Reasoning about Active Object Programs

Jelte Zeilstra

November 15, 2016

Contents

Chapter 1

Introduction

1.1 Checking correctness of software

It is important to know that a large software project functions according to its specification. It must be free of errors and unwanted behaviour. Designing and programming a large system however is difficult and error-prone.

To prevent or find errors in programs several techniques are available. A good and clear specification and design to start with can already improve the program. During or after programming code analysis, testing and verification can find (potential) problems in the program. Each of these techniques has its own advantages and disadvantages.

Code analysis is automatic static analysis on the source code of a program. Code analysis can be performed by the compiler or an external tool. These tools can, depending on the programming language, include type guessing, detection of undeclared or unused methods and fields, duplicate code detection, and complexity analysis. Code analysis is relatively quick and easy but also superficial. It is solely based on the programming code itself and not on the intended behaviour.

Testing is performed by actually executing the code and comparing its result with the expected behaviour. Testing can focus on single methods (unit tests), larger components or the system as a whole. Test cases describe the input for the method or program and the expected output. Tests can be executed either manually or automatically. Testing is limited to predefined test cases and cannot detect errors in unexpected situations.

Verification is based on a formal description of the program. Such a description must be made for each method and class that is being verified. This formal description can be used by the programmer as specification and by tools to check if the program respects its description. Tools can use both

run-time checking as well as static checking.

Because the description of methods uses a formal language, multiple interpretations are prevented. By using static checking, all possible code paths are checked. Downsides include the extra time and knowledge needed to write the formal specifications. Formal verification is still an active research area and is not available for all programming languages and program structures.

Ensuring correctness of concurrent and distributed programs is especially difficult as the number of possible traces grows exponentially with the number of concurrent processes (threads) and the interleaving of processes is non-deterministic by nature.

In the last years several techniques to verify concurrent have been developed, for example in the VerCors project. In this study we will research the possibility to use those techniques for verification of distributed programs. We will focus on two styles of distributed programs, namely message passing and active object programs.

1.2 Message passing

Message passing is a general approach for communication between concurrent or distributed actors. Actors can send messages to a queue, and another actor (or multiple actors) can read from the queue.

Multiple implementations exist, both programming language specific and language independent. Some queues guarantee lossless and in-order delivery, others do not.

In general, a *send* operation takes a queue identifier and a message as parameters. The *receive* operation takes a queue identifier as parameter and returns a message that was sent to the queue. There are multiple variants of the send and receive operations, both blocking (synchronous) or non-blocking (e.g. buffering).

Message passing is an one-way communication protocol. The sender is not notified what the result is of any actions taken by the receiver. Two-way communication can be build by using a second queue for returning messages.

1.3 Active objects

Active objects are a special type of objects. Methods invoked on an active object are not executed in the current thread, but in a separate thread [16], possibly on a remote machine. An active object can have a worker thread executing each method sequentially or start a new thread for each invoked

method. These methods could also return a value, allowing for two-way communication. After invocation, the caller can continue executing code, or wait for the callee to finish and collect the return value.

One could consider Remote Procedure Calls and even web services as variants on active objects. RPC services and web services (almost) solely act when there is an incoming call/request and send a response based on the request and their own data.

1.4 Research question

The goal of this research is to create and verify an implementation of active objects using MPI and VerCors.

This project develops a method for verification of active object programs using the existing VerCors tool. Also an implementation of active objects using MPI is developed.

The research question answered in this report is:

To what extent can an active object implementation with MPI be verified?

This question is divided in several sub-questions:

1. How can active objects be described and verified using permission-based separation logic?
2. How can active objects be implemented using MPI?
3. Can the active objects implementation in MPI be verified?
4. Can an example program be verified using the developed techniques?

The first question is discussed in chapter 3. The second question in 4. The third question in 5. The fourth question and the main question are answered in the conclusion in chapter 6.

Chapter 2

Background

To start, this chapter first gives an overview of techniques existing for the verification of distributed systems.

A lot of research on concurrent and distributed processes has already been done. In 1978 communicating sequential processes have been developed to reason about general parallel processes (section 2.1). To reason about concurrent programs at source code level, permission-based separation logic has been developed (section 2.2).

Starting in February 2011 the FMT group at the University of Twente developed a verification technique combining permission-based separation logic with histories to allow modular verification of concurrent programs (section 2.3). They also developed the VerCors toolset to automate verification of concurrent programs using permission-based separation logic and histories (section 2.4).

Jinjiang Lei et al. developed a techniques for verification of message passing programs (section 2.5 and 2.6).

Also section 2.7 gives an overview of the Message Passing Interface specification and implementations.

2.1 Communicating sequential processes

Communicating sequential programs (CSPs) were developed as a programming language by Hoare [12] but has been generalized into a formal language or process algebra for describing concurrency and non-determinism [29].

This formal language is based on events and processes. The syntax has various features for describing these processes. A process in CSP is essentially a sequence of events. Parallelism of events can be denoted by a special syntax.

One of the syntax elements in CSP is the guarded alternative. A simple

guard is written $(a \to P | b \to Q)$. This indicates that process P is executed if event a is matched and process Q is executed if event b is matched. When both events can be matched, one of the two process will be executed.

This syntax can be extended to a set of events with the notation $?x : A \to P(x)$. This indicates an event of set A which is then bound to variable x, followed by the process P with parameter x.

Also a notation of channels is introduced. Channels can be seen as just a message containing a channel name, a dot, and than a value. A special syntax is introduced to make the channels easier to use: $c?x : T \to P(x)$. This indicates a message from channel c with type T. If any type for x is allowed, $c?x \to P(x)$ is also used. For symmetry the syntax $c!x$ is introduced which indicates a message x being sent over the channel c. This however is equal to $c.x$.

We can use this notation for example to create a buffer.

An endless empty buffer $B^{\infty}_{<>}$ is a simple process, only an element x can be added with the incoming event $\mathit{left}?x : T$:

$$B^{\infty}_{<>} = \mathit{left}?x : T \to B^{\infty}_{<x>}$$

A buffer $B_{s^\frown<y>}$ with a sequence s followed by y has two choices: either an element x is added with the incoming event $\mathit{left}?x : T$ or the element y is removed from the buffer with the outgoing event $\mathit{right}!y$:

$$B^{\infty}_{s^\frown<y>} = (\mathit{left}?x : T \to B^{\infty}_{<x>^\frown s^\frown<y>} | \mathit{right}!y \to B^{\infty}_{s})$$

Guarded alternatives can be generalized into the external choice operator $\Box$. This is a binary operator which operates on two processes. It goes to the process of which the first event in the process is available.

With $P \Box Q$, if only events in P are enabled this goes to P, if only events in Q are enabled this goes to Q. And if events in both P and Q are enabled then either P or Q will be executed.

The external choice operator is a replacement for the guarded choice operator in all cases. It also allows to specify more processes then are possible with only the guarded choice operator.

In addition to the external choice operator a non-deterministic choice or internal choice operator is introduced. This is the operator $\sqcap$. The difference with the external choice is the choice for which of the two subprocesses is executed is made by the current process instead of depending on external events.

So the process $P \sqcap Q$ means either process P or Q and the current process can choose. If only events from one of the two subprocesses is available, this process can result in the $STOP$ process. The process is only required to

$$
\begin{aligned}
Counter(n,m) = &(down \rightarrow Counter(n-1,m)) \triangleleft n > 0 \triangleright STOP \\
&\Box (up \rightarrow Counter(n+1,m)) \triangleleft n < 7 \triangleright STOP \\
&\Box (left \rightarrow Counter(n,m-1)) \triangleleft m > 0 \triangleright STOP \\
&\Box (right \rightarrow Counter(n,m+1)) \triangleleft m < 7 \triangleright STOP
\end{aligned}
$$

Figure 2.1: Counter with two variables [29]

advance to one of the processes if events of both processes are available. Because the choice of the process can be made internally, the process P is a valid implementation of the specification $P \sqcap Q$.

The $\sqcap$ operator is also being used in the definition of refinement. If $P \sqcap R = R$, it is said that P refines R; P is more deterministic than R. So every event in P is also possible in R but not the other way around, which makes P more deterministic than R.

The third choice operator is the conditional choice operator. This is a choice based on a binary formula instead of the availability of events. The binary choice is written as $P \triangleleft b \triangleright Q$ which works exactly the same as an `if` b `then` P `else` Q statement in many other programming languages.

The conditional choice operator can also be used for making general specifications by enabling events only in specific situations. For example given a counter with two variables, one down-up and one left-right, both ranging from 0 up to 7. When both variables are 0, only the up and right events must be enabled. This can be done with the specification given in figure 2.1.

Another feature are multipart data events which are events with multiple data attributes. This can be noted as $d?x?y!z!t$. The variables x and y are received from the environment and the variables z and t are put in the environment. These can be used to send and receive multiple parts of data at one time.

Parallel operators CSP defines several parallel operators. The simplest is the synchronous parallel operator $\|$. In $P \parallel Q$, P and Q must synchronise on all actions, i.e. all events in P must also happen in Q and vice versa. The second variant is the alphabetized parallel operator $P \ {}_X\|_Y \ Q$: P can only use events in X, Q only events in Y and they must synchronise on events in $X \cap Y$. This gives $P \parallel Q = P \ {}_\Sigma\|_\Sigma \ Q$ if Σ is the set of all possible events.

The third operator is the interleaving operator $|||$. With this operator, no synchronisation occurs between the processes. If a is enabled in both P

and Q, only one will match:

$$(a \to P) \;|||\; (a \to Q) = a \to \Big(\big(P \;|||\; (a \to Q)\big) \sqcap \big((a \to P) \;|||\; Q\big)\Big)$$

The last operator is the generalized parallel operator. In $P \underset{X}{\parallel} Q$ can use all events and synchronise on events in X. The other operators can be replaced by the generalized parallel operator: $P \parallel Q = P \underset{\Sigma}{\parallel} Q$, $P \;|||\; Q = P \underset{\{\}}{\parallel} Q$. Also if P only uses events in X and Q only uses events in Y, then $P \;{}_X\|_Y\; Q = P \underset{X \cap Y}{\parallel} Q$.

