Challenges of specifying concurrent program components*

Ian J. Hayes
School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

The purpose of this paper is to review some of the challenges of formally specifying components of shared-memory concurrent programs. The focus is to provide an abstract specification of a component that is suitable for use both by clients of the component and as a starting point for refinement to an implementation of the component. We present some approaches to devising specifications, investigating different forms of specification suitable for different contexts. We examine handling atomicity of access to data structures, blocking operations and progress properties, and transactional operations that may fail and need to be retried.

1 Introduction

The objective of this paper is to present challenges to do with specifying concurrent program components in order to promote discussion about possible different alternatives. Our main foci are atomicity, blocking operations and transactional operations in the context of rely/guarantee specifications. Our aim is to present the ideas rather than a fully formal development. Specifications play an important role in decoupling the use of a component from its detailed implementation. Often the role of specifications as a starting points for refinement to an implementation is emphasised but here we would like to balance that with their role of being used by other components. Hence we try to focus on a top-down approach to concurrent program specification, rather than a bottom-up approach.

Sequential programs. For sequential programs conventional Floyd/Hoare-style specifications [3, 9] in terms of *preconditions* and *postconditions* form the basis of component specifications, however, just pre and post conditions are inadequate for specifying concurrent operations because they do not handle interference between the operations.

Shared variable concurrency. First, to state the obvious, variables that are local to a thread are not subject to interference and hence can be treated in a manner similar to a sequential program. For variables shared between parallel threads, interference becomes an issue. An important consideration is whether access (e.g. read or write) of variables is *atomic* or not. At the lowest level, atomicity is determined by the machine hardware and properties like its atomic access "word" size. A further complication at the hardware level is that, due to caches and write buffers, the order of write/read accesses to shared memory may not respect the sequential order of instruction execution. But perhaps we get a bit ahead of ourselves if we worry about these issues when considering specifications.

A program component that needs to perform multiple atomic accesses can be subject to *data races* where variables are updated in parallel by concurrent threads because the component may see inconsistent data. At a more abstract level operations may be required to be atomic with respect to a shared data

*This work was supported by Australian Research Council (ARC) Discovery Project DP130102901.

Submitted to:
Refinement Workshop 2018

structure. The implementation of such operations may require locks to ensure sequentialisation of access to the data structure or it may use more sophisticated non-blocking algorithms that achieve the effect of operation atomicity by utilising hardware-level atomicity [22].

A concept commonly used to show an implementation is valid is *linearisability* [8], whereby parallel execution of a set of operations on a shared data structure is considered valid if it is equivalent to some linear (sequential) execution of the same operations (subject to certain requirements).

Verifying concurrent programs. Early rules for reasoning about parallel programs by Hoare [11] utilised preconditions and postconditions but had strict disjointness requirements on program variables occurring in parallel threads, which effectively ruled out interference between parallel threads. The approach of Owicki and Gries [19, 20, 21] treats parallel components like sequential programs with intermediate assertions between each atomic step but then requires an extensive interference-freedom proof. Concurrent separation logic [1, 18] also leverages disjointness but does so in a more fine-grained and dynamic manner.

An early compositional approach to handle interleaved interference on shared variables was the rely/guarantee approach of Jones [12, 13, 14]. This extended pre/post specifications with a *rely* condition, a binary relation between program states expressing an assumption about the allowable interference that any step of the environment of the component can impose on the shared variables. To constrain the interference generated by a thread, Jones uses a *guarantee* condition, also a binary relation on states limiting the changes the components can make to the shared variables. The guarantee is required to be reflexive (i.e. it contains the identity relation) so that the program may make stuttering steps that do not change the observable variables. The guarantee of each parallel thread must imply the relies of all threads that run in parallel with it. The rely/guarantee approach does not dictate any particular granularity of atomicity, however, it does require all steps of a thread to satisfy its guarantee as long as all steps of the thread's environment satisfy its rely.

Operations that may block. At the specification level, operations may block waiting for "communication" from another thread. For example, an operation wanting to read a message from a communication channel may need to wait for a message to be written to the channel by another thread. Specifications of such operations need to be able to express such *waiting* criteria. This affects the termination behaviour of the operation. For example, if a message is never written to a channel, a read from the channel will never terminate (block forever). There are two approaches to specifying such operations.

- Using an explicit **await** construct that allows both nonterminating behaviour if the **await** blocks forever and terminating behaviour if it becomes unblocked.
- An implicit approach that specifies under what conditions an operation is guaranteed to terminate as well as its behaviour when it does terminate, e.g. a read on a message channel is guaranteed to terminate if the channel is non-empty.

