MASTER

Proving correctness of threaded parallel executable code generated from models described by a domain specific language

Roede, S.

Award date:
2012

Link to publication
Proving correctness of threaded parallel executable code generated from models described by a Domain Specific Language

By
S. Roede

Supervisor & tutor

Dr. R. Kuiper (TU/e)
Ir. J.M.A.M Gabriels (TU/e)

Eindhoven, July 2012
Proving correctness of threaded parallel executable code generated from models described by a Domain Specific Language

Author: ing. Sybren ROEDE
s.roede@student.tue.nl

Graduation supervisor: dr. Ruurd KUIPER

Graduation tutor: ir. Joost M.A.M GABRIELS

July 7, 2012
Preface

This thesis was written in partial fulfilment of the requirements for graduation in a master degree in computing science at the Eindhoven University of Technology. This graduation project was done in the area of Software Technology for the research group Software Engineering Technology.

The author would like to give a special thanks to my graduation supervisor Ruurd and my graduation tutor Joost. They provided an excellent job in supervising the research work and correcting the thesis. During the process they made useful critical remarks and give valuable suggestions to strengthen the work. They made my graduation project a joyful process and learningful experience.

I want to thanks K. Rustan M. Leino of the Research in Software Engineering (RiSE) group at Microsoft Research. In an early stage of my graduation project he give me the change to use his expertise in the field of formal verification.

My deep thanks go to Bart Jacobs of the Department of Computer Science of the Katholieke Universiteit Leuven, Belgium. The time he spend on me and the opportunity to learn from him is greatly appreciated. The verification in VeriFast could not been done without his guidance. The two afternoons in Leuven where very helpful and a great experience.

The strong support and love of my friends and family is a blessing during my study. I am especially grateful to my family and lovely girlfriend who were tireless in motivating and supporting me to finish my master degree.
Abstract

We investigate correctness of threaded parallel executable code generated from models described by a Domain Specific Language (DSL). There are challenges in developing correct multi-threaded code. Model Driven Engineering (MDE) is a promising approach: the modeling can be done at a high abstraction level and with a chain of model transformations the implementation code is generated. During my Capita Selecta, I made a realization of the code generation step of a particular MDE-chain, the transformation of state-machine like SLCO models into multi-threaded C# code. In the generation process, use of and inheritance from multi-threaded generic code was applied. The goal of the present research is to investigate formal specification and verification of the generic code. This was done by first making a small overview of available tools and selection of the best suited tool for the verification. After that, we performed a parameterized verification with VeriFast on an essential, non-trivial part of the framework. Finally we reflect on the lessons learned.
1 Introduction

A phenomenon in software engineering is that programs get bigger and more complex every year. In the last couple of years the emergence of concurrent programming received more attention. Programming those complex systems with concurrency is difficult: threading and synchronization can lead to a range of errors, e.g., race conditions and deadlocks.

There are techniques available to help cope with these defects. Some techniques check programs after the code has been made, e.g. static analyzers (without running), dynamic analyzers (during running) and model checkers (on a mathematical representation of the program). With other techniques such as Model-driven Engineering (MDE), it is possible to tackle those defects before any code is made. By investigating models of the software and generating code from those models, it is possible to detect defects earlier in the software development trajectory. In most cases this means that less effort is required to fix the defects.

With MDE, models are not used just for documentation and communication purposes, but are first-class artifacts. Developing programs depends on constructing models that capture the intended behavior and properties, and eventually generating the source code from those models. Constructing a model enables the developer to deal with difficult aspects at a less complex and higher level of abstraction. Transformations are used to create new models, source code, test scripts and other artifacts. Models, possible generated with model transformation from other models, can be used to check whether the executed behavior is compliant with the specified behavior. Also other properties like deadlock freedom and the absence of race conditions can be checked.

One question that remains is, whether the transformations are correctness preserving. In other words: If we have proven that certain properties hold in the model, how can we be sure those properties still hold in the generated source code?

In a Capita Selecta project at the TU/e, the author provided a concrete instantiation of an MDE-style transformation chain; transformations were made to generate source code out of a model. There we gave an informal argument to the correctness problem. In this M.Sc. project, the goal is to apply formal verification to answer the correctness problem.

1.1 Capita Selecta project

As part of a Capita Selecta project [1] at the TU/e in 2011, the author implemented a chain of model transformations to transform a model in SLCO to threaded C# code. SLCO (Simple Language of Communicating Objects) [2] is a Domain Specific Language (DSL) based on state machines. SLCO provides constructs for specifying systems consisting of state machines that operate concurrently and communicate with each other. In SLCO, the structure of a system is modeled using special structure artifacts. The behavior is modeled by state machines.

After transformation, the resulting C# code consists of model independent (generic) code, and model dependent (specific) code. The generic code contains the general concepts such as state machines, transitions, and channels. The specific code contains the actual behavior of these concepts for a specific case such that when the specific code inherits from the generic code, it becomes a fully functional program. A framework called TOOSM (Threaded Object-Oriented State Machines) created in this project contains the generic code. The MDE transformations generate the model specific code. In this thesis, we refer to the whole solution (framework and model transformations) as TOOSM and only differentiate when necessary.

In the Capita Selecta project, only an informal argument was given for the correctness of the code resulting from the transformations. Although a detailed comparison of the model state space and execution logs of the resulting code provided a reasonably convincing argument for correctness, a
formal proof of the correctness is what is really required. This is what the present M.Sc. project is aiming to provide.

1.2 Goals and approach

Because of the separation of generic and specific code, it is not enough to prove only the correctness of the model transformations since we would miss significant details about the generic code. The separation does however also have a clear maintenance advantage. The concepts and general workings of a state machine are located in one place and do not need to be generated for every instance. Because the same code is inherited from for every transformation, we can use verification to verify the correctness of the generic code just once. This strategy is significantly different from more common strategies used today in MDE that verify a specific instance or only the model transformations, whereas we attempt to verify all instances (within certain bounds) based on generic code.

While investigating the correctness of the generic code, the focus will be on obtaining an automatically checked formal proof. Therefore, a suitable tool has to be found.

We formulate three main research questions:

A What is the current state of art in tool-based verification of concurrent, object-oriented code?
   • Which tools are available for verification of concurrent, object-oriented code?
   • What are the techniques used by the tools?
   • What restrictions do these tools have?
   • Which tool is best suitable for the verification of TOOSM?

B Can one of the tools verify the correctness of concurrent object-oriented code, especially the generic code of TOOSM?
   • To what extend can we verify a generic part in the TOOSM framework?

C How practical and useful is the tool-based approach for verification?
   • What lessons can be learned from this approach?

1.2.1 Approach

The first stage of the research (question A) is spent on making a broad overview of the verification techniques and tools available for verification of code. We need to get familiar with the techniques to be able to categorize and compare the tools. From this broad overview we will select the most promising tools and investigate them further. We also will define tool requirement. Based on our problem and needs, these tools will be evaluated against our tool requirements in order to choose one tool for the next stage of the master’s project.

Because we want to start the verification (question B) with a simple, but still close to the original, case, we make an abstraction from the TOOSM communication channel. The communication channel is an important concept and contains a lot of the generic functionality we wish to verify. Based on this abstraction, the verification process will be done in three iterations. Each iteration consists of a three steps: the first is to specify the requirements of the communication channel. These requirements are not related to any tool and technique. The second step is to transform these requirements in a form that the chosen verification tool (verifier) understands. The last step is to iteratively run the verifier, providing additional annotations of the code needed to verify the correctness.
In the first iteration, we verify only the communication channel itself. In the second iteration the communication channel is used by a client program. We try to verify the complete program to see if the communication channel can be used in a multi-threaded setting. The client program will have only one sender and one receiver on the endpoints of the communication channel. In this verification we want to parameterize the number of sent and received messages, that is, we want to show that we can send and receive $k$ messages. The third iteration is to have multiple senders and receivers. The number of senders and receivers will also be parameterized, thus having $n$ senders and $m$ receivers. We end up of having a program with a parameterized number of senders, receivers, and messages (with $k$ sent and $l$ received). Furthermore, the verifier has to prove deadlock freedom and the absence of race conditions for all iterations.

The three iterations of the verification process are shown in figure 1.

(i)

\[\text{Communication Channel}\]

(ii)

\[\text{Sender (S) to Receiver (R) via Communication Channel}\]

(iii)

\[\text{Multiple Senders (S$_1$, S$_2$, ..., S$_n$) to Receivers (R$_1$, R$_m$) via Communication Channel}\]

Figure 1: Verification iterations

In the last stage (question C) we address the lessons learned from the verification process.

1.3 Overview

The rest of the thesis is organized as follows. In chapter 2 an overview of SLCO and Toosm is given. In Chapter 3 some verification techniques are outlined, this includes Owicki-Gries and Separation Logic. Chapter 4 gives an overview of some verifications tools we encountered during this project, and we compare them to make a well considered choice in tools for further use in the project. The verification technique chosen for this project, Separation logic, is explained further in chapter 5. The chosen verification tool, VeriFast, is explained in Chapter 6. In Chapter 7 we provide the verification of the Toosm communication channel together with the specification and limitations. The last chapter, 8, describes the conclusions of this project and practicality and usefulness is the tool-based approach for verification and VeriFast. Furthermore, the chapter describes threats to validity and possible future work.
2 Preliminaries

In this chapter we explain a few concepts which may not yet be known to the reader. We describe what SLCO is and which constructs the language contains. We also introduce the Toosm framework, which was developed during the Capita Selecta project at TU/e.

2.1 SLCO

Van Amstel, Engelen and Van den Brand created SLCO [2], when investigating iterative DSL design. SLCO provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. The structure and behavior of the objects is described in a state machine approach. In their paper [2], they created a model of a conveyor belt in SLCO to investigate different model transformations. One transformation is used to transform an SLCO model into a model described in the Parallel Object-Oriented Specification Language (POOSL) [7], for simulation and performance analysis. Another transformation is used to transform the model into a model that can be read by the model checker Spin [4]. The last transformation transforms the SLCO model into executable code in the language Not Quite C (NQC). NQC can be executed on the Lego Mindstorms platform [5], e.g., a physical conveyor belt system could be built from Lego parts. The different target models have different language characteristics, i.e., POOSL does not support asynchronous communication. To bridge the gap a number of transformations are defined that transform an SLCO model to a different SLCO model with equivalent observable behavior, e.g. synchronous communication channels to asynchronous channels or bidirectional channels to unidirectional channels.

2.1.1 State machine

State machines are well known for defining the behavior of (parts of) a system. SLCO has a textual model. As it is sometimes easier to give a graphical representation there is also a graphical model of SLCO and a graphical editor in the Eclipse environment [6]; their expressive power is the same and both can reflect the same model. In figure 2 we give a graphical representation of two state machines (Note that to keep the picture simple, this is not exactly the SLCO graphical representation). In the next section those two state machines are used to describe most language constructs of SLCO.