For the syntax elements in CSP a number of algebra rules are defined to indicate the symmetry, associativity, distributivity and idempotence of operators. Also step-rules are defined to show the behaviour of a process.

For each process the set of traces can be determined. The traces of a process P, $traces(P)$ is a possibly infinite set of event sequences. Traces can both be timed and untimed but mostly untimed traces are used.

In the set of traces the $\Box$ and $\sqcap$ operators are indistinguishable because traces have no notion of where possible choices are made. Traces are used to formulate specifications of CSP processes. P *sat* $R(tr)$ means all possible traces tr of P satisfy the condition $R(tr)$: $\forall tr \in traces(P) \cdot R(tr)$.

CSP has been used to verify the T9000 virtual channel processor [4] and for finding and fixing a vulnerability in the Needham-Schroeder public-key protocol [21].

2.2 Permission-based separation logic

Separation logic [28] is an extension of Hoare logic [11] which allows to reason about the heap of a program. It introduces the separating conjunction operator $*$. The formula $f * g$ is valid for heap h (notation: $h \vdash f * g$) if the heap h can be split into two separate heaps h' and h'' such that f is valid for h' ($h' \vdash f$) and g is valid for h'' ($h'' \vdash g$). Intuitively: f and g must be valid formulas about different parts of the heap.

Also the separating implication or magic wand operator $-\!\!*$ is introduced. This is the separation logic equivalent of the implication $\to$. $h \vdash f \mathrel{-\!\!*} g$ is valid if the heap h can be extended with a disjoint heap h' and if $h' \vdash f$ then $(h \cup h') \vdash g$. From $h \vdash f * (f \mathrel{-\!\!*} g)$ we can conclude with the modus ponens rule $h \vdash g$. There is also a separating equivalence or two-way magic wand operator $*\!\!-\!\!*$, similar to $\leftrightarrow$. $f *\!\!-\!\!* g$ means that both $f \mathrel{-\!\!*} g$ and $g \mathrel{-\!\!*} f$ hold.

With these operators we can reason about threads of concurrent programs in isolation. If two threads operate on disjoint parts of the heap they can be proven separate from each other and later be combined using the $*$ operator.

Permissions However requiring that threads operate on completely separate heaps is unnecessarily restricting. Multiple threads can read a shared variable without problems if it is guaranteed no thread is writing to the variable at the same time.

To solve this problem permissions are introduced. A thread can have a read (partial) or write (full) permissions for a certain variable/memory location [6]. A permission is a value in $(0, 1]$. The sum of permissions for a variable for all threads cannot exceed 1. It is therefore guaranteed that if a thread as a partial (read) permission ($0 < p < 1$), no other thread can have a full (write) permission for the same variable. A permission π to variable x with value v can be written as the predicate $x \overset{\pi}{\mapsto} v$. Permissions can be split and combined with the following rule: $x \overset{\pi_1}{\longmapsto} v * x \overset{\pi_2}{\longmapsto} v \mathrel{*\!\!-\!\!*} x \overset{\pi_1+\pi_2}{\longmapsto} v$.

2.3 Histories and Futures

When using locking to protect shared variables, verification of functional properties using permission-based separation logic is difficult. When a thread releases its lock it can no longer make guarantees about the unlocked variables because another thread can change it at any moment. Ghost variables can be used to track local modifications to a variable but this approach is not modular.

To specify the functional behaviour of concurrent programs with finite executions, so-called histories can be used [33]. With histories you can locally specify the actions of a thread using process algebra and later combine the actions of all threads. An action is an atomic modification of variables and can have pre- and post-conditions.

Histories are specified with the history predicate $\mathrm{Hist}(L, \pi, R, H)$ [5, 31]. L is the set of locations which belong to this history. π indicates the fraction of this history: 1 indicates a full history, any fraction thereof a fractional (local, incomplete) history. R is a predicate over L specifying the initial state. H is the process algebra term representing the history. A history is sometimes noted as $\pi\mathrm{Hist}(L, R, M)$ [32] (with $M = H$).

A history can be initialised with a location set L and a predicate R about the initial state with the specification command $\mathrm{crhist}(L, R)$. A full permission for each location $l \in L$ is required, i.e. $i \overset{1}{\mapsto} v$. This will produce

this history predicate $\text{Hist}(L, 1, R, \epsilon)$ and history permissions predicates $i \overset{1}{\mapsto}_h v$ for each $l \in L$. $\mapsto_h$ indicates that changes to the variable must be recorded in the history. The history permissions are typically stored in a lock.

The history can be split to allow it to be used by multiple threads with the following rule:

$$\text{Hist}(L, \pi, R, \epsilon) \multimap \text{Hist}(L, \frac{\pi}{2}, R, \epsilon) * \text{Hist}(L, \frac{\pi}{2}, R, \epsilon)$$

Also non-empty histories can be split, but then a synchronisation barrier is required [32]. Histories can be combined again with the rule:

$$\text{Hist}(L, \pi_1, R, H_1) * \text{Hist}(L, \pi_2, R, H_2) \multimap \text{Hist}(L, \pi_1 + \pi_2, R, H_1 \parallel H_2)$$

When a threads wants to update variables/locations covered by a history, it must do so in a so-called action segment. This requires a (partial) history predicate $\text{Hist}(L, \pi, R, H)$. The modifications are specified in the action contract and the action a is stored in the history predicate: $\text{Hist}(L, \pi, R, H \cdot a)$. The allowed operations in the action segment are limited to ensure it is seen as an atomic action by the other threads. For example, no permissions can be released and no other threads can be started or joined in the segment.

When a thread has a full history, it can reinitialise the history. With the specification command $\text{reinit}(L, R')$ the predicate $\text{Hist}(L, 1, R, H)$ can be converted to $\text{Hist}(L, 1, R', \epsilon)$ if R' can be proven to hold after all possible traces w in process H, i.e. $\forall w \in \text{Traces}(H).\{R\}w\{R'\}$.

If the history is not needed any more it can be destroyed with $\text{dsthist}(L)$. It requires the $\text{Hist}(L, 1, R, \epsilon)$ and the $i \overset{1}{\mapsto}_h v$ predicates for all $l \in L$. This command will return again the normal permission predicates $i \overset{1}{\mapsto} v$.

Futures Histories are limited to finite processes because only after termination of the individual processes you can reason about the global result. To allow the verification of infinite processes *futures* are developed [31, 25, 24]. Instead of obtaining the global process at the end of the program, the global process is defined in advance. In essence a future is a reversed history, e.g. in an action segment an action is removed from the future instead of being added to the history. This allows us to specify and verify infinite processes.

2.4 VerCors

VerCors is a tool set for verification of concurrent programs written in Java, PVL [2] or C. The syntax used by VerCors is an extension of JML (Java Modelling Language) [17, 13], an annotation language for Java.

VerCors syntax	**Description**
`**`	separating conjunction ($*$)
`-*`	separating implication ($-\!\!*$)
`\forall*`	universal separating quantification ($p_1 * p_2 * ...$)
Perm(x, p)	permission on x with fraction p $\in (0, 1]$ ($x \overset{p}{\mapsto} \_$)
Perm(x, **read**)	permission on x with unspecified fraction ($x \overset{\epsilon}{\mapsto} \_$)
Value(x)	same as **Perm**(x, **read**)
PointsTo(x, p, v)	permission on x with frac. p and value v ($x \overset{p}{\mapsto} v$)
	PointsTo(x, p, v) $\Leftrightarrow$ **Perm**(x, p) $*$ x = v
`create` hist;	crhist(L, R)
`split` hist, 1/2, empty, 1/2, empty;	Hist($L, 1, R, \epsilon$) $-\!\!*$ Hist($L, \frac{1}{2}, R, \epsilon$) $*$ Hist($L, \frac{1}{2}, R, \epsilon$)
`merge` hist, 1/2, h_1, 1/2, h_2;	Hist($L, \frac{1}{2}, R, h_1$) $*$ Hist($L, \frac{1}{2}, R, h_2$) $-\!\!*$ Hist($L, 1, R, h_1 \parallel h_2$)

Table 2.1: Overview of VerCors syntax [1]

VerCors adds the permission-based separation logic elements to JML, such as the Permission property Perm, the separating conjunction `**` (double star to distinguish from the multiplication operator), the magic wand operator `-*` and the universal separating quantification `\forall*`. There is also support for histories and futures in VerCors. An overview of the most import syntax additions is given in table 2.1.

VerCors additionally supports abstract predicates [27], i.e. parameterized predicates of which de definition can depend on the actual subclass of an object. Abstract predicates can be defined in classes as a member with type `resource`.

VerCors supports several back ends, for example Chalice[1]-Boogie[2], Carbon and Silicon (both part of the Viper suite[3]). The Silicon back end is currently worked on the most. All these back ends use the Z3 theorem prover[4] for proving verification conditions.

```
send (2, pt);   ||  x := recv (pt);
send (3, pt);   ||  y := recv (pt);
```

Figure 2.2: 2 threaded sender-receiver [18]

```
agent5 () {
  v1 = recv (1@5); v2 = recv (2@5);
  while (v1 = 0 || v2 = 0){
    if (v1 > v2 ) {send (v2 , 2@6); v2 := recv (2@5);}
    else {send (v1, 2@6); v1 := recv (1@5);}}
  while (v1 = 0) {send (v1 , 2@6); v1 := recv (2@5);}
  while (v2 = 0) {send (v2 , 2@6); v2 := recv (1@5);}
  send (0, 2@6);
}
```

Figure 2.3: Agent 5 of Merge sort [18]

2.5 Modular Reasoning for Message-passing Programs

In "Modular Reasoning for Message-passing Programs" [18, 19] Jinjiang Lei et al. introduce a method for modular verification of message-passing programs using event traces. In the event trace all send and receive actions are stored. The event trace is then used to verify the correctness of the message passing part of the program. This method is based on rely-guarantee-based reasoning [14].

