $\langle pop_{inv}(), Q\rangle$, $\langle pop_{res}(16), Q\rangle$, the invocation of push is also pending. Since the pop operation succeeding this invocation terminates successfully and returns 16, the push operation has not yet changed the state of the system. This invocation can be removed.

In Example 2.6, the history H_1 is linearizable. There exists a legal sequential history H_2, such that H_1 and H_2 are equivalent, satisfying the first requirement of the definition. Furthermore, the ordering induced by H_1 is included into the ordering induced by H_2, satisfying the second requirement. In this case, H_2 is a linearization of H_1.

Linearizability is a *local* property, meaning that a concurrent system is said to be linearizable whenever each individual object is linearizable. This locality principle supports the assumption of one shared object in the system. Additionally, an object is linearizable if all its concurrent histories are linearizable with respect to some sequential specification [15].

2.2. Labeled Transition Systems

As mentioned above, a concurrent system is a parallelization of some sequential processes. While parallel processes can perform one or more actions at a given moment, a sequential process can perform at most one action at a given moment [23]. The order in which the actions occur within the processes defines the behavior of the system [10]. To capture this behavior, a *labeled transition system*(LTS) can be used.

Definition 2.11. (LTS) A labeled transition system is a four-tuple $L = (S, Act, \rightarrow, s)$, where S is a set of states, Act is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, and $s \in S$ is the initial state.

The communication in a concurrent system can occur through a concurrent data structure. In this context, methods are used to access and manipulate the data structure. A method consists of a number of statements. Then, the set Act contains actions that represent those statements. Furthermore, a transition label can be either a visible or an invisible action. The τ-transition is the only transition labeled with an invisible action, and $\tau \in Act$. Given an LTS L, a *finite path* is a sequence of alternating states and actions, starting and ending with states $\pi = \langle s_0, \alpha_0, s_1, \alpha_1 ... s_n\rangle$, where $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$, for all i. Then, a path of the LTS is a path in which the first state corresponds to the initial state, i.e. $s_0 = s$. Given a path π, the *weak trace* of π is then the sequence obtained by omitting states and invisible actions [17].

The LTS can capture the behavior of a system by recording its possible executions. This implies that an LTS can serve as the underlying *semantic model* for the defined processes. Furthermore, two different processes might behave in an equivalent fashion. Thus, one can check if the semantic models of two processes are equivalent, expressed by giving an equivalence relation (e.g. bisimilarity) between them. The chosen equivalence relation might vary depending on the requirements

on the system, for example ensure that the two processes behave equivalently or that the two processes contain the same traces. Two equivalence relations relevant in the context of linearizability are described in the next two definitions.

Definition 2.12. (Weak trace equivalence [10]) Let $L = (S, Act, \rightarrow, s)$ be a labelled transition system. The set of *weak traces* for a state $t \in S$ is the minimal set $weakTraces(t)$ satisfying:

1. $\epsilon \in weakTraces(t)$, where ϵ denotes the empty trace
2. if there is a state $t' \in S$ such that $t \xrightarrow{a} t'$, where $a \in Act$, $a \neq \tau$, and $\sigma \in weakTraces(t')$, then $a\sigma \in weakTraces(t)$
3. if there is a state $t' \in S$ such that $t \xrightarrow{\tau} t'$, and $\sigma \in weakTraces(t')$, then $\sigma \in weakTraces(t)$

Two states t and u are called *weak trace equivalent* iff $weakTraces(t) = weakTraces(u)$. Two labelled transition systems are *weak trace equivalent* iff their initial states are weak trace equivalent.

Definition 2.13. (Branching bisimulation [10]) Let $L = (S, Act, \rightarrow, s)$ be a labelled transition system. A relation $R \subseteq S \times S$ is a branching bisimulation relation if for all $t, u \in S$ such that tRu, the following conditions hold for all actions $a \in Act$:

1. if $t \xrightarrow{a} t'$, then either $a = \tau$ and $t'Ru$, or there is a sequence $u \xrightarrow{\tau} \cdots \xrightarrow{\tau} u'$ of τ-transitions such that tRu' and $u' \xrightarrow{a} u''$ with $t'Ru''$.
2. symmetrically, if $u \xrightarrow{a} u'$, then either $a = \tau$ and tRu', or there is a sequence $t \xrightarrow{\tau} \cdots \xrightarrow{\tau} t'$ of τ-transitions such that $t'Ru$ and $t' \xrightarrow{a} t''$ with $t''Ru'$.

Two states t and u are *branching bisimilar*, denoted by $t \leftrightarrow_b u$, iff there is a branching bisimulaton relation R such that tRu. Two labelled transition systems are *branching bisimilar* if their initial states are branching bisimilar.

Since an LTS captures the executions of a system, it can also be used to represent the histories of a system. As mentioned above, an object provides a number of methods, that allow threads to interact with the object. A method consists of an invocation, a number of processing steps and a response. However, linearizability is only concerned with operations, which are formed of invocations and responses. Hence, processing steps should be hidden (i.e. abstracted from). An LTS is *operational* when all its visible actions are restricted to invocations and responses of operations, while all the other actions are invisible (i.e. silent actions). Let o range over the set O of operations contained in a history. Then $Act_e = \{inv(o), res(o) | \forall o \in O\} \cup \{\tau\}$. Consequently, for an operational LTS L_o, a trace through the LTS will then consist only of invocations and responses, exactly capturing an execution history.

Thus, an operational LTS can be used to represent the relevant execution histories of a concurrent system, composed of shared objects. To prove linearizability for an object, all its concurrent histories should be proven linearizable. This can be translated into a question on LTSs: given an object shared by parallel processes, generate the LTS of this system, then prove that all traces satisfy some condition (e.g. traces are linearizable). These proof techniques will be elaborated in a subsequent section.

2.3. mCRL2

To generate an LTS, process algebra specifications can be used. These algebras are tools to formally specify complex systems, in particular distributed and concurrent systems [3]. Furthermore, these process algebras are usually equipped with operational semantics, that can be used to generate transition systems.

One process algebra is mCRL2, which is a specification language with an associated toolset [10]. This toolset can receive as input the specification of a process, and then verify certain properties of that process. This verification may also include proving equivalence with other processes. Then, two process specifications are given, and the tool can be used to check whether they are equivalent to each other (e.g. bisimilar or weak trace equivalent). The process specifications need to be expressed in the mCRL2 language, which is based on the Algebra of Communicating Processes (ACP) **??**.

Operator	**Syntax**	**Informal Semantics**
multi-action	$\alpha\|\beta$	actions α and β occurring at the same moment
alternative composition	$p + q$	non-deterministic choice between the two processes
sequential composition	$p.q$	process p followed by process q
conditional operator	c -> p <> q	if condition c is true, then proceed with process p, else proceed with process q
sum operator	$sum\ n$:Nat. $p(n)$	generalization of the alternative operator: non-deterministic choice between $p(0)$, $p(1)$, $p(2)$...
parallel composition	$p\|\|q$	the first action in process p, the first action in process q, or the communication of the two processes
communication operator	$comm(\{\alpha\|\beta$ -> $\gamma\}, p)$	in process p, the actions α and β must communicate to γ
allow operator	$allow(\{\alpha, \beta\}, p)$	in process p, only actions α, β and τ can be executed; all other actions are blocked
hide operator	$hide(\{\alpha\}, p)$	in process p, all instances of α are transformed into invisible actions
rename operator	$rename(\{\alpha$ -> $\beta\}, p)$	in process p, all instances of α are renamed to β

Table 2.1: mCRL2 operators

The Algebra of Communicating Processes provides elements to specify complex behavior. One of the building blocks of such a specification are actions. These can be user-defined actions (i.e. *ReturnPush*), representing simple events of a system. One can formally compose and define complex behavior from these actions using a set of operators. These operators can be applied on two processes, actions, or constants, and are described in Table 2.1. Through these constructs, processes can communicate with each other, and this communication can also be enforced [3].

The specification language mCRL2 extends ACP with data types. Both actions and processes can be parameterised with data. Then, processes can communicate with each other data via these parameters. Additionally, by checking conditions on these data, conditional behavior can be specified. One mechanism to verify that the data fulfills certain conditions is the conditional operator. Furthermore, data transformation and initialization rules can be defined in an mCRL2 specification by using functions and constructors. Thus, besides composing processes, an mCRL2 specification also allows data manipulation in processes [10].

A concurrent system is comprised of processes running in parallel, that operate on a shared object. To examine the possible executions of the object, the most general client can be used, as in [26] and [2]. The *most general client* of a concurrent object is a process that nondeterministically invokes all the object's methods, with all the possible parameters, in an infinite loop. Therefore, each process represents the most general client. The next example encompasses all the described concepts.

Example 2.14. (Stack process specification) In this process specification, a stack is a collection of nodes, where each node stores some value and a reference to the next node. The stack process needs to only store the top, which is either null or a list with a pointer to the next node. A specification in mCRL2 is given in Listing 2.1.

This specification has a *Stack* process, used as a single point of interaction for reading or modifying the *top* variable. This variable has the type *Node*, which stores an integer value and a reference to the next node. Furthermore, the *Stack* process provides actions for accessing and modifying the *top* variable.

```
sort Node = struct null | node(dt: Nat, nxt: Node);
map data: Node -> Int;
    next: Node -> Node;
var d: Nat; n: Node;
eqn data(node(d, n)) = d;  data(null) = -1;
    next(node(d, n)) = n;  next(null) = null;
act CallPush: Int#Nat; CallPop: Int;
    ReturnPush: Int; ReturnPop: Int#Int;
    snd_PushNode: Nat; rcv_PushNode: Nat; PushNode: Nat;
    snd_PopNode: Int; rcv_PopNode: Int; PopNode: Int;
    rcv_ReadTop: Node; snd_ReadTop: Node; ReadTop: Node;

proc Stack(top: Node) = rcv_ReadTop(top).Stack(top) +
    sum v: Nat. rcv_PushNode(v). Stack(node(v, top)) +
    snd_PopNode(data(top)). Stack(next(top));

proc Push(tid: Int,  v: Nat) = CallPush(tid, v). snd_PushNode(v). ReturnPush(tid);

proc Pop(tid: Int) = CallPop(tid). sum n: Int. rcv_PopNode(n). ReturnPop(tid, n);

proc Thread(id: Nat) = Push(id, 128). Push(id, 4). Push(id, 8). Push(id, 16).
     sum t1: Node. snd_ReadTop(t1). Pop(id).
     sum t2: Node. snd_ReadTop(t2);

init hide({ PushNode, PopNode },
     allow({CallPush, CallPop, ReturnPush, ReturnPop, ReadTop, PushNode, PopNode},
     comm({ rcv_ReadTop | snd_ReadTop -> ReadTop,
            rcv_PushNode | snd_PushNode -> PushNode,
            rcv_PopNode | snd_PopNode -> PopNode
   }, Thread(1) || Stack(null))));
```

Listing 2.1: Stack process specification in mCRL2

As mentioned, there are two methods to interact with a stack: push and pop. These methods are modeled as different processes in the specification. This way of modeling is motivated by the desire to verify linearizability. Since a linearizability proof requires execution histories, each method can be annotated with an invocation action and a response actions. Also, each of these processes has as parameter the thread id, to distinguish between method calls initiated from different processes. The *Push* process is also parameterized with the value to be added in the stack. To push an element v on the stack, communicate this value to the *Stack* process, which will create a new node whose value is v and its *next* pointer references the current top. Finally, update the top of the stack to be the new node. To pop an element from the stack, communicate this to the *Stack* process, which will read and return the value stored in the current top, then update *top* to point to the next node of *top*.

The thread interacting with the *Stack* process is modeled as a process that has an identifier and calls some methods of the object. In this scenario, the thread has predefined values that are pushed on the stack. After all values have been added on the stack, the process reads the content of the stack through the action *ReadTop*. The purpose of executing this action is to inspect the contents of the stack. The resulting stack is the one given as an example at the beginning of the section. Finally, one value is removed from the stack.

Furthermore, the specification enforces communication and abstracts from certain actions. The emphasis of such process specifications is on invocations, modeled as calls of the methods (e.g. $CallPop(tid)$), and

on responses, modeled as returns of the methods (e.g. *ReturnPop*(*tid*)). All the other actions should be abstracted from, and thus they may appear as arguments for the *hide* operator. For the purpose of showing how methods behave on the stack, the *ReadTop* actions is not hidden.

Finally, the system is represented as a parallel composition of the *Stack* process and a number of *Thread* processes. In this example, only one thread is included. From this mCRL2 specification, a labeled transition system can be generated. The labeled transition systems then can be reduced modulo a behavioral equivalence (e.g. branching bisimulation). The resulting LTS is shown in Figure 2.3.

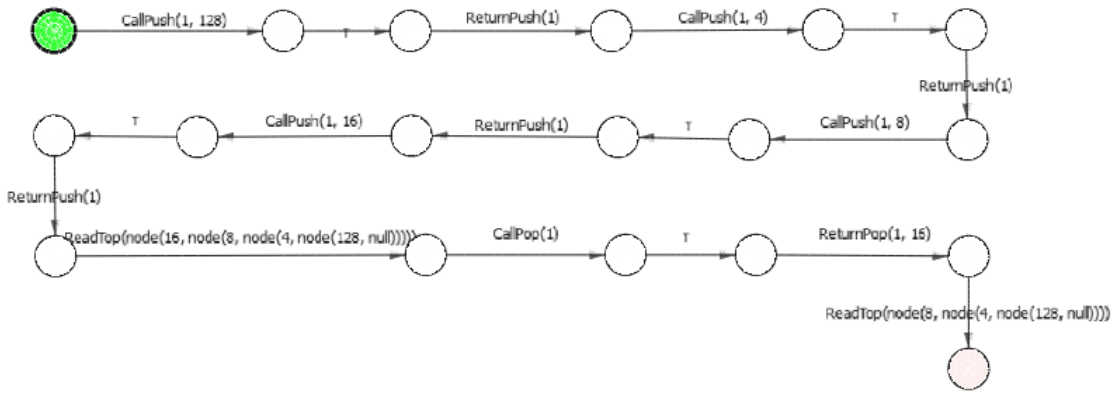

Figure 2.3: LTS generated from the stack process specification

A thread could also be modeled to represent the most general client of the stack. In this case, the thread would be a process that has an identifier. The thread is making progress by non-deterministically calling one of the possible methods of an object. In the case of the stack, the thread makes a choice between calling the *Push* process or the *Pop* process. Thus, this process would consist of an infinite non-deterministic sequence of method calls, as given in Listing 2.2. However, mCRL2 imposes some limitations on the number of states that a generated LTS could contain. An infinite sequence of method calls, with different parameters, can be specified, but the state space cannot be generated. To mitigate this, each process should have a fixed number of operations that it can execute. Given a positive number of operations *nOp*, the thread still proceeds in a non-deterministic manner. This represents a *finite client* of the object.

```
proc Thread(id: Nat, nDone: Nat) = (Push(id, nDone) + Pop(id)).
     Thread(id, nDone+1);
```

Listing 2.2: Most general client of a stack

In the specification given in the previous example, by replacing the given *Thread* process with a finite client that executes two methods, the LTS in Figure 2.4 is generated. Furthermore, by abstracting from all internal actions, and allowing only the invocations and responses of methods to be visible, the resulting LTS is operational. The thread consists of calling two times all object's methods, in any order. Since the stack's methods are only push and pop, the possible interleavings are: push-push, push-pop, pop-push, and pop-pop. It can also be noticed that all possible interleavings are present in the LTS.

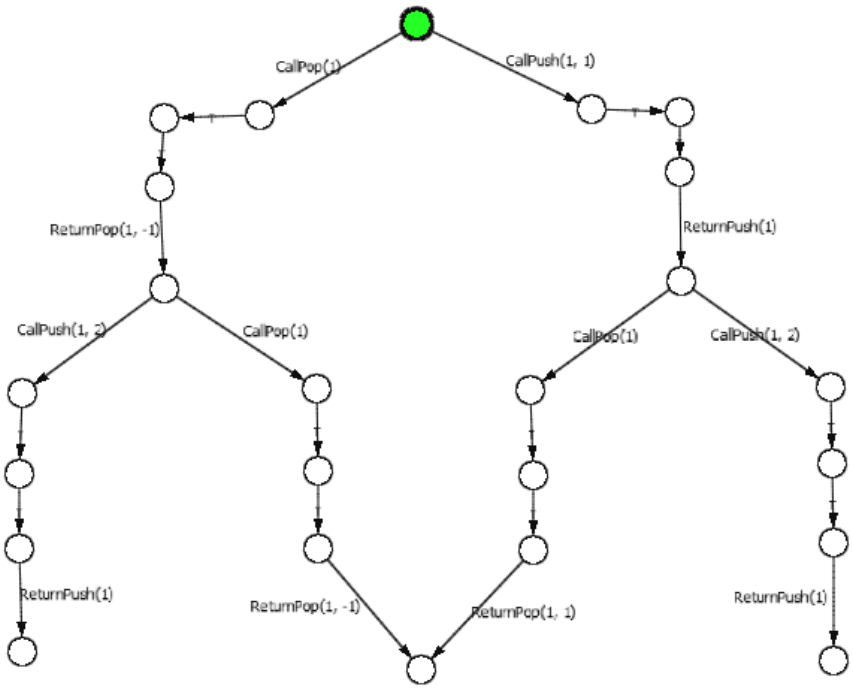

Figure 2.4: LTS generated from the stack process specification

2.4. Verifying Linearizability

Formally, linearizability is defined in terms of the invocations and responses of high-level methods. In a real concurrent program, the high-level methods are implemented by algorithms that operate on concrete shared data structures. Therefore, the execution of high-level methods may allow complicated interleavings at lower levels. Linearizability of a concurrent object requires that, despite complicated low-level interleaving, the history of high-level invocation and response events still has a sequential permutation that respects both the runtime ordering among operations and the sequential specification of the object [17].

Linearizability can be verified through different methods, using both informal and formal reasoning. The informal definition of linearizability states that, given a system, each method call should appear to take effect instantaneously at some moment between its invocation and response. This time point is considered a linearization point (LP). Thus, the most intuitive approach is to determine these linearization points for all methods of a concurrent object implementation [14].

Linearization points were also used in [2]. Their approach formalizes the reasoning and interpretation of linearization points. They present a technique for automatically verifying linearizability. However, this approach is limited to methods of concurrent linked data structures, that contain fixed linearization points. This solution employs *correlating semantics*, which consists of simultaneously analyzing the concurrent implementation with a sequential implementation. Two disjoint instances of the data structure are maintained: the candidate state, which represents the interleaved execution and is built by the concurrent implementation, and the reference state, which is used to build the sequential history. The manipulation of the two instances occurs as follows: whenever a linearization point in a concurrent method is encountered, the execution on the candidate state is temporarily suspended to invoke this method with the same arguments on the reference state. Then, the reference response is saved and compared with the corresponding candidate response, once the method terminates. If these two responses do not match, then one

can conclude that the interleaved execution is not linearizable. This solution builds during runtime the sequential history (witness) needed to prove that an interleaved execution is linearizable [2]. Furthermore, this algorithm was used for checking linearizability in the tool implemented in [24], which constructs linearizable concurrent algorithms starting from a given sequential implementation. This approach demonstrates how linearization points can be used to built the sequential history necessary for proving linearizability.

Both approaches require linearization points for all methods of a concurrent object. However, these ideas might not be applicable when the LPs are not fixed in the code of object methods. For a large class of lock-free algorithms with helping mechanisms, such as [12], the LP of one method might be in the code of some other method. Additionally, in optimistic and lazy algorithms (see [11], [8]), the LPs might depend on unpredictable future interleavings [16]. Thus, it is desirable to reason about linearizability without requiring the knowledge of LPs.