![Figure 2: Communicating state machines](image)

2.1.2 Language constructs

The textual representation of SLCO uses of a number of language constructs. We use the textual representations of parts of an SLCO model to illustrate the language constructs. Listing 1 we show a basic example of textual SLCO. With the use of this example we briefly explain a few language constructs, note that we have omitted two parts for readability, we will fill in those gaps in listings 2 and listing 3.
An SLCO model always has one model construct. The model consists of a number of classes, objects and communication channels between objects. Objects are instances of classes. A channel can be one of three types: synchronous, asynchronous and lossless, and asynchronous and lossy. A channel is connected to a port of an object, allowing all state machines of that object to communicate over the channel. A class consists of a set of variables, ports and state machines. A port consists only of an identifying name. The variables are of the type integer, Boolean or string and can have an initial value.

A state machine is an object that has variables, nodes and transitions. Variables can be of type integer, Boolean or string and can have an initial value. A node can be the initial state, this is the starting location of the state machine, a final state, in this state a state machine may end, or a regular state. Transitions are relations between two nodes, a source node and a target node. The source node and target node can be the same node.

A transition can have a trigger. This can be a deadline or a signal reception. If the amount of time specified by the deadline has passed or if a signal is received, the transition is enabled. The countdown for the deadline starts when the state machine enters the source state of the corresponding transaction. The signal reception can have an expression. This expression can be over the arguments and over the variables. The expression yields to a Boolean value indicating if the signal can be received.

A transition can also have a guard. A guard is a Boolean expression over the variables of the class and state machine that must hold to enable a transition. Triggers and guards are used in the state machine definition in listing 2. This is the state machine definition of ReceiverStateMachine in listing 1. The receive trigger has an expression over it, the second argument has to be equal to the value 2 and also the guard requires that the first argument is positive. If both criteria are satisfied, the first argument is stored in the variable j and the third variable is stored in the variable a.
ReceiverStateMachine {
  variables
  Integer j = 0
  Boolean a
  initial A
  final B
  transitions
  A2A from A to A {
    trigger
    receive G(j,[2],a) from P
    guard
    (j >= 0)
  }
  A2B from A to B {
    trigger
    after 1000 ms
    guard
    (j >= 2)
  }
}

When the transition is made from a source state to a target state, the effects of the transition are executed. Effects are statements such as assignments or signal sending. An assignment construct is an assignment of a value or an expression to a variable. Signal sending sends a signal over a channel which is connected to a port of the class. Sending signals can be done in a synchronous manner and an asynchronous manner, depending on the channel type. Channels are one-place buffers to which the sending party can assign a value, and the receiving parties can read from. When the buffer is full, sending is not possible and therefore the transition is not enabled. With sending over asynchronous channels, the sending party can transition to the target state after placing the signal, possibly with arguments into the buffer.

With synchronous sending it is a bit trickier. First we have to distinguish two different cases, whether the sending is the first action of a the effects or not. In the first case, sending of a signal, possibly with arguments, can only be done if, and only if, the receiver can receive that signal, i.e., the corresponding name and the expressions over the arguments must also evaluate to true, otherwise the transition cannot be taken. In the second case, the send statement will wait until the receiving party picks up the signal, and then proceeds to the next action or state transition. The concept’s effects are presented in listing 3, which is the initialization of the state machine SenderStateMachine of listing 1.

Listing 3: Textual SLCO of communicating state machines (3)

SenderStateMachine {
  variables
  Integer i = 1
  initial C
  final D
  transitions
  C2C from C to C {
    guard
    (i <= 2)
    effect
    send G(i,2,true) to Q;
    i := i + 1
  }
  C2D from C to D {
}
2.2 Toosm

In the Capita Selecta project [1] the Toosm (Threaded Object-Oriented State Machine) was build to translate SLCO execution in C#. Toosm is an environment that has a translation from SLCO to C# and has generic code for artifact out of SLCO, e.g. the state machines and communication channels, described earlier. Toosm can not be executed on its own. It requires code that is generated by the Toosm transformations and the Toosm framework. The Toosm transformations take an SLCO model as input and produce code in C#. The Toosm framework consists of common functionality, from this generic code the model specific code inherits. Figure 3 shows the process that transforms the SCLO model to executable code.

![Figure 3: Toosm flow](image)

The reason for choosing to store the common functionality in a framework and not generating all required code is reuse of code. It is well known that maintaining duplicated code is harder than code without duplication. Most classes of the framework realize a single concept defined by the specification such as a state, a transition, a communication channel, etc. Using inheritance, the common functionality is available in the generated code - except for the communication channel class: which is used as an object, it is instantiated not inherited. Figure 4 depicts the essential parts of the framework as a class diagram.

In this project the decision was made to keep the transformation simple, therefore we mapped each language construct to a counterpart in the code. We could, for example, have encoded intelligence about having different code generated if the send effect is the first action of the transition or not; the transformation now is unaware of this and it is determined at in the code. In listing 4 we present a small impression of the transformation file, here the model artifact is used to generate a file and in that file the C# model class is defined. In the constructor first the communication channels are created, followed by the objects which are instantiations of class artifacts, the last step in the constructor is to join the objects and the communication channels to each other (the generateChannelObjectJoin is not presented).
Listing 4: Generation of model class in Xpand2

```csharp
<DEFINE generateModel FOR Model>
<FILE "mdl" + this.name + ".cs"->
using System;
using toosm.Framework;

namespace toosm
{
    // The class that represent the model '<<this.name>>'
    public class mdl<<this.name>> : Model {

        // The default constructor
        public mdl<<this.name>>() : base("<<this.name>>")
        {
            // declarations for each channel with a type of Channel
            <FOREACH this.channels AS cnl>
                Channels.Add("<<cnl.name>>", new Channel("<<cnl.name>>"));
                Channels["<<cnl.name>>"].SynchronousChannel = <<IF cnl.channelType==
                ChannelTypeEnum::sync >> true <<ELSE>> false <<ENDIF>>;
                Logger.Write("Channel: '<<cnl.name>>'", LoggerMessageType.ChannelInit);
            <ENDFOREACH>

            // declarations for each class
            <FOREACH this.objects AS obj>
                Objects.Add("<<obj.name>>", new Class<<obj.class.name>("<<obj.class. name>>", "<<obj.name>>", this));
            <ENDFOREACH>

            // join the channel to the objects (classes with stateMachines)
            <FOREACH this.channels AS cnl><EXPAND generateChannelObjectJoin FOR cnl><ENDFOREACH>
        }
    }

    <FOREACH this.classes AS cls><EXPAND generateClass FOR cls><ENDFOREACH>
    <ENDFILE><ENDDEFINE>
```

Figure 4: Toosm framework class diagram
State machines are executed concurrently in threads. Each state machine object has its own thread. A lot of code for this is in Toosm, only the model specific parts are in the generated code. Model specific parts are the lines of code that are model specific, like evaluating a specific guard.

The first way of information sharing between Toosm state machines can be via an instantiation of the communication channel object; the two most important methods of the channel are send and receive. Those methods contain all synchronization logic for controlling the threads. A communication channel has two end points, a sender side and a receiver side, which are modeled with Toosm classes. A Toosm class can hold multiple state machines, but on the receiver side only one state machine at a time can peek at the content and decide if it wants to receive it; this decision is determined by the conditional signal reception and guard evaluation.

A second way two state machines in the same class can communicate is via a class variable. To prevent race conditions a lock is placed around the access to this variable. This type of communication is always generated by the Toosm transformations.

The result of the Toosm transformations on the SLCO model shown in listing 1, 2 and 3 is presented in next three listings. In the listings we have removed some empty lines, comments and logging commands for readability. In listing 5 we give the code which is generated for the model object. Here we create the objects, channels and join them according to the model. In listing 6 we give the code for the receiver class and state machine. Notice that we add the states and transitions of the state machine in the constructor of the class. For each guard evaluation of a transition a method that is responsible for its task is created, like trying to receive signals and evaluating variables. In listing 7 the code for the sender is given, here we can see how actions, like signal sending, are executed.

Listing 5: Model specific code: model class

```csharp
public class mdlPaperExample1 : Model {
    public mdlPaperExample1(): base("PaperExample1")
    {
        Channels.Add("Q_to_P", new Channel("Q_to_P"));
        Channels["Q_to_P"].SynchronousChannel = false;
        Objects.Add("sender", new ClassSenderClass("SenderClass", "sender", this));
        Objects.Add("receiver", new ClassReceiverClass("ReceiverClass", "receiver", this));
    }
}
```

Listing 6: Model specific code: receiver class class and receiver state machine class

```csharp
public class ClassReceiverClass : Class {
    public ClassReceiverClass(string classname, string objectname, Model parent) : base(classname, objectname, parent)
    {
        StateMachines.Add("ReceiverStateMachine", new smReceiverStateMachine("ReceiverStateMachine", this));
        Init();
    }
```
public class smReceiverStateMachine : StateMachine{
    public smReceiverStateMachine(string name, Class parent): base(name, parent)
    {
        AddState("A", true, false);
        AddState("B", false, true);
        AddTransition(States["A"], null, true, GuardA2A, null, States["A"], "A2A");
        AddTransition(States["A"], 1000, false, GuardA2B, null, States["B"], "A2B");
        BeginAtomicStep();
        Init();
        EndAtomicStep();
    }
    protected override void Init()
    {
        Variables.Add("j", 1);
        Variables.Add("a", new object());
        base.Init();
    }
    private bool GuardA2A(object sender, smTriggerGuardEventArgs args)
    {
        SignalMessage signal = Parent.Ports["P"].Peek(this); // peek and lock
        if (signal != null) // there is a signal
        {
            if ((signal.Name == "G") && (signal.Args.Length == 3)) // correct signal name and arguments
            {
                object oldSMj = Variables["j"];
                Variables["j"] = (int)signal.Args[0];
                object oldSMa = Variables["a"];
                Variables["a"] = (bool)signal.Args[2];
                if ( ((int)Variables["j"] >= 0) ) //conditional signal reception (guard)
                {
                    Parent.Ports["P"].RemoveLast(this); //remove and unlock
                    return true;
                }
            //signal not received place values back;
            Variables["j"] = oldSMj;
            Variables["a"] = oldSMa;
        }
    }
    //The signal is not for this sm.
    Parent.Ports["P"].Release(this); //release peek lock;
    return false;
}
private bool GuardA2B(object sender, smTriggerGuardEventArgs args)
{
    return ((int)Variables["j"] >= 2) && (args.Transition.DeadlineExpired); //Delay guard
}

Listing 7: Model specific code: sender class class and sender state machine class

public class ClassSenderClass : Class {
    public ClassSenderClass(string classname, string objectname, Model parent)
    : base(classname, objectname, parent)
    {
        StateMachines.Add("SenderStateMachine", new smSenderStateMachine("SenderStateMachine", this));
        Init();
    }
}

public class smSenderStateMachine : StateMachine {
    public smSenderStateMachine(string name, Class parent) : base(name, parent)
    {
        AddState("C", true, false);
        AddState("D", false, true);
        AddTransition(States["C"], null, false, GuardC2C, ActionC2C, States["C"], "C2C");
        AddTransition(States["C"], null, false, GuardC2D, null, States["D"], "C2D");

        BeginAtomicStep();
        Init();
        EndAtomicStep();
    }

    protected override void Init()
    {
        Variables.Add("i", 1);
        base.Init();
    }

    private bool GuardC2C(object sender, smTriggerGuardEventArgs args)
    {
        return ((int)Variables["i"] <= 2); // Expression guard
    }

    private bool ActionC2C(object sender, smActionEventArgs args)
    {
        bool result = true;
        bool isFirstActionOfTransition = true;
        result = result && Parent.Ports["Q"].Send(new SignalMessage("G", new object[] { (int)Variables["i"], 2, true }), this, isFirstActionOfTransition);
        if (!result) return false;

        Variables["i"] = (int)((int)Variables["i"] + 1);
        return result;
    }
}
2.2.1 Logging and state space

In the Capita Selecta we used logging and generation of a state space to give a reasonable convincing argument for correctness, here we briefly describe how this was done.