This paper includes two examples that are verified on paper, a two-threaded sender-receiver (figure 2.2) and a multi-threaded merge sort (figure 2.3).

Message sends and receives are modelled as events in *event traces* or *event graphs*, which are a way to model the relations between these events. Events contain a port (or channel) identifier and a value and are linked to the previous event in the agent (local direct predecessor). Receive events are also linked to the corresponding send event. Using these two references, a Happens-Before relation for a trace tr can be defined: event e happened before event e' (notation: $e \prec e'$) iff e is the local direct predecessor of e', e

[1] `http://research.microsoft.com/en-us/projects/chalice/`
[2] `http://research.microsoft.com/en-us/projects/boogie/`
[3] `http://www.pm.inf.ethz.ch/research/viper.html`
[4] `https://github.com/Z3Prover/z3`

is the sending event for receiving event e' or there is an event e'' for which $e \prec e'' \wedge e'' \prec e'$ holds.

This paper defines a logic based on Hoare logic to reason about a program containing send and receive instructions for message queues. It splits the pre- and post-conditions P and Q of the classical Hoare triple ($\{P\}\ C\ \{Q\}$) in local state and assumptions about the environment (other threads/agents), resulting in a triple $\{r, p\}\ C\ \{r', q\}$ in which r and r' represent the assumptions about the environment and p and q the local states.

In these triples, a send event can be noted as $pt!m$ in which pt is the port and m the message. A receive event is similar notated as $pt?m$. $p * q$ denotes a separating conjunction similar to separation logic, $p \circ q$ is the sequential conjunction which also (roughly) requires that events in p happen before q. emp is an empty trace.

The sender agent from the first example (figure 2.2) would produce the Hoare triple $\{\text{emp}, \text{emp}\}$ `send (2, pt); send (3, pt);` $\{\text{emp}, pt!2 \circ pt!3\}$. This indicates no starting assumptions, and it generates two send events without new environment assumptions.

The receiver agent is more complicated: $\{\text{emp}, \text{emp}\}$ `x := recv (pt); y := recv (pt);` $\{pt!2 \circ pt!3, (pt?2 \circ pt?3) \wedge x = 2 \wedge y = 3\}$. It ensures two receive events and sets $x = 2$ and $y = 3$. It also assumes the environment has two send events, the first one containing 2 and the second 3.

The composition of the two agents results in $\{\text{emp}, \text{emp}\}$ `send (2, pt); send (3, pt);` $\|$ `x := recv (pt); y := recv (pt);` $\{\text{emp}, x = 2 \wedge y = 3 \wedge (pt!2 \circ pt!3) * (pt?2 \circ pt?3)\}$. The assumptions about the environment of the receiver are met by the sender, so the combined system has no environmental assumptions.

The paper contains a set of inference rules to reason about the programs. These inference rules formalize the reasoning with the new Hoare triples.

The proposed method has some limitations. It assumes that all message arrive without loss and in the same order. There is as far as I know, no tool support available.

2.6 Rely-Guarantee Based Reasoning for Message-Passing Programs

Jinjiang Lei et al. later published the article "Rely-Guarantee Based Reasoning for Message-Passing Programs" [20]. This is an extension of their previous paper [18] discussed in the previous section. In this paper they add an example with non-deterministic behaviour (figure 2.4) and they verify a

leader election algorithm (figure 2.5). The rest of the paper is essentially the same as the previous.

2.7 Message Passing Interface

The Message Passing Interface[5] (MPI) is a library interface specification for communication between parallel and distributed processes. The interface is initially defined for C/C++ and Fortran but language bindings for several other languages such as Java are also available. The fist version was published in 1994 by Dongarra et al. [10] and the latest version is version 3.1 published in 2015 [22].

MPI has several communication modes for send operations:

Synchronous mode The message will be directly copied from the sender to the receiver. The send operation only finishes when a matching receive operation is started.

Buffered mode In buffered mode the message will first be copied to an internal buffer. The operation finishes when the message is placed in the internal buffer, but the message might not be delivered yet. The operation returns an error when there is no buffer space available.

Standard mode The MPI library will automatically choose buffered mode when there is buffer space available or otherwise use synchronous mode.

Ready mode Ready mode send can be used when the matching receive operation is already started. This allows for some optimizations in some systems. The semantics are the same as for the standard or synchronous mode. When no matching receive is available, the behaviour is undefined.

For all modes of the send operation there exists a blocking and a non-blocking variant:

Blocking A call to a blocking operation only returns when the operation finished. The used send buffers can be safely reused as the message is either send to the receiver or copied to the internal buffer.

Non-blocking A call to a non-blocking returns immediately and uses a Request object to refer to ongoing operation. With a `test` and `wait` call the process can check if the send operation is finished. Non-blocking methods are prefixed with an `i`, e.g. `isend`.

[5] `http://mpi-forum.org/`

There are no different communication modes for the receive operation, the receive operation can receive messages from all send modes. The receive operations does have a blocking and a non-blocking variant.

To differentiate between messages send to the same destination a tag can be added to a message. The receiver can filter incoming messages on source and/or tag.

Besides point-to-point communication, MPI also supports broadcast messages and barriers. Other features include multiple communication universes, persistent communication requests and data types. These features are however not used in this paper and therefore not discussed here.

Java In Java there are two mayor MPI-like interfaces: mpiJava [3, 8] and MPJ [9]. MPJ was developed as a common interface for MPI implementations in Java as successor of the functionality comparable but different interfaces: mpiJava, JavaMPI [23] and MPIJ [15].

MPJ Express[6] is popular and actively developed [30] MPI implementation. Despite its name and original intentions, MPJ Express implements the mpiJava 1.2 API. MPJ/Ibis[7] is an implementation of the MPJ interface from the VU Amsterdam [7].

[6] `http://mpj-express.org/`

[7] `http://www.cs.vu.nl/ibis/mpj.html`

```
send (2, pt); || send (3, pt); || x := recv (pt);
                                  y := recv (pt);
```

Figure 2.4: 3 threaded sender-receiver [20]

```
agent_i(){
  ld_i := ff;
  send (token_i, i + 1);
  tk_i := recv (i - 1);
  while (tk_i = 0){
    if (token_i < tk_i) {send (tk_i, i + 1);}
    if (token_i = tk_i) {ld_ i = tt; send (0, i + 1);}
    tk_i := recv (i-1);
  }
  if (ld_i = ff){send (0, i + 1);}
}
```

Figure 2.5: Leader election [20]

Chapter 3

Specification of asynchronous methods

A key part of active objects is the asynchronous method call. This chapter shows how asynchronous method calls can be scribed using the VerCors extension of JML. It also shows that this specification can be used by VerCors to automatically verify a local asynchronous method call. The formal description of asynchronous method calls is used in later chapters in the specification of active objects.

An asynchronous method call differs from a normal, synchronous, method call because the caller does not wait for the called method to be finished, but executes some other code in parallel. This can be useful when the method takes a longer time to complete, for example because it needs to access slow resources or executes remotely using active objects.

When the caller needs the result of the asynchronous method, it either can get it directly if the call is already finished, or the caller will block till the call is finished. How this last part is implemented differs between programming languages. In Java the `Future`[1] class can be used for this.

3.1 Specification of the `Future` class

The Java language has no built-in mechanism for returning values of asynchronous calls, therefore a dedicated class or interface must be used to provide this feature. The `Future` interface of the `java.util.concurrent` package seems to be the best interface for this, as the Java API specifies: "A `Future` represents the result of an asynchronous computation." [26].

[1] Not to be confused with the Future in specifications described in section 2.3

Listing 1: Specification of `Future` class

```
public class Future {
  //@ public resource getToken();
  //@ public resource postGet(Object value) = true;

  /*@
    ensures getToken();
  @*/
  public Future() {
    //@ assume getToken();
  }

  /*@
    given frac p;
    requires [p]getToken();
    ensures [p]postGet(\result);
  @*/
  public Object get();
}
```

The `Future` interface is used as follows: an asynchronous method call will return a `Future` instance instead of its actual return value. When the caller needs the actual return value it will call the `get` method on the `Future` instance. If the result is not yet available, the call will block it it is available and return the value. If the value is already available, it will return it directly.

We have based our specification of the `Future` interface on the specification of the `Thread` class by Amighi et al. [1]. The specification of `Future` is given in Listing 1.

For specification purposes we changed the `Future` interface to a class, so it can have a constructor which returns the `getToken`, a token which is required by the `get` method. This token is used to get permissions back (using the `postGet` predicate) from the active object call. This is similar to the `joinToken` and `postJoin` predicate of the `join` method in `Thread`.

The `get` method is parametrized with the fraction `p` to allow to retrieve only a part of the post-condition `postGet`. This enables sharing the `Future` object between multiple threads or other objects. Because there is only one full `getToken`, only one full `postGet` can be retrieved from the `get` method.

If the `postGet` predicate refers to the parameters of the asynchronous method, these parameters must also be stored in the Future object as specification-only fields.

The original interface has four other methods: `cancel`, `get` with a time-out, `isCancelled` and `isDone`. Because these methods are not essential, we

have not included them in our specification.

3.2 Example: ForkJoinTask

To demonstrate the usage of the specification of the `Future` class, we have implemented a `ForkJoinTask`, based on the idea of the `ForkJoinTask` from the Java API.

A `ForkJoinTask` is a mechanism in the Java API to run a tasks in parallel. The `exec` method defines the task of the `ForkJoinTask` object. The task is executed in a different thread than the main thread and the result can be collected using the `get` method (from the `Future` interface).

The source is given in listing 2. In this example the task will always return the value 5 in a `ValueContainer` instance. The correctness of this implementation is verified using the VerCors tool set.

Two extra predicates are added, `preExec` and `execToken`. `preExec` containing the precondition for the `exec` method, similar to `preFork` in `Thread`. In this example, the precondition is just `true`.