2.4.1 Formalization of proof techniques

Liu et al. proposed expressing linearizability through trace refinement, which is a subset relationship between traces of two given systems. The methods of a concurrent object can be captured through invocations and responses. Thus, linearizability can then be captured through trace refinement of these invocations and responses from a specification to an implementation, where the specification is correct with respect to the sequential semantics [17].

To reason about shared objects, a shared memory model should be defined. A shared memory model M is a triple (O, $init_O$, P), where O is a finite set of shared objects, $init_O$ is the initial valuation of the objects in O, and P is a finite set of processes accessing the objects. Then, an execution of a shared memory model M is modeled by a history. The behavior of M is then defined as the set H of all possible histories together [17]. Considering that linearizability is local, one can assume that the set O consists of one shared object, without loss of generality.

To capture the behavior of shared memory models, Liu et al. use labeled transition systems (LTSs) as their semantic model. Thus, linearizability can be defined as the refinement relationship between two system models (or equivalently two LTSs).

Definition 2.15. (Weak trace refinement) Given two LTSs L_1 and L_2, L_1 refines L_2, written as $L_1 \sqsupseteq_T L_2$, if and only if $weakTraces(L_1) \subseteq weakTraces(L_2)$.

To prove linearizability through trace refinement of two labeled transition systems, several notions have to be introduced. Firstly, these LTSs capture two different implementations of a concurrent object, an abstract one and a concrete one. The abstract implementation is described through a linearizable specification.

The underlying concept behind the abstract specification is that any linearizable object has within its methods linearization points, whether fixed or non-fixed. Thus, in any linearizable history, the moment when the method manipulates the data structure is concentrated in the linearization point, which modifies the data structure atomically. One way to achieve this, as seen in the correlating semantics solution, is to consider the method as occurring in an atomic block. However, Liu et. al relax this principle by decomposing a method o, i.e. an operation in a process, into three atomic steps: the invocation action, the linearization action $lin(o)$, and the response action. The linearization action performs the computations based on the sequential specification of the object. In particular, it maps the invocation and the object state before the method call to a new object state and a response, based on the sequential specification of the object and the old object state. Then, it changes the object to the new state, and stores the response locally [17]. This approach allows interleavings of the three actions, but not of the actions within the method body.

In this context, each operation o performed by a process is defined as a circular state machine with three states: (1) an idle state s_0, (2) a state s_1 after the invocation of the method, but before the linearization action, and (3) a state for every response $s(res)_2$, representing the state after the linearization action but before returning the response [17]. Figure 2.5 depicts the state machine for the executions of the method *pop*, executed by a single process.

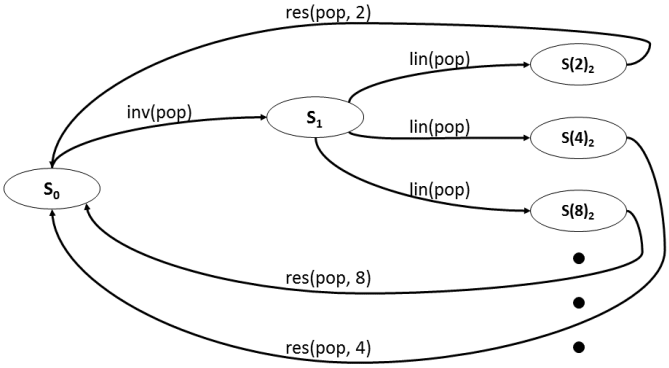

Figure 2.5: Circular state machine for the method *pop*

The invocation and response actions are visible, while all linearization actions are invisible actions. Each process is then defined as the nondeterministic choice of invoking all the allowed methods. Then, the semantic model of the linearizable specification is defined as the composition of the state machines for all processes [17].

Informally, in the abstract specification, the mechanisms for synchronization are removed, since all state changes happen atomically at the linearization action. This atomicity is enforced when building the specification, for example by encapsulating all actions in an atomic block. Thus, there is no need to compare previously read information or to block other processes. This specification might not be sequential, but it fulfills the linearizability property.

The second specification needed in the trace refinement proof is for the concrete implementation. The starting point for this specification is the concrete concurrent algorithm, implementing the methods of a shared object. The implementation model consists of a parallel execution of a number of processes. The behavior of each process is defined as nondeterministic invocations of all the methods supported by the object. The execution of program statements are then considered as actions. Furthermore, local statements, that have no influence over the system state, may be grouped into one atomic action to reduce the state space of the resulting LTS, since a reduced LTS might ensure in a faster comparison of the two semantic models. To prove linearizability of the concurrent algorithm, its histories have to be built. Thus, an invocation action is added at the beginning of each operation and a response action is added to every return statement of each operation. All other actions inside the algorithm are treated as invisible actions since they do not contribute to the histories [17].

Liu et al. formalized linearizability in terms of these two specifications. The underlying assumptions for this formalization is that all shared objects have finite domains, and that all processes have finitely many local states [17]. These assumptions are needed to ensure that the LTSs of the implementations can be constructed. The following theorem characterizes the refinement relation.

Theorem 2.16. ***(Linearizability as trace refinement)*** *Let L_{im} be the implementation LTS and L_{sp} be the corresponding specification LTS. All traces of L_{im} are linearizable if and only if $L_{im} \sqsupseteq_T L_{sp}$.*

The idea behind this theorem is that, for any trace $\sigma \in weakTraces(L_{sp})$, there is a sequential permutation π of σ, that is built based on the order of the linearization actions of all the operations. This order is established based on the irreflexive partial order induced amongst operations. That is, if a response of an operation op_1 has occurred before the invocation of an operation op_2, this also implies that the linearization action of op_1 has occurred before the linearization actions of op_2, since the linearization actions always occur between the invocation and the response. Thus, the sequential permutation π follows the definition of linearizability as given by Herlihy and Wing, maintaining the irreflexive partial order induced by σ. Furthermore, π is a legal sequential history of the object, since the object state is only influenced by the linearization action of every operation, and π respects the order of these operations as defined in the execution history.

Liu et al. also proposed an optimization to this approach, using the knowledge of LPs, if available. The concepts behind this optimization are quite straightforward: when building the abstract specification, the invocation and response actions are hidden, while the linearization actions are visible. Additionally, when building the concrete specification, invocation and response actions are not added anymore, and the only visible actions are again the linearization actions. In both specifications, the returned values are attached to the linearization actions. The resulting models contain fewer states, and thus the verification is performed faster, since the comparison is done on these fewer states.

Yang et al. proposed to verify linearizability by checking if the abstract and the concrete specifications are branching bisimilar. The abstract specifications are built from abstract objects, which can be interpreted as concurrent specifications, where each method body of every object method is a single atomic operation. The concrete specifications are built from the implementation of the objects, which involve more intricate interleavings and have low-level synchronization mechanisms. For both types of objects, the methods start with an invocation and end with a response [26].

The choice for branching bisimulation is justified by the fact that client programs expect that the observable behavior of concrete object programs is equivalent to that of abstract ones. This equivalence relationship supports the concept, since it is an action-based version of stutter bisimulation that basically allows to abstract from sequences of internal steps [26]. Their claim is expressed in the following theorem.

Theorem 2.17. ***(Linearizability through branching bisimulation)*** *Let L_c be the concrete LTS and L_a be the abstract LTS. If L_c and L_a are branching bisimilar, then L_c is linearizable.*

The idea behind this theorem is that, since L_a and L_c are branching bisimilar, this implies that they are also weak trace equivalent. That is, $traces(L_a) = traces(L_c)$, which implies that $traces(L_c) \subseteq traces(L_a)$. Thus, by Definition 2.15, L_c refines L_a. Finally, by Theorem 2.16, one can conclude that L_c is linearizable.

The use of abstract and concrete specifications are fundamental to the proof techniques expressed thus far. By construction, the abstract specification represents the set of linearizable histories. This is ensured by having atomic blocks or clear linearization actions. The concrete specification should be constructed following the methods of the concurrent object. By finding an equivalence between these two specifications, such as trace refinement or branching bisimulation, one can conclude that all the traces of the concrete specification are linearizable. Then, the concurrent object is also linearizable.

As mentioned, linearizability is a safety property, so its violation can be detected with a finite prefix of an execution history [17]. Given a tool that can check whether the two specifications are equivalent, this tool might generate a counterexample when the equivalence relation is not found. This counterexample represents an execution history generated by the system that is not linearizable, i.e. the invocations and responses do not respect the sequential specification of the object. Then, the concurrent object is not linearizable, since a non-linearizable history was found.

3. Treiber's Stack

The Treiber's stack [21] serves as a motivating example for using the methodology described in the next chapter. In general, a stack is a first-in-last-out data type, which provides two methods: *push*, which adds an element at the top of the stack, and *pop*, which removes the element from the top of the stack. Any implementation given for the two methods would be broken down into low-level instructions, executed by the processor. For example, in the *pop*, the method should access the top of the stack and modify it to reference the next node (e.g. $Top = Top.next$). However, this statement is represented by three instructions: access the current data stored in Top, get the next node that should become the top of the stack, and replace the data stored in Top. Between any two instructions, interleavings from other processes can occur, and the references to the current data can become outdated. Treiber's stack mitigates this erroneous manipulation of data by employing a synchronization mechanism. The implementation details of the two methods are shown in Listing 3.1.

```
class Node {
  int data; Node next;
  Node(int d) {
    data = d; next = null;
  }}
Node Top = null; //shared variable
void push(int v) {
  bool done = false;
  Node x = new Node(v);
  while(!done) {
    Node old = Top;
    x.next = old;
    done = CAS(&Top, old, x);
  } return;
}
int pop() {
  bool done = false;
  while(!done) {
    Node old = Top;
    if(old == null)
      return EMPTY;
    Node x = old.next;
    done = CAS(&Top, old, x);
  } return old.data;
}
```

Listing 3.1: Implemetation of *push* and *pop* methods of Treiber's stack

The stack is implemented as a linked list of nodes, which is referenced by the Top object. A node stores some kind of data, and a reference to the next node in the list. When adding or removing an element from the list, the Top has to be updated. Furthermore, to ensure that no data is lost or overwritten when updates occur, a synchronization mechanism should be used. In this case, the synchronization mechanism is non-blocking, meaning that, if a process fails, it cannot propagate that failure to block other processes. Additionally, Treiber's stack is also lock-free, meaning that, from all threads present in the system, at least one is guaranteed to make progress and return from its method.

The Compare-And-Swap (CAS) instruction is used to achieve the synchronization. This is an atomic instruction, and it requires three parameters: a pointer to the variable to be updated p, the old value of this variable *old*, and the new value of this variable *new*. This instruction checks if the current value of the variable, accessed through p, is the same as the old value; if this check is successful, then the variable gets updated to the new value and the instruction returns true. Otherwise, the variable retains the old value, and the instruction returns false. This synchronization mechanism is employed in both methods of the stack. The CAS instruction is placed inside a loop, which repeats as long as the instruction is not

successful. If the instruction is not successful, it implies that another thread modified the data structure and made progress. Eventually, the loop in one of the threads exits and the method terminates.

3.1. mCRL2 specifications

To prove linearizability of a given concurrent data structure, two specifications should be defined: the concrete one and the abstract one. These two specifications are then compared to each other to check whether they are weak trace equivalent or branching bisimilar, and thus if linearizability of the concurrent data structure can be established. The abstract specification is manually obtained from the concrete one. For the comparison of the two specifications, mCRL2 is used. This requires that the processes representing the objects should be specified in the mCRL2 specification language.

The first specification is the concrete one, given in Listing 3.2, which contains seven processes. The first process is the *LinkedList*, which contains the low-level instructions of managing the memory transactions on the stack. The implementation of a stack uses pointers to reference the memory locations, as well as to access or modify those memory locations. However, the mCRL2 language lacks the concept of global memory and variables, which could be referenced from within the processes. Therefore, a process is introduced to represent the shared memory. The linked list can synchronize with other processes, to enable them to read or update the contents at the top of the list. The update is achieved by communicating data through the CAS instruction, whose low-level behavior is implemented by the linked list. However, actions that do not modify the shared memory are not fine-grained into low-level instructions, since the emphasis is put on the interleavings of actions that manipulate the shared memory.

The processes *Push* and *PushWhile* model the implementation of the *push* method. Since the purpose of this specification is to serve as an input for verifying linearizability, invocations and responses of the methods are relevant in the verification. Thus, every method of the data structure should be annotated with an invocation and its matching response. Due to this reasoning, the first action of the process *Push* is *CallPushPrime*. For each statement in the implementation, a matching action is added in the concrete specification, e.g. *AssignNext*. Additionally, the implementation of the *push* method contains a while-loop, which is executed until the CAS instruction succeeds. The only way of expressing a loop in mCRL2 is through a recursive process, *PushWhile*. The actions representing the CAS instruction can synchronize and carry data. The effect of communicating data impacts the local variables of a process, namely the new value of the *Top* variable carried by the actions might change the value stored in the *LinkedList* process. The last action in the while-loop is *ReturnPushPrime*. In the implementation of the *push* method, the return is empty.

The *pop* method is modeled through two processes, *Pop* and *PopWhile*. The same reasoning regarding invocations and responses applies, thus the first action of the *Pop* process is *CallPopPrime*. The *pop* method also contains a while-loop, modeled in the same way as the previous method. This process can either try to pop an empty stack, and then the action *ReturnEmptyPrime* is taken, or try to pop a non-empty stack until the CAS succeeds. In the latter scenario, the last action of this process is *ReturnPopPrime*, containing the element removed from the stack.

The previous processes employ the CAS instruction, which consists of two actions. The first one is accessing the linked list, which will then check the received information against its current value. The second action is providing a result, representing the success of the instruction. These two actions should be modeled as one atomic step, to represent the execution of the CAS instruction. One way of modeling this behavior is to use the multi-action facility of mCRL2.

```
sort Node = struct null | node(data: Nat, next: Node);

act rcv_ReadTop: Node; snd_ReadTop: Node; ReadTop: Node;
    rcv_CAS: Node#Node; snd_CAS: Node#Node; CAS: Node#Node;
    rcv_Result: Bool; snd_Result: Bool; Result: Bool;
    CallPush: Nat#Nat; CallPop: Nat;
    ReturnPush: Nat; ReturnPop: Nat#Int; ReturnEmpty: Nat;
    CallPushPrime: Nat#Nat; CallPopPrime: Nat;
    ReturnPushPrime: Nat; ReturnPopPrime: Nat#Int; ReturnEmptyPrime: Nat;
    NewNode: Nat; AssignNext: Node; GetNext: Node;
```

```
proc LinkedList(top: Node) = rcv_ReadTop(top).LinkedList(top) +
     sum old: Node, new: Node. (rcv_CAS(old, new) | snd_Result(old == top)).
     ((old == top) -> LinkedList(new) <> LinkedList(top));

proc Push(tid: Nat, v: Nat) = CallPushPrime(tid, v). NewNode(v). PushWhile(tid, v);
proc PushWhile(tid: Nat, v: Nat) = sum old:Node. snd_ReadTop(old).
     AssignNext(node(v, old)).
     ((snd_CAS(old, node(v, old)) | rcv_Result(true). ReturnPushPrime(tid)) +
     (snd_CAS(old, node(v, old)) | rcv_Result(false)).PushWhile(tid, v));

proc Pop(tid: Nat) = CallPopPrime(tid). PopWhile(tid);
proc PopWhile(tid: Nat) = sum old:Node. snd_ReadTop(old).
     (old == null) -> ReturnEmptyPrime(tid)
        <> GetNext(next(old)).
        ((snd_CAS(old, next(old)) | rcv_Result(true)).
        ReturnPopPrime(tid, data(old)) +
        (snd_CAS(old, next(old)) | rcv_Result(false)).PopWhile(tid));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) = (nOp > 0) ->
     ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Push(id, head(elms)) + Pop(id)).
     ThreadProgress(id, nOp, tail(elms), nDone+1);

init
    hide({ NewNode, AssignNext, GetNext, ReadTop, CAS, Result
    }, allow({ CallPush, CallPop, ReturnPush, ReturnPop, ReturnEmpty,
        NewNode, AssignNext, GetNext, ReadTop, CAS | Result
    }, comm({ rcv_ReadTop | snd_ReadTop -> ReadTop,
        rcv_CAS | snd_CAS -> CAS, rcv_Result | snd_Result -> Result
    }, rename({ CallPushPrime -> CallPush,
        CallPopPrime -> CallPop, ReturnPushPrime -> ReturnPush,
        ReturnPopPrime -> ReturnPop, ReturnEmptyPrime -> ReturnEmpty
    }, LinkedList(null) ||  Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing 3.2: Concrete specification of Treiber's stack

The last two processes, *Thread* and *ThreadProgress*, model the behavior of a finite approximation of the most general client. Ideally, the most general client would be the one included in the specification. However, as concluded in the previous chapter, this approach is unfeasible to generate the state space. Thus, the finite client is used to support the verification process. Each thread has an identifier, the number of operations that should be executed by the thread and the a list of elements to be added on the stack. The second parameter, representing the number of operations, can be tuned to allow for different scenarios, depending on the speed and efficiency of the hardware. The process starts making progress by choosing non-deterministically one of the two possible methods of this data structure. When two or more threads are run in parallel, the resulting LTS will contain all the possible combinations, regarding the type of operations.

As mentioned above, histories are analyzed when verifying linearizability. A history consists of only invocations and responses. Thus, all the other actions are hidden from the LTS. Finally, the actual process is a parallel composition of two threads and the linked list. This parallel composition mimics the implementation and usage of concurrent data structures, where there is a shared object at a given memory location and various threads access and modify that object through its methods.

The concrete specification can be transformed into an LTS, which depends on the finite client used. One example of such an LTS is shown in Figure 3.1, which was generated from a client consisting of two threads, each thread calling once the *push* method. To visualize the possible interleavings of this concrete specification, the actions *Read*, *CAS*, and *Result* are not hidden. These actions are used to display how the information was modified through the system. However, all the other actions are internal, do not modify the stack, and should still be hidden. Furthermore, to be able to analyze how the actions occur and the data is carried, the number of states and transitions should be reduced. This is achieved by reducing the LTS modulo branching bisimulation, which abstracts from the invisible actions. From this LTS, it becomes visible that the behavior described initially, where data gets overwritten, cannot happen. On the

right side of the LTS, two *push* methods start executing concurrently. Both methods read the top of the stack. Once the first method modifies the top of the stack, the CAS instruction of the second method fails and the loop in *push* method restarts. Eventually, the CAS instruction succeeds, and the stack contains all the values that have been pushed.

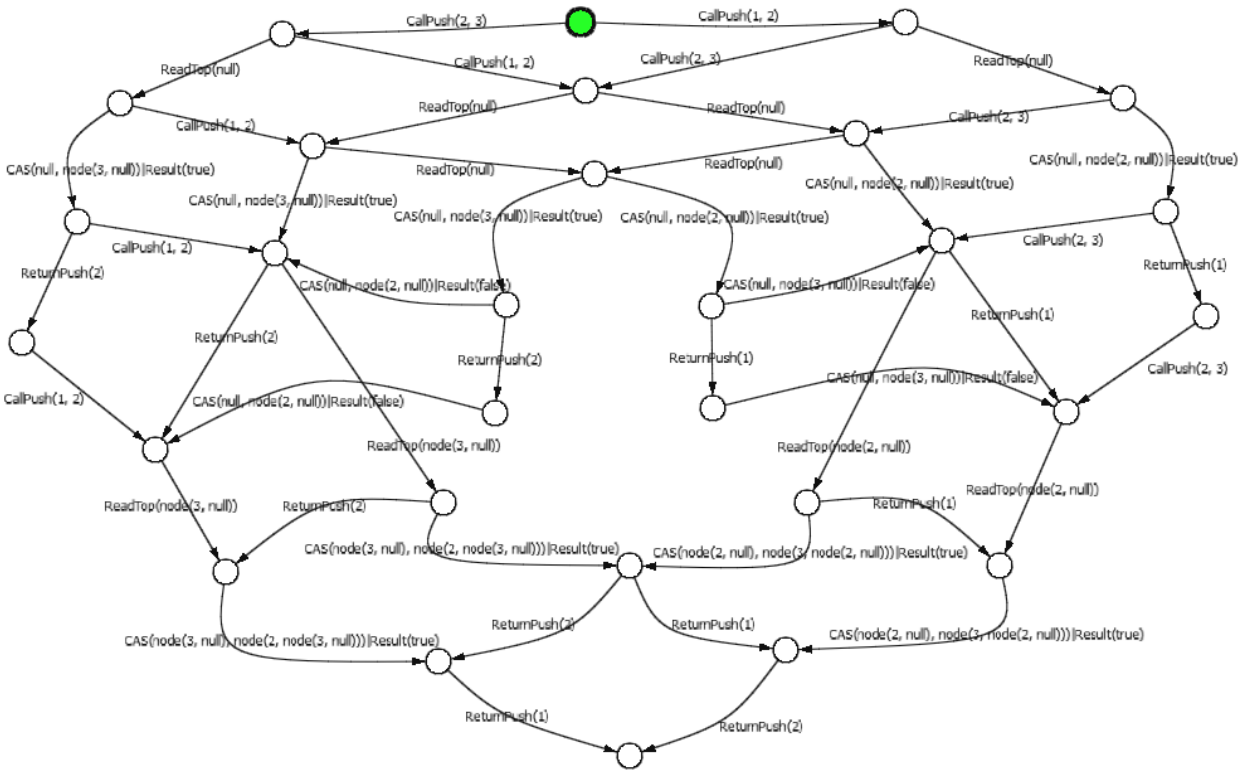

Figure 3.1: LTS generated from the Treiber's stack concrete specification

The second specification is the abstract one, given in Listing 3.3. This specification is obtained from the concrete one, by adding a layer of atomicity. The processes, as defined in the concrete specification, remain unchanged. To ensure atomicity, the process *Atomic* is introduced, which has two options: to define an atomic block and to define an atomic statement. This process can be interpreted as a resource, i.e. time-slice on a CPU or a semaphore representing a lock [14], that needs to be shared by other processes, in a mutual exclusion manner. Furthermore, the bodies of the methods have to be encapsulated in an atomic block, which is achieved by ensuring that the action *StartAtomicBlock* occurs immediately after an invocation, and that the action *EndAtomicBlock* occurs immediately before a response. To achieve this behavior, five simple processes are introduced, that serve as a tool to refine the prime invocations and responses as introduced in the concrete specification. In this context, the only actions outside the atomic blocks are the invocations and responses. To ensure that the invocation and responses do not interleave during atomic blocks, they are synchronized with *AtomicStatement*.

To model an approximation of the most general client, the same processes are used as in the concrete specification. Additionally, the only visible actions are the invocations and responses of each method. Finally, the actual process is a parallel composition of two threads, the linked list, and the atomic process. The last process has to be included, since this becomes a shared resource of the system.

This abstract specification encompasses three atomic behaviors: the invocation action, the atomic block containing the body of the method, and the response action. Considering that any changes incurred on the system happen in the body of the method, this atomic block behaves as a linearization action within the system. Namely, considering that there are no interleavings during the execution of the body, one can observe that any change on the system occurs instantaneously some time between the invocation and

the response of the method. Thus, as mentioned in Section 2.4, this abstract specification contains all the linearizable histories that can be generated from the parallelization of the finite clients. This implies that a legal sequential history can be found for every concurrent history generated through the abstract specification. Additionally, the LTS obtained from the concrete specification contains all the execution histories, both concurrent and sequential, that are executed from the finite clients. Furthermore, if branching bisimulation or weak trace equivalence can be established for the two specifications, this implies that all the histories captured in the concrete specification are also found in the abstract specification. Since it has been established that all histories in the abstract specification are linearizable, then all the histories in the concrete specification are linearizable as well. Thus, the concurrent object modeled through the concrete specification is linearizable.