The models execution can be presented as a state space, this state space can be generated with a model transformation of the model. The model transformation was made by creators of SLCO. The state space depicting states with state information and state transitions with corresponding SLCO actions. This state space can be compared against the execution of the program.

The code of the listings in the previous section can be executed, after adding the framework code, by initializing the `mdlPaperExample1` class and calling the `run` method. When the code is executed it generates a log file where we can inspect important steps, for example guard evaluation and value changes of variables. Such a log can be visualized by a program especially written for this purpose, see figure 5. The program searches the log for a begin and end log-record and combines them into a single event. All events are presented on a time line, which supports zooming. This tool was immensely helpful during the development of Toosm.

![Figure 5: Toosm visualization tool GUI](image)

With this program it is also possible to execute the generated code a specified number of times. The log of each execution is saved and those logs can be used to generate a “state space” of the program. That generated state space can be compared to the state space calculated from the
source SLCO model. A generated state space of the example used in this chapter is given in figure 6.

Figure 6: Generated state space of the code from the SLCO model in this chapter
3 Verification techniques

Concurrent programs these days are used in many different areas. There are compelling reasons for executing code concurrently, for example, the computation of certain data can be done much faster using true parallelism than when using sequential computation. Also, concurrency sometimes better fits the intuition designing a program than sequential execution. Programs today grow in size and complexity, which makes the presence of defects in especially concurrent programs more likely and more difficult to detect.

Techniques for verification have been developed since the 1950’s. Some techniques are based on testing or reasoning about the behavior of possible executions. These techniques help to find bugs and, for example, improve efficiency, but do not provide complete verification of correctness. Model checking achieves this completeness for programs that have an, essentially, finite state space. Many programs however cannot be verified in this manner, simply because the number of possible executions is too large (or even infinite). Especially for programs used in safety-critical situations we therefore turn to formal verification methods based on theorem proving to verify that the specification is satisfied.

In this thesis, we aim to verify specific concurrent code, the generic part of the TOOSM framework, in a compositional fashion, using theorem proving. Compositionality is important both to keep the verification manageable as well as to localize the effects of adaptations on the re-implementation, the re-specification and the re-verification.

In this chapter we briefly discuss various theorem-proving based approaches, from simple formal verification for sequential programs to the compositional verification of concurrent programs that we need. This provides some context for our approach and also explains our choice of a particular logic and proof approach.

The field started with Turing in the fifties and passed a milestone with Hoare’s approach in the sixties. In 1975 a significant extension was made by Susan Owicki and David Gries who invented a method for verification of concurrent programs [7]. This led to the further development of other techniques like the compositional rely-guarantee method for shared-variable parallelism [8] and, somewhat later, the Separation Logic approach [9] [10].

3.1 Hoare Logic

The origins of (axiomatic) reasoning about program verification can be traced back to Turing in the 1950s, and, in the early sixties, McCarthy’s Mathematical Theory of Computation. But the first useable approaches should be attributed to Floyd and Hoare. Floyd introduced assertional reasoning on flowcharts for proving partial (discarding termination) and total correctness (including termination). In Floyd’s method, partial correctness is proved by annotating each of the program’s control points with a mathematical specification, an assertion, on the state of the program that should hold whenever control is at that point. The program with the assertions leads to verification conditions, which can be checked by hand or with a theorem prover for the logic in which the conditions are expressed. For total correctness, termination is proved by special counting-down arguments in looping constructions, showing that the loop is finite.

Hoare developed this method further into a syntax-directed approach dealing with WHILE programs directly rather than with their representations as flowcharts. WHILE programs are programs with a small grammar with only a few statements, but they can express powerful and complex programs. The key concept Hoare introduced in his logic is a formula in the form of a Hoare triple, \( \{P\} S \{Q\} \), denoting that if the assertion \( P \) is true before initiation of program(fragment) \( S \), then the assertion \( Q \) will be true upon \( S \)'s termination. An assertion \( P \) holds locally if it is either:

- an initial assertion of a component and is implied by the precondition of the program, or
• if $P$ is established by a preceding atomic action $S$ with pre-assertion $Q$.

This concept can be used in a compositional way, building up from individual program statements to whole programs: to verify $\{P\} S \{Q\}$, nothing about the program around $S$ needs to be known except that always when control arrives at $S$, $P$ holds - and about $P$ that then, if $S$ terminates, $Q$ holds.

Hoare’s approach received a great deal of attention, and many Hoare-style proof systems dealing with various programming constructs have been proposed since then; it is still an inspiration for current research. Thanks to Floyd and Hoare we view programs as state transformers.

Another significant contributor to this subject was Dijkstra, who, among other things, introduced in 1975 the idea of weakest precondition, i.e. predicate transformers that provided are the basis for today’s verification condition generators used in automated verification [11].

3.2 Owicki and Gries

Hoare logic was extended to concurrent programs by Owicki and Gries [7] [12]. The Owicki-Gries proof system represents the first and probably the simplest extension of Hoare logic to parallelism, with shared variables. They extended the logic with only one proof rule, for the parallel `cobegin` ... `coend` statement. In the same manner as Hoare, one breaks down the parts of the verification into simpler pieces. First, the sequential components of the program are annotated with assertions. The components are then verified separately, as in the sequential case: local correctness. Then it is shown additionally, that each assertion in the annotation of a component is invariant under the execution of other components: global correctness. An assertion $P$ is interference free, if for each atomic action $\{Q\} S$ taken from another component, $\{P \land Q\} S \{P\}$. An assertion holds if it holds both locally and is interference-free.

The interference-freedom requirement is also the main drawback of this method: Owicki-Gries is not compositional: To perform the interference-freedom test for the assertions in some component we require information about the implementation of ALL concurrent components. As Lamport stated [13], this means that for a concurrent program with $n$ statements, there are $O(n^2)$ verification conditions instead of the $O(n)$ conditions needed for `while` programs with Hoare logic. The need to have knowledge about the inner workings of two components is a major drawback. Dijkstra formulated this in 1965 in Cooperating Sequential Processes [14] that processes are disconnected except for explicit communication, as follows: “We have stipulated that processes should be loosely connected; by this we mean apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other.”. This limits the class of concurrent programs for which verification is feasible to languages that have quite constrained communication primitives - our programs with channels and senders and receivers are more complex than can be feasibly handled by the Owicki-Gries approach.

3.3 Rely/guarantee

The idea to obtain a more compositional proof method for general parallel programs is to enrich the specification of each component with additional, but more abstract, information about the interaction with its environment during its execution. Jones introduced in 1983 his Rely/Guarantee verification method [8] [15] [16] for shared memory parallel programs that has this feature.

Instead of having only the precondition and postcondition, Jones added two new predicates: a rely condition and guarantee condition. The rely condition specifies what the component, minimally, expects from the environment, i.e., that the rely condition expresses requirements on all atomic actions of the environment, describing the maximal interference the component can tolerate. The rely condition states these requirements in specification terms, meaning that we do not need
to know the implementation of the environment. Conversely, the guarantee condition expresses the task performed by that component, and how this task may influence the environment - also abstractly. As a result, the proof rule for parallel composition can be formulated in terms of the specifications rather than the implementation of the components (as in the Owicky-Gries case), i.e. the resulting proof method is compositional.

Although the rely-guarantee method represents a significant step forward in the methodology of program verification, it does not make the Owicky and Gries method obsolete. In concurrent shared variable programs where processes require a lot of reading from and writing to the same shared variable, non-compositional methods are sometimes more successful than compositional ones, i.e., where precise information about the information is required, using the implementation may be easier than separately writing specifications that are equally detailed.

### 3.4 Separation Logic

The rely/guarantee in fact satisfies our desire for a compositional verification method, able to prove functional properties about programs with complex communication via channels and senders and receivers. Nevertheless, we look further to a more recent development, Separation Logic. The reason is, that besides functional requirements, we are interested in dealing with deadlock and race conditions. Separation logic provides a novel way to handle shared resources that is very well suited to deal with these two issues. Separation Logic ca be used to treat channels, viewing channels as shared resources - and be stretched to also deal with functional properties.

In this section we just introduce the essential, basic idea of Separation Logic to deal with shared resources; in section 5 we present how Separation Logic handles concurrency, in section 7 we present the specialization to our specific case, channels with senders and receivers.

Orthogonal to compositionality of program parts, these is the issue of aliasing: decomposing the memory. A modern system for reasoning about program correctness that focusses on this issue is Separation logic [9] [10]. It was developed by Reynolds and O’Hearn. It is an extension of Hoare logic with a built-in notion of a resource, and is based on the logic of bunched implications (BI) [17].

They key concept behind Separation Logic is local reasoning. In 1972 Burstall [18] made the observation that separate program code which works on separate sections of the memory can be reasoned about independently. In Separation Logic, the specifications and proofs of a program component mention only the portion of memory used by the component, and not the entire global state of the system. Thus, only the parts of the memory needed for the reasoning are used, and all other parts remain unchanged.

Its main application has been reasoning about pointer programs. Separation logic gives an elegant solution for the problem of aliasing, which is a common problem in verifying pointer programs. Aliasing happens when a single memory object is referenced by multiple pointers in different places. Separation logic keeps track of the memory usage, e.g. that a single memory object is aliased, and also handles deallocation of unused memory. As Separation Logic is a recent development, there are various versions of the logic with complementary features, but there is not yet a standard uniform presentation of all of these. In the initial version of the logic there is no support for object oriented programs and permission based variable access. Those concepts are important for our research, and fortunately they are provided as a number of extensions, described in various papers.