The `execToken` is a token which guarantees that the `exec` method is executed at most once.. It also contains the permissions to store the result value and modify the `done` state variable.

The variable `done` indicates if the result is available. The variable `value` contains the result value when `done` is true.

The specification variable `dummy` is used by the `getToken` predicate and the invariant. Only the permissions on this variable are used, not the value. When the `get` method is called, the permissions on the `dummy` variable are obtained from the `getToken`. Subsequently the permissions on the `dummy` variable are exchanged for permissions on the post-condition (`postGet` predicate) using the invariant. In the specification variable `getExecuted` is stored which amount of the `postGet` predicate is already returned using the `get` method.

The CSL invariant stores a half permission to the `done` variable, full permission to the `getExecuted` variable and a `getExecuted` fraction of the `dummy` variable. When `done` is false, the `getExecuted` fraction must be none. When `done` is true, it also contains an additional read permission for the `done` variable indicating it cannot be changed back to false, read permission for the result value and a $1 - $ `getExecuted` fraction of the `postGet` predicate. The sum of the permissions on the `dummy` variable and the `postGet` predicate is then always 1. This ensures that no more than a full `postGet` can be returned by the `get` method.

The constructor will return both the `execToken` and the `getToken`. In

multi-threaded programs, the `execToken` will normally be passed to the executing thread and the `getToken` to the calling thread.

The `exec` method does the real execution of the task. In this example it will simply return a `ValueContainer` with the value 5.

The `doExec` method calls the `exec` method, stores the result in the `value` variable and sets `done` to true.

The `get` will wait till `done` is true using a while-loop. After that the fraction `p` is added to `getExecuted` to get the `postGet` predicate from the invariant. The result value and a `p` fraction of `postGet` will be returned.

Listing 2: Specification of `ForkJoinTask` class

```
public class ForkJoinTask /* extends Future */ {
  /*@
    public static resource preExec() = true;
    public resource execToken() = Value(this.done) ** PointsTo(this.done.val, 1/2,
      false)
                                    ** Perm(this.value, 1);
    public resource getToken() = Perm(this.dummy, 1);
    public resource postGet(ValueContainer value) = PointsTo(value.x, 1, 5);

    private boolean dummy;
    private frac getExecuted;
    resource csl_invariant () = Value(this.done) ** Perm(this.done.val, 1/2)
        ** Perm(this.getExecuted, 1)
        ** (this.getExecuted != none ==> Perm(this.dummy, this.getExecuted))
        ** (!this.done.val ==> this.getExecuted == none)
        ** (this.done.val ==> (Value(this.done.val) ** Value(this.value)
                              ** [1 - this.getExecuted]postGet(this.value)));
  @*/

  private AtomicBoolean done;
  private ValueContainer value;

  /*@
    ensures execToken();
    ensures getToken();
  @*/
  public ForkJoinTask(){
    this.done = new AtomicBoolean();
    //@ this.getExecuted = none;
    //@ fold this.getToken();
    //@ fold this.execToken();
  }

  /*@
    requires preExec();
    ensures postGet(\result);
```

```
36    @*/
      private ValueContainer exec() {
38      ValueContainer res = new ValueContainer();
        res.set(5);
40      //@ fold postGet(res);
        return res;
42    }

44    /*@
        requires preExec() ** execToken();
46      ensures  Value(this.done) ** Value(this.done.val) ** this.done.val;
      @*/
48    public void doExec() {
        //@ unfold this.execToken();
50      this.value = exec();
        this.done.set(true);
52    }

54    /*@
        given frac p;
56      requires p > none ** [p]getToken();
        ensures  [p]postGet(\result);
58    @*/
      public ValueContainer get() {
60      boolean d = false;

62      //@ loop_invariant d ==> Value(this.done) ** Value(this.done.val) ** this.done
        .val ** Value(this.value);
        while(!d) {
64        d = this.done.get();
        }
66
        //@ unfold [p]this.getToken();
68      /*@
          atomic(this) {
70          frac oldGetExecuted;
            oldGetExecuted = this.getExecuted;
72          assert oldGetExecuted + p <= write;
            this.getExecuted = oldGetExecuted + p;
74        }
        @*/
76
        return this.value;
78    }
}
```

Listing 3: `get` method of `ValueContainer` class

```
/*@
  given frac p;
  given int res;
  requires p != none ** PointsTo(x, p, res);
  ensures PointsTo(x, p, res) ** \result == res;
@*/
public int get() {
  return x;
}
```

3.3 Limitations and bugs in VerCors

During the development and verification of the `Future` and `ForkJoinTask` classes we have discovered some limitations and bugs in the VerCors tool. Some issues are already solved, but there are others remaining:

- Because class parameters are not yet implemented in VerCors, for each `Future` with a different 'postcondition' (`postGet` predicate) a new subclass is required.
- It is not possible to access a ghost parameter of an overridden method, such as the `p` parameter of the `get` method. This was overcome by making `ForkJoinTask` not a real subclass of `Future`.
- It is also not yet possible to use generics. Subclasses of `Future` which use a result type other than `Object` can therefore not be real subclasses.
- It is not possible to do arithmetic operations on instance ghost variables with type `frac` directly. A workaround is to first store the value in a local ghost variable, as on line 71 of listing 2.
- Declaring and assigning a local (ghost) variable in one statement sometimes fails due to a bug. A workaround is to split the assignment to a second statement, as on lines 70 and 71.
- The property that a `frac` is always larger than zero is not automatically available everywhere. The `get` method of `ValueContainer` (listing 3) can not be verified without `p != none` in the precondition (line 4).

Support for comparing and subtracting fractions was not yet available in VerCors. We have added this functionality to VerCors.

3.4 Asynchronous methods

Because of the mentioned limitations, the specification of asynchronous methods requires a separate `Future` subclass for each method. In an ideal situation it would be nice to use the general `Future` class for specification of asynchronous methods. A specification could then be something like this:

```
/*@
requires true;
ensures \result.getToken() ** \result.contract == {PointsTo(value.x, 1, 5)};
@*/
public Future<ValueContainer> getFive();
```

Chapter 4

Implementation of active objects using MPI

This chapter describes our implementation of active objects using MPI messages in Java. With this implementation, a programmer can define and use active objects in a distributed Java program. The communications between the processes is done via message passing using an MPI library.

The basics of the implementation are quite simple. Each active object method call is translated to two MPI-messages: a message from the caller to the callee containing the name of the called method and the parameters and a message from the callee to the caller containing the result of the method call.

This chapter first gives an overview of the MPI operations used in our implementation, then we explain in more detail how we encoded active objects using the MPI operations. Subsequently we will discuss how we implemented this encoding in Java and tested it.

4.1 MPI

We first give a brief overview of the main operations in the Message Passing Interface. In MPI many different operations are available for sending and receiving messages. However for our implementation we only used the basic send, receive and operations. The send and receive operations have a non-blocking variant (starting with an `i`) and a blocking variant. For each operation we describe its parameters, return value and the effects of the operation.

- `isend`: non-blocking send operation
  parameters: destination, tag, datatype and message buffer with mes-

sage
returns: request handle
This operation sends a message to the given destination. This method returns immediately. With the `wait` method the program can wait till the message is copied to the internal buffer or is sent.

- `irecv`: non-blocking receive operation
  parameters: source (or any), tag (or any), datatype and message buffer in which the received message will be stored
  returns: request handle
  This operations receives a message from the given source with given tag. This method returns immediately. With the `wait` method the program can wait till the message is actually received.

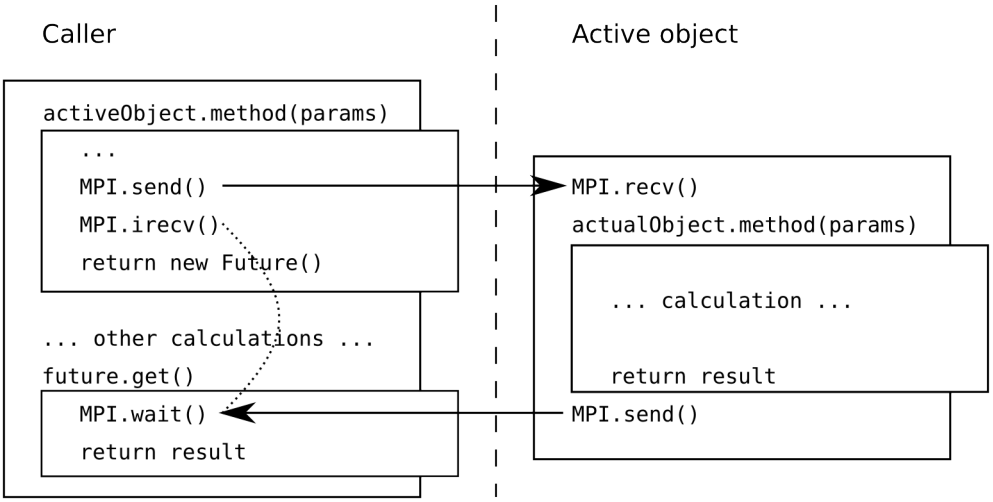
- `wait`: wait for a non-blocking request to be finished.
  parameters: request handle
  returns: status information, depending on request type
  This method returns when the related request is finished, e.g. a message is received.

- `send`: blocking send operation
  parameters: same as `isend`
  returns: nothing
  This operation sends a message to the given destination and blocks until the message is copied to the internal buffer or is sent. This method is equivalent to `r = isend(...); wait(r);`.

- `recv`: blocking receive operation
  parameters: same as `irecv`
  returns: status information: source, tag, message length
  This operations receives a message from the given source with given tag. This method returns when the message is received. Equivalent to `r = irecv(...); wait(r);`.

MPI supports many more operations, for example sending broadcast messages and barrier synchronization. These are not used in our implementation and are not discussed further.