```
sort Node = struct null | node(data: Nat, next: Node);

act rcv_ReadTop: Node; snd_ReadTop: Node; ReadTop: Node;
    rcv_CAS: Node#Node; snd_CAS: Node#Node; CAS: Node#Node;
    rcv_Result: Bool; snd_Result: Bool; Result: Bool;
    CallPush: Nat#Nat; CallPop: Nat; ReturnPush: Nat;
    ReturnPop: Nat#Int; ReturnEmpty: Nat;
    NewNode: Nat; AssignNext: Node; GetNext: Node;
    rcv_StartAtomicBlock; snd_StartAtomicBlock; StartAtomicBlock;
    rcv_EndAtomicBlock; snd_EndAtomicBlock; EndAtomicBlock;
    AtomicStatement;

proc LinkedList(top: Node) = rcv_ReadTop(top).LinkedList(top) +
     sum old: Node, new: Node. (rcv_CAS(old, new) | snd_Result(old == top)).
     ((old == top) -> LinkedList(new) <> LinkedList(top));

proc Atomic = rcv_StartAtomicBlock. rcv_EndAtomicBlock. Atomic +
     AtomicStatement. Atomic;
proc CallPushPrime(tid: Nat, v: Nat) = CallPush(tid, v) . snd_StartAtomicBlock;
proc CallPopPrime(tid: Nat) = CallPop(tid) . snd_StartAtomicBlock;
proc ReturnPushPrime(tid: Nat) = snd_EndAtomicBlock . ReturnPush(tid);
proc ReturnPopPrime(tid: Nat, v: Int) = snd_EndAtomicBlock . ReturnPop(tid, v);
proc ReturnEmptyPrime(tid: Nat) = snd_EndAtomicBlock . ReturnEmpty(tid);

proc Push(tid: Nat, v: Nat) = CallPushPrime(tid, v). NewNode(v). PushWhile(tid, v);
proc PushWhile(tid: Nat, v: Nat) = sum old:Node. snd_ReadTop(old).
     AssignNext(node(v, old)).
     ((snd_CAS(old, node(v, old)) | rcv_Result(true). ReturnPushPrime(tid)) +
     (snd_CAS(old, node(v, old)) | rcv_Result(false)).PushWhile(tid, v));

proc Pop(tid: Nat) = CallPopPrime(tid). PopWhile(tid);
proc PopWhile(tid: Nat) = sum old:Node. snd_ReadTop(old).
     (old == null) -> ReturnEmptyPrime(tid)
        <> GetNext(next(old)).
        ((snd_CAS(old, next(old)) | rcv_Result(true)).
        ReturnPopPrime(tid, data(old)) +
        (snd_CAS(old, next(old)) | rcv_Result(false)).PopWhile(tid));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) = (nOp > 0) ->
     ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Push(id, head(elms)) + Pop(id)).
     ThreadProgress(id, nOp, tail(elms), nDone+1);

init
    hide({ NewNode, AssignNext, GetNext, ReadTop, CAS, Result,
        AtomicStatement, StartAtomicBlock, EndAtomicBlock
    }, allow({ CallPush | AtomicStatement, CallPop | AtomicStatement,
        ReturnPush | AtomicStatement, ReturnPop | AtomicStatement,
        ReturnEmpty | AtomicStatement, NewNode, AssignNext, GetNext,
    ReadTop, CAS | Result, StartAtomicBlock, EndAtomicBlock
    }, comm({ rcv_ReadTop | snd_ReadTop -> ReadTop,
        rcv_CAS | snd_CAS -> CAS, rcv_Result | snd_Result -> Result,
        rcv_StartAtomicBlock | snd_StartAtomicBlock -> StartAtomicBlock,
        rcv_EndAtomicBlock | snd_EndAtomicBlock -> EndAtomicBlock
```

```
    }, LinkedList(null) || Atomic ||
       Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8]) )));
```

Listing 3.3: Abstract specification of Treiber's stack

Both these specifications are given as input to the mCRL2 tool. Each specification is transformed to an *LPS* (linear process specification), which is then transformed to the corresponding LTS. The last step of the verification is comparing the two LTSs using branching bisimulation. For this verification to be accurate, one requirement emerges regarding the most general client. Both specifications should have the same instances of the most general client, i.e. same number of threads and same number of operations per thread. Given the two above specifications, the tool returns that their LTSs are both branching bisimilar and weak trace equivalent.

This approach of verifying linearizability has a limitation, which manifests itself in the most general client. To exhaustively verify linearizability, the client should invoke an arbitrary number of methods, in any order and with all possible parameters, as described in Section 2.4. However, since mCRL2 cannot produce infinite state spaces, infinite domains pose some challenges in this context. Thus, the threads consider a finite abstraction of the most general client, which considers only a finite set of parameters. The reason behind this is that the value of the parameters do not influence the correctness of the methods. Additionally, these threads call a bounded number of operations.

The question still remains which type of conclusions can be drawn from these equivalences. This methodology verifies linearizability, but with certain restrictions, namely a bounded number of threads, as well as bounded number of operations per thread. As mentioned above, an object o is linearizable if and only if all its execution histories are linearizable. Obtaining these histories depends on the specification of the concurrent system. Let n and m be two positive, fixed numbers. Consider histories generated by a concurrent system that consists of n clients of o, where each client invokes at most m methods. Proving that these execution histories are linearizable leads to establishing $\textit{linearizability}_{m,n}$ of the object, i.e. a restricted version of linearizability.

Given an object that is linearizable for unbounded histories, one can conclude that this object is also $\text{linearizable}_{m,n}$, for all positive numbers m and n. Establishing $\text{linearizability}_{m,n}$ follows from the fact that all histories generated by m clients, each client invoking n methods, should be verified as linearizable. This is already implied from the statement that the object is linearizable for all possible histories. However, it is not as clear whether the converse can also be so easily established. Hence, if one can conclude that an object is $\text{linearizable}_{m,n}$, one still needs to investigate whether this leads to the conclusion that the object is linearizable. Naturally, if one can establish that the object is $\text{linearizable}_{m,n}$, for all m and n, the one can also conclude that the object is linearizable. However, this requires an exhaustive search over all positive numbers, which is unfeasible in the current setting.

Using the methodology described thus far, one can conclude $\text{linearizability}_{m,n}$ of the object. This originates from the construction mechanism of the two specifications. However, if the two LTSs are not equivalent (i.e. branching bisimilar or weak trace equivalent), then the object is not $\text{linearizable}_{m,n}$. When mCRL2 determines an inequivalence, it can generate a counterexample, which is a trace that is found in one specification, but not in the other. Thus, this trace represents an execution history that does not follow the sequential specification, e.g. two *push* methods return the same Top. This cannot happen in a sequential setting, since the *Top* would be overwritten by the *push* method. Additionally, if an object is not $\text{linearizable}_{m,n}$, then it is also not linearizable, since an execution history that is not linearizable has already been found in the bounded environment.

Returning to the Treiber's stack, the two specifications given are branching bisimilar. In this context, there are two clients, each invoking two operations. Thus, Treiber's stack is $\text{linearizable}_{2,2}$ in this case. This conclusion is not surprising, since Treiber's stack has already been illustrated as linearizable by previous work ([17], [25]). Thus, a counterexample could not have been found by mCRL2.

4. Linearizability in mCRL2

In the context of mCRL2, verifying linearizability consists of defining two specifications, the concrete and the abstract one, that derive from the implementation of a concurrent data structure. These specifications are checked to verify whether they are *branching bisimilar* (or trace equivalent). This model checking approach establishes linearizability$_{m,n}$, due to the restrictions imposed to avoid state space explosion.

4.1. Defining specifications

To define the specifications, there are several underlying properties that the modeled system should incorporate. Firstly, the system is characterized as being shared-memory multiprocessor, in which threads operate on concurrent data structures, e.g. Treiber's stack. Nevertheless, the concurrent data structures can function properly even in a single processor system that have multiple threads running in parallel. Several challenges arise regarding reading and writing data in such a context, such as reading old data or losing data. Thus, it is necessary to define a memory consistency model, which specifies how instructions on the memory are ordered and processed. The most commonly assumed model, as stated in [1], is the sequential consistency model, which requires that all memory operations appear to execute one at a time and that all operations of a single thread appear to execute in the order described by that thread's program. Secondly, the concurrent data structure should employ some synchronization mechanism, to avoid interleavings that result in incorrect data. There are two types of synchronization mechanisms: blocking (the delay of a thread causes a delay in all the other threads), and nonblocking (the delay of a thread does not cause the delay of all the other threads). As described in [19], blocking techniques include locks (coarse-grained or fine-grained) or combining trees. Additionally, non-blocking techniques concerns themselves with atomic hardware operations (e.g. Compare-and-Swap), which are supported by modern multiprocessor systems.

The properties described above provide an overview of how memory is administered in the modeled system. This is the basis for constructing the specifications, since the specifications should adhere to the memory model. Thus, the specifications should include some tasks supported by an operating system: memory management and access (ordering of memory operations, accessing of data), as well as process scheduling (defining the behavior of a thread, scheduling time-slices on the available resources).

The concurrent data structure should be robust in shared-memory multiprocessor systems. The implementation defines the synchronization mechanism used. Additionally, this implementation provides details about the type of information stored and the methods to access that information. Treiber's stack provided in the previous chapter fulfills these properties. The synchronization mechanism used in the stack is non-blocking, since it employs the atomic instruction CAS. This CAS instruction was modeled through two actions, one that sends the data to the stack and one that returns the result of updating the stack. This represents the low-level behavior that occurs on a processor.

In the approach suggested, two specifications are required to verify linearizability for a concurrent data structure. The concrete specification closely mimics the implementation of the concurrent data structure, allowing interleavings at every possible moment. When modeling the methods that the data structure provides, two actions should be added for each method, the invocation and the response of that method. The invocation action should occur before any action in the method has executed, while the response action should be the last action that is executed by the method. The abstract specification should be obtained from the concrete specification and it disallows interleavings while executing the method's body. This restriction is achieved by adding an atomic layer on top of the concrete specification, where each body of a method is encapsulated in an atomic block. The resulting methods in this abstract specification can have interleavings only at the following moments: before the invocation action; after the invocation action and before the body of the method; after the body of the method and before the response action.

One way of achieving the atomicity of the abstract specification is annotating the start and end of the atomic blocks at all the appropriate moments, namely exactly after the invocation and exactly before the

response of a method. However, this is manual work that changes the original specification, resulting in a solution that is not maintainable. To generalize this approach, the notion of prime action is going to be used. These *prime actions* are invocations and responses that are marked with the keyword *Prime*. These actions serve different purposes in the two specifications. In the concrete specification, each method starts with a prime invocation and end with a prime response. In the abstract specification, these prime actions are refined, where one prime invocation/response is replaced by the actual invocation/response and the start, respectively the end, of an atomic block. To ensure that the these two specifications capture the same idea semantically, the prime actions in the concrete specification are going to be renamed to the actual invocations and responses. Then, the comparison between the traces of the two specifications evaluate the same type of invocations and responses. This modeling principle is illustrated in the next sections.

4.1.1 Concrete specification

The concrete specification reproduces the implementation of the concurrent data structure. There are several steps that should be undertaken to build this specification:

1. Identify how the concurrent data structure should be stored and define the necessary structures (e.g. a linked list of nodes, an array of nodes). These structures will serve as a parameter to the process representing the concurrent data structure (e.g. the *LinkedList* process in Treiber's stack). This is one shared resource of the system. The next step is to identify all the other shared resources (e.g. locks). These shared resources employ a certain memory management technique (e.g. references, CAS), which are possibly expressed as low-level instructions. Thus, introduce one or more mCRL2 processes that can model the shared resources. In Treiber's stack example, the only shared resource is the process representing the linked list.

   Additionally, identify the synchronization mechanism (e.g. acquiring a lock, executing a CAS statement) and introduce the required actions in the shared resources. This is operating under the assumption that synchronization is only needed for shared resources. The *LinkedList* process contains actions for synchronizing on the CAS instruction, as well as returning the value of executing the instruction.

2. For each method of the concurrent data structure, define a process called *<MethodName>*. This process should have as parameters the thread id, to identify where the call of the method initiated from, and the necessary parameters for the execution of the method. Furthermore, it should start with a prime invocation, denoted by *Call<MethodName>Prime*, and should end with a prime response, denoted by *Return<MethodName>Prime*. Both these events should be parameterized with the thread id, as well as all the required parameters. For each of these events, introduce the invocation and response actions containing the same parameters, of the shape *<Call/Return><MethodName>*. These actions are the ones that will be visible in both specifications, while all the other actions are hidden. Namely, the traces generated by both specifications will contain only invocations, responses and the invisible actions τ.

   Furthermore, for each statement in the body of the method, add an action in the specification. Additionally, each action used for synchronization in the method body should communicate with the counterpart of that action in the shared resource. The processes representing the methods of the Treiber's stack was obtained rather straightforwardly by introducing actions for each statement in the code, and adding the prime invocations and responses (e.g. *CallPushPrime*, *ReturnEmptyPrime*).

3. Define a process for the finite client, called *Thread*, parameterized with a process id and the number of operations *nOp* that can be executed. The thread should make progress by repeatedly calling the methods supported by the object. This poses the need for a loop. Thus, a second process is needed, consisting of a summation of all the possible methods. At each step, there is a nondeterministic choice that this thread can do amongst all the possible methods. These processes are added to carry out the analysis of the data structure, by representing multiples threads in the system and also by tuning *nOp* to evaluate different scenarios.

In the processes representing the shared resources and the methods, the actions identified are either prime invocations, prime responses, or internal actions. As linearizability is only concerned with events, the actions that should be visible in the system are the non-prime invocation and responses. This is achieved through the following tasks. First, the internal actions should be abstracted from the concrete specification. This manifests itself as hiding all actions, except the invocations and responses, marked with either *Call* or *Return*. Secondly, the set of actions that should be allowed in the specification contains all defined actions, except the prime actions. From this method of modeling, the only visible allowed actions in the specification are the invocations and responses, e.g. *CallPush* and *ReturnPush*. However, this actions are not executed in the specification currently, since the prime invocations and responses are added in the methods, as can be seen from Listing 4.1. Thirdly, a rename block should be introduced, where every prime event is renamed to its corresponding event (e.g. *CallPushPrime -> CallPush*).