Specifically, standard Separation logic focuses on the memory model, which in Separation Logic has a store and a heap. A store is a function mapping variables to values. The heap is modeled as a partial map from addresses to values. A heap with no entries is called the empty heap and is denoted as \( \text{emp} \). A heap with an entry on address \( E \) and with content \( F \) is written as \( E \mapsto F \). If the value is unknown or we do not care about the value, we can write \( E \mapsto \_ \), which is shorthand...
for \( \exists v : E \mapsto v \). To reason about multiple values in a heap, two heaps can be combined, using a new symbol, separating conjunction. Heap \( h \) and \( g \) for which the address domains are disjoint, can then be combined as \( h \ast g \). It is now easy to express that a program is in a state where the program has two values \( x \) and \( y \) with the values 2 and 7, \( x \mapsto 2 \ast y \mapsto 7 \).

To enable reasoning about the effect of the execution of program fragments on store and heap, another important requirement added to Separation Logic is the requirement of “tight specification”\(^1\). A specification \( \{P\}C\{Q\} \) is tight when all of the resources accessed by \( C \) are described in its precondition \( P \), or acquired through explicit resource transfer. No other resources will be accessed or affected by \( C \) when executed from a resource satisfying \( P \). The tight interpretation is essential for separation logic. Suppose we have \( \{P\}C\{Q\} \), then every resource that \( C \) modifies is in \( P \) and every resource not modified is not in \( P \). Every resource not in \( P \) can safely be used in other parts of the program, thus can be safely used in other threads.

One result of this tight interpretation is the frame rule, which allows a small specification to be embedded into a larger context. It captures that \( C \) changes only the values of variables present in predicate \( P \), then it will not change values of other memory locations. This additional requirement on the reasoning, going beyond Hoare logic, enables using Separation Logic for compositional reasoning.

\[
\frac{\{P\} \ C \ \{Q\}}{\{P \ast R\} \ C \ \{Q \ast R\}} \text{ modifies}(C) \cap \text{vars}(R) = \emptyset
\]

The frame rules side-condition is inherited from Hoare logic. It is required because \( \ast \) only describes the separation of heap locations and not variables\(^2\). This frame rule is proven sound in the context of Separation Logic\(^2\). It is now possible to reason about low level imperative programs in Separation Logic, for example, \( \{x \mapsto 2 \ast y \mapsto 7\}x := 5\{x \mapsto 5 \ast y \mapsto 7\} \).

At this stage of our logic evaluation, Owicki-Gries, Rely-Guarantee and Separation Logic are all still viable options. We now consider verification tools to further guide our choice.

---

\(^1\)“tight specification” is occasionally referenced in other papers as “precise specification”.

\(^2\)O’Hearn\(^19\) observes that this side-condition can be removed, which is shown in chapter 5.
4 Verification tools

This chapter gives a partial overview of verification tools that are available for formal verification of programs. The focus is on tools that use a form of axiomatic reasoning on code. The first part has two goals: one is perform an exploration into the field of verification tools in general, to get a good understanding of the today’s tools, the other to create a list of tools that are promising for our aims. The second part is a selection process: we select the tool which is best suited for our verification goal.

Because producing a complete list of verification tools that is available is outside the scope of the project, we consider a limited number of tools. The reasons for looking at these tools are quite diverse. Some features were very concrete, like the programming language supported. But even such a feature would not be fully decisive: we might choose to re-write our code to fit a tool, if that tool was the only realistic option for other reasons. Also, for example, even just having contacts that have good experiences with a tool, or a tool being supported by a larger company, made us have a closer look at such tool. We report on these initial explorations, as they provide useful information about the field and also because these explorations shaped the criteria that we used to select tool.

First we give a short introduction to the tools, in section 4.2 we state the requirements for the tools such that we can make a well-considered decision, and then evaluate the tools on these requirements. In section 4.3 we end with a conclusion together with the tool choice.

The first four tools are from Microsoft, since Microsoft is the creator of C#, the programming language of TOOSM. These are Spec#, Code Contracts, Dafny and Chalice. The RISE group of Microsoft Research has a number of projects that focus on formal verification. Those tools turned out not to be applicable for the research question, therefore a wider search wind was made, and more tools were investigated; they are VeriFast and ProPar.

4.1 Tool introductions

In the next section, we present the tools by means of short descriptions.

4.1.1 Spec#

Spec# [21] is an object-oriented .NET language with design-by-contract features for method preconditions and postconditions and object invariants and also a non-null type system. With a non-null type system a variable can not have the null value, it always needs a concrete value or reference. In many programming languages, like Java and C#, an Integer is defined as a non-null type. Microsoft Research designed it to explore the possibilities of formal program verification. Spec# permits specification and reasoning about object invariants even in the presence of callbacks and multithreading. It supports both dynamic and static checking of certain properties, although it is not always possible to check all properties in both settings. The Spec# programming language is an extension of the object-oriented language C# 2.0. The compiler allows the source to be compiled to an executable or library, with additional code and data to support dynamic checking while executing the code. Beside the possibility of dynamic checking, Spec# has a static program verifier, Boogie. Boogie generates all verification conditions from the Spec# code, with the use of the classical weakest-precondition calculus. An automatic theorem prover, standard it uses Z3, tries to prove all verification conditions to prove the correctness of the program or find errors in it. The theory behind Spec#, or what Microsoft likes to call it the Spec# methodology, is based on boolean logic. The verification conditions created by Spec# are operations and assertions on a heap. It uses ownership of objects to find possible race conditions, and keeps track of locking orders to detect deadlocks. In Spec# one can use the value of a variable that it holds on entering...
a method, read when the precondition was evaluated, side by side with the current value in a postcondition.

Programming language-wise and also with respect to the features offered for specification and verification this seemed a good candidate. However, although the Spec# project is not official discontinued, there is no more effort spent on it. Microsoft gave the code to the open source community, support is via an Internet forum, but the activities here are limited. In personal communication with dr. R. Leino [22] it became clear that the system is outdated compared to modern program verification tools. All this would seriously limit the value of our verification, especially as regards value for future research.

4.1.2 Code Contracts

Another verification tool of Microsoft Research is Code Contracts [23]. Code Contracts uses the same theory and backend (Boogie) as Spec#. For Code Contracts Microsoft took the lessons learned in Spec# and added some new requirement and insides.

Code Contracts provides a language-agnostic way to express coding assumptions, called contracts, in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants. As Code Contracts work on all .net languages, we have all the benefits of those frameworks, thus also threads and all required synchronization constructs. The specification, in the form of contracts, can be checked for validity by a static checker, the same as in Spec#. There is also the possibility for dynamic checking. Code Contracts bring the advantages of design-by-contract programming to all .NET programming languages. Microsoft states that Code Contracts is industry ready, and it is shipped with the latest ultimate edition of Microsoft Visual Studio EDI. While the Code Contracts approach and tooling were released over two years ago now, it seems not to be commonly used while programming in .NET today.

A major difference between Code Contracts and Spec# is that there is no way of explicitly specifying loop invariants on loop bodies. The static verifier has mechanisms to derive loop invariants itself.

Code Contracts seem a good candidate for our purposes.

4.1.3 Dafny

Another tool of the Rise group of Microsoft Research is Dafny [24]. Dafny is an imperative object-based language with built-in specification constructs. It has a lot of aspects in common with Spec# and Code Contracts. Dafny is designed to support the static verification of programs. In particular, it can be used to verify the functional correctness of programs.

The language is imperative, sequential, supports generic classes and dynamic allocation, and has build-in specification constructs. The specifications include preconditions and postconditions, frame specifications (read and write sets), and termination metrics. A nice feature of Dafny is that it is able to prove termination, if the required annotation is supplied. The verifier is thus able to proof total correctness, a feature not all tools support. To ease the usage of specification, Dafny supports ghost variables, recursive functions, and types like algebraic datatypes, sets, and sequences. Specifications and ghost constructs are only used during verification; the compiler omits them from the executable code. The Dafny verifier is run as part of the compiler. While compiling the code, the code is checked against the specification. Boogie and the Z3 SMT solver again powers the verifier.

Dafny thus is a user friendly, mature tool, and as such rapidly gaining acceptance. A significant, and for us show-stopping, limitation is, that it is developed for sequential verification instead of concurrent verification.
4.1.4 Chalice

Chalice is another experimental language and tool from Microsoft Research [25], it explores specification and verification of concurrency in programs. Chalice supports contracts such as preconditions and postconditions and loop invariants. Chalice translates its input program, with the contracts, to the intermediate verification language Boogie for which verification conditions in first-order logic can be generated. Like Spec#, Code Contracts and Dafny these verification conditions can then be solved by the Z3 SMT solver.

Chalice is object-based, it supports the creation of objects but does not support inheritance. A program consists of one or more classes, which contains zero or more methods and fields. Each method has, like the previous tools, a precondition and a postcondition. In the method body one can use constructs like assignment, conditional constructs and looping constructs. When a loop is used in Chalice, it is possible to specify a loop invariant.

Chalice keeps track of fractional permissions for all memory locations, all objects stored in the heap. With those permissions the verifier can determine whether certain code can only read or read and write to that location of memory. A method can not do a read or write to the class fields, unless we mention the required permission in the precondition of the method. Permission of fields should also be returned to the method caller, via the postcondition, otherwise the field becomes inaccessible after the method has finished. Permissions can be less than the full permission, a fractional permission, which allows only a read action on the particular memory location. Fractional permission are specified as an integer value between 1 and 100, where 100 means the full permission.

We will later encounter permissions in the context of our chosen logic and verifier - we defer more detailed explanation till then.

Chalice supports threads. A thread in Chalice starts executing a method of a class in a new execution trace. Starting a new thread will transfer all permissions mentioned in the precondition of the starting method to the new thread. At a later stage in the execution path of the original thread, the two threads can be joined again. At this point, the return value of the starting method is returned, and the postcondition from the method is assumed, all permissions mentioned in the postcondition are transferred back to the original thread.

An important concept in concurrent programming are synchronization constructs. In Chalice, monitors are used for this. Every object can be used as a monitor. A thread can give its permission to a memory location up to the monitor. When the monitor is acquired, the permissions are transferred from the monitor to the thread and on releasing, the permission goes back to the monitor. The monitor can be shared via a fractional permission by multiple threads, and when those threads acquire the lock it can work on the shared memory location. The permissions held by the monitor are specified in an invariant in the object. This invariant can also be used as object invariant for the related object.

As should be clear from the detailed description of Chalice, this is a serious candidate for our verification. The reason that we looked for further approaches and tools is, the lack on support for inheritance. Because the code of Toosm makes frequently uses the concept inheritance, support for this concept in the verification is desirable. Also, we wanted to look beyond what was offered by Microsoft.