These forms can give equivalent specifications, where the explicit form may be more useful for refining the operation, while the implicit form makes it easier to reason about using the operation. Another source of blocking is at the implementation level where atomicity constraints on operations may lead to the use of locks that lead to an implementation blocking awaiting a lock.

Transactional operations. Utilising locks can generate bottlenecks because operations requiring the locks are sequentialised and on multi-processor architectures they also generate more costly memory

synchronisation primitives. One approach to avoiding (or minimising) locks is to implement operations that do most of their work locally and then perform a final atomic commit step that may fail if another operation has committed while the first operation was executing based on the old data [22]. Such implementations suffer issues similar to livelock where they can repeatedly try and fail, potentially forever if there is continual interference from competing parallel operations. Note that in these situations, one of the competing threads may succeed but an individual thread may be pre-empted every time and never succeed. Fair scheduling is assumed, i.e. every thread is executed eventually, but that does not mean an operation a thread is executing will succeed.

Specifying such operations has to allow for the case in which an operation is continually thwarted and may never terminate, while guaranteeing termination if the interference on the data structure eventually quiesces. Using c^ω to represent the execution of a command c zero or more times, including possibly infinitely many times, such specifications have the general form

$$\mathit{fail}^\omega ; \mathit{succeed} \tag{1}$$

where *fail* represents the operation failing due to interference (and not changing the state) and *succeed* represents the operation successfully completing (and updating the state once). The iteration fail^ω may execute *fail* infinitely many times representing the continual thwarting by interference from parallel operations. Arguments about termination usually need to resort to either

- timing arguments based on minimal separation between operations in any single thread leading to a situation in which interference will eventually quiesce for long enough for the operation to succeed, and
- arguments based on probabilities of two (or more) operations overlapping and competing.

Note that probability bounds can be derived from timing bounds. The probability can be sensitive to load, i.e. the more threads competing, the lower the probability of success of any single operation. And probabilities can be sensitive to the execution time of an operation: the longer it executes, the more likely it is to overlap with a competing operation. Of course, such timing and probability arguments depend on the context of the use of the data structure and can be tricky in practice.

Implicit specifications can also be used for such operations by specifying the conditions under which they are guaranteed to terminate.

Overview. Sect. 2 addresses specifying atomic operations on a shared data structure (or resource). It examines the use of Hoare's **with** statement [10] and how it interacts with rely and guarantee conditions. Sect. 3 examines blocking operations giving both explicit waiting conditions and more implicit specifications using a temporal logic formula under which an operation terminates. Sect. 4 looks at transactional operations that may either succeed, or try and fail, possibly indefinitely. Both explicit waiting and implicit temporal logic specifications are considered.

2 Specifying atomicity

As an example, consider a message queue with operations to enqueue and dequeue messages. If there are separate concurrent threads enqueuing and dequeuing, each operation needs to (appear to be) be atomic, i.e. other operations cannot observe the state part way through the operation. If the queue were used

in a purely sequential program, the *enqueue* and *dequeue* operations can be specified by Morgan-style specification commands [16, 17] as follows.

$$\begin{array}{rcl} enqueue(v : Val) & \widehat{=} & qu{:}\big[qu' = qu \frown [v]\big] \\ dequeue()res : Val & \widehat{=} & \{qu \neq [\,]\}\,;res, qu{:}\big[qu = [res'] \frown qu'\big] \end{array}$$

The *enqueue* operation takes a value v to append to the queue. Its postcondition is $qu' = qu \frown [v]$, where qu' stands for the final value of the queue, qu stands for the initial value, "$\frown$" is sequence concatenation, and $[v]$ is the singleton sequence containing v. It modifies only the queue and hence it has a frame of qu (before the colon). The *dequeue* operation returns a value *res* that is the head of the queue and removes it from the queue in the process; it has a precondition that the queue is non-empty.

To extend the operation specification to handle concurrency, as presented in Fig. 1, the operations need to be augmented with rely and guarantee conditions and the atomicity of the operations needs to be handled. In Fig. 1 the relies and guarantees are represented as rely and guarantee commands [2, 5, 7]. The guarantee command (**guar** g) restricts every atomic step of the thread to satisfy g. The rely command (**rely** r) represents an assumption that every environment step satisfies r; it aborts if the environment performs a step not satisfying r, in a manner similar to the precondition command $\{p\}$ aborting if the initial state does not satisfy p. The rely and guarantee commands are combined with the remainder of the specification using weak conjunction "$\Cap$" [6, 4]. Weak conjunction is a specification operator, rather than a programming operator. The weak conjunction $c \Cap d$ performs steps allowed by both c and d unless either c or d aborts at some point, in which case their weak conjunction aborts from that point.