```
proc Push(tid: Nat, v: Nat) = CallPushPrime(tid, v).
     NewNode(v). PushWhile(tid, v);
init %...
    allow({ CallPush
    }, rename({ CallPushPrime -> CallPush
        %all the remaining prime actions
    }))
```

Listing 4.1: Snapshot of the Treiber's stack concrete specification

The system is a parallelization of all shared resources and m threads, where each thread executes n operations. The concrete specification obtained serves as input to the process of building the abstract specification, where the prime actions are going to be refined to include the atomic block.

4.1.2 Abstract specification

The abstract specification restricts the interleavings that the concurrent data structure can have by enforcing that the method body executes sequentially. This is achieved by adding an atomic layer on top of the concrete specification, where all internal actions in a method body are encapsulated in an atomic block. From an external perspective, the methods in the abstract specification consist of three actions: the invocation action, the body action, and the response action. This follows the reasoning described in [17], where a method is decomposed into these three actions. This decomposition ensures that the change on the state of the system occurs consistently during the body action, and this block can be considered as the linearization point of the method.

As can be observed from the example of Treiber's stack, the start of the atomic block should occur immediately after an invocation, while the end of the atomic block should occur immediately before the response. To generalize this annotation of atomic blocks, the prime actions are used. Namely, the prime invocation action should be replaced by an invocation and the start of the atomic block. A similar reasoning should be used regarding the prime response actions. Thus, to build the abstract specification, the following steps should be undertaken:

1. Define the process *Atomic*. This process has two choices: to build an atomic block or to build an atomic statement. This enforces the atomicity protocol, where the method body is executed without interleavings. Additionally, this process can be considered as a shared resource in the system, where all the clients are trying to get a time-slice on the CPU. This process is identical in all the abstract specifications, and is given in Listing 4.2.

   The action *StartAtomicBlock* should be executed immediately after a call action, while the action *EndAtomicBlock* should be executed immediately before a return action. By the construction of the *Atomic* process, once the start of an atomic block has occurred, the process needs to wait for the end of an atomic block to occur. This is equivalent to acquiring a lock and releasing a lock. Hence, any other method that wants to execute its body is forced to wait for the previous method to finish its execution. However, the invocations and responses are placed outside atomic blocks to ensure that the method consists of three uninterrupted moments: the invocation action, the atomic block, and the response action. If no communication is added to enforce that the invocations and

responses communicate with the *Atomic* process, then these actions can interleave at any moment. Thus, the action *AtomicStatement* is used to ensure that the invocations and responses are executed only outside of atomic blocks. This action should always be part of a multi-action of the shape *<Call/ReturnAction> | AtomicStatement*.

2. To ensure that the body of each method is placed within an atomic block, the specification should contain some kind of action refinement. Considering the prime actions already introduced and the markers of atomic blocks, it is clear that each prime call action should be refined to a call action followed by an action representing the start of the atomic block. Subsequently, each prime return action should be refined to an action representing the end of the atomic block followed by a return action. Thus, for each event as defined in the general specification, namely *Call<MethodName>Prime* and *Return<MethodName>Prime*, introduce a process with the same name, consisting of the corresponding action and the marker of the atomic block (e.g. *CallPushPrime(id: Nat, item: Nat) = CallPush(id, nat . snd_StartAtomicBlock)*). To avoid naming conflicts, the actions for the events, which have been refined, should be removed from this specification. Additionally, the rename block should be removed. An example can be seen in the abstract specification of the Treiber's stack.

```
act <other_actions>
    rcv_StartAtomicBlock; snd_StartAtomicBlock; StartAtomicBlock;
    rcv_EndAtomicBlock; snd_EndAtomicBlock; EndAtomicBlock;
    AtomicStatement;
<other_processes>
proc Atomic = rcv_StartAtomicBlock. rcv_EndAtomicBlock. Atomic +
     AtomicStatement. Atomic;

init
     hide({ <other_actions>
          StartAtomicBlock, EndAtomicBlock, AtomicStatement
     }, allow({ <other_actions>
          CallPushPrime | AtomicStatement,
          ReturnPushPrime | AtomicStatement,
          StartAtomicBlock, EndAtomicBlock
     }, comm({ <other_actions>
          rcv_StartAtomicBlock | snd_StartAtomicBlock -> StartAtomicBlock,
          rcv_EndAtomicBlock | snd_EndAtomicBlock -> EndAtomicBlock
     }, )));
```

Listing 4.2: Atomic process defined in an abstract specification

The concurrent system as generated by this specification should still concern itself with only invocations and responses. Thus, the atomic actions should be hidden from the system. As the *Atomic* process is a shared resource, it should also be added in the parallelization given in the concrete specification. One remark is that the abstract system should consider the same number n for threads, and same number m for number of operations as the system generated from the concrete specification.

The concrete specification obtained through this mechanism generates all possible execution histories, when the object is accessed by a finite client. This is achieved through the nature of the parallel composition in mCRL2. When n threads are executed in parallel, where each thread calls m operations, then all the possible interleavings of the actions in the methods are generated. After hiding the internal actions, the result of this specification is an LTS whose weak traces represent all the execution histories of a system calling $m \cdot n$ operations. Similarly, the abstract specification obtained through this mechanism generates all linearizable execution histories. The reason behind this phenomenon is that the body of the methods executes atomically, meaning that no interleaving can occur. Interleavings at the level of method statements are the source of incorrect concurrent data structures. This results in a specification that is linearizable, since the only interleavings that can occur is among the invocations, responses and execution bodies. These interleavings do not affect the changes that happen on the system.

4.2. Verification

Due to the way the specifications are constructed, to verify linearizability it suffices to compare them using either branching bisimulation or weak trace equivalence. One underlying assumption for the specifications is that they employ finite domains and finite number of threads, to ensure that there are finitely many states in the generated LTSs. The first step towards the comparison is transforming each specification into a linear process specification (LPS), automatically done via the tool mCRL2. These LPSs are then transformed into labeled transitions systems, also done via the toolset. The obtained LTSs are then compared against each other to verify if they are equivalent (branching bisimilar or weak trace equivalent). This verification mechanism functions because the abstract specification is linearizable, which is a notion claimed by the following theorem.

Theorem 4.1. ***(Linearizable abstract specification)*** *Let L_a be the LTS generated from the abstract specification. Then all weak traces of L_a are linearizable.*

An informal rationale behind this theorem follows from the method in which the abstract specification is constructed. Let $H \in WeakTraces(L_a)$. Let T be an execution path that generated this trace. This implies that T consists of invocations, responses, and invisible actions. Additionally, the body of each method is executed within an atomic block. This block ensures that any change on the system occurs in a sequential manner (i.e. no interleavings from the other processes while accessing and modifying the data). Thus, there is a sequence of invisible actions in the trace that represents that atomic block, which can be considered as the linearization point of the method. Then, a sequential history can be built following the order of these linearization points. This follows from the fact that any two operations can be either concurrent or partially ordered. Let o be a shared object, which provides the method *Modify*. Then, a trace that can be generated from the abstract specification when calling this method from two parallel processes is given next. For illustrating the linearization points, the atomic blocks are not hidden.

T: CallModify(1, 1), CallModify(2, 2), StartAtomicBlock, InternalActions(2), EndAtomicBlock, StartAtomicBlock, InternalActions(1), EndAtomicBlock, ReturnModify(1), ReturnModify(2), CallModify(1, 3), StartAtomicBlock, InternalActions(1), EndAtomicBlock, ReturnModify(1)

This trace contains operations that are partially ordered, namely *Modify(1, 1)* $<$ *Modify(1, 3)* and *Modify(2, 2)* $<$ *Modify(1, 3)*. Additionally, the operations *Modify(1, 1)* and *Modify(2, 2)* are concurrent. The linearization point of each method occurs between the start of the atomic block and the end of the atomic block. Then, one can build sequential trace S that respects both the ordering of the operations, as well as the ordering of the linearization points. This trace is given next.

S: CallModify(2, 2), StartAtomicBlock, InternalActions(2), EndAtomicBlock, ReturnModify(2), CallModify(1, 1), StartAtomicBlock, InternalActions(1), EndAtomicBlock, ReturnModify(1), CallModify(1, 3), StartAtomicBlock, InternalActions(1), EndAtomicBlock, ReturnModify(1)

By hiding all the atomic and internal actions, one obtains a weak trace that is a legal sequential history. The change on the system occurs only during atomic blocks, in a purely sequential manner, ensuring that this history respects the sequential specification of o. Thus, since there exists a legal sequential history equivalent with H, one can conclude that H is linearizable.

The toolset produces a result based on the two specifications. The first comparison of the two specifications could be done with branching bisimulation. As van Glabbeek claimed in [23], branching bisimulation implies weak trace equivalence. Thus, if the two labeled transitions are branching bisimilar, one can conclude that their respective sets of weak traces are equal. Deciding branching bisimulation is more time efficient than weak trace equivalence in mCRL2. However, presuming that the result returned for checking branching bisimulation is false, then the two systems should be checked using weak trace equivalence. A counterexample generated by mCRL2 in this scenario includes a trace that can be found in the concrete LTS, but not in the abstract LTS. This trace represents a subhistory that is not linearizable, presenting an invalid sequence of invocations and responses. Hence, provided that the concurrent data structure is linearizable, then the concrete LTS and the abstract LTS should contain the same set of weak traces. This result is presented by the following theorem.

Theorem 4.2. ***(Capturing linearizability)*** *Let L_c be the LTS generated from the concrete specification and L_a be the LTS generated from the abstract specification, given a finite client (i.e. n threads, m operations per thread). Then L_c is linearizable$_{m,n}$ if and only if $WeakTraces(L_c) = WeakTraces(L_a)$.*

An informal argument for this theorem follows from the construction mechanism of the specifications, as well as from establishing that the abstract specification is linearizable. Firstly, assume L_c is linearizable$_{m,n}$, and let $H \in WeakTraces(L_c)$. Then H is a linearizable$_{m,n}$ history. Following the construction of the abstract specification and the restriction on the finite client, L_a contains all linearizable$_{m,n}$ histories. Thus, H must also be an element of L_a. Additionally, L_c captures all execution histories generated from the finite client, and it is also linearizable$_{m,n}$. Since L_a is obtained from L_c and is also linearizable$_{m,n}$, intuitively it must hold that L_c captures all linearizable$_{m,n}$ execution histories that are also contained in L_a. Secondly, assume $WeakTraces(L_c) = WeakTraces(L_a)$. It is established that all the weak traces in L_a are linearizable. Since the concrete LTS and the abstract LTS contain the same weak traces, one can conclude that all the weak traces in L_c are linearizable. Thus, L_c is linearizable, and thus also linearizable$_{m,n}$.

In conclusion, if the two LTSs do not contain the same weak traces, then one can conclude that the concurrent data structure is not linearizable. A counterexample can be produced by the toolset, which clearly demonstrates an execution history that is not linearizable. However, in the case of concluding that the two LTSs are branching bisimilar, then the concurrent data structure is linearizable$_{m,n}$. The two parameters n and m originate from the two specifications, representing the number of selected threads, as well as the number of operations per thread.

5. Case Studies

This chapter presents the models for some given concurrent data structures, in order to test the general approach presented in the previous chapter. The implementations of the concurrent data structures are given in Appendix A, while the full concrete specifications are given in Appendix B. The verification results, as well as statistics on the state space, are given in the next chapter.

5.1. Concurrent set

Herlihy and Shavit [14] present multiple algorithms for implementing a concurrent set. This set can be manipulated through two methods: adding or removing an element, both returning a boolean depending on whether the element is already contained in the set. The common notion among all these algorithms is the representation of a set as a linked list of nodes. Each node stores the actual value of the element, its hash value and a reference to the next node. The nodes are sorted by their hash values. The difference among these algorithms is the type of synchronization used, ranging from lock-based to lock-free algorithms.

5.1.1 Coarse-grained set

The coarse-grained algorithm is lock-based, using a single lock for the entire list. The data requirements are given in Listing 5.1. The set is stored as a linked list, with a reference *head* to the first element in the list. The two methods provided by the object should acquire the lock before manipulating the list. This is shown in Listing 5.3, where the implementation of the *add* method is shown. Thus, both methods access and modify the list only when holding the lock. This results in executing the body of the methods in a sequential manner. Following this reasoning, it becomes clear that the coarse-grained set is linearizable. The methodology described thus far should reach the same conclusion.

```
private class Node {
   T item; int key; Node next;
}
public class CoarseList<T> {
   private Node head;
   private Lock lock = new ReentrantLock();
   public CoarseList() {
      head = new Node(Integer.MIN_VALUE);
      head.next = new Node(Integer.MAX_VALUE);
   }
}
```

Listing 5.1: Data structures of the coarse-grained set

The concrete specification is constructed following the steps described in Section 4.1. To achieve this, the first step is identifying the data requirements, which are shown in Listing 5.1. The coarse-grained set is stored as a linked list of nodes. A node contains three elements: the item representing the data to be stored, the key of that item, and a pointer to the next node in the list. The items in the linked list should be sorted by the value of the key. The necessary structures and functions to manipulate the list should be created. Additionally, there are two shared resources, namely the coarse list and the lock. Thus, a process should be introduced for each of these resources, with the respective needed actions. Modeling these concepts results in the data specification shown in Listing 5.2, which represents an integral part of the concrete specification. Further specification details can be found in Appendix B, Listing B.1.

```
sort Node = struct null | node(item: Int, key: Int, next: Node);
map MinKey: Int; MaxKey: Pos; hashCode: Nat -> Nat;
    insert: Nat#Node -> Node; insertHelper: Nat#Nat#Node#Node -> Node;
    remove: Nat#Node -> Node; removeHelper: Nat#Nat#Node#Node -> Node;
eqn MinKey = -1; MaxKey = 100;

act rcv_ReadHead: Node; snd_ReadHead: Node; ReadHead: Node;
    rcv_InsertItem: Nat; snd_InsertItem: Nat; InsertItem: Nat;
```

```
    rcv_RemoveItem: Nat; snd_RemoveItem: Nat; RemoveItem: Nat;
    rcv_Lock: Nat; snd_Lock: Nat; Lock: Nat;
    rcv_Unlock: Nat; snd_Unlock: Nat; Unlock: Nat;

proc LinkedList(nodes: Node) = rcv_ReadHead(nodes). LinkedList(nodes) +
    sum item1: Nat. rcv_InsertItem(item1). LinkedList(insert(item1, nodes)) +
    sum item2: Nat. rcv_RemoveItem(item2). LinkedList(remove(item2, nodes));

proc ReentrantLock = sum id: Nat. rcv_Lock(id). rcv_Unlock(id). ReentrantLock;
```

Listing 5.2: Data specification of the coarse-grained set

The second step is identifying all the methods provided by the concurrent data structure, and introduce a process for each of the methods. The concurrent set provides two methods: *add* and *remove*. The implementation of the *add* method is given in Listing 5.3. The implementation of the *remove* method is omitted here, due to employing a similar reasoning, but can be found in Appendix A. The processes, and corresponding actions, for modeling the *add* method are given in Listing 5.4.

```
public boolean add(T item) {
   Node pred, curr;
   int key = item.hashCode();
   lock.lock();
   try {
      pred = head;
      curr = pred.next;
      while (curr.key < key) {
         pred = curr;
         curr = curr.next;
      }
      if (key == curr.key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = curr;
         pred.next = node;
         return true;
      }
   } finally {
    lock.unlock();
}}
```

Listing 5.3: Implementation of the *add* method of the coarse-grained set

The process *Add* models the behavior of the *add* method. As mentioned in the methodology, the process should be parameterized with the id of the thread that initiated the call. Since the *add* method takes as a parameter the item to be added, this parameter should be also included in the process. Furthermore, the process should start with a prime invocation, denoted by the action *CallAddPrime*, which contains the same parameters as the process. There is a one-to-one mapping between the statements in the method implementation and the actions in the method specification. To model the while-loop in the method, the recursive process *AddWhile* has been introduced, containing the necessary parameters to traverse the list. The traversal stops when the right position has been reached. After the loop finishes, there are two possible cases: either the element exists, in which case the method returns false, or the element does not exist, in which case the list is modified and the method returns true. This response is achieved through the action *ReturnAddPrime*, which contains the value to be returned. On top of that, it also the identifier of the invoking thread. The actions for the invocations and responses are also defined, to ensure that the histories generated by this specification can be compared to the histories generated by the abstract specification.

```
act CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    GetHashCode: Nat; NextIteration;

proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). snd_Lock(id).
    sum predNode: Node. snd_ReadHead(predNode).
    AddWhile(id, itemToAdd, predNode, next(predNode));
```

```
proc AddWhile(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
    -> NextIteration. AddWhile(id, itemToAdd, curr, next(curr))
    <> ((key(curr) == hashCode(itemToAdd))
        -> snd_Unlock(id). ReturnAddPrime(id, false)
        <> snd_InsertItem(itemToAdd). snd_Unlock(id). ReturnAddPrime(id, true));
```

Listing 5.4: Method specification of the coarse-grained set

The third step in building the concrete specification is defining the finite client, which can be found in Listing 5.5. The process *Thread* has an identifier, a bounded number of operations that can execute, and a list of items that can be added or removed from the set. To allow the thread to make progress, a loop is needed, which is modeled through a second process. This process calls non-deterministically the two methods provided by the object.

```
proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
    (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
    (nDone <= nOp) -> (Add(id, head(elms)) + Remove(id, head(elms))).
    ThreadProgress(id, nOp, tail(elms), nDone + 1);
```

Listing 5.5: Client of the coarse-grained set

The initial process for this specification can be found in Listing 5.6. The internal actions are hidden, to ensure that the generated LTS consists only of invocations and responses. The allowed actions in this specification are all the ones defined, except the prime actions, and the required communication is enforced. Finally, the prime invocations and prime responses are renamed to their non-prime counterparts. Then, the process is a parallel composition of the two shared resources, the linked list and the lock, and two threads, each executing two operations.