4.1.5 VeriFast

Another research tool for verification of certain correctness properties is VeriFast [26] [27]. VeriFast is developed at the Department of Computer Science of the Katholieke Universiteit Leuven. VeriFast takes an annotated program as input. Annotations include method preconditions and postconditions and loop invariants in a form of Separation Logic. Originally developed for C
programs, the current version is also able to handle a subset of Java. VeriFast is able to process the Java grammar, but it also requires contracts on the methods of the standard Java classes. The creators of VeriFast supplied a number of contracts for the standard Java classes but not all; only those they needed in the past for their verifications have contracts. Most papers and help documents are focused on the C part, this “weak point” is acknowledged by the creators. The tool reports either “0 errors found” or indicates the location of a potential error. If the tool reports “0 errors found”, this means

- does not perform illegal memory accesses, such as reading or writing an object instance field before an object is instantiated, or after the object instance has been freed, and
- does not include a certain type of concurrency errors known as data races, i.e. unsynchronized conflicting accesses of the same field by multiple threads. Accesses are considered conflicting if at least one of them is a write access.
- complies with function contracts (preconditions and postconditions) specified by the programmer in the form of special comments (known as annotations) in the source code.

VeriFast verifies each function by stepping through it, while keeping track of a symbolic representation of the relevant program state. It will have some assumptions about the current state of the program and also heap chunks that represent the data of the heap. The verification is done via a symbolic execution, using a Separation Logic-based representation of the memory. VeriFast, as the name indicates, can, for the properties it is designed to verify, produce a result very fast compared to other provers.

So a preliminary verdict: promising tool for our purposes, downside may be that it is using complex Separation Logic and aimed for Java rather than the C# we use.

4.1.6 ProPar

The research tool ProPar [28][29] (Prover of Parallel Programs) is a tool written in Python that supports verification of parallel programs that annotated in Owicki and Gries style. ProPar is developed by W. Wesslink and A. Mooij of the Technische Universiteit Eindhoven. ProPar checks the correctness of annotated programs by taking an annotated program as input, and generates proof obligations. The generated proof obligations are checked by the PVS proof checker. The proof checker validates the proof obligations or fails, just stating whether or not the program meets the specifications. A details of which proof obligations are proven and which not is given to give a better feedback to the user.

ProPar has a basic language which is close to the language constructs used by Owicki and Gries. The grammar of the language is not close to C#, most other tools have a grammar which is or is closer the C#. ProPar is in this area a outlier. We have not found any support for Object Oriented programming in the language, but the most basic language constructs as assignment, if, loop and parallel composition exists in the language. Between the language constructs one can place assertions, predicates on the systems state, to verify the systems state at that point. Also the way programs and assertions are expressed is completely different than the other tools do it. In ProPar one file specifies the structure of the program and all assertions, guards and statements are replaced by identifiers. A second file contains a PVS translation of the assertions, guards and statements. For repetition and parallel composition there is also a notion of invariants; invariants are in a way shortcuts to specify that a certain predicate holds for a certain part of the code, they hold at the beginning up to the end of the related code.

A preliminary verdict is that usage beyond the TU/e community and long-term stability and maintenance of the tool are quite unclear, also a factor that plays a negative role is the programming

---

3Stated by the VeriFast developers “There are a few known small reasons (known unsoundness) why the tool may sometimes incorrectly report “0 errors found”.”
language.

4.2 Tool selection

To make a well-considered decision which tool we select, we established four main tool requirements: concurrency, object-oriented, language, availability and easy of use. If other aspects played a role at preferring or discarding the tool, they will be stated in the section about that tool.

An essential requirement is that the tool supports some kind concurrency. Tools without this can not deal with our program and can not proof the absence of concurrency defects. We would only be able to prove whether some non-concurrent parts of the code in isolation meet some requirement, but it would be hard to say something about the whole system.

Another requirement is some kind of support for dealing with objects. It would be nice to have support for a complete object oriented language. There are tools that support objects but do not support subtype polymorphism. Support for subtype polymorphism is important due to the regularly use in TOOSM, but it is not mandatory for the verification we perform in this thesis, as there we only specify and verify classes, but do not address more advanced OO features yet. There are relevant things we can prove where we do not need subtype polymorphism. Recoding of the code to remove the subtype polymorphism is maybe possible, but this would require additional arguments why this code reflects the original code, and why the verification on this code is a valid verification for the original code.

We focus on tools that support the programming language where TOOSM is made in, C#. This would degrease the effort of rewriting the code into a different language. As our initial tool survey indicates that this requirement limits the tool choice dramatically, we keep an open mind for tools with different languages.

The last issue is the tools availability and the ease of usage. Because program verification is, in the sense of formal verification, no common good in the software development world, the number of available tools can be limited. This in contrast to the available programming languages, IDEs, test tools and other programming supporting tools. The ease of use of the tool is important, mainly the effort needed to use the tool, especially as regards tools from the academic world. For those tools, it is known that the documentation is not always complete or hard to read. Therefore, those aspects play an important part in the tool selection. This is a difficult to weigh requirement: even by-chance contacts with experienced users might influence the choice.

Recap in sort:

- Concurrency (mandatory);
- Object-oriented (at least support for objects);
- Language (preferably C#);
- Availability and ease of use (rather “soft” requirement).

4.2.1 Spec#

As stated before Spec# uses a super set of C#, therefore the first three requirement are easily met. We now give Spec# a closer inspection, we want to get some experience in verification with Spec#. In the next example we show a simple Spec# class with three methods. The methods must calculate a simple mathematical formula, in this way we can see how to formulate the specification together with the coding. At the same time the we can see if the verifier is able to conclude some straightforward mathematical calculations.
Listing 8: SpecSharp summation example.

class Summation {

    // most simple summation of a polynomial expression
    public static int SimpleSummation1(int m, int n)
    {
        int som = 0;
        int i = m;
        while (i <= n)
        {
            som += i;
            i ++;
        }
        return som;
    }

    // simple summation of a polynomial expression (special case of arithmetic series)
    public static int SimpleSummation2(int n)
    {
        int i = 0;
        int som = 0;
        while (i < n)
        {
            assert i != n;
            som += (i + 1);
            i ++;
        }
        return som;
    }

    // simple summation of a polynomial expression (arithmetic series)
    public static int SimpleSummation3(int m, int n)
    {
        int som = m;
        int i = m;
        while (i < n)
        {
            som += i;
            i ++;
        }
        return som;
    }

    // most simple summation of a polynomial expression
    public static int SimpleSummation1(int m, int n)
    {
        int som = 0;
        int i = m;
        while (i <= n)
        {
            som += i;
            i ++;
        }
        return som;
    }

    // simple summation of a polynomial expression (special case of arithmetic series)
    public static int SimpleSummation2(int n)
    {
        int i = 0;
        int som = 0;
        while (i < n)
        {
            assert i != n;
            som += (i + 1);
            i ++;
        }
        return som;
    }

    // simple summation of a polynomial expression (arithmetic series)
    public static int SimpleSummation3(int m, int n)
    {
        int som = m;
        int i = m;
        while (i < n)
        {
            som += i;
            i ++;
        }
        return som;
    }
}
This class can be verified by Spec#’s static checker. This means the postconditions are valid and therefore, the method meets its specification. There is one comment to make about the third method: the commented assertion in the method are unprovable in contrast to the complete method. The first assertion follows directly from the loop guard, the second commented assertion from the loop invariant and the fact we increase \( i \) by one, and the third is clearly the same as the loop invariant. The same holds for the fourth and fifth commented assertion. This shows a potential weak point in the static verification (or the verification condition generator).

We also investigated the verification of multithreaded programs. We found that verification of multithreaded programs was hard, and where not able to have a working verification of multithreaded code.

For a number of other factors this tool is not a good choice. The factors that played a role where:

- Effort needed for the verification is great.
- Documentation was sometimes not clear and there where no running examples on concurrent verification to show how this kind of verification could be done.
- No further development by Microsoft of this tool.
- Outdated. Newer tools succeeded this tool.
- Other tools seemed to provide better change for the verification.

In the appendix we show a multithreaded program with locking and sharing of variables, which we tried to get verified. The rather simple example shows up to level which we got, it could be that with simple modification this program is verifiable, but in search of tool we could not get it to work.

### 4.2.2 Code Contracts

Code contracts can be seen as the successor of Spec#. This tool was therefore a natural candidate in the selection process. Because Code Contracts is shipped with the .NET framework 4.0, initial setup was not needed and the support was already built in the IDE, Visual Studio 2010. Because the languages targeted seemed to allow concurrency, we expected Code Contacts to support concurrency, but when investigating it, it appeared that the support for this was limited or non-existing. On the Internet someone stated even, “... using Code Contracts ... with multithreading can be downright dangerous sometimes”. So although the programming languages support concurrency, as we have the complete .NET functionality to our proposal, and one can use specification in the code, without good support for concurrency in the verifier, we can not use the approach for concurrency verification.

A second disadvantage of Code Contracts is the design decision of Microsoft of leaving out explicit loop invariants. Inferring loop invariants automatically has it disadvantages. In the example of

---

4 It is common the specify the goal of a method in its postcondition. In the situation of fine grained locking and a postcondition, this can not be done, another thread could have changed the situation between end of locking and before checking the postcondition, in such a way the postcondition does not hold anymore.
Listing 9: Code Contracts summation example.

```
public static int SimpleSummation1 (int m, int n)
{
    Contract.Requires (0 <= m && m <= n);
    Contract.Ensures (Contract.Result<int>() == n + 1 - m);
    int som = 0;
    int i = m;
    while (i <= n)
    {
        som = som + 1;
        i = i + 1;
    }
    return som;
}

// simple summation of a polynomial expression (special case of arithmetic series)
public static int SimpleSummation2 (int n)
{
    Contract.Requires (0 <= n);
    Contract.Ensures (Contract.Result<int>() == ((n * n + n) / 2));
    Contract.Assert (0 <= n);
    int i = 0;
    int som = 0;

    while (i != n)
    {
        // Contract.Assert (0 <= i && i <= n); /* loop invariant */
        //Contract.Assert (som == ((double)i * i + i) / 2); /* loop invariant */
        //Contract.Assert (i != n); /* loop condition */
        som = som + (i + 1);
        i = i + 1;
    }
    Contract.Assert (i == n); /* negation loop condition */
    return som;
}
```

The static verifier verifies the method SimpleSummation1, but can not prove correctness of the postcondition of the method SimpleSummation2. Adding assertions similar to the loop invariant of Spec# in the loop had not the effect that it becomes verifiable. The verifier then fails at the commented assertion of line 27. Apparently removing explicit loop invariants in code contracts made writing specifications easier, as coming up with good loop invariants is considered to be hard, but this design decision of Code Contracts comes at a high cost: If the loop invariant can not be correctly generated, then the functionality of the loop, and therefore also the method containing the loop, can not be proven.

Due to the disadvantages of poor concurrency support and the awkward design decisions described above. Code Contracts, after this closer look, turned out not to be applicable for us.
4.2.3 Dafny

Due to no support for concurrency, this tool will not be investigated further. Concurrency is a requirement we can not dispense with.

4.2.4 Chalice

At first sight Chalice seems a promising tool for our research question: the tool was designed to deal with concurrency problems like we have and it is able to prove functional specifications. Chalice supports concurrency via threads and this is for us important.