For the message queue we assume there is a single writer thread performing *enqueue* operations and a single reader thread performing *dequeue* operations. A suitable rely condition for *enqueue* is that elements are only ever removed from the front of the queue by *dequeue*, i.e. the queue after any interference is a suffix of the queue before. The rely condition for *dequeue* is that the interference from concurrent *enqueue*s only ever adds elements to the end of the queue and hence the queue before the interference is a prefix of the queue after. The guarantees of each operation match the rely of the other operation.

For sequential programs and in the original rely/guarantee approach the postconditions of operations are considered end-to-end, i.e. they must hold between the states at the start and end of an operation invocation. Such a postcondition is problematic in the context of a parallel thread modifying the queue, for example, after an *enqueue* operation is initiated but before it can complete (or lock the data structure), the reader thread may *dequeue* a value. If the writer then completes the *enqueue* without further interference, the end-to-end effect is that of both the *dequeue* and the *enqueue*, not just the *enqueue*.

From the above example, it is apparent that an end-to-end postcondition is not suitable in this case. The alternative is a specification whereby the postcondition holds for some "atomic" step between the start and end of the operation and the operation makes no changes to the queue before or after that step, although other threads may. Here one needs to be careful about what is meant by "atomic". For a simple operation it may be possible to implement it by utilising hardware-level atomicity but a more complex operation may need to lock the data structure. In the latter case, the lock is used to prevent concurrent operations on the same data structure but does not preclude concurrent operations on unrelated data structures overlapping their execution.

In the rely/guarantee approach a rely condition can be conditional on whether a thread has the lock or not, so that (part of) the rely condition states that, if the thread has the lock on a data structure d, the environment does not change d.

$$has\_lock(d) \quad \Rightarrow \quad d' = d \tag{2}$$

resource qu : **seq** Val **initially** $qu = [\,]$

$enqueue(v : Val) \mathrel{\widehat{=}}$
  (**rely** qu' **suffixof** qu) $\Cap$ – implies single writer
  (**guar** qu **prefixof** qu') $\Cap$ – implies the rely of dequeue
  with qu **do** $qu{:}[qu' = qu \frown [v]]$ **od**

$dequeue()res : Val \mathrel{\widehat{=}}$
  (**rely** qu **prefixof** qu') $\Cap$ – implies single reader
  (**guar** qu' **suffixof** qu) $\Cap$ – implies the rely of enqueue
  $\{qu \neq [\,]\}$; – stable under the rely condition
  with qu **do** $qu, res{:}[qu = [res'] \frown qu']$ **od**

Figure 1: Message queue with read and write operations

2.1 Resources

The early work of Hoare [10] introduced the idea of a resource and a **with** statement that provides access to the resource. A *resource* represents a shared data structure that is only accessible to a thread within a command of the form,

$$\textbf{with}\ d\ \textbf{do}\ c\ \textbf{od} \tag{3}$$

that ensures the resource d is not modified by the environment while the thread is executing c. A data structure d is declared as a resource by a declaration of the form **resource** d, and within the scope of the resource declaration, all uses of d must be within a **with** d **do** ... **od** statement.[1] It is assumed that the data structure of the resource is only accessed within **with** statements; this may be checked syntactically. The implementation is responsible for ensuring the data structure isn't modified by the environment while executing within the **with** statement. The **with** statement allows stuttering steps before the body and finite stuttering after the body of the **with**, c, is executed. As entry to a **with** statement by one thread may block other threads wishing to gain access to the same resource, it is prudent to require that the bodies of **with** statements terminate; that is required within this paper. The data structure of the resource often has a data-type invariant associated with it that is established by its initialisation and maintained by each operation (see the example in Section 3).

Fig. 1 presents the specification of a message queue with *enqueue* and *dequeue* operations. The *enqueue* appends a value v to the end of the queue atomically (with respect to qu), and the *dequeue* operation removes and returns the first element of the queue atomically. The *dequeue* operation has a precondition that the queue is non-empty; the precondition is stable under the rely condition which assumes the queue is only ever extended. This version assumes that there is just one reader thread and one writer thread because the rely condition of *dequeue* assumes the queue can only be extended and hence it is not concurrently being dequeued by another thread, and the rely condition of *enqueue* assumes the queue can only become a suffix of its previous state and hence it is not concurrently being enqueued by another thread.