```
init
    hide({ NextIteration, ReadHead, InsertItem, RemoveItem, Lock,
        Unlock, GetHashCode
    }, allow({ CallAdd, ReturnAdd, CallRemove, ReturnRemove,
        ReadHead, InsertItem, RemoveItem, Lock, Unlock, GetHashCode, NextIteration
    }, comm({ rcv_ReadHead | snd_ReadHead -> ReadHead,
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_Lock | snd_Lock -> Lock, rcv_Unlock | snd_Unlock -> Unlock
    }, rename({ CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
    }, LinkedList(node(MinKey, MinKey, node(MaxKey, MaxKey, null))) ||
       ReentrantLock || Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing 5.6: Start process of the coarse-grained set

The abstract specification is obtained from the concrete specification, as described in the methodology. This means that the process *Atomic* is added in the abstract specification, with the corresponding actions and communication. This process is also added in the parallel composition, as it is a shared resource. Additionally, the processes representing the action refinement are added to the specification, as shown in Listing 5.7. To avoid conflicts, the actions with the same name as the processes should be removed from the abstract specification, as well as the *rename* operator.

```
CallAddPrime(id: Nat, itemToAdd: Nat) =
    CallAdd(id, itemToAdd) . snd_StartAtomicBlock;
ReturnAddPrime(id: Nat, res: Bool) = snd_EndAtomicBlock. ReturnAdd(id, res);
CallRemovePrime(id: Nat, itemToRemove: Nat) =
    CallRemove(id, itemToRemove). snd_StartAtomicBlock;
ReturnRemovePrime(id: Nat, res: Bool) = snd_EndAtomicBlock. ReturnRemove(id, res);
```

Listing 5.7: Action refinement of the coarse-grained set

5.1.2 Fine-grained set

The fine-grained algorithm is also lock-based, using a lock for every node in the list, rather than using a lock for the entire list. The data requirements imposed by the implementation are the same as given in the coarse-grained set. The two methods provided by the object acquire locks as they traverse the list, in a hand-over-hand manner: except for the initial node, acquire the lock for the current node only when holding the lock for the predecessor of the node [14]. Currently, threads can traverse the list together, which was not possible with the coarse-grained implementation. This mechanism is shown in Listing 5.8, where the implementation of the *add* method is shown. Both methods access and modify the list only when holding the appropriate locks. As the manipulation of the list occurs when locks are held, the statements executing in between locking a node and unlocking that node contain linearization points. Consequently, the modification of a particular node happens in an exclusive manner. Therefore, the authors claim that this concurrent data structure is linearizable.

```
public boolean add(T item) {
   int key = item.hashCode();
   head.lock();
   Node pred = head;
   try {
      Node curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
         return false;
         }
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         return true;
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
}}
```

Listing 5.8: Implementation of the *add* method of the fine-grained set

The concrete specification for the fine-grained set is modeled first. To achieve this, identify the data requirements. In the implementation, the fine-grained set is stored as a linked list of nodes. Using references, each node can be locked individually. However, as mCRL2 lacks the notion of global memory, there is no way to add references to a linked list of nodes. To mitigate this issue, the linked list of nodes is stored as an array in the mCRL2 model, allowing for access to individual nodes. This array is modeled as a function, that maps a number to a node. The necessary structures and functions to manipulate the array should be created. The nodes now include a variable representing whether this node is locked, and a variable representing its memory location (i.e. the index of the node in the array). There is one shared resources, namely the linked list. Thus, a process is introduced for this resource, with the respective needed actions. Modeling these concepts results in the data specification shown in Listing 5.9, which represents an integral part of the concrete specification. The variables and mappings are omitted from this specification, and can be found in Appendix B, Listing B.2.

```
sort Node = struct null | node(item: Int, key: Int, next: Int, lock: Bool, mem: Nat
    );
sort ArrayNodes = Int -> Node;

act snd_ReadNode: Int#Node; rcv_ReadNode: Int#Node; ReadNode: Int#Node;
    snd_ReadNext: Int#Node; rcv_ReadNext: Int#Node; ReadNext: Int#Node;
    snd_LockNode: Int; rcv_LockNode: Int; LockNode: Int;
    snd_UnlockNode: Int; rcv_UnlockNode: Int; UnlockNode: Int;
```

```
    CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallRemove: Nat#Nat; ReturnRemove: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    CallRemovePrime: Nat#Nat; ReturnRemovePrime: Nat#Bool;
    rcv_InsertItem: Nat#Nat; snd_InsertItem: Nat#Nat; InsertItem: Nat#Nat;
    rcv_RemoveItem: Nat; snd_RemoveItem: Nat; RemoveItem: Nat; GetHashCode: Nat;

proc LinkedList(nodes: ArrayNodes, nextLoc: Nat) =
    sum element: Int. snd_ReadNode(element, nodes(element)).
        LinkedList(nodes, nextLoc) +
    sum elementN: Int. snd_ReadNext(elementN, nodes(next(nodes(elementN)))).
        LinkedList(nodes, nextLoc) +
    sum n: Int. (!lock(nodes(n))) -> rcv_LockNode(n).
        LinkedList(nodes[n->lockNode(nodes(n))], nextLoc) +
    sum nn: Int. rcv_UnlockNode(nn).
        LinkedList(nodes[nn->unlockNode(nodes(nn))], nextLoc) +
    sum item1, loc: Nat. rcv_InsertItem(item1, loc).
        LinkedList(insertNode(nodes, nextLoc, item1, loc), (nextLoc+1)) +
    sum loc: Nat. rcv_RemoveItem(loc). LinkedList(removeNode(nodes, loc), nextLoc);
```

Listing 5.9: Data specification of the fine-grained set

The second step is identifying all the methods provided by the concurrent data structure, and introduce a process for each of the methods. The concurrent set provides two methods: *add* and *remove*. The implementation of the *add* method is given in Listing 5.8. The implementation of the *remove* method is omitted here, due to employing a similar reasoning, but can be found in Appendix A. The processes for modeling the *add* method are given in Listing 5.10.

The process *Add* models the behavior of the *add* method. As mentioned in the previous section, the process should be parameterized with the id of the thread and the item to be added. Furthermore, the process should start with a prime invocation, denoted by the action *CallAddPrime*, which contains the same parameters as the process. There is a one-to-one mapping between the statements in the method implementation and the actions in the method specification. As can be seen, the *head* node is locked before reading the next node. To model the while-loop in the method, the recursive process *AddWhile* has been introduced, containing the necessary parameters to traverse the list. During the traversal, the hand-over-hand locking occurs, and the traversal stops when the right position has been reached. After the loop finishes, the same two cases apply as in the previous implementation. Finally, the method concludes with the action *ReturnAddPrime*.

```
proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). snd_LockNode(0).
    sum predNode: Node. rcv_ReadNode(0, predNode).
    sum curr: Node. rcv_ReadNext(mem(predNode), curr). snd_LockNode(mem(curr)).
    AddWhile(id, itemToAdd, predNode, curr);
proc AddWhile(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
      -> snd_UnlockNode(mem(predNode)). sum nn: Node. rcv_ReadNext(mem(curr), nn).
         snd_LockNode(mem(nn)). AddWhile(id, itemToAdd, curr, nn)
      <> ((key(curr) == hashCode(itemToAdd))
         -> snd_UnlockNode(mem(curr)). snd_UnlockNode(mem(predNode)).
            ReturnAddPrime(id, false)
         <> snd_InsertItem(itemToAdd, mem(predNode)). snd_UnlockNode(mem(curr)).
            snd_UnlockNode(mem(predNode)). ReturnAddPrime(id, true));
```

Listing 5.10: Method specification of the fine-grained set

The third step in building the concrete specification is defining the finite client, which is the same as the one defined for the coarse-grained set. The initial process for this specification can be found in Listing 5.11. The internal actions are hidden, to ensure that the generated LTS consists only of invocations and responses. The allowed actions in this specification are all the ones defined, except the prime actions, and the required communication is enforced. Finally, the prime invocations and prime responses are renamed to their non-prime counterparts. Then, the process is a parallel composition of the linked list and two threads, each executing two operations.

```
init
    hide({
        GetHashCode, InsertItem, RemoveItem, LockNode, UnlockNode, ReadNode,
            ReadNext
    }, allow({
        CallAdd, ReturnAdd, CallRemove, ReturnRemove, InsertItem, RemoveItem,
        GetHashCode, LockNode, UnlockNode, ReadNode, ReadNext
  }, comm({
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_LockNode | snd_LockNode -> LockNode,
        rcv_UnlockNode | snd_UnlockNode -> UnlockNode,
        rcv_ReadNode | snd_ReadNode -> ReadNode,
        rcv_ReadNext | snd_ReadNext -> ReadNext
  }, rename({
        CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
  }, LinkedList((lambda n: Nat.null)[0->node(MinKey, MinKey, 1, false, 0)]
        [1->node(MaxKey, MaxKey, -1, false, 1)], 2)
        || Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing 5.11: Start process of the fine-grained set

The abstract specification is obtained from the concrete specification, as described in the methodology. This means that the process *Atomic* is added in the abstract specification, with the corresponding actions and communication. This process is also added in the parallel composition, as it is a shared resource. Additionally, the processes representing the action refinement are added to the specification, and they are the same as for the abstract specification of the fine-grained set. From this abstract specification, the elements considering the prime actions should be removed.

5.1.3 Optimistic set

The optimistic algorithm is lock-based, having a lock for every node in the list in a similar manner as with the fine-grained set. However, the fine-grained set requires many lock acquisitions and releases. The optimistic set mitigates this overload of locking by traversing the list without acquiring any locks. Then, in the methods provided, the node that should be modified is locked, as well as its predecessor. Finally, the nodes are validated to ensure that they have not been modified by another thread. If the validation fails, the locks are released and the methods restart. The motivation for this design principle is that the probability that the method would restart is small. The data requirements are the same as given as for the fine-grained set. The methods provided are equivalent to the methods given in the previous versions of the set. Both methods modify the list only when holding the appropriate locks. On top of the *add* and *remove* methods, there is a *validate* method that checks the correctness of the locked nodes. The implementation *add* and *validate* is given in 5.12. As the manipulation of a node occurs only when the corresponding locks are acquired, the node is modified in an exclusive manner. Therefore, the authors claim that this concurrent data structure is linearizable.

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = pred.next;
      while (curr.key <= key) {
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred, curr)) {
            if (curr.key == key) {
               return false;
            } else {
               Node node = new Node(item);
               node.next = curr;
               pred.next = node;
               return true;
```

```
            }
          }
      } finally {
          pred.unlock(); curr.unlock();
   }}}

private boolean validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) {
      if (node == pred)
         return pred.next == curr;
      node = node.next;
      }
   return false;
   }
```

Listing 5.12: Implementation of the *add* method of the optimistic set

The concrete specification for the optimistic set is modeled first. As mentioned above, the data requirements are identical with the requirements for the fine-grained set, namely the set is stored as a linked list of nodes, where each node can be locked individually. The same principles for modeling the linked list given for the fine-grained set apply to the optimistic set. On top of the process representing the shared linked list, there is a *Validate* process, representing the validation method of the set. Listing 5.13 described this process, as well as the required actions and the modifications applied to the linked list. The process *Validate* checks that the node to be modified and its predecessor still exist in the linked list, and that the next node of the predecessor points to the current node. The remainder of the data specification can be found in Appendix B, Listing B.3.

```
act snd_ValidateNodes: Node#Node; rcv_ValidateNodes: Node#Node;
    ValidateNodes: Node#Node;
    snd_Validation: Bool; rcv_Validation: Bool; Validation: Bool; NextIteration;

proc LinkedList(nodes: ArrayNodes, nextLoc: Nat) =
    sum element: Int. snd_ReadNode(element, nodes(element)).
        LinkedList(nodes, nextLoc) +
    sum elementN: Int. snd_ReadNext(elementN, nodes(next(nodes(elementN)))).
        LinkedList(nodes, nextLoc) +
    sum n: Int. (!lock(nodes(n))) -> rcv_LockNode(n).
        LinkedList(nodes[n->lockNode(nodes(n))], nextLoc) +
    sum nn: Int. rcv_UnlockNode(nn).
        LinkedList(nodes[nn->unlockNode(nodes(nn))], nextLoc) +
    sum item1, loc: Nat. rcv_InsertItem(item1, loc).
        LinkedList(insertNode(nodes, nextLoc, item1, loc), (nextLoc+1)) +
    sum loc: Nat. rcv_RemoveItem(loc). LinkedList(removeNode(nodes, loc), nextLoc)
       +
    sum predNode, curr: Node. rcv_ValidateNodes(predNode, curr).
        Validate(nodes, predNode, curr, 0). LinkedList(nodes, nextLoc);
proc Validate(nodes: ArrayNodes, predNode: Node, curr: Node, nd: Int) =
    (nodes(nd)!= null && key(nodes(nd)) <= key(predNode))
       -> ((nodes(nd) == predNode)
           -> ((nodes(next(nodes(nd))) == curr)
              -> snd_Validation(true) <> snd_Validation(false))
           <> NextIteration. Validate(nodes, predNode, curr, next(nodes(nd))))
       <> snd_Validation(false);
```

Listing 5.13: Data specification of the optimistic set

The second step consists of introducing a process for each of the two method, namely *add* and *remove*. The implementation of the *add* method is given in Listing 5.12. The implementation of the *remove* method can be found in Appendix A, Listing A.3. The processes for modeling the *add* method are given in Listing 5.14. The process *Add* is the start point of modeling the *add* method. As mentioned in the previous sections, the process should be parameterized with the id of the thread and the item to be added, and it should start with a prime invocation, denoted by the action *CallAddPrime*. The method contains two nested loops, which are modeled by introducing two processes, *AddWhile* and *AddWhile2*. The first while-loop reads the head of the list and its successor. The second while-loop starts from these two nodes and traverses the list without locking until the correct position is reached. At this point, the current node and

its predecessor are locked and validated. If the validation succeeds, meaning that the nodes have not been changed by a parallel thread, then the process continues with one of two options: either the node exists, and the item is not added; or the converse occurs. However, if the validation fails, the process restarts with reading the head of the list. The method concludes with the action *ReturnAddPrime*.

```
proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). AddWhile(id, itemToAdd);
proc AddWhile(id: Nat, itemToAdd: Nat) = sum predNode: Node.
    rcv_ReadNode(0, predNode).
    sum curr: Node. rcv_ReadNext(mem(predNode), curr).
    AddWhile2(id, itemToAdd, predNode, curr);
proc AddWhile2(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
        -> sum nextCurr: Node.  rcv_ReadNext(mem(curr), nextCurr).
           AddWhile2(id, itemToAdd, curr, nextCurr)
        <> snd_LockNode(mem(predNode)).  snd_LockNode(mem(curr)).
           snd_ValidateNodes(lockNode(predNode), lockNode(curr)).
           (rcv_Validation(true). ((key(curr) == hashCode(itemToAdd))
           -> snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
              ReturnAddPrime(id, false)
           <> snd_InsertItem(itemToAdd, mem(predNode)).
              snd_UnlockNode(mem(predNode)).
              snd_UnlockNode(mem(curr)). ReturnAddPrime(id, true)) +
           rcv_Validation(false). snd_UnlockNode(mem(predNode)).
           snd_UnlockNode(mem(curr)). AddWhile(id, itemToAdd));
```

Listing 5.14: Method specification of the optimistic set

The third step in building the concrete specification is defining the finite client, which is the same as the one defined for the previous sets. Finally, the initial process can be found in Listing 5.15. The reasoning concerning the previous sets applies for this process as well, when it relates to hiding, allowing, communicating and renaming actions. Then, the system can be defined as the parallel composition of the linked list and two threads, each executing two operations.

```
init
    hide({
        GetHashCode, InsertItem, RemoveItem, LockNode, UnlockNode, ReadNode,
        ReadNext, ValidateNodes, NextIteration, Validation
    }, allow({
        CallAdd, ReturnAdd, CallRemove, ReturnRemove, InsertItem, RemoveItem,
        GetHashCode, LockNode, UnlockNode, ReadNode, ReadNext, ValidateNodes,
        NextIteration, Validation
    }, comm({
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_LockNode | snd_LockNode -> LockNode,
        rcv_UnlockNode | snd_UnlockNode -> UnlockNode,
        rcv_ReadNode | snd_ReadNode -> ReadNode,
        rcv_ReadNext | snd_ReadNext -> ReadNext,
        snd_ValidateNodes | rcv_ValidateNodes -> ValidateNodes,
        snd_Validation | rcv_Validation -> Validation
  }, rename({
        CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
  }, LinkedList((lambda n: Nat.null)[0->node(MinKey, MinKey, 1, false, 0)]
        [1->node(MaxKey, MaxKey, -1, false, 1)], 2)
        || Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing 5.15: Start process of the optimistic set

The abstract specification is obtained from the concrete specification, as described in the methodology. The process *Atomic* and the corresponding elements are added to the abstract specification. Additionally, the processes concerning the invocations and responses are added to the specification, and the relevant elements pertaining to the prime actions should be removed. The processes resulting from this mechanism are equivalent to those defined in the abstract specification of the previous sets.

5.1.4 Lazy set

The lazy algorithm optimizes the optimistic algorithm by ensuring that the list is traversed once, compared to the two traversals done by the optimistic algorithm. This is achieved by adding a *marked* field to each node that reflects whether that node still belongs to the set. This concept impacts the implementation of the *validate* method, which can be seen in Listing 5.16, which does not need to traverse the list in order to check that a node is reachable from the head of the list. Alternatively, this method verifies that the current node, as well as its predecessor, have not been marked for removal, and that the predecessor's next field still references the current node. In this context, the method traverse the list in a similar manner as the optimistic set, without acquiring any locks. The addition of an element follows closely the algorithm given in the optimistic set, while using the update *validate* method. However, the removal of an element currently occurs in two steps: first, the node is logically removed by marking the node; second, the node is physically removed by redirecting the next field of the predecessor node. The implementation of the *remove* method is also given in Listing 5.16. The data requirements have been updated to include the *marked* field in each node, as well as the corresponding mappings to change this field, as can be seen in Listing 5.17. Additionally, this listing describes the modifications realized on the *Validate* process.

```
public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = head.next;
      while (curr.key < key) {
         pred = curr; curr = curr.next;
      }
      pred.lock();
      try {
         curr.lock();
         try {
            if (validate(pred, curr)) {
               if (curr.key != key) {
                  return false;
               } else {
                  curr.marked = true;
                  pred.next = curr.next;
                  return true;
            }}
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
   }}}

private boolean validate(Node pred, Node curr) {
   return !pred.marked && !curr.marked && pred.next == curr;
}
```

Listing 5.16: Implementation of the *remove* method of the optimistic set

The concrete specification for the lazy set replicates the concrete specification of the optimistic set, and can be found in Appendix B, Listing B.4. As mentioned above, the difference lies in each node having an additional field, the Boolean *marked*, that indicates the removal of the node. The required variables and actions to access and modify this field have been added to the specification.

The implementation for both methods, *add* and *remove*, is given in Listing A.4. The processes for these methods behave equivalently with the method processes for the optimistic set. Thus, each process has the necessary parameters (i.e. id of the thread, item to be added or removed), starts with a prime invocation, continues with the required internal actions, and concludes with a prime response. Both methods employ nested loops, which are modeled through two recursive processes per method. However, the validation is realized using the *marked field*. Additionally, the processes required for the *remove* method include two actions for removing a node, an action for the logical removal of the node and an action for the physical removal of the node.

```
sort Node = struct null | node(itm: Int, ky: Int, nxt: Int,
            mkd: Bool, lck: Bool, mm: Nat);
act snd_MarkRemoved: Int; rcv_MarkRemoved: Int; MarkRemoved: Int;

proc LinkedList(nodes: ArrayNodes, nextLocation: Nat) =
    %equivalent behavior as optimistic set +
    sum nnn: Int. rcv_MarkRemoved(nnn).
        LinkedList(nodes[nnn->markNode(nodes(nnn))], nextLocation);
proc Validate(nodes: ArrayNodes, predNode: Nat, curr: Nat) =
    (!marked(nodes(predNode)) && !marked(nodes(curr)) &&
    next(nodes(predNode)) == curr)
        -> snd_Validation(true) <> snd_Validation(false);
```

Listing 5.17: Data specification of the lazy set

Subsequently, the concrete specification includes the finite client and the initial process. These elements are similar to the ones described in the previous set. The action for marking a removed node is added in the respective blocks of the initial process, specifically the hide, allow and comm blocks. Finally, the system equals the parallel composition of the linked list and two threads, each executing two operations.

The abstract specification is obtained from the concrete specification, as described in the methodology. The processes concerning the atomic protocol, the prime invocations and the prime responses are added to the abstract specification, as well as the corresponding elements. The relevant elements pertaining to the prime actions should be removed. The processes resulting from this mechanism are equivalent to those defined in the abstract specification of the previous sets.

5.2. Non-blocking queue

Shann et al.[20] present an algorithm for a non-blocking queue, and claim that this queue is linearizable. However, it has already been established that this queue is not linearizable [17]. In this section, the queue is modeled according to the methodology described in the previous chapter.

The data requirements for the non-blocking queue are given in Listing 5.18. The queue is stored as a circular array, with fixed length L. Each item of the array consists of two parts: the value that should be stored in the queue and a counter that represents the number of references that a particular memory location has held so far. Additionally, two methods are possible on a queue: *enqueue*, which adds an element at the rear of the queue, and *dequeue*, which removes an element from the front of the queue. There are also two variables that represent the *front* and the *rear* of the queue. The queue uses non-blocking synchronization, achieved through the CAS instruction.

```
Q: array [0..L-1] of structure {val: qitem; ref: counter}
FRONT, REAR: counter
```

Listing 5.18: Data structures of the non-blocking queue

The implementation of the *enqueue* method is given in Listing 5.19. This methods adds an element in the queue only when the queue still has capacity and the *rear* stores no element. If the queue is full, then the process waits until a parallel process dequeues an element. If the variable *rear* stores an element, then another process might operate on the queue at the same time. In this case, the current process helps the other processes by incrementing the *rear* variable. The *dequeue* behaves in a simialr fashion, and can be found in Appendix A, Listing A.5.