One downside is that it is not object-oriented, but object-based which means we do not have inheriting. The code we want to verify heavily depends on inheriting, this is a potential problem. As suggested before, some of our verifications can do without it and it maybe possible to make a verification on redesigned code without subtype polymorphism, but this is not preferred.

Another disadvantage of Chalice is that it has its own grammar for the programming and specification language. This means that we have to make a translation from C# to this language, this is not preferred.

The third disadvantage is the usability and support. The crucial verification of multithreaded code with semaphores is hard, we tried to get help from the tools forum. Questions posted to the Internet forum of Chalice where left unanswered, this could be due to the question or to the limited activities there.

While at first glance it looked promising, the disadvantages for us were too big, therefore this tool is not our first choice.

In itself Chalice is a nice tool so to give an idea of how verification with Chalice works we give again the same simple summation program used previously but now in Chalice, see listing 10. Note that we were unable to code return values of methods, for this we use a trick with an additional variable to store the return value. The verifier has no problem verifying the three methods.

Listing 10: Chalice summation example.

class Summation {
  // most simple summation of a polynomial expression
  var SimpleSummation1Result: int;
  method SimpleSummation1(m: int, n: int)
    requires acc(SimpleSummation1Result) && 0 <= m && m <= n;
    ensures acc(SimpleSummation1Result) && SimpleSummation1Result == n + 1 - m;
  {
    var som: int := 0;
    var i: int := m;
    while (i<=n)
      invariant 0 <= i && i <= n+1; /* loop invariant */
      invariant som == i - m; /* loop invariant */
      {
        som := som + 1;
        i := i + 1;
      }
    SimpleSummation1Result := som;
  }
  // simple summation of a polynomial expression (special case of arithmetic series)
4.2.5 VeriFast

VeriFast is advertised as a verification tool for concurrent programs, that can deal with race conditions and deadlocks as well as functionality. The first two items are very desirable for us, especially if the support makes the verification smooth and easy, the second we are willing to put effort into investigating the possibilities.

One thing that seemed problematic is that VeriFast uses C or Java. This is a disadvantage, but
a translation from C# to Java is, in most cases, not that hard. An argument of correctness can easily be made, because the two languages are very similar and based on the same programming principles.

Java is, like C#, object oriented, so we can use code with inheritance. This is preferable to other techniques.

While evaluating the tool the usability seemed to be very good. There is a lot of documentation and examples, most of them in the C language. The feedback of the creators was quick, and helpful allowing us to proceed to more daring problems.

Our initial hesitance to go the direction of Separation Logic was assuaged when we found out that the part used to describe channels-as-a-shared-resource turned out to be quite manageable.

All things considered, we have chosen this tool to proceed with. We therefore leave it at this point and in chapter 6 will give a more extensive description of VeriFast.

4.2.6 ProPar

This tool is not investigated further because another tool, VeriFast, which was promising was selected during the tool investigation. At first glance ProPar has a few disadvantages. The first is it seems to miss the support for object oriented programming. Secondly, the language required to encode the program in, is not close to C#. The last disadvantage is the documentation supplied with ProPar was very concise; because to I did not try to verify any programs; I can not say if this concise documentation is enough.

4.3 Conclusions

To conclude the tool selection, we present the tools and the four requirement in table 1. In the table a “1” means fulfilment of the requirement and a “0” means no fulfilment, we used a half to indicate fulfilment of the requirement is not complete.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Concurrency</th>
<th>Object-Oriented</th>
<th>Language</th>
<th>Availability and ease of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpecSharp</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Code Contracts</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dafny</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>Chalice</td>
<td>1</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
</tr>
<tr>
<td>VeriFast</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>ProPar</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
</tbody>
</table>

In the last couple of sections we already excluded for different reasons all tools except one. The most promising tool at the end of this tool investigation is VeriFast, therefore we will continue this project with that tool as selected verification tool.

We see differences in the power of the prover, even if the same backend and SAT solver is used. Spec#, Code Contracts and Chalice all convert their input to the intermediate language of Boogie and use Z3. The selected tool was most promising and had only one disadvantage the language, C# vs Java, but we feel like this can be overcome.

*Some state that C# is inspired by Java or even an imitation of Java, others state it is evolved out C++ and a common runtime environment, lets leave that discussion as it is.
We see that verifications tools to do verification on annotate source code with concurrency is not yet a common good. We give here a small overview of tool we encountered but did not do any further research in due to different reasons:

- **VCC** is an industrial-strength verification environment for low-level concurrent system code written in C. (Microsoft) Main disqualify reason is the non Object Oriented C language.

- **Smallfoot** is an automatic verification tool that checks Separation Logic specifications of sequential and concurrent programs that manipulate recursive dynamically-allocated (linked) data structures. (Caldera Systems/The SCO Group)

- **VerCors** is a verification tool for Concurrent Data Structures, its specification language combines features of Separation Logic with JML. (Universiteit Twente) Experimental tool, no open beta.

- **ESC/Java** (Extended Static Checker for Java) is a programming tool that attempts to find common run-time errors in Java programs by static analysis of the program text. (Kind-Software Research Group at University College Dublin) Main disqualify reason, no releases the last 5 years.
5 A closer look at Separation Logic

In section 3.4 Separation Logic is introduced. This section repeats the basics and then provides
the extensions for concurrency and permissions. This forms the basis for many verification tools
that use Separation Logic and concurrency.

This chapter includes a minor disclaimer. There are a few notable extensions to Separation Logic.
There is no claim that the techniques or extensions described here are the same as the techniques
that drive VeriFast.

5.1 Basics

First we will repeat the most fundamental concepts demonstrated in section 3.4:

- \{P\} C \{Q\} - Hoare triple, the precondition P is met, the command C establishes the
  postcondition Q;
- emp - empty heap;
- \[E \mapsto* F\] (points-to) - a singleton heap with address E and contents F;
- \[E \mapsto_\_\text{shorthand for } \exists v : E \mapsto v;\]
- \[h \ast g\] - (separating conjunction) a heap with two heap cells h and g, which do not point to
  the same memory location;
- \[\frac{\{P\} C \{Q\}}{\{P\ast R\} C \{Q\ast R\}} \text{ modifies}(C) \cap \text{vars}(R) = \emptyset\] - frame rule.
- Tight specification (precise specification) - All resources required to execute the statement
  are present in the precondition of the statement and no more.

Separation Logic is very well suited for reasoning about pointer programs; conversely, the standard
Hoare logic rules are unsound when aliasing is allowed. Consider the following block of code
\[x.f = 3; y.f = 4; \{x.f < y.f\}\]. The specification seems to hold, but in a language with pointers
it may not be the case. Take for example the situation where both \(x\) and \(y\) point to the same
memory location, then the assertions of the specification do not hold. The technical term of
multiple pointers to the same memory location is called aliasing. Hoare logic has the Assignment
axiom, \([E/x]P x := E\{P\}\), which is clearly is unsound if aliasing is allowed. In Separation Logic
this situation can not occur, the frame rule disallows it, because there is a command that modifies
a variable that is also free. An example of a small proof for a pointer program is shown in listing
11.

Listing 11: Separation Logic pointer example

```
{ emp }
local x, y;
{ x \mapsto_\_ \ast y \mapsto_\_ }
x = new Cell(3, nil);
{ x \mapsto 3, nil \ast y \mapsto_\_ }
y = x;
{ y \mapsto 3, nil \land x == y }
y.head = 4
{ y \mapsto 4, nil \land x == y }
```
5.2 Variables as resource

In classical Separation Logic variables are separate from the heap. They form a stack, and cannot be pointed to; many rules need side conditions, e.g., the frame rule. Bornat et al. [30] showed that one can eliminate the variable-use side conditions on the rules: variables are treated as resources in the same way as heaps are treated as resources. This is a gentle change in the logic, but it permits Separation Logic to deal with operations which control access to variables. A resource is identified by a resource name, and a set of variables is connected to resource.

The frame rule can now be expressed without the side condition:

\[
\frac{P \ \ C \ \ Q}{P \ast R \ \ C \ \ Q \ast R}
\]

Note that this reduces the amount of work needed for verification by a great factor, because \(R\) can be of any form.

5.3 Concurrency

Reynolds insight of separating conjunctions, described in [10], and the idea of tight specification of Ishtiaq and O’Hearn [31] form the basis of Separation Logic. Bornat’s extension made it possible to reason much clearer with the frame rule. This basis is not sufficient to tackle our verification problem. Back to that, our problem requires us to be able to deal with concurrency. The need for parallel composition of commands, \(C_1 \parallel C_2\), is evident. O’Hearn also developed a form of the logic for dealing with concurrent programs [19]. Separation logic is very well suited to handle concurrency. The key idea is that threads (processes) can only access disjoint resources, and then we can verify the different processes as disjoint sequential programs.

The definition of the (disjoint) parallel composition rule is as follows.

\[
\frac{P_1 \ C_1 \ Q_1 \ \ \ \ P_2 \ C_2 \ Q_2}{P_1 \ast P_2 \ C_1 \parallel C_2 \ (Q_1 \ast Q_2)} \quad \text{if } C_i \text{ does not modify any free variable in } P_j, C_j, Q_j \text{ for } i \neq j
\]

This rule says that if two threads have disjoint memory requirements, they can execute safely in parallel, and the postcondition is simply the composition of the two threads postconditions. We think of the memory location changed by the process \(C_i\), which are specified in \(P_i\), as being “owned” by process \(C_i\). The other process can not interfere due to the side requirement of the parallel composition. In listing 12 a disjoint concurrency example is given, original presented in [19].

Listing 12: Separation Logic disjoint concurrency example

<table>
<thead>
<tr>
<th>Line</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{ x \mapsto 3 \ast y \mapsto 4 }</td>
<td>{ x \mapsto 3 }</td>
<td>{ y \mapsto 4 }</td>
<td>x = 4; \parallel y = 5;</td>
<td>{ x \mapsto 4 \ast y \mapsto 5 }</td>
</tr>
</tbody>
</table>

---

The importance of this paper is huge, in 2011 ACM SIGPLAN awarded the 10-Year 'Most Influential POPL Paper’ award to Ishtiaq and O’Hearn for their paper they presented 10 years ago on POPL, it turned out to be ground-breaking work on Separation Logic.
This works well for verifying the rather trivial case of parallel programs without any shared variables, but how about communication between processes? It has been stated that a thread owns a heap or part of a heap, but now heaps can also be associated with a resource identifier. When a heap, e.g. a variable, is associated with a variable, it is described by an invariant. With the rule \( \text{resource } r \ (I_r) \) we establish that the variables described in \( I_r \) are owned by the resource \( r \). The Conditional Critical Region rule, \( \text{with } r \ \text{when } G \ \text{do } C \ \text{od} \), gives mutually exclusive access to the heap described invariant \( I_r \). The execution of the command goes as follows, the thread waits for exclusive ownership of the resource identifier \( r \). When the thread gets the exclusive ownership the guard \( G \) is evaluated. If \( G \) is false then the thread releases ownership of \( r \) and starts executing the command again. If \( G \) is true, then the thread executes \( C \). Ownership of \( r \) is released when the execution of \( C \) terminates. Both \( C \) and \( G \) may refer to both the threads and the resource identifier resources.

\[
\{(P * I_r) \land G\}C\{Q * I_r\}

(P) \ \text{with } r \ \text{when } G \ \text{do } C \ \text{od} \{Q\}
\]

Two important remarks have to be made here. First the rules imply partial correctness; the rule does not guarantee termination of the program, because we might never get into a situation that we have exclusive access to \( r \). Second, the rule is only sound if \( P * I_r, Q * I_r \) and \( I_r \) are precise. This requirement reflects Dijkstra’s principle and slogan “Well specified processes mind their own business” [19].