[1] A more general resource construct would allow a resource to encompass a set of variables but for the examples here a resource will correspond to a single variable, so we'll identify the resource with the variable.

2.2 Rely/guarantee laws for resource access

The concept of a resource may be combined with the rely/guarantee approach. When a thread enters the body of a **with** d **do** c **od** statement, the rely can be strengthened for the duration of c with $d' = d$ and the guarantee weakened so that d only need satisfy the guarantee over the complete operation, not every step. The initial step of the refinement of operations specified via a **with** statement needs to "move" the rely and guarantee conditions into the body of the **with** but in the process the rely and guarantee conditions are transformed. A rely condition surrounding a **with** statement may be strengthened when it is moved inside the **with** to state that d is not modified.

$$(\textbf{rely}\ r)\Cap \textbf{with}\ d\ \textbf{do}\ c\ \textbf{od} \quad \sqsubseteq \quad \textbf{with}\ d\ \textbf{do}\ (\textbf{rely}\ r \wedge d' = d)\Cap c\ \textbf{od} \tag{4}$$

For the *enqueue* operation above (ignoring the guarantee for the moment) this law can be applied as follows.

$$\begin{array}{ll} & (\textbf{rely}\ qu' \ \textbf{suffixof}\ qu)\Cap \textbf{with}\ qu\ \textbf{do}\ qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \\ \sqsubseteq & \text{by (4)} \\ & \textbf{with}\ qu\ \textbf{do}\ (\textbf{rely}\ qu' \ \textbf{suffixof}\ qu \wedge qu' = qu)\Cap qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \\ = & \text{as } qu' = qu \ \Rrightarrow\ qu' \ \textbf{suffixof}\ qu \\ & \textbf{with}\ qu\ \textbf{do}\ (\textbf{rely}\ qu' = qu)\Cap qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \end{array}$$

A guarantee condition surrounding a **with** may be weakened so that it only has to apply for the resource data structure over the body of the **with** command. It is assumed that the guarantee is of the form $g_d \wedge g_x$, where d is the only shared variable g_d refers to, and g_x does not refer to d. The weakened guarantee g_x is retained to handle references to variables other than d within the guarantee. The specification $\left[g_d\right]$ requires g_d to hold end-to-end over the body of the **with**. The lack of a frame allows any variables to be modified but when it is combined with c, any frame of c will apply to their weak conjunction.

$$(\textbf{guar}\ \ g_d \wedge g_x)\Cap \textbf{with}\ d\ \textbf{do}\ c\ \textbf{od} \quad \sqsubseteq \quad \textbf{with}\ d\ \textbf{do}\ (\textbf{guar}\ \ g_x)\Cap \left[g_d\right]\Cap c\ \textbf{od} \tag{5}$$

For the *enqueue* operation above (ignoring the rely for the moment) this law can be applied as follows.

$$\begin{array}{ll} & (\textbf{guar}\ \ qu \ \textbf{prefixof}\ qu')\Cap \textbf{with}\ qu\ \textbf{do}\ qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \\ \sqsubseteq & \text{by (5) with } g_d \mathrel{\widehat{=}} qu \ \textbf{prefixof}\ qu' \text{ and } g_x \mathrel{\widehat{=}} \mathsf{true} \\ & \textbf{with}\ qu\ \textbf{do}\ (\textbf{guar}\ \mathsf{true})\Cap \left[qu \ \textbf{prefixof}\ qu'\right]\Cap qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \\ = & \text{as } (\textbf{guar}\ \mathsf{true}) \text{ requires no guarantee and } \left[q_1\right]\Cap x{:}\left[q_2\right] = x{:}\left[q_1 \wedge q_2\right] \\ & \textbf{with}\ qu\ \textbf{do}\ qu{:}\left[qu \ \textbf{prefixof}\ qu' \wedge qu' = qu \frown [v]\right]\ \textbf{od} \\ = & \text{as } qu' = qu \frown [v] \ \Rrightarrow\ qu \ \textbf{prefixof}\ qu' \\ & \textbf{with}\ qu\ \textbf{do}\ qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \end{array}$$

Combining the above applications of (4) and (5), the following refinement of the *enqueue* operation holds.

$$\begin{array}{ll} & (\textbf{rely}\ qu' \ \textbf{suffixof}\ qu)\Cap (\textbf{guar}\ \ qu \ \textbf{prefixof}\ qu')\Cap \textbf{with}\ qu\ \textbf{do}\ qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \\ \sqsubseteq & \text{using (5) and (4)} \\ & \textbf{with}\ qu\ \textbf{do}\ (\textbf{rely}\ qu' = qu)\Cap qu{:}\left[qu' = qu \frown [v]\right]\ \textbf{od} \end{array}$$