```
enqueue(X: qitem)
    enq_try_again:
    rear := REAR
    x := Q[rear mod L]
    if rear != REAR then goto enq_try_again endif
    if rear == FRONT+L then goto enq_try_again endif
    if x.val == NULL then
        if CAS(Q[rear mod L], x, <X || x.ref+1>) then
            CAS(REAR, rear, rear+1)
            return
        endif
```

```
    elseif
        CAS(REAR, rear, rear+1) # help others increment REAR
    endif
    goto enq_try_again
```

Listing 5.19: Implementation of the *enqueue* method of the non-blocking queue

The concrete specification should be constructed according to the method described in Section 4.1. To achieve this, the first step is identifying the data requirements, which are shown in Listing 5.18. The non-blocking queue is stored as an array of items, modeled as a function. The necessary structures and functions to manipulate the array should be created. This specification contains a process representing the shared resources, the *QueueInterface*. This process models the behavior of the low-level instruction CAS. This object contains three different resources that should be accessed, i.e. an element in the array, the rear, and the front. Thus, there are three instances of the CAS instruction, each defined per shared resource. Modeling these concepts results in the data specification shown in Listing 5.20. Further specification details can be found in Appendix B, Listing B.5.

```
sort Null = struct null;
sort Item = struct item(nl: Null, ref: Nat) | item(value: Nat, ref: Nat);
sort Array = Nat -> Item;
map L: Pos; isNull: Item -> Bool;

act rcv_ReadRear: Nat; snd_ReadRear: Nat; ReadRear: Nat;
    rcv_ReadFront: Nat; snd_ReadFront: Nat; ReadFront: Nat;
    rcv_ReadElement: Nat#Item; snd_ReadElement: Nat#Item; ReadElement: Nat#Item;
    rcv_Result: Bool; snd_Result: Bool; Result: Bool;
    rcv_CASRear: Nat#Nat; snd_CASRear: Nat#Nat; CASRear: Nat#Nat;
    rcv_CASFront: Nat#Nat; snd_CASFront: Nat#Nat; CASFront: Nat#Nat;
    rcv_CASElement: Nat#Item#Item; snd_CASElement: Nat#Item#Item;
    CASElement: Nat#Item#Item;

proc QueueInterface(q: Array, rear: Nat, front: Nat) =
     rcv_ReadRear(rear). QueueInterface(q, rear, front) +
     rcv_ReadFront(front). QueueInterface(q, rear, front) +
     sum n: Nat. rcv_ReadElement(n, q(n)). QueueInterface(q, rear, front) +
     sum oldR, newR: Nat. (rcv_CASRear(oldR, newR) | snd_Result(oldR == rear)).
        ((oldR == rear) -> QueueInterface(q, newR, front)
       <> QueueInterface(q, rear, front)) +
     sum oldF, newF: Nat. (rcv_CASFront(oldF, newF) | snd_Result(oldF == front)).
         ((oldF == front) -> QueueInterface(q, rear, newF)
        <> QueueInterface(q, rear, front)) +
     sum n: Nat, oldX, newX: Item. (rcv_CASElement(n, oldX, newX) |
         snd_Result(q(n) == oldX)).
         ((q(n) == oldX) -> QueueInterface(q[n->newX], rear, front)
        <> QueueInterface(q, rear, front)) ;
```

Listing 5.20: Data specification of the non-blocking queue

The second step is identifying all the methods provided by the concurrent data structure, and introduce a process for each of the methods. The non-blocking queue provides two methods: *enqueue* and *dequeue*. The processes, and corresponding actions, for modeling the *enqueue* method are given in Listing 5.21. The processes *Enqueue* and *EnqueueTryAgain* model the behavior of the *enqueue* method. The processes are parameterized with the id of the thread that initiated the call, and the item to be added. Additionally, the method call should start with a prime invocation. Thus, the action *CallEnqueuePrime* is added as the initial action, and it contains the same parameters as the processes. There is a one-to-one mapping between the statements in the method implementation and the actions in the method specification. To model the jump statement represented by *goto*, the recursive process *EnqueueTryAgain* was introduced. The method first checks that the required conditions for adding an element are fulfilled. If the conditions are met, then the process executes two CAS instructions, one for the element to be added and one for the *rear* of the list. Conversely, the process executes one CAS instruction to update the *rear*. The method concludes with a response, denoted by the action *ReturnEnqueuePrime*. The actions for the non-prime invocations and responses are also defined. These actions are used to ensure that the histories generated by this specification can be compared to the histories generated by the abstract specification.

```
act CallEnqueue: Nat#Nat; ReturnEnqueue: Nat;
    CallEnqueuePrime: Nat#Nat; ReturnEnqueuePrime: Nat;

proc Enqueue(tid: Nat, v: Nat) = CallEnqueuePrime(tid, v). EnqueueTryAgain(tid, v);
proc EnqueueTryAgain(tid: Nat, v: Nat) = sum r1: Nat. snd_ReadRear(r1).
     sum x: Item. snd_ReadElement((r1 mod L), x).
     sum r2: Nat. snd_ReadRear(r2).
     ((r1 != r2) -> EnqueueTryAgain(tid, v)
     <> sum f1: Nat. snd_ReadFront(f1).
     ((r1 == f1 + L) -> EnqueueTryAgain(tid, v)
         <> ((isNull(x)) ->
            ((snd_CASElement((r1 mod L), x, item(v, ref(x)+1)) | rcv_Result(true)).
            (sum b1: Bool. (snd_CASRear(r1, (r1+1)) | rcv_Result(b1))).
            ReturnEnqueuePrime(tid) +
            (snd_CASElement((r1 mod L), x, item(v, ref(x)+1)) | rcv_Result(false)).
            EnqueueTryAgain(tid, v))
            <> sum b2: Bool. (snd_CASRear(r1, (r1+1)) | rcv_Result(b2)).
            EnqueueTryAgain(tid, v))));
```

Listing 5.21: Method specification of the non-blocking queue

The third step in building the concrete specification is defining the finite client, which can be found in Listing 5.22. The process *Thread* has an identifier and a bounded number of operations that can execute. To allow the thread to make progress, a loop is needed, which is modeled through a second process. This process calls non-deterministically the two methods provided by the object.

```
proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
     (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Enqueue(id, head(elms)) + Dequeue(id)).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);
```

Listing 5.22: Client of the coarse-grained set

The initial process for this specification can be found in Listing 5.23. The internal actions are hidden, to ensure that the generated LTS consists only of invocations and responses. The allowed actions in this specification are all the actions that are not prime, and the required communication is enforced. Finally, the prime invocations and prime responses are renamed to their non-prime counterparts. Then, the process is a parallel composition of the queue and two threads, each executing two operations.

```
init
    hide({ ReadRear, ReadFront, ReadElement, CASElement, CASFront, CASRear, Result
    }, allow({ CallEnqueue, ReturnEnqueue, CallDequeue, ReturnDequeue,
         ReadRear, ReadFront, ReadElement,
         CASRear | Result, CASFront | Result, CASElement | Result
    }, comm({ rcv_ReadRear | snd_ReadRear -> ReadRear,
         rcv_ReadFront | snd_ReadFront -> ReadFront,
         rcv_ReadElement | snd_ReadElement -> ReadElement,
         rcv_Result | snd_Result -> Result,
         rcv_CASRear | snd_CASRear -> CASRear,
         rcv_CASFront | snd_CASFront -> CASFront,
         rcv_CASElement | snd_CASElement -> CASElement
    }, rename({ CallEnqueuePrime -> CallEnqueue, CallDequeuePrime -> CallDequeue,
         ReturnEnqueuePrime -> ReturnEnqueue, ReturnDequeuePrime -> ReturnDequeue
  }, QueueInterface((lambda n:Nat.item(null, 0)), 0, 0) ||
       Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing 5.23: Start process of the non-blocking queue

The abstract specification is obtained from the concrete specification, as described in the methodology. This means that the process *Atomic* and its corresponding elements are added in the abstract specification. Additionally, the processes representing the action refinement are added to the specification, as shown in Listing 5.24. To avoid conflicts, the actions with the same name as the processes should be removed from the abstract specification, as well as the *rename* operator.

```
proc CallEnqueuePrime(tid: Nat, v: Nat) = CallEnqueue(tid, v).
     snd_StartAtomicBlock;
proc ReturnEnqueuePrime(tid: Nat) = snd_EndAtomicBlock . ReturnEnqueue(tid);
proc CallDequeuePrime(tid: Nat) = CallDequeue(tid). snd_StartAtomicBlock;
proc ReturnDequeuePrime(tid: Nat, v: Nat) = snd_EndAtomicBlock.
     ReturnDequeue(tid, v);
```

Listing 5.24: Action refinement of the non-blocking queue

The LTSs generated by these two specifications are given as input to mCRL2, to check for branching bisimilarity. The tool returns that the two specifications are not branching bisimilar. The verification of weak trace equivalence returns false as well, and produces a trace that serves as a counterexample to linearizability. The trace is given in Figure 5.1. This trace is a history that is found in the concrete specification, but not in the abstract one. Thus, the object allows the following behavior: call to dequeue an element, enqueue element 3 (which becomes the first element in the queue), enqueue element 4, and return the dequeue method with value 4. This is a violation of the FIFO (First-In-First-Out) principle of the queue, since the element that should have been dequeued is 3. Consequently, there is no legal sequential history equivalent to this execution history, implying that this concurrent history is not linearizable. Thus, the queue is not linearizable.

CallDequeue(1)
CallEnqueue(2, 3)
ReturnEnqueue(2)
CallEnqueue(2, 4)
ReturnDequeue(1, 4)

Figure 5.1: Trace produced by mCRL2 as counterexample for non-blocking queue

6. Results

This chapter presents the results of providing the case studies as an input to the mCRL2 tool. The experiments were run on the tool version 201707.1, on a computer equipped with 5-core Intel CPU @ 2.80 GHz and 4GB of memory. All case studies were verified with a finite client consisting of n threads, each thread calling m operations, where $m, n \in \{2, 3\}$. Thus, if the two specifications are equivalent, then the data structure is linearizable$_{m,n}$, for the selected parameters. Table 6.1 includes the results. For each case study, the number of states and transitions in the concrete and abstract specifications are presented, denoted by the columns **St/Tr**$_{conc}$ and **St/Tr**$_{abs}$. Finally, the times required to check the equivalences of the given specifications are given in the last columns, where **BB** stands for branching bisimulation and **WTE** stands for weak trace equivalence. These times were calculated as the average of five runs of the tool.

Case Study	**Linearizable**$_{m,n}$	**St/Tr**$_{conc}$	**St/Tr**$_{abs}$	**BB**	**WTE**
Treiber's stack	Yes [2, 2]	1724 / 3275	1662 / 2121	0.04	0.05
Treiber's stack	Yes [2, 3]	12750 / 24391	10097 / 12853	0.31	0.43
Treiber's stack	Yes [3, 2]	205634 / 564862	87389 / 124740	5.8	8.91
Coarse-grained set	Yes [2, 2]	1545 / 2844	1749 / 2106	0.04	0.05
Coarse-grained set	Yes [2, 3]	6377 / 11824	7133 / 8570	0.15	0.21
Coarse-grained set	Yes [3, 2]	55444 / 145043	50488 / 64729	1.63	2.20
Fine-grained set	Yes [2, 2]	5077 / 9006	3305 / 3720	0.09	0.12
Fine-grained set	Yes [2, 3]	37666 / 68665	21991 / 24548	0.75	0.92
Optimistic set	Yes [2, 2]	29312 / 47467	3713 / 4128	0.40	0.48
Optimistic set	Yes [2, 3]	234332 / 3893344	25435 / 28154	3.92	5.36
Lazy set	Yes [2, 2]	24496 / 41431	3565 / 3980	0.35	0.42
Non-blocking queue	No [2, 2]	2966 / 5707	1198 / 1423	0.05	0.06
Non-blocking queue	No [3, 2]	870243 / 2440900	41967 / 54109	18.42	27.06

Table 6.1: Results of verifying the case studies

From the previous table, it is noticeable that branching bisimulation and weak trace equivalence produce the same results for all case studies. Intuitively, this observation is sustained by the mode in which the specifications are built. That is, the abstract specification is built from the concrete specification by restricting its behavior, but this restriction preserves the branching structure of the labeled transitions systems. The number of possible interleavings is reduced in the abstract specification, which affects the ordering of the internal actions. The coarse-grained set deviates from this idea, since this set behaves in a similar manner with the abstract specification (i.e. invocation, acquiring a lock, releasing a lock, response). Thus, the atomic protocol does not restrict the possible interleavings, but increases the number of states. Provided that the concurrent data structure is linearizable, the external behavior of both LTSs the same. The advantage of branching bisimulation is its efficiency, outputting the result in less time, which is visible in all case studies.

From the table, it becomes apparent that the number of states increase with the complexity of the algorithm. In the optimistic and lazy implementations of the set, the increase in the number of states is considerable. These set implementations do not have fixed linearization points. Both implementations traverse the set without locking, and lock only when reaching the appropriate nodes. Then, if the validation of the nodes fails, both implementations restart the process from the beginning of the set. These versions of the set should be used when conflicts are rare. However, the mCRL2 tool generates all the possible traces, containing all the possible invalidations that can be present in the system. This greatly reduces the efficiency of these sets, since each method has to restart many times, resulting in the big amount of states and transitions.

This approach does not scale well when the number of threads is increased, but scales better when only the number of operations is increased. Adding an extra thread in the system results in having many more interleavings of the different provided methods. Furthermore, these case studies suggest the validity of the approach. The results of these case studies match those that have been claimed by previous work.

7. Related Work

Verifying linearizability has been the focus of many studies, since linearizability is one of the main correctness properties of concurrent data structures. Herlihy and Wing [15] defined linearizability and introduced a method of verification. Their approach requires defining a function that maps every state of the object to the set of possible abstract values that can occur in that state. This approach results in a manual proof, requiring the domain knowledge of the analyzed object. Manual proofs are prone to errors that are hard to identify.

Lowe [18] proposes a testing framework for linearizability. The first step is to randomly generate histories of the concurrent object. Then, each history should be checked whether it is linearizable. To perform this check, the paper uses five different algorithms, varying in efficiency and generality. However, testing does not serve as a formal verification technique, but rather complementary to one.

Liang and Feng [16] propose a program logic with an instrumentation mechanism that verifies algorithms with non-fixed linearization points. This mechanism relates concrete implementations with abstract operations, allowing the abstract operations to execute simultaneously with the concrete code. One challenge they encounter is proving that their logic is sound with respect to linearizability. Their work is similar to Vafeiadis' [22], which presents a technique for automatically verifying linearizability. This technique was implemented into a tool, and applied on different concurrent algorithms. Furthermore, Vafeiadis also contributed to giving an aspect-oriented proof for linearizability [13]. This proof mechanism involves concluding linearizability by verifying four simpler properties, where each property can be established individually. However, this proof mechanism is neither complete, nor fully automated.

Bouajjani et al. tackle the problem of formally proving linearizability of a concurrent priority queue [5]. They represent the queue as an automaton. Using this representation, verification of linearizability can be transformed into a reachability analysis or invariant checking. This implies that establishing linearizability is decidable for finite-state systems. This proposed solution is based on their previous work [4], which also involves the reduction of checking linearizability to control-state reachability.

Model checking has also been employed in [6]. This method focuses on deterministic sequential specifications. Their contribution includes building an automatic linearizability checker. The verification consists of systematically identifying the sequential executions of the object, and then verifying if every concurrent execution is equivalent to some sequential one. This approach is complete, but only sound regarding some given inputs. Cerný et al. [7] use model checking to verify linearizability for concurrent linked-list implementations. Their verification consists of checking reachability in a method automaton that follows the manipulation of the concurrent data structure. This solution also has the limitation of bounding the number of operations. As can be observed, the challenge encountered in all model checking techniques is the state-space explosion, which enforces the bound on the number of operations and processes.

8. Conclusion

This work investigated the process of modeling and verifying concurrent data structures, where the main property to be verified is linearizability. As defined by Herlihy and Wing [15], linearizability considers the execution histories of an object, where an object is an instance of the data structure. An object can be manipulated only through its methods, where each method starts with an invocation (i.e. the method call is initiated) and ends with a response (i.e. the method finishes executing). Then, an execution history consists only of these invocations and responses. To prove linearizability, prove that every concurrent history is equivalent to some sequential history. When this property is met, the concurrent data structure can be analyzed using sequential reasoning rather than complex concurrent reasoning. The reason behind this follows from the fact that linearizability ensures that all possible interleavings respect the sequential specification of the object.

The approach explored in this work consists of building two specifications, a concrete one and an abstract one. These specifications are built using the mCRL2 specification language, which is a process algebra. The concrete specification formalizes the behavior of the implementation by introducing processes for each method and each shared resource. By definition, investigating linearizability requires invocations and responses of the methods. Actions representing these events are added in the corresponding processes. To express the executions of methods, a process representing a thread should be introduced, which calls all the provided methods non-deterministically. To allow for concurrent processing of the object, a number of threads run in parallel, which represent the client of the object. Both the number of threads and the number of methods called should be bounded, resulting in a finite client. This client is needed to ensure that the corresponding LTS can be generated. The restriction on the client motivates the need for $\mathit{linearizability}_{m,n}$, where n represents the number of threads, and m represents the number of operations per threads. Thus, it follows that this methodology can establish only $\mathit{linearizability}_{m,n}$, a restricted version of linearizability that considers the bounds imposed by the model checking approach.

The abstract specification is obtained from the concrete specification by adding an atomic layer on top of it. This is realized by appending an atomic block around the internal actions in the body of every method. Thus, the possible interleavings occur between the invocations, responses, or atomic blocks. The execution of the internal actions is purely sequential. The same finite client should be used in this specification to generate the LTS. By construction, this abstract specification is linearizable, in the sense that all the histories contained in its labeled transition system are linearizable.

The last step in verifying linearizability is comparing these two specifications using branching bisimulation. When mCRL2 validates their equivalence, then one can conclude that the object on which these specifications are based is linearizable. However, if the two specifications are not branching bisimilar, then the two specifications should be checked using weak trace equivalence. The tool produces a counterexample trace when the weak trace equivalence cannot be concluded. This trace serves as an execution history that is not linearizable, which suffices to conclude that the object is not linearizable.

This methodology was applied on a number of case studies. The results matched the literature, namely correct and incorrect implementations of objects were detected in all cases. Regarding the correct implementations, one can conclude that they are $\text{linearizable}_{m,n}$, where the two parameters originate from the finite client defined in the specifications. Unfortunately, this approach does not scale well, since the number of states increases exponentially with the number of threads or operations.

Future work could investigate how to improve the scalability of the approach, by possibly analyzing only parts of the traces or reducing the number of internal actions used in a specification. Furthermore, the arguments for verification should be formalized to ensure that linearizability can be verified in a formal setting. As mentioned, this work establishes $\text{linearizability}_{m,n}$. One could explore whether it is possible to generalize this approach, by concluding linearizability from establishing it for n threads and m operations. Finally, an algorithm could be developed to automate the translation from a concrete specification to an abstract specification.

Bibliography

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29:66–76, 1996.

[2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. *Proc. 19th Int'l Conf. Computer Aided Verification*, pages 477–490, 2007.

[3] J. Baeten, T. Basten, and M. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2010.

[4] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza. On reducing linearizability to state reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 95–107, 2015.

[5] A. Bouajjani, C. Enea, and C. Wang. Checking linearizability of concurrent priority queues. *CONCUR*, 2017.

[6] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340, 2010.

[7] P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 465–479, 2010.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. *FORTE'04*.

[9] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. Snzi: Scalable nonzero indicators. *Proc. 26th Ann. ACM Symp. Principles of Distributed Computing*, pages 13–22, 2007.

[10] J. Groote and M. Mousavi. *Modeling and analysis of communicating systems*. 2014.

[11] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. S. III, and N. Shavit. A lazy concurrent list-based algorithm. *OPODIS '05*.

[12] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *SPAA '04*.