In listing 13 we give an example where a shared resource is used, adapted from [19]. The buffer resource in line 5, takes both the variable full and \( c \), they are not available any more until the ownership of \( \text{buf} \) is taken and the guard is evaluated to true, this happens at line 9. After this the \( \text{RI} \) is available again, and because it is known what the value of full is it is known which part of the resource invariant is true.

Listing 13: Shared variable via resource example

```plaintext
{emp}
local full = false, c;
{full ⇔ false * emp}
{RI ≡ (¬full ∧ emp) ∨ (full ∧ c ⇔ _)}
resource buf(c, full)
{emp}

local m = "msg";
{m ⇔ "msg"}
with buf when ¬full do
{RI * m ⇔ "msg"}
c = m;
{full ⇔ false * c ⇔ "msg" ∧ c = m}
full = true;
{full ⇔ true * c ⇔ _}
{RI}
with buf when full do
{RI * n ⇔ _}
{full ⇔ true * c ⇔ _ * n ⇔ _}
n = c;
{full ⇔ true * n ⇔ _ ∧ c = n}
full = false;
{full ⇔ false * n ⇔ _}
{RI}
od;
{RI}

full = true;
full = false;
```

5.4 Semaphores

A way to implement Conditional Critical Regions is by protecting the region with a semaphore. Semaphores were invented to be guards on ‘critical sections’ of code.

37
A common synchronization construct is the lock, we can model a lock with a semaphore. Acquiring a lock is used to guard the entrance to the section, and releasing it to exit the section. A semaphore can be set to allow a number of threads into the critical section, sometimes we call a binary semaphore, a semaphore of one, a lock. Lock acquiring can be accomplished with the `with r when G do c od` rule, as `with s when s > 0 do s := s − 1 od`; and lock releasing as `with s when true do s := s + 1 od`. The semaphore name `s` doubles as a resource name and a program variable.

In listing 14 we give an example where locks are used, adapted from [19]. P and V are used for the acquiring and releasing of a lock; their definitions are at the bottom of the listing.

### Listing 14: Semaphore example

```plaintext
{emp}
local free = 1, busy = 0, c;
{free := 1 * busy := 0 * c := _}
{RIfree ≡ (free = 0 ∧ emp) ∨ (free = 1 ∧ c := _) }
{Rlibusy ≡ (busy = 0 ∧ emp) ∨ (busy = 1 ∧ c := _) }
resource free(free), busy(busy);
{emp}

while true do
{emp} while true do
{emp}
local m = produce();
{m := _}
P(baby) {c := _}
P(free); local n = c; {c := _ ∧ n = c}
{c := _ * m := _} ||
c := m; V(free) {emp}
{c := _ ∧ m := c} consume(n); {emp}
V(baby); {emp}
{emp}
od; od;
{emp ∧ ¬true} {emp ∧ ¬true}
{RIfree * Rlibusy * false}
{false}

P(s) ≡ with s when s > 0 do s := s − 1 od;
V(s) ≡ with s when true do s := s + 1 od;
```

### 5.5 Permissions

Separation logic can now deal with two groups of variables, those are exclusively owned by a thread, which can be read or written, and those in mutual exclusion who are owned by a semaphore (temporary transfer of the ownership from the semaphore to a thread is possible). But there is a third group of variables, those of read only shared owned variables. It is easy to see that shared variables where the threads do not write to, can not lead to a race condition. To deal with those variables in Separation Logic, Boyland suggested a model with fractional permission. Ownership of a heap can be split into a number of fractional permissions, each of which allows a read only read access. Exclusive ownership can be written as $E \mapsto F$, and a fractional permission can be written as $E \mapsto z F$ with $0 < z \leq 1$. Permission of a variable can be cut up into a number of fractions, the sum of those fractions has to be 1. $E \mapsto z F * E' \mapsto z' F'$, if $E = E'$ then they point to the same memory location, thus $F = F'$ and $z + z' \leq 1$.

In listing 15 we give a example where the permission of a variable is split into two read only

---

38
fractions, in this way the variable can be used in two concurrent process, adapted from [19].

5.6 Further reading

The thesis provides a concise description of Separation Logic and a few of its extensions; for further reading there are many papers on this subject, for example [32].

Separation Logic is still actively researched. One of the topics researched is to combine the compositional reasoning with local reasoning [33] [34] [35].

Also the tooling which uses separation logic as a base is actively researched, new experimental tools are emerging [36] [37] [38] [39] [40].
6 A closer look at VeriFast

This section 6.1 first presents the main features of VeriFast some features are easily related to the concepts given in the previous chapter. The VeriFast IDE is presented in the next section, section 6.2 As last we present in section 6.3 a simple verification example; that explains how to verify a program with the necessary specifications.

6.1 VeriFast features

The VeriFast Tool is a program verifier which uses Separation Logic as its basis. It can verify sequential and concurrent C and Java programs. The verifier can check for NullPointerException or ArrayIndexOutOfBoundsException. For concurrent programs it checks that the program does not contain data races (memory safety), and that locks do not create deadlocks. When the verification succeeds and it reports no error, the assertions and method contracts (preconditions and postconditions) are respected in every program execution.

VeriFast is based on modular verification, symbolic execution and Separation Logic. The modular verification of VeriFast is crucial for achieving scalability of verification. Each method is checked in isolation, thus converging all possible traces from its precondition to it postcondition. When verifying a method, all method calls are replaced by the respective method contracts. This means that the precondition is asserted, and then the postcondition is assumed. If there are no verification errors in all methods, then the complete program is error free. In the verification, VeriFast executes the method body symbolically. The symbolic state represents an arbitrary concrete state. It will start with the symbolic state that satisfies the precondition, and checks whether these result in a state that satisfies the postcondition.

During the verification process, VeriFast executes each method symbolically. Symbolic execution is like ordinary, concrete execution, except that symbolic values are used instead of concrete values. A symbolic state represents an arbitrary concrete state that satisfies a certain condition. At the start of a method it satisfies the precondition of that method. VeriFast explores all (feasible) paths through the code of the methods body, from the methods entry point to an exit point of the method, exit points can be the return statement or uncaught exceptions. At method exit points VeriFast verifies that the respective symbolic state satisfies the postcondition to the method.

VeriFast employs Separation Logic as a basis logic, this allows VeriFast to handle aliasing of memory location in a natural manner. The use of the concept of ownership allows VeriFast to reach a conclusion about data races. VeriFast actually enforces correct use of variables; when the program writes to a memory location the program should have the correct permission. Making sure a write can only be done on a memory location, when no other thread can do a read on the location, leads to race condition free programs. An important feature of the approach is that VeriFast breaks down the program memory (heap) into separate chunks, the chunks are passed from method to method during method calls and returns, or distributed between concurrent threads. Within VeriFast it is possible to break down the ownership of a memory location into multiple permissions, or more precisely have multiple read-only permissions to a memory location. The broken down permissions are the so called, fractional permissions. The memory location can in this way be used in differed parts of the code. VeriFast enforces that all fractional permissions to a memory location combined give a full write permission; the sum of the fractions should not exceed one.

A nice feature of VeriFast is the possibility to create custom predicates, with this data abstraction is possible. Specifications can be made more concise and abstract, for example, instead of repeating a partial specifications in multiple preconditions, the repeating part can be written down as a

\[ \text{It is possible to write down fractions greater then one in VeriFast, improper use of this can lead errors in VeriFast (this is a bug).} \]
predicate and the precondition uses only the predicate. Data abstractions like hiding private fields of an object can also be archived with predicates.

VeriFast has also more advanced features like generics, inductive data types, fixpoint functions, specification lemmas and atomic spaces. For those features refer to the VeriFast \cite{27} website, where these are explained in a number of papers.

### 6.2 VeriFast IDE

This section briefly presents the IDE of VeriFast, this IDE is very helpful for the developer. To make it easier for developers to diagnose verification errors, VeriFast has an IDE that supports symbolic debugging. That is, when verification fails, one can inspect the symbolic states encountered during symbolic execution on the path to the failure. A screenshot of the IDE is shown in figure 7.

![Figure 7: VeriFast IDE: debugging a faulty program.](image)
First we shortly describe the other panels we see in the IDE. In the main panel, the editor, we can program the code, left to that we see the local variables together with its symbolic values, the symbolic store. The right bottom panel contains the heap chunks, heap chunks are field chunks or predicate chunks that represent the separated heap parts. This panel is referred to as the symbolic heap. The center bottom panel includes assumptions on the current execution path made over the local variables or heap chunks, or referred to as the path conditions. The bottom left panel shows the steps made by the verifier during the symbolic execution of the current execution path. When a step is selected in the steps panel, VeriFast shows which command is executed in the editor and also the symbolic store, the symbolic heap and the path conditions are updated to that selected state. Another feature of VeriFast is the “run to cursor”, with this a new execution is started running up to the command where the cursor is placed in the main panel. Viewing the state of the program when selecting a step and the possibility to run up to a certain point in the code, makes debugging an easier process.

Notice in figure 7 first of all the red bar in the toolbar, showing the verification error. The red bar indicates that verification has failed, and the message suggests the reason for the failure. This bar turns green if the verification is successful. If verification fails, the location in the program where VeriFast encountered the failure is indicated in the editor by a double red underline. In the example, the postcondition is shown with such a double red underline. This indicates that verification of method deposit failed because, on one of the methods execution paths, VeriFast could not prove the postcondition. The next section explores why this verification fails.

### 6.3 Verification example

A simple example demonstrates the VeriFast specification and verification. This verification is a preparation for understanding the verification of chapter 6. This is a small non-concurrent trivial example of a bank account class. Chapter 6 presents more complex concepts when they are needed in the verification.

There is a VeriFast tutorial available [41], which presents programming concept like loops, arrays, lists, generics, multithreading and specification concepts as predicates, lemmas, inductive datatypes, fixpoint functions, function pointers and permissions. There is also a paper [42] about verifying Java programs, but it covers a limited number of concepts like permissions, data abstraction, inductive data types, fixpoint functions lemmas and inheritance. VeriFast can also be used to verify Java Card applications [43]. Java Card applications are applications to be run securely on smart cards and similar small memory footprint devices.