Note that in the body of the **with** there is no explicit guarantee and the rely condition assumes that qu is never modified by the environment and hence the refinement of the body of the **with** is effectively a sequential refinement (as one would expect).

resource $qu : \mathbf{seq}\, Val$ **initially** $qu = [\,]$ **invariant** $\#qu \leq N$

$write(v : Val) \mathrel{\widehat{=}}$
  (**rely** qu' **suffixof** qu) $\Cap$ – implies single writer
  (**guar** qu **prefixof** qu') $\Cap$ – implies the rely of read
  with qu **await** $\#qu < N$ **do** – stable under the rely condition
    $\{\#qu < N\}\,; qu{:}\big[qu' = qu \frown [v]\big]$ **od**

$read()res : Val \mathrel{\widehat{=}}$
  (**rely** qu **prefixof** qu') $\Cap$ – implies single reader
  (**guar** qu' **suffixof** qu) $\Cap$ – implies the rely of write
  with qu **await** $qu \neq [\,]$ **do** – stable under the rely condition
    $\{qu \neq [\,]\}\,; qu, res{:}\big[qu = [res'] \frown qu'\big]$ **od**

Figure 2: Message queue with blocking read and write operations

3 Specifying operations that may block

3.1 Blocking using an explicit await condition

Consider the example in Fig. 2 of a message queue with a bounded capacity of N messages. It has a *write* operation that waits until the queue is not full and appends a value to the tail of the queue, and a *read* operation that waits until there is a message in the queue and returns the head of the queue, removing it in the process. The queue has a data-type invariant that its size is bounded by N. The initialisation establishes the invariant (assuming N is a positive integer) and each operation on the queue can assume the invariant when it starts and must be re-establish the invariant when it terminates.

A common approach to specifying potentially blocking operations is to use an **await** statement. The form used here also includes a **with** component. The statement **with** d **await** b **do** c **od** waits until condition b holds and then executes c. The resource d is attained each time b is evaluated, and retained for the execution of c if b is true. For both the *write* and *read* operations the await conditions are stable under the rely condition, i.e. if the await condition b holds before a step that satisfies the rely condition r, then b holds in the after state.

3.2 Blocking using temporal logic termination conditions

The specifications of *read* and *write* in Fig. 2, by including **await** statements, allow non-terminating behaviour in the cases where the await condition never becomes true, e.g. a *read* will wait forever if the queue remains empty because no writes are performed. However, *read* terminates if the queue is eventually non-empty and *write* terminates if the queue is eventually non-full. That leads to an alternative specification using temporal logic termination conditions in Fig. 3. To accommodate the termination conditions, a command of the form,

$$\mathbf{terminate}\, t \cdot c \tag{6}$$

is introduced, in which t is a temporal logic formula and c is a command. If the temporal logic formula t holds, the operation must terminate and specification c must be satisfied, but if t does not hold c must be

resource qu : **seq** Val **initially** $qu = [\,]$ **invariant** $\#qu \leq N$

$write(v : Val) \mathrel{\widehat{=}}$
  terminate $\Diamond(\#qu < N) \cdot$
    (**rely** qu' **suffixof** qu) $\Cap$ – implies single writer
    (**guar** qu **prefixof** qu') $\Cap$ – implies the rely of read
    with qu **do** qu:$\big[qu' = qu \frown [v]\big]$ **od**

$read()res : Val \mathrel{\widehat{=}}$
  terminate $\Diamond(qu \neq [\,]) \cdot$
    (**rely** qu **prefixof** qu') $\Cap$ – implies single reader
    (**guar** qu' **suffixof** qu) $\Cap$ – implies the rely of write
    with qu **do** qu, res:$\big[qu = [res'] \frown qu'\big]$ **od**

Figure 3: Message queue with conditions to ensure termination

satisfied and termination is not required, but is allowed. Neither of the specifications of *write* and *read* in Fig. 3, contain the explicit **await**s used in Fig. 2. The postcondition of the *read* is unsatisfiable if the queue always remains empty because

$$[\,] = [res'] \frown qu' \quad \equiv \quad \mathsf{false}\,.$$

However, if the condition $\Diamond(qu \neq [\,])$ holds, i.e. the queue is eventually non-empty, the postcondition eventually becomes feasible from the state in which the queue is non-empty. Note that $qu \neq [\,]$ is stable under the rely condition and so it will not be falsified by the environment.