4.2 Encoding of active objects with MPI



Caller When a caller calls an active object method, it will first call the MPI-method `send` with the method name and parameters as message and the callee as destination. Then it will call `irecv` to indicate it wants to receive the result[1]. When the caller wants to use the result, it calls the `wait` method. If the result is not yet available, the caller will block. When the result is available the `wait` method returns and the caller can use the result.

Active object The active object (callee) will first call `recv` to wait for an incoming active object method call. When an active object method call is received, `recv` returns. The active object will then execute the actual method. When the method call returns, the result will be sent to the caller with `send`. After that, the active object will loop back to the `recv` call to receive another method call.

Tags To distinguish between messages sent to the same MPI node, *tags* can be used. For method call messages, an identifier for the active object is used. This identifier must be unique for all active objects on the same MPI node. For result messages an identifier for the related call is used. This identifier is generated by the caller and sent to the callee in the method call

[1] `irecv` can also be called before `send`. This allows the callee to use `rsend` (ready-mode send) to send the result, since it is guaranteed that `irecv` is already called. This could be more efficient in some situations.

message. This identifier must be unique for all (concurrent) method calls from the same caller MPI node and must also not conflict with active object identifiers on the same MPI host.

This allows multiple active objects on the same MPI node, as the incoming messages are routed to the correct active object based on the message tag. Similarly an MPI node can have multiple outstanding active object method calls, as the `irecv`/`wait` methods can wait on the correct result based on the tag for the call.

4.3 Implementation in Java

We have created an implementation of active objects using MPI in Java. We have created this program using the MPJ Express library, an MPI implementation for the Java programming language.

This implementation has a limitation: the parameters of active object method calls must either be primitive types, serializable objects or references to other active objects. Non-serializable objects cannot be used as they cannot be converted to a byte stream and sent to another node. We think however that this restriction is reasonable and not restricting any actual programs.

4.3.1 Active object

The implementation of the active object is relative simple. Each active object will be a thread. The thread will receive a message for its object, execute the named method with the given parameters, send the result back to the caller and repeat this process.

For the execution of the method two methods can be chosen: in the same thread as receiving the messages or a separate thread per method call.

Executing the method calls in the thread which receives the messages in simpler to implement and verify. Also probably no locking in the active object is required as only a single method can be called at the same time. This process does not allow for simultaneous or recursive active object method calls to the same object since all calls will be executed sequentially. Recursive calls will deadlock since the inner call cannot start before the outer call is finished.

Creating a separate thread per method call allows for simultaneous and recursive calls. The active object might require a locking mechanism depending on its data structure. This implementation can be harder to verify, but this can be solved by using histories or futures.

4.3.2 Caller

To be able to call methods on active objects, the caller node must have a reference to the active object. This is achieved by making a proxy object for the active object, containing the MPI node id and the tag of the active object. This proxy object has the same methods as the remote object.

When a method on the proxy object is called, a message must be sent to the MPI node on which the active object resides, containing the method name, parameters and call identifier. To encode the information a `Call` object is used. This is essentially a tuple containing the method name, the parameters and the call identifier (the tag for the response).

To allow asynchronous method calls, these proxy methods do not return the actual result value, but a `Future` object to access the return value at a later moment, as discussed in chapter 3.

When the caller calls an active object method, a `Future` object is created. This `Future` object is associated with the request handle return by the `irecv` call. When the caller wants to use the result value (calls `future.get()`) the MPI `wait` function is called. Because MPI allows only one thread to call `wait` on a single handle, the `Future` object makes sure it is only called once. Multiple threads calling the `get` function are handled in the `Future` object internally.

As Java supports method overloading, multiple methods with the same name but different parameter types can exists. To also support overloading of active object methods, the method name in the `Call` object can be extended to also encode parameter types, for example using method descriptors as defined in section 4.3.3. of the Java Virtual Machine Specification.

4.3.3 Object creation

To allow creation of objects we made a `Factory` active object which is created at the start for each MPI node. The constructors of other classes can be called via this factory. The `Factory` will then create the active object and the corresponding proxy object. It will return the proxy object to the caller of the constructor. The `Factory` is an active object itself, so it allows for asynchronous creation of objects.

To simplify the program, it is also possible to start a program with predefined instances at specific MPI nodes and disable active object creation and the `Factory` object.

4.3.4 Proxy classes

For all active object classes a proxy class must be created containing the code to forward method calls to the actual active object. These proxy classes must currently be created by hand. However the required information can be extracted from the actual classes and automatic generation of the proxy classes should be possible.

4.4 Testing

We have tested the implementation using a couple of small programs. We have tested with multiple MPI nodes (2 - 8) on a single computer. We have not tested the programs in a real distributed setup.

The source code is available at `http://fmt.ewi.utwente.nl/education/master/145/`.

Chapter 5

Verification of the active objects implementation

To verify the correctness of our program, a couple of steps have been taken. MPI programs are concurrent and/or distributed and thus possibly non-deterministic. To be able to specify and verify our implementation in a modular fashion, we have chosen to used permission-based separation logic extended with futures as described in sections 2.2 and 2.3.

To be able to use those futures, first action and processes must be defined (sections 5.1 and 5.2). Using these actions and processes we annotated the MPI library (section 5.3). These annotations use a predicate describing what valid MPI messages, this is explained in section 5.4. With these building block we then annotated our implementation (starting section 5.5). In the final section we describe the status of tool support for our specification.

This chapter shows it is possible to annotate the MPI operations in the library using futures. It also show it is possible to annotate most parts of our active objects implementation.

5.1 Actions

To create a specification using futures, actions must be defined as these are used in the process algebra terms in the future predicates.

We based the actions on the MPI operations as discussed in section 4.1. Compared to the MPI operations, the actions have more parameters. These parameters are used to store extra information needed for the specification, such as return values.

The defined actions are:

- **isend**$(source, dest, msg, tag, r)$: Non-blocking send action.

isend(0, 1, 5, 100, 1) ·		**irecv**(ANY_SOURCE, 1, 100, 1) ·
irecv(1, 0, ANY_TAG, 2) ·	‖	**waitrecv**(0, 1, 5, 100, 1) ·
waitsend(1) ·		**isend**(1, 0, 6, 200, 2) ·
waitrecv(1, 0, 6, 200, 2)		**waitsend**(2)

Figure 5.1: Example of two parallel processes (0 and 1)

- **waitsend**(r): Wait for non-blocking send identified by r to finish.
- **irecv**($source, dest, tag, r$): Non-blocking receive action. *source* and *tag* can be wildcard (any source and/or any tag).
- **waitrecv**($source, dest, msg, tag, r$): Wait for non-blocking receive to finish. *source* and *tag* are actual values here (no wildcards).

The parameters of the action are defined as follows:

- *source*: MPI identifier of the sender.
- *dest*: MPI identifier of the receiver.
- *msg*: Message value sent/received.
- *tag*: Tag to distinguish between messages sent to the same receiver.
- r: Result handle to link **isend** and **waitsend** or **irecv** and **waitrecv** actions.

In these actions a single message value is assumed. The MPI library can send an array of values in each message, but this was not needed for our program. As future work, the actions could be modified to use an array of message values.

Although there is only one `wait` command in MPI, it is split in a **waitsend** and a **waitrecv** action because of semantic differences. The **waitsend** only waits for the send operation to finish, but the **waitrecv** action also receives values which are not known at the time of the **irecv** action: the actual source and tag and the message value.

Example The example in figure 5.1 contains two processes in parallel to show the usage of actions. The left process, numbered 0, starts a send operation from 0 (itself) to process 1 with message 5, tag 100 and request handle 1. It then starts a receive operation from process 1 to 0 (itself) with any tag and request handle 2. Then it waits till the send is finished (with handle 1)

and finally till the receive is finished. The receive operation has actual source 1, destination 0, message value 2, actual tag 200 and request handle 2.

The second process first starts a receive operation from any source to process 1 (itself) with tag 100 and request handle 1. It then waits till the receive is finished, this has actual source process 0 and the destination, actual tag, and request handle are 1, 100 and 1. Next it starts a send operation from process 1 (itself) to process 0 with message 6, tag 200 and request handle 2. Finally it will wait till the send operation with handle 2 is finished.

As you can see, the send in the left process matches with the receive in the right process and vice versa.

5.2 Processes

The actions defined in the previous section can be combined to processes. These processes can be used to specify the composed MPI operations **send** and **recv** from section 4.1 and to specify the encoding of active objects as discussed in section 4.2. The processes used in the verification are discussed below.

Blocking send The **isend** and **waitsend** actions can be combined to the *send* process, a blocking send process:

$$\begin{array}{l} send(source, dest, msg, tag) \\ = \\ \sum_r \quad \mathbf{isend}(source, dest, msg, tag, r) \cdot \\ \qquad \mathbf{waitsend}(r) \end{array}$$

The blocking *send* process is a non-blocking **isend** action directly followed by a **waitsend** action to wait till the send operation is completed.

The sum $\sum_r$ indicates that the parameter r is not defined up front, but is determined by the first action in the sum. This is sometimes called an *output parameter*. It can be used for a generated identifier such as r, but it can also be used for received values such as msg. In this case it indicates the value r is needed by the actions, but not relevant for the general *send* process.