[13] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. *CONCUR*, pages 242–256, 2013.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[15] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[16] H. Liang and X. Feng. Modular verification of linearizability with non-fixed linearization points. *PLDI*, pages 459–470, 2013.

[17] Y. Liu, W. Chen, and et al. Verifying linearizabiity via optmized refinement checking. *IEEE Trans. on Soft. Eng.*, 39(7):1018–1039, 2013.

[18] G. Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[19] M. Moir and N. Shavit. Concurrent data structures. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.

[20] C. H. Shann, T. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. *Proc. Seventh Int'l Conf. Parallel and Distributed Systems*, pages 470–475, 2000.

[21] R. Treiber. Systems programming: Coping with parallelism. *Technical Report RJ 5118, IBM Almaden Research Center*, 1986.

[22] V. Vafeiadis. Automatically proving linearizability. *CAV*, 2010.

[23] R. van Glabbeek. The linear time-branching time spectrum i. *CONCUR*, 1990.

[24] M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 125–135, 2008.

[25] X. Yang, J.-P. Katoen, H. Lin, and H. Wu. Proving linearizability via branching bisimulation. *CoRRabs/1609.07546*, 2016.

[26] X. Yang, J.-P. Katoen, H. Lin, and H. Wu. Verifying concurrent stacks by divergence-sensitive bisimulation. *CoRR abs/1701.06104*, 2017.

A. Implementation of concurrent data structures

Coarse-grained set

```
public class CoarseList<T> {
   private Node head;
   private Lock lock = new ReentrantLock();

   public CoarseList() {
      head = new Node(Integer.MIN_VALUE);
      head.next = new Node(Integer.MAX_VALUE);
   }

   public boolean add(T item) {
      Node pred, curr;
      int key = item.hashCode();
      lock.lock();
      try {
         pred = head;
         curr = pred.next;
         while (curr.key < key) {
            pred = curr;
            curr = curr.next;
         }
         if (key == curr.key) {
            return false;
         } else {
            Node node = new Node(item);
            node.next = curr;
            pred.next = node;
            return true;
         }
      } finally {
       lock.unlock();
   }}

   public boolean remove(T item) {
      Node pred, curr;
      int key = item.hashCode();
      lock.lock();
      try {
         pred = head;
         curr = pred.next;
         while (curr.key < key) {
            pred = curr;
            curr = curr.next;
         }
         if (key == curr.key) {
            pred.next = curr.next;
            return true;
         } else {
            return false;
         }
      } finally {
         lock.unlock();
   }}
```

Listing A.1: Implementation of the coarse-grained set

Fine-grained set

```
public boolean add(T item) {
   int key = item.hashCode();
   head.lock();
   Node pred = head;
   try {
      Node curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
         return false;
         }
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         return true;
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
   }}

public boolean remove(T item) {
   Node pred = null, curr = null;
   int key = item.hashCode();
   head.lock();
   try {
      pred = head;
      curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            pred.next = curr.next;
            return true;
         }
         return false;
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
   }}
```

Listing A.2: Implementation of the fine-grained set

Optimistic set

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = pred.next;
      while (curr.key <= key) {
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred, curr)) {
            if (curr.key == key) {
               return false;
            } else {
               Node node = new Node(item);
               node.next = curr;
               pred.next = node;
               return true;
         }}
      } finally {
         pred.unlock(); curr.unlock();
   }}}

public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = pred.next;
      while (curr.key < key) {
         pred = curr; curr = curr.next;
      }
      pred.lock(); curr.lock();
      try {
         if (validate(pred, curr)) {
            if (curr.key == key) {
               pred.next = curr.next;
               return true;
            } else {
               return false;
         }}
      } finally {
         pred.unlock(); curr.unlock();
   }}}

private boolean validate(Node pred, Node curr) {
   Node node = head;
   while (node.key <= pred.key) {
      if (node == pred)
         return pred.next == curr;
      node = node.next;
      }
   return false;
   }
```

Listing A.3: Implementation of the optimistic set

Lazy set

```
public boolean add(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = head.next;
      while (curr.key < key) {
         pred = curr; curr = curr.next;
      }
      pred.lock();
      try {
         curr.lock();
         try {
            if (validate(pred, curr)) {
               if (curr.key == key) {
                  return false;
               } else {
                  Node node = new Node(item);
                  node.next = curr;
                  pred.next = node;
                  return true;
               }}
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
   }}}

public boolean remove(T item) {
   int key = item.hashCode();
   while (true) {
      Node pred = head;
      Node curr = head.next;
      while (curr.key < key) {
         pred = curr; curr = curr.next;
      }
      pred.lock();
      try {
         curr.lock();
         try {
            if (validate(pred, curr)) {
               if (curr.key != key) {
                  return false;
               } else {
                  curr.marked = true;
                  pred.next = curr.next;
                  return true;
               }}
         } finally {
            curr.unlock();
         }
      } finally {
         pred.unlock();
   }}}

private boolean validate(Node pred, Node curr) {
   return !pred.marked && !curr.marked && pred.next == curr;
}
```

Listing A.4: Implementation of the lazy set

Non-blocking queue

```
Q: array [0..L-1] of structure {val: qitem; ref: counter}
FRONT, REAR: counter
enqueue(X: qitem)
    enq_try_again:
    rear := REAR
    x := Q[rear mod L]
    if rear != REAR then goto enq_try_again endif
    if rear == FRONT+L then goto enq_try_again endif
    if x.val == NULL then
        if CAS(Q[rear mod L], x, <X || x.ref+1>) then
            CAS(REAR, rear, rear+1)
            return
        endif
    elseif
        CAS(REAR, rear, rear+1) # help others increment REAR
    endif
    goto enq_try_again

dequeue(): qitem t
    deq_try_again:
    front := FRONT
    x := Q[front mod L]
    if front != FRONT then goto deq_try_again endif
    if front == REAR then goto deq_try_again endif
    if x.val != NULL then
        if CAS(Q[front mod L], x, <NULL || x.ref+1>) then
            CAS(FRONT, front, front+1)
            return(x.val)
        endif
    elseif
        CAS(FRONT, front, front+1) # help others increment FRONT
    endif
    goto deq_try_again
```

Listing A.5: Implementation of the non-blocking queue

B. Concrete specifications of case studies

Coarse-grained set

```
sort Node = struct null | node(item: Int, key: Int, next: Node);
map MinKey: Int; MaxKey: Pos; hashCode: Nat -> Nat;
    insert: Nat#Node -> Node; insertHelper: Nat#Nat#Node#Node -> Node;
    remove: Nat#Node -> Node; removeHelper: Nat#Nat#Node#Node -> Node;
var it : Nat; i, k: Int; n: Node;
    ci, ck: Int; cn: Node; el, kEl: Nat;
eqn MinKey = -1; MaxKey = 100;
    hashCode(it) = if(it >= MaxKey, it mod MaxKey, it);
    insert(el, node(i, k, n)) = insertHelper(el, hashCode(el), node(i, k, n), n);
    insertHelper(el, kEl, node(i, k, n), node(ci, ck, cn)) =
        if(ck == kEl, node(i, k, n),
            if(ck < kEl, node(i, k, insertHelper(el, kEl, n, next(n))),
                node(i, k, node(el, kEl, n))));
    remove(el, node(i, k, n)) = removeHelper(el, hashCode(el), node(i, k, n), n);
    removeHelper(el, kEl, node(i, k, n), node(ci, ck, cn)) =
         if(ck == kEl, node(i, k, cn),
             if(ck < kEl, node(i, k, removeHelper(el, kEl, n, next(n))),
                 node(i, k, n)));

act rcv_ReadHead: Node; snd_ReadHead: Node; ReadHead: Node;
    rcv_InsertItem: Nat; snd_InsertItem: Nat; InsertItem: Nat;
    rcv_RemoveItem: Nat; snd_RemoveItem: Nat; RemoveItem: Nat;
    rcv_Lock: Nat; snd_Lock: Nat; Lock: Nat;
    rcv_Unlock: Nat; snd_Unlock: Nat; Unlock: Nat;
    CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    CallRemove: Nat#Nat; ReturnRemove: Nat#Bool;
    CallRemovePrime: Nat#Nat; ReturnRemovePrime: Nat#Bool;
    GetHashCode: Nat; NextIteration;

proc LinkedList(nodes: Node) = rcv_ReadHead(nodes). LinkedList(nodes) +
    sum item1: Nat. rcv_InsertItem(item1). LinkedList(insert(item1, nodes)) +
    sum item2: Nat. rcv_RemoveItem(item2). LinkedList(remove(item2, nodes));

proc ReentrantLock = sum id: Nat. rcv_Lock(id). rcv_Unlock(id). ReentrantLock;

proc Add(id: Nat, itemToAdd: Nat) =  CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). snd_Lock(id).
    sum predNode: Node. snd_ReadHead(predNode).
    AddWhile(id, itemToAdd, predNode, next(predNode));
proc AddWhile(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
    -> NextIteration. AddWhile(id, itemToAdd, curr, next(curr))
    <> ((key(curr) == hashCode(itemToAdd))
        -> snd_Unlock(id). ReturnAddPrime(id, false)
        <> snd_InsertItem(itemToAdd). snd_Unlock(id). ReturnAddPrime(id, true));

proc Remove(id: Nat, itemToRemove: Nat) = CallRemovePrime(id, itemToRemove).
    GetHashCode(hashCode(itemToRemove)). snd_Lock(id).
    sum predNode: Node. snd_ReadHead(predNode).
    RemoveWhile(id, itemToRemove, predNode, next(predNode));
proc RemoveWhile(id: Nat, itemToRemove: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToRemove))
    -> NextIteration. RemoveWhile(id, itemToRemove, curr, next(curr))
    <> ((key(curr) == itemToRemove)
        -> snd_RemoveItem(itemToRemove). snd_Unlock(id).
           ReturnRemovePrime(id, true)
        <> snd_Unlock(id). ReturnRemovePrime(id, false));
```

```
proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
  (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Add(id, head(elms)) + Remove(id, head(elms))).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);

init
    hide({
        NextIteration, ReadHead, InsertItem, RemoveItem, Lock, Unlock, GetHashCode
    }, allow({
        CallAdd, ReturnAdd, CallRemove, ReturnRemove,
        ReadHead, InsertItem, RemoveItem, Lock, Unlock, GetHashCode, NextIteration
    }, comm({
        rcv_ReadHead | snd_ReadHead -> ReadHead,
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_Lock | snd_Lock -> Lock, rcv_Unlock | snd_Unlock -> Unlock
    }, rename({
        CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
    }, LinkedList(node(MinKey, MinKey, node(MaxKey, MaxKey, null))) ||
       ReentrantLock || Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing B.1: Concrete specification of the coarse-grained set

Fine-grained set

```
sort Node = struct null | node(item: Int, key: Int, next: Int, lock: Bool, mem: Nat
    );
sort ArrayNodes = Int -> Node;
map MinKey: Int; MaxKey: Pos; hashCode: Int -> Nat;
    lockNode: Node -> Node; unlockNode: Node -> Node;
    insertNode: ArrayNodes # Nat # Nat # Nat -> ArrayNodes;
    removeNode: ArrayNodes # Nat -> ArrayNodes;
var i, k, n: Int; m: Nat; l: Bool;
    an: ArrayNodes; loc: Nat; nextL: Nat; it: Nat;
eqn MinKey = -1; MaxKey = 100; hashCode(i) = i mod MaxKey;
    lockNode(node(i, k, n, l, m)) = node(i, k, n, true, m);
    unlockNode(node(i, k, n, l, m)) = node(i, k, n, false, m);
    insertNode(an, nextL, it, loc) = an[nextL -> node(it, hashCode(it), next(an(loc
        )),
       false, nextL)][loc -> node(item(an(loc)), key(an(loc)), nextL, lock(an(loc))
           , loc)];
    removeNode(an, loc) = an[loc -> node(item(an(loc)), key(an(loc)),
       next(an(next(an(loc)))), lock(an(loc)), mem(an(loc)))][next(an(loc)) -> null
           ];

act snd_ReadNode: Int#Node; rcv_ReadNode: Int#Node; ReadNode: Int#Node;
    snd_ReadNext: Int#Node; rcv_ReadNext: Int#Node; ReadNext: Int#Node;
    snd_LockNode: Int; rcv_LockNode: Int; LockNode: Int;
    snd_UnlockNode: Int; rcv_UnlockNode: Int; UnlockNode: Int;
    CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallRemove: Nat#Nat; ReturnRemove: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    CallRemovePrime: Nat#Nat; ReturnRemovePrime: Nat#Bool;
    rcv_InsertItem: Nat#Nat; snd_InsertItem: Nat#Nat; InsertItem: Nat#Nat;
    rcv_RemoveItem: Nat; snd_RemoveItem: Nat; RemoveItem: Nat; GetHashCode: Nat;

proc LinkedList(nodes: ArrayNodes, nextLoc: Nat) =
    sum element: Int. snd_ReadNode(element, nodes(element)).
        LinkedList(nodes, nextLoc) +
    sum elementN: Int. snd_ReadNext(elementN, nodes(next(nodes(elementN)))).
        LinkedList(nodes, nextLoc) +
    sum n: Int. (!lock(nodes(n))) -> rcv_LockNode(n).
        LinkedList(nodes[n->lockNode(nodes(n))], nextLoc) +
    sum nn: Int. rcv_UnlockNode(nn).
        LinkedList(nodes[nn->unlockNode(nodes(nn))], nextLoc) +
    sum item1, loc: Nat. rcv_InsertItem(item1, loc).
        LinkedList(insertNode(nodes, nextLoc, item1, loc), (nextLoc+1)) +
    sum loc: Nat. rcv_RemoveItem(loc). LinkedList(removeNode(nodes, loc), nextLoc);

proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). snd_LockNode(0).
    sum predNode: Node. rcv_ReadNode(0, predNode).
    sum curr: Node. rcv_ReadNext(mem(predNode), curr). snd_LockNode(mem(curr)).
    AddWhile(id, itemToAdd, predNode, curr);
proc AddWhile(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
       -> snd_UnlockNode(mem(predNode)). sum nn: Node. rcv_ReadNext(mem(curr), nn).
          snd_LockNode(mem(nn)). AddWhile(id, itemToAdd, curr, nn)
       <> ((key(curr) == hashCode(itemToAdd))
          -> snd_UnlockNode(mem(curr)). snd_UnlockNode(mem(predNode)).
             ReturnAddPrime(id, false)
          <> snd_InsertItem(itemToAdd, mem(predNode)). snd_UnlockNode(mem(curr)).
             snd_UnlockNode(mem(predNode)). ReturnAddPrime(id, true));

proc Remove(id: Nat, itemToRemove: Nat) =  CallRemovePrime(id, itemToRemove).
    GetHashCode(hashCode(itemToRemove)). snd_LockNode(0).
    sum predNode: Node. rcv_ReadNode(0, predNode).
    sum curr: Node. rcv_ReadNext(mem(predNode), curr). snd_LockNode(mem(curr)).
    RemoveWhile(id, itemToRemove, predNode, curr);
proc RemoveWhile(id: Nat, itemToRemove: Nat, predNode: Node, curr: Node) =
```

```
    (key(curr) < hashCode(itemToRemove))
       -> snd_UnlockNode(mem(predNode)). sum nn: Node. rcv_ReadNext(mem(curr), nn).
          snd_LockNode(mem(nn)). RemoveWhile(id, itemToRemove, curr, nn)
       <> ((key(curr) == hashCode(itemToRemove))
          -> snd_RemoveItem(mem(predNode)). snd_UnlockNode(mem(predNode)).
             ReturnRemovePrime(id, true)
          <> snd_UnlockNode(mem(curr)). snd_UnlockNode(mem(predNode)).
             ReturnRemovePrime(id, false));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
  (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Add(id, head(elms)) + Remove(id, head(elms))).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);

init
    hide({
        GetHashCode, InsertItem, RemoveItem, LockNode, UnlockNode, ReadNode,
            ReadNext
    }, allow({
        CallAdd, ReturnAdd, CallRemove, ReturnRemove, InsertItem, RemoveItem,
        GetHashCode, LockNode, UnlockNode, ReadNode, ReadNext
  }, comm({
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_LockNode | snd_LockNode -> LockNode,
        rcv_UnlockNode | snd_UnlockNode -> UnlockNode,
        rcv_ReadNode | snd_ReadNode -> ReadNode,
        rcv_ReadNext | snd_ReadNext -> ReadNext
  }, rename({
        CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
  }, LinkedList((lambda n: Nat.null)[0->node(MinKey, MinKey, 1, false, 0)]
        [1->node(MaxKey, MaxKey, -1, false, 1)], 2) ||
        Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing B.2: Concrete specification of the fine-grained set

Optimistic set

```
sort Node = struct null | node(item: Int, key: Int, next: Int, lock: Bool, mem: Nat
    );
sort ArrayNodes = Int -> Node;
map MinKey: Int; MaxKey: Pos; hashCode: Int -> Nat;
    lockNode: Node -> Node; unlockNode: Node -> Node;
    insertNode: ArrayNodes # Nat # Nat # Nat -> ArrayNodes;
    removeNode: ArrayNodes # Nat -> ArrayNodes;
var i, k, n: Int; m: Nat; l: Bool;
    an: ArrayNodes; loc: Nat; nextL: Nat; it: Nat;
eqn MinKey = -1; MaxKey = 100; hashCode(i) = i mod MaxKey;
    lockNode(node(i, k, n, l, m)) = node(i, k, n, true, m);
    unlockNode(node(i, k, n, l, m)) = node(i, k, n, false, m);
    insertNode(an, nextL, it, loc) = an[nextL -> node(it, hashCode(it),
       next(an(loc)), false, nextL)][loc -> node(item(an(loc)), key(an(loc)),
       nextL, lock(an(loc)), loc)];
    removeNode(an, loc) = an[loc -> node(item(an(loc)), key(an(loc)),
       next(an(next(an(loc)))), lock(an(loc)), mem(an(loc)))];

act snd_ReadNode: Int#Node; rcv_ReadNode: Int#Node; ReadNode: Int#Node;
    snd_ReadNext: Int#Node; rcv_ReadNext: Int#Node; ReadNext: Int#Node;
    snd_LockNode: Int; rcv_LockNode: Int; LockNode: Int;
    snd_UnlockNode: Int; rcv_UnlockNode: Int; UnlockNode: Int;
    CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallRemove: Nat#Nat; ReturnRemove: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    CallRemovePrime: Nat#Nat; ReturnRemovePrime: Nat#Bool;
    rcv_InsertItem: Nat#Nat; snd_InsertItem: Nat#Nat; InsertItem: Nat#Nat;
    rcv_RemoveItem: Nat; snd_RemoveItem: Nat; RemoveItem: Nat; GetHashCode: Nat;
    snd_ValidateNodes: Node#Node; rcv_ValidateNodes: Node#Node;
    ValidateNodes: Node#Node;
    snd_Validation: Bool; rcv_Validation: Bool; Validation: Bool; NextIteration;

proc LinkedList(nodes: ArrayNodes, nextLoc: Nat) =
    sum element: Int. snd_ReadNode(element, nodes(element)).
        LinkedList(nodes, nextLoc) +
    sum elementN: Int. snd_ReadNext(elementN, nodes(next(nodes(elementN)))).
        LinkedList(nodes, nextLoc) +
    sum n: Int. (!lock(nodes(n))) -> rcv_LockNode(n).
        LinkedList(nodes[n->lockNode(nodes(n))], nextLoc) +
    sum nn: Int. rcv_UnlockNode(nn).
        LinkedList(nodes[nn->unlockNode(nodes(nn))], nextLoc) +
    sum item1, loc: Nat. rcv_InsertItem(item1, loc).
        LinkedList(insertNode(nodes, nextLoc, item1, loc), (nextLoc+1)) +
    sum loc: Nat. rcv_RemoveItem(loc). LinkedList(removeNode(nodes, loc), nextLoc)
       +
    sum predNode, curr: Node. rcv_ValidateNodes(predNode, curr).
        Validate(nodes, predNode, curr, 0). LinkedList(nodes, nextLoc);
proc Validate(nodes: ArrayNodes, predNode: Node, curr: Node, nd: Int) =
    (nodes(nd)!= null && key(nodes(nd)) <= key(predNode))
       -> ((nodes(nd) == predNode)
           -> ((nodes(next(nodes(nd))) == curr)
              -> snd_Validation(true) <> snd_Validation(false))
           <> NextIteration. Validate(nodes, predNode, curr, next(nodes(nd))))
       <> snd_Validation(false);

proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). AddWhile(id, itemToAdd);
proc AddWhile(id: Nat, itemToAdd: Nat) = sum predNode: Node.
    rcv_ReadNode(0, predNode). sum curr: Node.
    rcv_ReadNext(mem(predNode), curr). AddWhile2(id, itemToAdd, predNode, curr);
proc AddWhile2(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
        -> sum nextCurr: Node.  rcv_ReadNext(mem(curr), nextCurr).
           AddWhile2(id, itemToAdd, curr, nextCurr)
        <> snd_LockNode(mem(predNode)).  snd_LockNode(mem(curr)).
```