For the *read* operation, the negation of $\Diamond(qu \neq [\,])$ is $\Box(qu = [\,])$, i.e. the queue is always empty. If $\Box(qu = [\,])$ the *read* operation is not required to terminate. In addition, if in every state $qu = [\,]$, the postcondition of *read* is unsatisfiable because $[\,] = [res'] \frown [\,]$ is false, and hence the terminating behaviour of *read* is infeasible, and therefore the only possible behaviour if *read* is to not terminate.

More subtly, any finite prefix of a trace of a *read* operation for which $qu = [\,]$ in every state cannot have satisfied the postcondition of *read* and hence cannot have terminated. However, it is still possible that another thread may execute a *write* at some later time establishing $qu \neq [\,]$ and allowing the *read* to terminate. For the finite prefix of the trace, the behaviour of the *read* must correspond to the stuttering allowed by the **with** statement before it enters its body. Hence for every finite prefix of a trace of *read* in which $qu = [\,]$ in every state, the operation performs only finite stuttering steps and, further, nontermination is allowed if $\Box(qu = [\,])$. Hence the behaviours allowed by the specifications in Fig. 3 are actually equivalent to those of the specifications in Fig. 2.

4 Transactional operations

Some implementations of operations are optimistic in that they complete most of the operation locally to the thread and then have a final commit phase that may fail if another operation has committed. Such operations consist of a repeated failure behaviour (that does not change the shared data structure) in the presence of interference followed by a successful commit phase. Of course, in the presence of repeated interference the successful commit may never occur.

resource s : **seq** Val **initially** $s = [\,]$

$push(v : Val) \mathrel{\widehat{=}} push\_fail^{\omega} ; push\_success(v)$
where $push\_fail \mathrel{\widehat{=}} \varepsilon\langle s' \neq s\rangle$
$push\_success(v : Val) \mathrel{\widehat{=}}$ **with** s **do** $s{:}\big[s' = [v] \frown s\big]$ **od**

$pop()res : Val \mathrel{\widehat{=}} pop\_fail^{\omega} ; res := pop\_success()$
where $pop\_fail \mathrel{\widehat{=}} \varepsilon\langle s' \neq s\rangle$
$pop\_success()res : Val \mathrel{\widehat{=}}$ **with** s **do** $s, res{:}\left[\begin{array}{l}(s \neq [\,] \Rightarrow s = [res'] \frown s') \wedge \\ (s = [\,] \Rightarrow res' = \mathsf{null})\end{array}\right]$ **od**

Figure 4: Stack with possibly failing push and pop operations

4.1 Specification using explicit failure

Treiber [23] provided a non-blocking lock-free implementation of a stack in which the push and pop operations may try and fail due to interference from a parallel push or pop operation, and hence may need to be retried until they succeed. Fig. 4 gives a specification of a stack with *push* and *pop* operations that may fail and need to retry, possibly indefinitely. The *push_fail* operation may be executed any number of times but each time it is executed the environment makes a step that changes the stack s. If from some point of time the environment never changes s, then the *push_fail* becomes infeasible and the operation must perform *push_success* which pushes the value on the stack. The definition of *pop* is similar. The command $\varepsilon\langle s' \neq s\rangle$ corresponds to the environment performing a step that modifies s; it may also perform a finite number of stuttering program steps (i.e. steps that do not change observable variables). If the environment performs a step modifying s, $\varepsilon\langle s' \neq s\rangle$ terminates but if not, it becomes infeasible. The number of times *push_fail* is iterated is non-deterministic but it cannot execute an infeasible *push_fail*, i.e. one in which s is never changed by the environment, and hence termination of the iteration is forced in that case, so that the *push_success* alternative is taken.

4.2 Specification using temporal logic termination conditions

Note that if the stack s is never changed by the environment, the behaviour of the *push* (or *pop*) operation reduces to just its successful behaviour. More subtly, if eventually the environment stops changing s, then there can only be a finite number of failure iterations before the operation succeeds. This latter condition can be converted into a temporal logic termination condition $\Diamond\Box_{\varepsilon}(s' = s)$, i.e. eventually all environment steps do not change the value of the stack, which leads to the specification given in Fig. 5. An extended form of temporal logic is used here that distinguishes program and environment steps and allows one to specify a constraint on a step in the form of a relation, in this case $s' = s$. If parallel activity on the stack eventually quiesces, Treiber's push and pop operations are guaranteed to terminate, and hence the specifications with the quiescence termination conditions do not need to include the failure possibilities.