Blocking receive The **irecv** and **waitrecv** actions can be combined to the *recv* process, a blocking receive process:

$$
\begin{array}{l}
recv(req\_source, act\_source, dest, msg, req\_tag, act\_tag) \\
= \\
\sum_r \quad \mathbf{irecv}(req\_source, dest, req\_tag, r) \cdot \\
\qquad \mathbf{waitrecv}(act\_source, dest, msg, act\_tag, r)
\end{array}
$$

The *req_source* and *req_tag* are the requested source and tag and can contain the wild cards `ANY_SOURCE` and `ANY_TAG`. The *act_source* and *act_tag* contain the actual values. When the requested source is the wild card, the actual source can be any valid source. When a specific source is requested, the requested source and the actual source must be equals. The same holds for the requested and actual tag. In a formula this would be:

$$req\_source = \texttt{ANY\_SOURCE} \vee req\_source = act\_source$$

or

$$req\_source \neq \texttt{ANY\_SOURCE} \Rightarrow req\_source = act\_source$$

And similar for the tag:

$$req\_tag = \texttt{ANY\_TAG} \vee req\_tag = act\_tag$$

or

$$req\_tag \neq \texttt{ANY\_TAG} \Rightarrow req\_tag = act\_tag$$

Method call A typical client would execute the following process when calling an active object method:

$$
\begin{array}{l}
callMethod(i, j, t, m, p, res) \\
= \\
\sum_c \quad send(i, j, < c, m, p >, t) \cdot \\
\qquad \sum_r \quad \mathbf{irecv}(j, i, c, r) \cdot \\
\qquad\qquad \mathbf{waitrecv}(j, i, res, c, r)
\end{array}
$$

with:

- i: MPI identifier of the caller
- j: MPI identifier of the callee
- t: Active object identifier
- c: The unique call identifier generated by the caller
- m: The method name

- p: Call parameters
- $< c, m, p >$: The `Call` object
- r: Result handle returned by the `irecv` call
- res: Result of the active object method call

Before *send* the precondition of the active object method should hold. After the *waitrecv* finishes, the postcondition holds.

Active object process A typical active object process would be:

$$
\begin{aligned}
&activeObject(j,t)\\
&=\\
&\textstyle\sum_{i,<c,m,p>} \quad \mathbf{recv}(i,j,<c,m,p>,t) \cdot\\
&\qquad\qquad \textstyle\sum_{res} \quad excecuteMethod(p,res) \cdot\\
&\qquad\qquad\qquad\quad \mathbf{send}(j,i,res,c)\\
&\cdot\ activeObject(j,t)
\end{aligned}
$$

$executeMethod$ is a dummy process representing the execution of the method. Before $executeMethod$ the precondition of the method holds, after $executeMethod$ the postcondition should hold. The method could include other active object method calls, so the $executeMethod$ process can include **send**, **irecv** and **waitrecv** actions.

After receiving a method call, executing the method and sending the result back, the process repeats itself. This variant doesn't allow parallel execution of methods of the same active object.

Composition The processes of the clients and the active objects are combined with the MPI system process defined by Wytse using parallel composition.

5.3 Annotation of the MPJ library

To use the existing MPJ Express library in the verification, it must be annotated first. This means adding pre- and postconditions to all methods used in our implementation. To do so dummy classes have been created.

Most method specifications use the ghost parameters `Future fut`, `frac p` and `process P`. These are required for reasoning with futures. `fut` contains a reference to the future specification object, the fraction `p` is the amount of permissions on the future and the process `P` is the process remaining after

Listing 4: Specification of the `Isend` method of the MPJ `Intracomm` class

```
/*@
    given Future fut;
    given frac p;
    given process P;
    given frac bufp;
    requires fut != null ** p != none;
    requires buf.length == 1 ** offset == 0 ** count == 1;
    requires Perm(buf[0], bufp);
    requires valid(Rank(), dest, buf[0], tag);
    requires Future(fut, p,
        \sum(int r; ; fut.isend(Rank(), dest, buf[0], tag, r) * P(r)));
    ensures Value(\result.recv) ** \result.recv == false **
        Value(\result.bufp) ** \result.bufp == bufp **
        Value(\result.buf) ** \result.buf = buf;
    ensures \result.waitToken();
    ensures Future(fut, p, P);
@*/
public Request Isend(Object buf, int offset, int count, Datatype datatype,
    int dest, int tag) throws MPIException {
}
```

executing the current method. `P` is sometimes parametrized with an output parameter of one of the actions.

Related to this are the pre-conditions (`requires`) `fut != null ** p != none` and `Future(fut, p, someProcess * P)` where *someProcess* is the process executed by the method. The post-condition (`ensures`) `Future(fut, p, P)` indicates that this process is executed.

As mentioned earlier we assume single value messages, therefore buffers must contain exactly one value: `buf.length == 1 ** offset == 0 ** count == 1`.

`Isend` method The specification of the `Isend` method (listing 4) contains an extra ghost parameter `frac bufp`. This indicates the amount of permission on the buffer is given in the pre-condition `Perm(buf[0], bufp)`. This permission is needed to read the message value from the buffer. The pre-condition `valid(Rank(), dest, buf[0], tag)` indicates that the combination of the source, destination, message value and the tag must be a valid message. More about the valid predicate in section 5.4.

The result of this method is a `Request` object. The post-condition grants read permissions to specification fields in this object and ensures that the contents are what you expect. It also grants the `waitToken` indicating `Wait`

Listing 5: Specification of the `Irecv` method of the MPJ `Intracomm` class

```
/*@
    given Future fut;
    given frac p;
    given process P(r);
    requires fut != null ** p != none;
    requires buf.length == 1 ** offset == 0 ** count == 1;
    requires Perm(buf[0], write);
    requires Future(fut, p,
        \sum(int r; ; fut.irecv(source, Rank(), tag, r) * P(r)));
    ensures Value(\result.recv) ** \result.recv == true **
        Value(\result.source) ** \result.source == source **
        Value(\result.dest) ** \result.dest == Rank() **
        Value(\result.tag) ** \result.tag == tag **
        Value(\result.r) ** \result.r == r **
        Value(\result.bufp) ** \result.bufp == write **
        Value(\result.buf) ** \result.buf = buf;
    ensures \result.waitToken();
    ensures Future(fut, p, P(r));
@*/
public Request Irecv(Object buf, int offset, int count, Datatype datatype, int
  source, int tag) throws MPIException {
}
```

can be called on the result object. The permission on the buffer won't be given back at the end of this method since the message value might not be read from the buffer yet.

The process executed by this method is a **isend** action with the given parameters.

`Irecv` **method** The `Irecv` method (listing 5) requires full permission on the buffer (`Perm(buf[0], write)`) as the incoming message value will be written to it. The result of this method is also a `Request` object and the post-conditions are similar to the `Isend` method. The process of this method is the **irecv** action with an arbitrary `r` value.

`Wait` **method** `Wait` (listing 6) is a method of the `Request` object returned by `Isend` or `Irecv`. The pre- and post-conditions of this method depend on whether the `Request` object is from a send or receive operation, indicated by the specification variable `recv`. In both cases the method requires read permissions on the `recv` and `r` variables and the `waitToken` to ensure this method is only called once. This method will also return the permissions on

Listing 6: Specification of the `Wait` method of the MPJ `Request` class

```
    /*@
2       given Future fut;
        given frac p;
4       given process P(msg);
        requires fut != null ** p != none;
6       requires Value(this.recv) ** Value(this.r);
        requires this.waitToken();
8       requires !recv ==> Future(fut, p, waitsend(this.r) * P(null));
        requires recv ==> Value(this.source) ** Value(this.dest) **
10          Value(this.tag);
        requires recv ==> Future(fut, p,
12          \sum(int act_source, int act_tag, Object msg;
                this.source != ANY_SOURCE ==> act_source == this.source **
14              this.tag != ANY_TAG ==> act_tag == this.tag;
                waitrecv(act_source, this.dest, msg, act_tag, this.r) *
16              P(msg)));
        ensures Value(bufp) ** Value(buf) ** Perm(buf[0], bufp);
18      ensures recv ==> Value(\result.source) ** \result.source == act_source;
        ensures recv ==> Value(\result.tag) ** \result.tag == act_tag;
20      ensures recv ==> buf[0] == msg;
        ensures recv ==> valid(act_source, dest, buf[0], act_tag);
22      ensures Future(fut, p, P(msg));
    @*/
24  public Status Wait() {
    }
```

Listing 7: Specification of the `Send` method of the MPJ `Intracomm` class

```
    /*@
2       given Future fut;
        given frac p;
4       given process P;
        given frac bufp;
6       requires fut != null ** p != none;
        requires buf.length == 1 ** offset == 0 ** count == 1;
8       requires Perm(buf[0], bufp);
        requires valid(Rank(), dest, buf[0], tag);
10      requires Future(fut, p, fut.send(Rank(), dest, buf[0], tag) * P);
        ensures Perm(buf[0], bufp);
12      ensures Future(fut, p, P);
    @*/
14  public void Send(Object buf, int offset, int count, Datatype datatype,
        int dest, int tag) throws MPIException {
16  }
```

the buffer required by the `Isend` or `Irecv` method. When the request is from `Isend`, the process is the **waitsend** action with parameter `r`.

When the request if from `Irecv`, also read permission on the `source`, `dest` and `tag` fields is required. The return value of this method is a `Status` object containing the actual source and actual tag as specified in the post-condition. The method ensures that the buffer contains the received message and that the combination of source, destination, message and tag is valid, as required by the `Send` method. The process executed by this method is $\sum_{act\_source,act\_tag,msg}$ **waitrecv**$(act\_source, dest, msg, act\_tag, r)$, i.e. the **waitrecv** action with input parameters *dest* and r and output parameters *act_source*, *act_tag* and *msg*. When the *source* parameter of the `irecv` method is not `ANY_SOURCE`, *act_source* is equal to *source*, the same holds for *tag* and *act_tag*.

`Send` method The `Send` method (listing 7) is the combination of the `Isend` and `Wait` method. The pre-conditions are therefore similar to the `Isend` method, and the post-conditions to the `Wait` method for send requests. The process is equal to the **isend** action followed by the **waitsend** action.

`Recv` method Similarly, the `Recv` method (listing 8) is the combination of the `Irecv` and `Wait` method. The pre-conditions are therefore similar to the `Irecv` method, and the post-conditions to the `Wait` method. The process is equal to the **irecv** action followed by the **waitrecv** action.