```
            snd_ValidateNodes(lockNode(predNode), lockNode(curr)).
            (rcv_Validation(true). ((key(curr) == hashCode(itemToAdd))
            -> snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
               ReturnAddPrime(id, false)
            <> snd_InsertItem(itemToAdd, mem(predNode)).
               snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
               ReturnAddPrime(id, true)) +
            rcv_Validation(false). snd_UnlockNode(mem(predNode)).
            snd_UnlockNode(mem(curr)). AddWhile(id, itemToAdd));

proc Remove(id: Nat, itemToRemove: Nat) = CallRemovePrime(id, itemToRemove).
    GetHashCode(hashCode(itemToRemove)). RemoveWhile(id, itemToRemove);
proc RemoveWhile(id: Nat, itemToRemove: Nat) = sum predNode: Node.
    rcv_ReadNode(0, predNode). sum curr: Node.
    rcv_ReadNext(mem(predNode), curr). RemoveWhile2(id, itemToRemove, predNode,
        curr);
proc RemoveWhile2(id: Nat, itemToRemove: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToRemove))
        -> sum nextCurr: Node. rcv_ReadNext(mem(curr), nextCurr).
           RemoveWhile2(id, itemToRemove, curr, nextCurr)
        <> snd_LockNode(mem(predNode)). snd_LockNode(mem(curr)).
           snd_ValidateNodes(lockNode(predNode), lockNode(curr)).
           (rcv_Validation(true). ((key(curr) == hashCode(itemToRemove))
           -> snd_RemoveItem(mem(predNode)). snd_UnlockNode(mem(predNode)).
              snd_UnlockNode(mem(curr)). ReturnRemovePrime(id, true)
           <> snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
              ReturnRemovePrime(id, false)) +
           rcv_Validation(false). snd_UnlockNode(mem(predNode)).
           snd_UnlockNode(mem(curr)). RemoveWhile(id, itemToRemove));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
  (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Add(id, head(elms)) + Remove(id, head(elms))).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);

init
    hide({
        GetHashCode, InsertItem, RemoveItem, LockNode, UnlockNode, ReadNode,
        ReadNext, ValidateNodes, NextIteration, Validation
    }, allow({
        CallAdd, ReturnAdd, CallRemove, ReturnRemove, InsertItem, RemoveItem,
        GetHashCode, LockNode, UnlockNode, ReadNode, ReadNext, ValidateNodes,
        NextIteration, Validation
    }, comm({
        rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_LockNode | snd_LockNode -> LockNode,
        rcv_UnlockNode | snd_UnlockNode -> UnlockNode,
        rcv_ReadNode | snd_ReadNode -> ReadNode,
        rcv_ReadNext | snd_ReadNext -> ReadNext,
        snd_ValidateNodes | rcv_ValidateNodes -> ValidateNodes,
        snd_Validation | rcv_Validation -> Validation
    }, rename({
        CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
    }, LinkedList((lambda n: Nat.null)[0->node(MinKey, MinKey, 1, false, 0)]
        [1->node(MaxKey, MaxKey, -1, false, 1)], 2) ||
        Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing B.3: Concrete specification of the optimistic set

Lazy set

```
sort Node = struct null | node(item: Int, key: Int, next: Int,
            marked: Bool, lock: Bool, mem: Nat);
sort ArrayNodes = Int -> Node;
map MinKey: Int; MaxKey: Pos; hashCode: Int -> Nat; markNode: Node -> Node;
    lockNode: Node -> Node; unlockNode: Node -> Node;
    insertNode: ArrayNodes # Nat # Nat # Nat -> ArrayNodes;
    removeNode: ArrayNodes # Nat -> ArrayNodes;
var i, k, n: Int; mk, l: Bool; m: Nat;
    an: ArrayNodes; loc: Nat; nextL: Nat; it: Nat;
eqn MinKey = -1; MaxKey = 100; hashCode(i) = i mod MaxKey;
    markNode(node(i, k, n, mk, l, m)) = node(i, k, n, true, l, m);
    lockNode(node(i, k, n, mk, l, m)) = node(i, k, n, mk, true, m);
    unlockNode(node(i, k, n, mk, l, m)) = node(i, k, n, mk, false, m);
    insertNode(an, nextL, it, loc) = an[nextL -> node(it, hashCode(it), next(an(loc
        )),
       marked(an(loc)), false, nextL)][loc -> node(item(an(loc)), key(an(loc)),
       nextL, false, lock(an(loc)), loc)];
    removeNode(an, loc) = an[loc -> node(item(an(loc)), key(an(loc)),
       next(an(next(an(loc)))), marked(an(loc)), lock(an(loc)), mem(an(loc)))];

act snd_ReadNode: Int#Node; rcv_ReadNode: Int#Node; ReadNode: Int#Node;
    snd_ReadNext: Int#Node; rcv_ReadNext: Int#Node; ReadNext: Int#Node;
    snd_LockNode: Int; rcv_LockNode: Int; LockNode: Int;
    snd_UnlockNode: Int; rcv_UnlockNode: Int; UnlockNode: Int;
    CallAdd: Nat#Nat; ReturnAdd: Nat#Bool;
    CallRemove: Nat#Nat; ReturnRemove: Nat#Bool;
    CallAddPrime: Nat#Nat; ReturnAddPrime: Nat#Bool;
    CallRemovePrime: Nat#Nat; ReturnRemovePrime: Nat#Bool;
    snd_InsertItem: Nat#Nat; rcv_InsertItem: Nat#Nat; InsertItem: Nat#Nat;
    snd_RemoveItem: Nat; rcv_RemoveItem: Nat; RemoveItem: Nat;
    snd_MarkRemoved: Int; rcv_MarkRemoved: Int; MarkRemoved: Int; GetHashCode: Nat;
    snd_ValidateNodes: Nat#Nat; rcv_ValidateNodes: Nat#Nat; ValidateNodes: Nat#Nat;
    snd_Validation: Bool; rcv_Validation: Bool; Validation: Bool;

proc LinkedList(nodes: ArrayNodes, nextLocation: Nat) =
    sum element: Int. snd_ReadNode(element, nodes(element)).
        LinkedList(nodes, nextLocation) +
    sum elementN: Int. snd_ReadNext(elementN, nodes(next(nodes(elementN)))).
        LinkedList(nodes, nextLocation) +
    sum n: Int. (!lock(nodes(n))) -> rcv_LockNode(n).
        LinkedList(nodes[n->lockNode(nodes(n))], nextLocation) +
    sum nn: Int. rcv_UnlockNode(nn).
        LinkedList(nodes[nn->unlockNode(nodes(nn))], nextLocation) +
    sum item1, loc: Nat. rcv_InsertItem(item1, loc).
        LinkedList(insertNode(nodes, nextLocation, item1, loc), (nextLocation+1)) +
    sum loc: Nat. rcv_RemoveItem(loc). LinkedList(removeNode(nodes, loc),
        nextLocation) +
    sum nnn: Int. rcv_MarkRemoved(nnn).
        LinkedList(nodes[nnn->markNode(nodes(nnn))], nextLocation) +
    sum predNode, curr: Nat. rcv_ValidateNodes(predNode, curr).
        Validate(nodes, predNode, curr). LinkedList(nodes, nextLocation);
proc Validate(nodes: ArrayNodes, predNode: Nat, curr: Nat) =
    (!marked(nodes(predNode)) && !marked(nodes(curr)) && next(nodes(predNode)) ==
        curr)
        -> snd_Validation(true) <> snd_Validation(false);

proc Add(id: Nat, itemToAdd: Nat) = CallAddPrime(id, itemToAdd).
    GetHashCode(hashCode(itemToAdd)). AddWhile(id, itemToAdd);
proc AddWhile(id: Nat, itemToAdd: Nat) = sum predNode: Node.
    rcv_ReadNode(0, predNode). sum curr: Node.
    rcv_ReadNext(mem(predNode), curr). AddWhile2(id, itemToAdd, predNode, curr);
proc AddWhile2(id: Nat, itemToAdd: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToAdd))
       -> sum nextCurr: Node.  rcv_ReadNext(mem(curr), nextCurr).
          AddWhile2(id, itemToAdd, curr, nextCurr)
```

```
        <> snd_LockNode(mem(predNode)). snd_LockNode(mem(curr)).
           snd_ValidateNodes(mem(predNode), mem(curr)).
           (rcv_Validation(true). ((key(curr) == hashCode(itemToAdd))
           -> snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
              ReturnAddPrime(id, false)
           <> snd_InsertItem(itemToAdd, mem(predNode)).
              snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
              ReturnAddPrime(id, true)) +
           rcv_Validation(false). snd_UnlockNode(mem(predNode)).
           snd_UnlockNode(mem(curr)). AddWhile(id, itemToAdd));

proc Remove(id: Nat, itemToRemove: Nat) = CallRemovePrime(id, itemToRemove).
    GetHashCode(hashCode(itemToRemove)). RemoveWhile(id, itemToRemove);
proc RemoveWhile(id: Nat, itemToRemove: Nat) = sum predNode: Node.
    rcv_ReadNode(0, predNode). sum curr: Node.
    rcv_ReadNext(mem(predNode), curr). RemoveWhile2(id, itemToRemove, predNode,
        curr);
proc RemoveWhile2(id: Nat, itemToRemove: Nat, predNode: Node, curr: Node) =
    (key(curr) < hashCode(itemToRemove))
       -> sum nextCurr: Node. rcv_ReadNext(mem(curr), nextCurr).
          RemoveWhile2(id, itemToRemove, curr, nextCurr)
       <> snd_LockNode(mem(predNode)). snd_LockNode(mem(curr)).
          snd_ValidateNodes(mem(predNode), mem(curr)).
          (rcv_Validation(true). ((key(curr) == hashCode(itemToRemove))
          -> snd_MarkRemoved(mem(curr)). snd_RemoveItem(mem(predNode)).
             snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
             ReturnRemovePrime(id, true)
          <> snd_UnlockNode(mem(predNode)). snd_UnlockNode(mem(curr)).
             ReturnRemovePrime(id, false)) +
          rcv_Validation(false). snd_UnlockNode(mem(predNode)).
          snd_UnlockNode(mem(curr)). RemoveWhile(id, itemToRemove));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
  (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Add(id, head(elms)) + Remove(id, head(elms))).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);

init
    hide({ GetHashCode, InsertItem, RemoveItem, LockNode, UnlockNode,
        ReadNode, ReadNext, ValidateNodes, Validation, MarkRemoved
    }, allow({ CallAdd, ReturnAdd, CallRemove, ReturnRemove,
        InsertItem, RemoveItem, GetHashCode, LockNode, UnlockNode,
        ReadNode, ReadNext, ValidateNodes, Validation, MarkRemoved
    }, comm({ rcv_InsertItem | snd_InsertItem -> InsertItem,
        rcv_RemoveItem | snd_RemoveItem -> RemoveItem,
        rcv_MarkRemoved | snd_MarkRemoved -> MarkRemoved,
        rcv_LockNode | snd_LockNode -> LockNode,
        rcv_UnlockNode | snd_UnlockNode -> UnlockNode,
        rcv_ReadNode | snd_ReadNode -> ReadNode,
        rcv_ReadNext | snd_ReadNext -> ReadNext,
        snd_ValidateNodes | rcv_ValidateNodes -> ValidateNodes,
        snd_Validation | rcv_Validation -> Validation
    }, rename({ CallAddPrime -> CallAdd, ReturnAddPrime -> ReturnAdd,
        CallRemovePrime -> CallRemove, ReturnRemovePrime -> ReturnRemove
    }, LinkedList((lambda n: Int.null)[0->node(MinKey, MinKey, 1, false, false, 0)]
        [1->node(MaxKey, MaxKey, -1, false, false, 1)], 2) ||
        Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing B.4: Concrete specification of the lazy set

Non-blocking queue

```
sort Null = struct null;
sort Item = struct item(nl: Null, ref: Nat) | item(value: Nat, ref: Nat);
sort Array = Nat -> Item;
map L: Pos; isNull: Item -> Bool;
var a: Array; v, r: Nat;
eqn L = 5; isNull(item(v, r)) = false; isNull(item(null, r)) = true;

act CallEnqueue: Nat#Nat; ReturnEnqueue: Nat;
    CallDequeue: Nat; ReturnDequeue: Nat#Nat;
    CallEnqueuePrime: Nat#Nat; ReturnEnqueuePrime: Nat;
    CallDequeuePrime: Nat; ReturnDequeuePrime: Nat#Nat;
    rcv_ReadRear: Nat; snd_ReadRear: Nat; ReadRear: Nat;
    rcv_ReadFront: Nat; snd_ReadFront: Nat; ReadFront: Nat;
    rcv_ReadElement: Nat#Item; snd_ReadElement: Nat#Item; ReadElement: Nat#Item;
    rcv_Result: Bool; snd_Result: Bool; Result: Bool;
    rcv_CASRear: Nat#Nat; snd_CASRear: Nat#Nat; CASRear: Nat#Nat;
    rcv_CASFront: Nat#Nat; snd_CASFront: Nat#Nat; CASFront: Nat#Nat;
    rcv_CASElement: Nat#Item#Item; snd_CASElement: Nat#Item#Item;
    CASElement: Nat#Item#Item;

proc QueueInterface(q: Array, rear: Nat, front: Nat) =
     rcv_ReadRear(rear). QueueInterface(q, rear, front) +
     rcv_ReadFront(front). QueueInterface(q, rear, front) +
     sum n: Nat. rcv_ReadElement(n, q(n)). QueueInterface(q, rear, front) +
     sum oldR, newR: Nat. (rcv_CASRear(oldR, newR) | snd_Result(oldR == rear)).
        ((oldR == rear) -> QueueInterface(q, newR, front)
       <> QueueInterface(q, rear, front)) +
     sum oldF, newF: Nat. (rcv_CASFront(oldF, newF) | snd_Result(oldF == front)).
         ((oldF == front) -> QueueInterface(q, rear, newF)
        <> QueueInterface(q, rear, front)) +
     sum n: Nat, oldX, newX: Item. (rcv_CASElement(n, oldX, newX) | snd_Result(q(n)
         == oldX)).
         ((q(n) == oldX) -> QueueInterface(q[n->newX], rear, front)
        <> QueueInterface(q, rear, front)) ;

proc Enqueue(tid: Nat, v: Nat) = CallEnqueuePrime(tid, v). EnqueueTryAgain(tid, v);

proc EnqueueTryAgain(tid: Nat, v: Nat) = sum r1: Nat. snd_ReadRear(r1).
     sum x: Item. snd_ReadElement((r1 mod L), x).
     sum r2: Nat. snd_ReadRear(r2).
     ((r1 != r2) -> EnqueueTryAgain(tid, v)
    <> sum f1: Nat. snd_ReadFront(f1).
     ((r1 == f1 + L) -> EnqueueTryAgain(tid, v)
        <> ((isNull(x)) ->
          ((snd_CASElement((r1 mod L), x, item(v, ref(x)+1)) | rcv_Result(true)).
          (sum b1: Bool. (snd_CASRear(r1, (r1+1)) | rcv_Result(b1))).
          ReturnEnqueuePrime(tid) +
          (snd_CASElement((r1 mod L), x, item(v, ref(x)+1)) | rcv_Result(false)).
          EnqueueTryAgain(tid, v))
         <> sum b2: Bool. (snd_CASRear(r1, (r1+1)) | rcv_Result(b2)).
          EnqueueTryAgain(tid, v))));

proc Dequeue(tid: Nat) = CallDequeuePrime(tid). DequeueTryAgain(tid);

proc DequeueTryAgain(tid: Nat) = sum f1: Nat. snd_ReadFront(f1).
     sum x: Item. snd_ReadElement((f1 mod L), x).
     sum f2: Nat. snd_ReadFront(f2).
     ((f1 != f2) -> DequeueTryAgain(tid)
    <> sum r1: Nat. snd_ReadRear(r1).
     ((f1 == r1) -> DequeueTryAgain(tid)
        <> ((!isNull(x)) ->
          ((snd_CASElement((f1 mod L), x, item(null, ref(x) +1)) | rcv_Result(true
              )).
          (sum b1: Bool. (snd_CASFront(f1, (f1+1)) | rcv_Result(b1))).
          ReturnDequeuePrime(tid, value(x)) +
```

```
            (snd_CASElement((f1 mod L), x, item(null, ref(x)+1)) | rcv_Result(false)
                ).
            DequeueTryAgain(tid))
            <> sum b2: Bool. (snd_CASFront(f1, (f1+1)) | rcv_Result(b2)).
            DequeueTryAgain(tid))));

proc Thread(id: Nat, nOp: Nat, elms: List(Nat)) =
     (nOp > 0) -> ThreadProgress(id, nOp, elms, 1);
proc ThreadProgress(id: Nat, nOp: Nat, elms: List(Nat), nDone: Nat) =
     (nDone <= nOp) -> (Enqueue(id, head(elms)) + Dequeue(id)).
     ThreadProgress(id, nOp, tail(elms), nDone + 1);

init
    hide({ ReadRear, ReadFront, ReadElement, CASElement, CASFront, CASRear, Result
    }, allow({ CallEnqueue, ReturnEnqueue, CallDequeue, ReturnDequeue,
         ReadRear, ReadFront, ReadElement,
         CASRear | Result, CASFront | Result, CASElement | Result
    }, comm({ rcv_ReadRear | snd_ReadRear -> ReadRear,
         rcv_ReadFront | snd_ReadFront -> ReadFront,
         rcv_ReadElement | snd_ReadElement -> ReadElement,
         rcv_Result | snd_Result -> Result,
         rcv_CASRear | snd_CASRear -> CASRear,
         rcv_CASFront | snd_CASFront -> CASFront,
         rcv_CASElement | snd_CASElement -> CASElement
    }, rename({ CallEnqueuePrime -> CallEnqueue,  CallDequeuePrime -> CallDequeue,
         ReturnEnqueuePrime -> ReturnEnqueue, ReturnDequeuePrime -> ReturnDequeue
  }, QueueInterface((lambda n:Nat.item(null, 0)), 0, 0) ||
       Thread(1, 2, [2, 4]) || Thread(2, 2, [4, 8])))));
```

Listing B.5: Concrete specification of the non-blocking queue