The negation of the termination condition is $\Box\Diamond_{\varepsilon}(s' \neq s)$, i.e. from every state there is eventually an environment step that modifies s. However –unlike for the blocking queue– that does not make either postcondition unsatisfiable and hence if the negation of the termination condition holds, each operation may either terminate satisfying its postcondition or never terminate. In a similar manner to the blocking queue, for a finite trace for which an operation has not yet satisfied its postcondition, it is still possible

$$\textbf{resource}\, s : \textbf{seq}\, Val \quad \textbf{initially}\, s = [\,]$$

$$push(v : Val) \mathrel{\widehat{=}} \textbf{terminate}\, \Diamond \Box_{\varepsilon}(s' = s) \cdot \textbf{with}\ s\ \textbf{do}\ s{:}\left[s' = [v] \frown s\right]\ \textbf{od}$$

$$pop()res : Val \mathrel{\widehat{=}} \textbf{terminate}\, \Diamond \Box_{\varepsilon}(s' = s) \cdot \textbf{with}\ s\ \textbf{do}\ s, res{:}\left[\begin{array}{l}(s \neq [\,] \Rightarrow s = [res'] \frown s') \wedge \\ (s = [\,] \Rightarrow res' = \mathsf{null}\end{array}\right]\ \textbf{od}$$

Figure 5: Stack with conditions to ensure termination

to extend the trace so that the postcondition is satisfied, and even so that the s is no longer modified by the environment, and hence the only allowable behaviour of the operation for a finite trace that has not yet satisfied its postcondition is finite stuttering. Hence if the termination condition is not satisfied an operation may either terminate successfully satisfying its postcondition or fail to terminate but only ever perform stuttering steps, i.e. it never modifies s. Hence the specifications in Fig. 5 are equivalent to those in Fig. 4.

If the termination conditions on the stack operations are replaced by *true*, that requires the operations always terminate even under interference from other threads performing *push* and *pop* operations. That gives strictly stronger specifications because their termination conditions are weaker. An implementation might then be required to make use of a lock that sequentialises access to the stack in the order in which the lock is requested (such as a ticket lock) in order to ensure termination.

5 Conclusions

To specify concurrent program components one needs to be able to address issues such as operation atomicity, operations blocking on conditions or locks, and transactional operations that may fail and need to be retried. Hoare's resource concept provides a notion of atomicity with respect to a resource. A contribution of this paper is to examine its interaction with rely and guarantee conditions in order to enable the initial refinement step of Hoare's **with** statements to code. Brookes [1] also makes use of the concept of a resource in concurrent separation logic. He generalises Hoare's concept to handle the heap as well as variables.

The specifications of operations using **with** statements do not dictate whether they are refined to implementations using locks or to non-blocking implementations or even a programming language that supports **with** statements. One issue not addressed here is that operations requiring multiple resources, e.g. an operation that needs to perform operations on two separate resources and needs to be considered atomic as a whole. In this case the **with** statement needs to allow for multiple resources, and if locking is used in the implementation of the operations, to avoid deadlock, the locking has to ensure that resources are locked in the same order.

Operations that block waiting for some condition have the potential for non-terminating behaviour. That has been addressed via two ways of specifying such operations: a form that explicitly includes both its terminating and non-terminating behaviours; and an implicit form that includes a condition that guarantees termination.

Non-blocking algorithms can provide more efficient solutions for managing shared data structures than using locks, but some algorithms have the issue that, under interference, they may fail and need to

be retried. In the worst case an operation may be continually thwarted and never get a chance to complete and hence its specifications needs to either allow for that possible behaviour or provide conditions under which it will terminate successfully, i.e. that the interference quiesces so that it can complete.

The approach taken in this paper is to indicate some directions for devising specifications for concurrent program components. In doing so we have shown that specifications with explicit await clauses can be expressed in a more abstract form with a temporal logic formula giving the condition under which termination is guaranteed. Such specifications have greater expressive power than those using explicit await constructs because temporal logic formulae allow termination conditions that cannot be expressed as await conditions, for example, if the blocking queue allowed multiple readers and hence its await condition was no longer stable, an alternative condition of $\Box\Diamond(qu \neq [\,])$ would require an implementation to guarantee the termination of each read operation under interference from other reads, provided a writer was also actively appending values to the queue. An implementation of *read* might, for example, use a lock that sequentialises access in the order in which the lock is requested (such as a ticket lock) in order to ensure termination.

Liang and Feng [15] have recently addressed handling progress conditions for blocking operations (which they refer to as partial methods). Their approach makes use of await statements but they give four different interpretations to await statements depending on whether one requires the operations to be starvation free or deadlock free, and depending on whether the enabling conditions are treated as weakly or strongly fair. These different interpretations give different termination behaviour for operations. The approach advocated here is to make use of different temporal logic conditions to cover these cases.