Listing 8: Specification of the `Recv` method of the MPJ `Intracomm` class

```
/*@
    given Future fut;
    given frac p;
    given process P(msg);
    requires fut != null ** p != none;
    requires buf.length == 1 ** offset == 0 ** count == 1;
    requires Perm(buf[0], write);
    requires Future(fut, p,
        \sum(int act_source, int act_tag, Object msg;
            this.source != ANY_SOURCE ==> act_source == this.source **
            this.tag != ANY_TAG ==> act_tag == this.tag;
            fut.recv(source, act_source, Rank(), msg, tag, act_tag, r) *
            P(msg)));
    ensures Value(\result.source) ** \result.source == act_source;
    ensures Value(\result.tag) ** \result.tag == act_tag;
    ensures Perm(buf[0], write);
    ensures buf[0] == msg;
    ensures valid(act_source, Rank(), buf[0], act_tag);
    ensures Future(fut, p, P(msg));
@*/
public Status Recv(Object buf, int offset, int count, Datatype datatype,
        int source, int tag) throws MPIException {
}
```

5.4 valid predicate

The $valid(source, dest, msg, tag)$ predicate is a predicate that specifies which messages (a combination of a source, destination, message value and tag) are valid messages in the current MPI program. What a valid message is and therefore the definition of the of the valid predicate depends on the MPI program.

In our active object implementation a message is valid if it is either a valid call message or a valid response message.

$$
\begin{aligned}
&valid(source, dest, msg, tag) = \\
&\quad (\ isCallMessage(dest, tag) \lor isResponseMessage(dest, tag)\ ) \land \\
&\quad (\ isCallMessage(dest, tag) \Rightarrow \\
&\qquad validCallMessage(source, dest, msg, tag)\ ) \land \\
&\quad (\ isResponseMessage(dest, tag) \Rightarrow \\
&\qquad validResponseMessage(source, dest, msg, tag)\ )
\end{aligned}
$$

Intuitively: a valid message is either a call or response, a call must be a valid call and a response must be a valid response.

Call message A call message is valid when the message value is a `Call` object with the correct destination and tag/object id, it represents a valid active object method call and a message back to the source with the callId as tag would be a response.

$$
\begin{aligned}
&validCallMessage(source, dest, msg, tag) = \\
&\quad msg \text{ instanceOf } \texttt{Call} * dest = msg.dest * tag = msg.objectId * \\
&\quad validCall(msg.dest, msg.objectId, msg.signature, msg.params) * \\
&\quad isResponseMessage(source, msg.CallId)
\end{aligned}
$$

An active object method call is valid when the combination of destination and object id refers to an existing active object, the signature is a valid method signature for the active object class, the number and types of the parameters are valid for the method, and the pre-condition of the method is met. This full definition of this predicate depends on the existing active object classes in the program.

Response message A response message is valid if the post-condition of the method call identified with the destination (caller) and the tag (call id) is met and the message value (return value) is of the expected type. This definition

of the validity of the response message requires some extra administration, since the post-condition can refer back to the parameters used in the method call. Therefore we wrapped the result value in an instace of the new `Result` class and added a reference to the related `Call` object as specification-only attribute to this new object. This allows us to refer to the method parameters in the *validResponseMessage* predicate.

$$validResponseMessage(source, dest, msg, tag) = \\ msg \text{ instanceOf } \texttt{Response} * source = msg.call.dest * tag = msg.call.callId * \\ postCondition(msg.call.dest, msg.call.objectId, msg.call.signature, \\ msg.call.params, msg.result)$$

Note that the destination of the original call ($msg.call.dest$) is the source of the response message and that the tag is now the $callId$ received from the caller. *postCondition* is the predicate that specifies that the post-condition for the called method is valid. This full definition of this post-condition predicate depends on the existing active object classes in the program.

5.5 Annotation of implementation methods

In this section we will describe the annotations of the methods we created and implemented ourself, especially the helper methods in the `MPAO` classed used by all active objects and all callers. The annotations of the methods we implemented follows generally the same structure as the annotated MPI methods discussed in section 5.3. We will describe the methods roughly in the order they are called: calling an active object, receiving an active object call, sending the result value and receiving the result value.

Notable will be a lot of administrative overhead: caring around a lot of read-only permissions (using the keyword `Value`) and variables containing information about sending and receiving messages. There are three main causes for this.

The first problem is that read permissions on field declared `final` (read-only) are not automatically derived while this should be possible.

The second problem is the limitation of the Java programming language to only allow returning a single value in a method. When multiple values need to be returned a wrapper class must be created introducing more annotations.

Finally when using futures in specifications requires information to be present at more points, which requires more specification-only variables and related annotations. It also prevents the hiding of some variables in predicates because the variable values are needed in action parameters in the futures.

Listing 9: Specification of the `callMethod` method of the `MPAO` class

```
    /*@
2       given Future fut;
        given frac p;
4       given process P(r);
        requires fut != null ** p != none;
6       requires validCall(workerId, objectId, signature, parameters);
        requires Future(fut, p,
8           \sum(int callId, int r; ;
                fut.send(Rank(), workerId, new Call(callId, signature,
10                                                  parameters), objectId) *
                        fut.irecv(workerId, Rank(), callId, r) * P(r)) *
12          P);
        ensures \result.getToken();
14      ensures Value(\result.request);
        ensures Value(\result.request.source) ** Value(\result.request.dest) **
16          Value(\result.request.tag) ** Value(\result.request.r) **
            Value(\result.request.buf);
18      ensures \result.request.source == workerId **
                \result.request.dest == Rank() **
20              \result.request.tag == callId **
                \result.request.r == r **
22              \result.request.bufp == write **
                \result.request.buf == \result.buffer;
24      ensures Future(fut, p, P(r));
    @*/
26  public static MPAOResult callMethod(int workerId, int objectId,
            String signature, Serializable [] parameters) {
28  }
```

Listing 10: Specification of the `receiveCall` method of the `MPAO` class

```
    /*@
2       given Future fut;
        given frac p;
4       given process P(msg);
        requires fut != null ** p != none;
6       requires Future(fut, p,
            \sum(int source, Call msg; ;
8           fut.recv(ANY_SOURCE, source, Rank(), msg, objectId, objectId, r) *
            P(msg)));
10      ensures Value(\result.call) ** \result.call == msg;
        ensures Value(\result.source) ** \result.source == source;
12      ensures \result.valid();
        ensures Future(fut, p, P(msg));
14  @*/
    public static ReceivedCall receiveCall(int objectId) {
16  }
```

This method requires access to the internal `Request` object and all of its fields

`callMethod` The `callMethod` method (listing 9) is used when an active object method is called. It forwards the method call to the correct MPI node followed by the start of the receiving process for the result value. These to steps are also shown in the future process, first a send action followed by an asynchronous receive (irecv) action. The precise call identifier r is not relevant, any unique identifier will do, hence the use of the **sum**. This method also requires the *validCall* predicate discussed in section 5.4 indicating, among other things, that the pre-condition is met.

The post-condition ensures that all the information in the `MPAOResult` result object is readable and correct and the `getToken` allows the `get` method of the `MPAOResult` result object to be called.

`receiveCall` The `receiveCall` method (listing 10) is called on the node containing the active object to receive method calls. The future specifies the **recv** action to receive a method call message send by the `callMethod` method. The source is not specified beforehand, instead the `ANY_SOURCE` constant is used as requested source parameter. The *tag* must be *objectId*, so this value is used for both the requested tag and actual tag in the action. There are no specific pre-conditions for this method.

The post-condition ensures the received call and the actual source are returned in the `ReceivedCall` object and that this object is valid, i.e. the

Listing 11: Specification of the `sendResult` method of the `MPAO` class

```
/*@
    given Future fut;
    given frac p;
    given process P;
    requires fut != null ** p != none;
    requires postCondition(Rank(), call.objectId, call.signature,
        call.parameters, result);
    requires Future(fut, p, fut.send(Rank(), call.source,
        new Response(result), tag) * P);
    ensures Future(fut, p, P);
@*/
public static void sendResult(int source, Call call,
        Serializable result) {
}
```

valid predicate form section 5.4 holds.

`sendResult` The `sendResult` method (listing 11) is used for sending the result of an active object method call back to the caller. The obvious pre-condition of this method is that the post-condition of the active object method call must hold. The future indicates that de result will be sent to the source of the call.

`MPAOResult.get` The pre-condition of the `MPAOResult.get` method (listing 12) is quite large because a lot of permissions are needed for the future **waitrecv** action. The pre-condition also requires the getToken to ensure it is only called once. The important post-condition of this method is the *valid* predicate, of which the post-condition of the active object method call can be derived. Because this method can only be called once, the `RemoteFuture` object is used to wrap the `MPAOResult` object to allow it to be used from multiple threads.

`RemoteFuture` **object** The `RemoteFuture` object implements the Java `Future` interface as discussed in chapter 3. It is based on the `ForkJoinTask` from listing 2. This class calls the `MPAOResult.get` method the first time its own `get` is called. We didn't succeed to get the annotation of this class correct, because the amount of extra administrative information that was needed to correctly call the `MPAOResult.get` with a future and to correctly derive the post-condition from the *valid* predicate. These information must be stored as specification-only fields and maybe included in the already complicated

Listing 12: Specification of the `get` method of the `MPAOResult` class

```
/*@
    given Future fut;
    given frac p;
    given process P(msg);
    requires fut != null ** p != none;
    requires Value(this.request);
    requires Value(this.request.source) ** Value(this.request.dest) **
        Value(this.request.tag) ** Value(this.request.r) **
        Value(this.request.buf);
    requires getToken();
    requires Future(fut, p,
            \sum(Object msg;; waitrecv(this.request.source, this.request.dest,
            this.request.msg, this.request.tag, this.request.r) * P(msg)));
    ensures Value(this.request.source) ** Value(this.request.dest) **
        Value(this.request.tag);
    ensures \result == msg;
    ensures valid(this.request.source, this.request.dest, \result,
        this.request.tag);
    ensures Future(fut, p, P(msg));
@*/
public Serializable get() {
}
```

csl_invariant as seen in listing 2.

5.6 Active objects

In a similar fashion also the object containing the actual callers, active objects and proxy classes can be annotated. These annotation are similar to the annotations in the previous section but parameters such as method signatures can be filled in and predicates such as *postCondition* can be made specific.

5.7 Verification

Automatic verification of the created annotations was not possible because we used some features which are not (yet) implemented in the tool. These are discussed in the next section. Although we tried hard to get the specifications correct, we did not verify them by hand because of time constraints.

5.8 Tool support

It was not possible to use the VerCors tool set with these specification. This is because it has no mechanism yet to specify future action parameters with unknown values. For example when receiving messages, the message value is not known beforehand, but it currently must be specified in the future. In our specifications we used the sum notation to solve this. Another possibility is to use explicit OUT (output) parameters in actions. Both options are not yet supported in VerCors.

Chapter 6

Conclusion

The main question of this research was

To what extent can an active object implementation with MPI be verified?

To a large extent an active object implementation can be specified using the current techniques although some specific parts in the specification language are not yet well defined. The specification is quite large compared to the actual source code and it is not trivial to write.

Automatically verifying the implementation is not yet possible since some of the required features are currently missing. Manual verification is possible but probably not feasible since the specification is quite large. When the necessary techniques are developed and implemented in the tool, automatic verification should be possible in the future.

How can active objects be described and verified using permission-based separation logic? It is possible to describe active objects using permission-based separation logic. We used a similar to describing threads with *preFork* and *postJoin* predicates. The Java interface `Future` is used as base for returning result values.

To test our specification we created a sample program and verified it using the VerCors tool set. There are some limitations with the tool. For example, a `Future` subclass must be crated (or generated) for each active object method.

How can active objects be implemented using MPI? Active objects can easily be implemented by sending an MPI message when an active object method is called and sending an MPI message back with the return value.

This can be done in Java using the MPJ Express library. Because Java does not support dynamic method creation, for each active object class a proxy class must be created which sends the active object method call to the actual active object. These proxy objects can probably be generated automatically. This implementation also uses the Java `Future` interface to return result values.

Can the active objects implementation in MPI be verified? To verify an implementation it must first be annotated. We annotated the the required methods of the MPJ Express library and most parts of our active objects implementation. However we could not annotate all parts.

Annotating the MPJ Express library was not straightforward. The MPJ Express library consists of several classes which functionality depend on each other. This required ghost variables and corresponding permissions to simulate the inner workings. This resulted in a lot of administrative work in the specification of the library and in the methods using the library.

Another difficulty was the specification of the future processes. A future process must be specified at the start of the program. However not all required values are known in advance since they are generated by the MPJ Express library or received from other MPI nodes. This can be solved by using the sum notation in the process term or using OUT parameters. However these theories are not mature yet.

Also because behaviour must be specified up front sometimes extra ghost parameters, ghost variables or permissions are needed which weren't needed for verification using histories. Future processes can also be counter-intuitive because you need to reason in the opposite direction of the program flow.

In one class these problems combined making it impossible to annotate the class without redesigning it or adding a lost of extra annotations and ghost variables.

Also automatic verification was not possible because specification techniques were used which are not yet supported by the tool set.

The current specification techniques seem to be suitable for programs and data structures which are simple or repetitive in a predictable manner, for example mathematical or algorithmic problems. These problems can often be described in a single mathematical property, predicate or process term.

When the program or the structures become more complex the specifications grow fast. A solution can be dividing a program or structure in smaller parts. However this creates its own problems. For example, all implicit knowledge and permissions must be specified explicitly in pre- and postconditions when splitting a method into multiple methods.

Can an example be verified using the developed techniques? As it was already difficult to specify the active objects implementation, we didn't specify and verify an example program. For programs many of the same problems exists as mentioned earlier. In our judgement specifying and verifying a small and simple active object program should be doable. However verifying larger, realistic and complex programs will require large efforts for making a complete specification.

Tool support During this project we encountered some problems and limitations in the verification tool VerCors. Some of the bugs were fixed or worked around. It is not possible to use the sum notation or OUT parameters in future processes which are required in our specification. Also other minor features were missing, such as class parameters, generics, static variables and methods and packages. Because of this we could not validate our full implementation using the tool. However the specification of active objects developed in chapter 3 can be used and validated using the tool.

6.1 Future work

In this project we concluded that there is no clear way for including unknown data in future processes. A technique and syntax for specifying input or other external data in futures and referring to that data in the rest of the specification can greatly improve the usability of future processes. The sum notation or OUT parameters should be developed further.

Currently there are only future processes for example programs. Defining a future process for larger programs can be a complex task. Developing a strategy or best practice for defining future process can help for specifying realistic programs.

In some cases like the Java `Future` interface and its implementations the specification of some methods, like assertions about return values, can greatly depend on the use cases. It would be nice to develop a way to have parametrized specifications, for example using class parameters.

Currently it is a lot of work to specify all permissions for variables used in pre- and postconditions. However there is no need to check for permissions on variables `final` since their value cannot be changed. Automatically assuming read permissions on `final` variables can reduce the specification overhead when a lot of immutable variables are used.

VerCors Most of the features mentioned above do not have support in VerCors yet, such as the sum notation or for future processes. There are also

some tool specific recommendations:

During this project there was no support for Java packages in VerCors. This limited the possibilities to add structure to a project. Also existing programs use packages to structure their program and resolve naming conflicts between classes. Package support allows larger project and existing code to be verified while maintaining structure.

Knowledge of the inner workings of the tool is sometimes needed to solve problems in the specification. Better documentation and more descriptive error messages can shorten the time needed to find and solve errors in the specification of a program.

The documentation on VerCors is short and for some parts non-existing. Documentation containing all the features, assumptions and magic names of the tool can greatly improve the learning time for new users of the tool. It also helps finding the right techniques for specifying particular constructs in the source code.

Bibliography

[1] Afshin Amighi, Stefan Blom, and Marieke Huisman. VerCors: A layered approach to practical verification of concurrent software. In 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pages 495–503, Feb 2016.

[2] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The VerCors project: Setting up basecamp. In Proceedings of the sixth workshop on Programming languages meets program verification, pages 71–82. ACM, 2012.

[3] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In First UK Workshop on Java for High Performance Network Computing, Europar, volume 98, 1998.

[4] Geoff Barrett. Model checking in practice: The T9000 virtual channel processor. Software Engineering, IEEE Transactions on, 21(2):69–78, 1995.

[5] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. 2015.

[6] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In ACM SIGPLAN Notices, volume 40, pages 259–270. ACM, 2005.

[7] Markus Bornemann, Rob V Van Nieuwpoort, and Thilo Kielmann. Mpj/ibis: a flexible and efficient message passing platform for java. In European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pages 217–224. Springer, 2005.

[8] Bryan Carpenter, Geoffrey C Fox, Sung-Hoon Ko, and Sang Lim. mpijava 1.2: Api specification. 1999.

[9] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey C Fox. MPJ: MPI-like message passing for Java. 2000.

[10] Jack Dongarra, D Walker, E Lusk, B Knighten, M Snir, A Geist, S Otto, R Hempel, E Lusk, W Gropp, et al. Special issue on MPI: a message-passing interface standard. International Journal of Supercomputer Applications and High Performance Computing, 8(3-4), 1994.

[11] Charles Antony Richard Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, 1969.

[12] Charles Antony Richard Hoare. Communicating sequential processes. Communications of the ACM, 21(8):666–677, 1978.

[13] Marieke Huisman, Wolfgang Ahrendt, Daniel Bruns, and Martin Hentschel. Formal specification with jml. 2014.

[14] Cliff B. Jones. Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 5(4):596–619, 1983.

[15] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed object group metacomputing architecture. Concurrency Practice and Experience, 10(11-13):977–983, 1998.

[16] R Greg Lavender and Douglas C Schmidt. Active object–an object behavioral pattern for concurrent programming. 1995.

[17] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes, 31(3):1–38, 2006.

[18] Jinjiang Lei and Zongyan Qiu. Modular reasoning for message-passing programs. In Theoretical Aspects of Computing–ICTAC 2014, pages 277–294. Springer, 2014.

[19] Jinjiang Lei and Zongyan Qiu. Modular reasoning for message-passing programs. Technical report, School of Mathematical Sciences, Peking University, June 2014.

[20] Jinjiang Lei and Zongyan Qiu. Rely-guarantee based reasoning for message-passing programs. Scientific Annals of Computer Science, 24(2):217, 2014.

[21] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In Tools and Algorithms for the Construction and Analysis of Systems, pages 147–166. Springer, 1996.

[22] Message Passing Interface Forum. MPI: A message-passing interface standard, version 3.1. `http://mpi-forum.org/docs/mpi-3.1/`, June 2015.

[23] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. In European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, pages 135–142. Springer, 1997.

[24] W. Oortwijn, S. Blom, D. Gurov, M. Huisman, and M. Zaharieva-Stojanovski. Future-based reasoning: Predicting concurrent program behaviour, 2017. Submitted.

[25] W. Oortwijn, S. Blom, and M. Huisman. Future-based static analysis of message passing programs. In PLACES, pages 65–72, 2016.

[26] Oracle and its affiliates. Java platform, standard edition 7, API specification. `http://docs.oracle.com/javase/7/docs/api/`.

[27] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In ACM SIGPLAN Notices, volume 40, pages 247–258. ACM, 2005.

[28] John C Reynolds. Separation logic: A logic for shared mutable data structures. In Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on, pages 55–74. IEEE, 2002.

[29] Bill Roscoe et al. The theory and practice of concurrency. 1998.

[30] Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Touriño, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. Science of Computer Programming, 78(5):425–444, 2013.

[31] Marina Zaharieva-Stojanovski. Closer to reliable software: Verifying functional behaviour of concurrent programs. 2015.

[32] Marina Zaharieva-Stojanovski, Stefan Blom, Dilian Gurov, and Marieke Huisman. Reasoning about concurrent programs with guarded blocks. Submitted.

[33] Marina Zaharieva-Stojanovski, Marieke Huisman, and Stefan Blom. A history of BlockingQueues. arXiv preprint arXiv:1209.2239, 2012.