At this stage our treatment has not been fully formalised and further work is required to support refinement of such specifications to code.

Acknowledgements. Thanks are due to Robert Colvin, Cliff Jones, Larissa Meinicke, and Kirsten Winter, and the anonymous reviewers for feedback on the ideas presented here. This research was supported Australian Research Council Discovery Grant DP130102901.

References

[1] S. Brookes (2007): *A semantics for concurrent separation logic*. Theoretical Computer Science 375(1–3), pp. 227–270.

[2] R. J. Colvin, I. J. Hayes & L. A. Meinicke (2016): *Designing a semantic model for a wide-spectrum language with concurrency*. Formal Aspects of Computing 29, pp. 853–875.

[3] R. W. Floyd (1967): *Assigning meanings to programs*. In: Proceedings of Symposia in Applied Mathematics: Math. Aspects of Comput. Sci., 19, pp. 19–32.

[4] I. J. Hayes (2016): *Generalised rely-guarantee concurrency: An algebraic foundation*. Formal Aspects of Computing 28(6), pp. 1057–1078, doi:10.1007/s00165-016-0384-0.

[5] I. J. Hayes, R. J. Colvin, L. A. Meinicke, K. Winter & A. Velykis (2016): *An algebra of synchronous atomic steps*. In J. Fitzgerald, C. Heitmeyer, S. Gnesi & A. Philippou, editors: FM 2016: Formal Methods: 21st International Symposium, Proceedings, LNCS 9995, Springer International Publishing, Cham, pp. 352–369, doi:10.1007/978-3-319-48989-6_22.

[6] I. J. Hayes, C. B. Jones & R. J. Colvin (2014): *Laws and semantics for rely-guarantee refinement*. Technical Report CS-TR-1425, Newcastle University.

[7] I.J. Hayes, L.A. Meinicke, K. Winter & R.J. Colvin (2017): *A synchronous program algebra: a basis for reasoning about shared-memory and event-based concurrency*. Ext. report at arXiv:1710.03352.

[8] Maurice Herlihy & Jeannette M. Wing (1990): *Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst.* 12(3), pp. 463–492.

[9] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Communications of the ACM* 12(10), pp. 576–580, 583.

[10] C. A. R. Hoare (1972): *Towards a Theory of Parallel Programming*. In: *Operating System Techniques*, Academic Press, pp. 61–71.

[11] C. A. R. Hoare (1975): *Parallel programming: an axiomatic approach*. *Computer Languages* 1(2), pp. 151–160.

[12] C. B. Jones (1981): *Development Methods for Computer Programs including a Notion of Interference*. Ph.D. thesis, Oxford University. Available as: Oxford University Computing Laboratory (now Computer Science) Technical Monograph PRG-25.

[13] C. B. Jones (1983): *Specification and Design of (Parallel) Programs*. In: *Proceedings of IFIP'83*, North-Holland, pp. 321–332.

[14] C. B. Jones (1983): *Tentative Steps Toward a Development Method for Interfering Programs*. *ACM ToPLaS* 5(4), pp. 596–619.

[15] Hongjin Liang & Xinyu Feng (2018): *Progress of Concurrent Objects with Partial Methods*. *Proc. ACM Program. Lang.* 2(POPL), pp. 20:1–20:31.

[16] C. C. Morgan (1988): *The Specification Statement*. *ACM Trans. Prog. Lang. and Sys.* 10(3), pp. 403–419.

[17] C. C. Morgan (1994): *Programming from Specifications*, second edition. Prentice Hall.

[18] P. W. O'Hearn (2007): *Resources, Concurrency and Local Reasoning*. *Theoretical Computer Science* 375(1-3), pp. 271–307.

[19] S. Owicki (1975): *Axiomatic Proof Techniques for Parallel Programs*. Ph.D. thesis, Department of Computer Science, Cornell University.

[20] S. S. Owicki & D. Gries (1976): *An axiomatic proof technique for parallel programs I*. *Acta Informatica* 6(4), pp. 319–340.

[21] Susan Owicki & David Gries (1976): *Verifying Properties of Parallel Programs: An Axiomatic Approach*. *Commun. ACM* 19(5), pp. 279–285.

[22] Michael L. Scott (2013): *Shared-Memory Synchronization*. Morgan & Claypool Publishers.

[23] R. K. Treiber (1986): *Systems Programming: Coping with Parallelism.* Technical Report RJ 5118, IBM Almaden Research Center.