**Listing 16: Verifast bank example (1).**

```java
// A predicate which represent the data structure Account.
predicate account(Account account, int balance, int limit) =
    account.balance -> balance & limit -> limit &
    limit <= balance; // an additional object invariant stating that the
    balance never drops below the limit

public class Account {
    private int balance = 0;
    private int limit = 0;

    public Account(int limit)
        //@ requires limit <= 0; //precondition
```
In listing 16, we have defined a class `Account` for a small bank example. The class has two fields, one for balance of the account and limit to set the maximum allowed amount that the account can be in depth. The account has one constructor function which sets the amount to zero and the limit to a certain amount. The specifications for this object are given inside a special annotation comment, a single line comment (`//@`) or a multi-line comment (`/*@@*/`). We represent the object in the verification by a single predicate, `account`. Each field of the class gets its own field predicate or heap chunk. The account predicate absorbs those field properties into the single account predicate. The account predicate also gives us the object invariant that the balance is never below the limit of account. Defining the object with custom predicates, instead of reasoning with the generated fields predicates, gives us the property of data abstraction. We describe the structure, and later also other properties, with a single predicate. This predicate can be used by specification of other objects, if the internal structure changes account we only need to change the predicate, instead of all places where the field predicate could have been used.

The constructor method has a precondition. This is shown in the first annotation under the constructor head. The specification forces to supply a limit parameter which is at most zero. The postcondition guarantees that we end up with a predicate for this account with a balance of zero and provided limit. The verifier executes the body and sets the field to zero and limit. After that it closes the account predicate, this means that it takes the field predicates and puts them in the account predicate. This predicate is put into the heap store. If we change the zero, for example in a one, of the balance on one of the three places, VeriFast will report a verification error. Depending on the place of the change, it can not prove the predicate "0==1" or find the predicate "account(this,0,limit)" in the heap store, it has only the produced predicate "account(this,1,limit)".

In listing 17, we extend the `Account` class with the methods `getBalance` which returns the current balance of the account, and `deposit` which adds a certain amount to the current balance. Both methods require the account predicate in the heap store, `deposit` has a second precondition that the deposit amount is not negative. If we would not require this, the condition "limit <= balance" in the account predicate can not be proven. Depending on the place of the change, the precondition can not be proven if the field is changed in one of the three places. VeriFast will report a verification error. Defining the object with custom predicates, instead of the bounded specification variables or programming variables. In that case the variables need to have the correct value, in that class case to zero and the limit to a certain amount.

The return keyword in the postcondition represents the return value of the method. We describe the structure, and later also other properties, with a single predicate. This predicate can be used by specification of other objects, if the internal structure changes account we only need to change the predicate, instead of all places where the field predicate could have been used.
As we already stated the verification in the screenshot of figure 7 failed; if we take another look at the code in the screenshot we will see why. The postcondition could not be reached in one of the execution paths; luckily for us this method has only one execution path. The postcondition of the deposit method stated that it requires a heap chunk where the balance of the account has been increased with value of the amount parameter. In the symbolic heap, there is a heap junk of the account predicate which represents this object in the memory space. The second parameter of this predicate represents the value of balance. The balance value has a symbolic value of the balance when the method started plus twice the value of the amount parameter. This is indeed not what the postcondition of the method requires, therefore we need to look at the programming code. In the code we see that the balance is increased with the doubled value of the amount parameter. Correcting this programming error would indeed result in an error free program, and the verification would succeed.

We extend the Account class for a second time in listing 18. We add a method for transfer an amount to another account. The transfer can only take place if the end amount of the account does not drop below his limit. While the postcondition of this method is defined as one conjunctive predicate, the method body uses another manner and even another method to make the same conclusion.

Listing 18: Verifast bank example (3).
This demonstrates the first basic steps of verification in VeriFast, chapter will show a more daring verification of multithreaded code.
7 Verification of the communication channels

This section presents the verification of an essential, representative part of the Toosm framework: the communication channel in a multiple sender/receiver setting. Using VeriFast, we verify the absence of race conditions and deadlocks. These are properties that VeriFast is known to be well-suited for. Beside these two properties we specify and verify some functional properties - this, in combination with concurrency, is a more novel application of VeriFast.

We consider a single communication channel with on one side of the channel \( n \) sender parties and on the other side \( m \) receiver parties. The parallelism is in the multi-party use of the channels: each party will execute in his own thread. The channels themselves are independent of one another, hence treating a single channel suffices to verify the essential properties. Each sender party will send \( k \) messages through the communication channel, and each receiver party will receive \( l \) messages. Prove is that the program does not suffer from deadlocks, that there are no race conditions, that after execution \( n \times k \) messages are sent and \( m \times l \) messages are received, that not more messages can be received than have been sent, that all sent messages will be received and finally that the number of messages in the communication channel is the number of sent messages minus the number of received messages. The sender has only one responsibility, it keeps trying to send a message until it succeeds, this will be done until it has send \( k \) messages. The receiver operates in a similar manner, it keeps trying to receive a message until it succeeds, this will be done until it has received \( l \) messages.

The verification is done incrementally and modular. We essentially follow the steps-outlined from the introduction. The approach outlined is to consider a channel in isolation, then add one sender and one receiver using the channel, and finally treat the multiple user situation.

In each step (except for the last one, where just VeriFast Annotation is added), we first present a description in a mix of pseudo-code and mathematical notation, of what the behavior of the part of the generic code should be: a general description. This easy to read description should lean credibility to the claim that indeed the intuitively required behavior is captured. Second, we provide the corresponding specification in the Separation Logic formalism that VeriFast uses and the Java code that is intended to satisfy this specification. This is a transformation from one formalism to another, so in principle amenable to formal verification. This however is not the aim of this research, so we again only informally argue this transformation. Our concern is, to verify that the Java code satisfies the VeriFast specification - this usually requires extra annotation, which is the third thing we provide at each step.

We encountered some obstacles that prompted us to add additional steps to explain how we dealt with these, see figure 8.

First we introduce a abstracted version of the communication channel in section \( 7.1 \). In section \( 7.2 \) we start with the communication channel class verification (ia). The specification and annotation of the channel in section \( 7.2 \) appeared unsuitable for verification of a program with senders and receivers, therefore we adapt the specification and annotation of the channel in section \( 7.3 \) (ib). In section \( 7.4 \) we give the first client program that uses the communication channel, with only one sender and receiver thread (ii). This client program is extended to support multiple senders and receivers in section \( 7.5 \) (iii)). Here we consider functional properties that hold after thread termination. Section \( 7.6 \) (iiib) we consider properties that hold during thread execution.

7.1 Abstraction

Our starting point is the source code of the communication channel in the C# language of the Toosm framework. In order to apply VeriFast, which operates on Java code, the C# code is translated to equivalent Java code. To ease the verification of the communication channels, we abstract from some aspects of the channel in this translation.
The first abstraction is that messages are objects in TOOSM, we are interested in the general working of the channel rather than in the message content. Hence, messages are identified by a string value instead of a complete object with a name and parameters. Two messages with the same string value are considered to be the same message. Thus, instead of sending an object with a serialized string as ‘A(true, 5, “a few words”)’ we just send ‘A’. As we are only concerned with the sending and receiving methods/properties of the communication channel, the peek method for peeking at the content of the buffer in the channel is removed. Finally, as we only look at the communication channel and not at the other artifacts of the framework, we model the endpoints of the communication channel differently. Instead of having state machines that can do a send or receive calls on the communication channel, the end points can be of any type of class, a class can perform send as well as receive calls on the communication channel.

### 7.2 Communication Channel

We start out by considering the communication channel by itself, without senders and receivers. The channel is specified in an intuitively easy to understand mix of mathematical notation and pseudo code, then the specification is translated to a VeriFast specification of Java code, and finally the annotation is added to enable Verifast to perform the verification.

#### 7.2.1 Channel specifications

This section describes the intended behavior of a single communication channel in an intuitive manner: a mix of mathematical specification and pseudo-code, presented in Channel specifications.
This mathematical specification captures the following requirements.

A channel should be able to hold a number of messages. It also should provide a way to add and remove messages: a send method and receive method. There should be no message losses, and the messages should be received in the same order as they are sent. The last is properly debatable in the real world, but in the setting of a single memory device this is reasonable. The data structure to store the messages is chosen to be a list. Messages are only added at the end, and removed from the front, thus the list is in fact used as a queue. In line with SLCO, message storage is limited. If the queue is full the send method will return this status, as a Boolean value (false), and it is up to the sender party to handle this situation. The receive method will return nothing if the queue is empty.

The channel will be used in a multi threaded environment, therefore the send and the receive operation are required to be done in atomic actions. For this purpose, a semaphore is included in the pseudo code: firstly to ensure the atomicity, secondly to make the access to the itemList conditional on having the semaphore.

The communication channel can be viewed as a thread-safe limited non-blocking queue.

<table>
<thead>
<tr>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemList : List&lt;String&gt; = []</td>
</tr>
<tr>
<td>queueMaxSize : integer</td>
</tr>
<tr>
<td>sema : Semaphore = 1</td>
</tr>
<tr>
<td>#itemList ≤ queueMaxSize</td>
</tr>
<tr>
<td>Send ( msg : String ) : bool</td>
</tr>
<tr>
<td>acquire sema</td>
</tr>
<tr>
<td>#itemList &lt; queueLength → itemList = itemList ++ [msg];</td>
</tr>
<tr>
<td>result = true</td>
</tr>
<tr>
<td>#itemList ≥ queueLength → result = false</td>
</tr>
<tr>
<td>release sema</td>
</tr>
<tr>
<td>Receive ( ) : String</td>
</tr>
<tr>
<td>acquire sema</td>
</tr>
<tr>
<td>itemList == [] → result = null</td>
</tr>
<tr>
<td>itemList == (x : xs) → itemList := xs;</td>
</tr>
<tr>
<td>result = x</td>
</tr>
<tr>
<td>release sema</td>
</tr>
</tbody>
</table>

General description Channel

To be able to argue the transformation of the mathematical specification into a VeriFast specification and Java code we list the verification properties in words:

1. Deadlock and race free.
2. Send/receive are the only operations that access itemList.
3. Send/receive are atomic.
4. Functionality of send/receive.
5. Numerical requirements on messages numbers.

*The queue datatype is not supported in VeriFast, support for this can be made, but we have chosen not to do this.*
7.2.2 Channel VeriFast specifications

The next step is to take the mathematical specification of the channel and transform it to a form that VeriFast can handle.

Rather than providing a formalism-to-formalism translation (which is not the focus of this research), we consider how the list of properties is reflected in the formalization.

For readability, we have split the code over two listings, the first one containing the set-up of the channel, the second one the send and receive methods. Note, that some specification details in the first listing can only be appreciated in combination with (the explanations of) the second one.

1. The VeriFast approach does not explicitly specify absence of race conditions and deadlock, but guarantees this indirectly for every verified program.

   Race conditions can only occur when a variable is referred to from more than one place: aliasing. Aliasing of a variable means shared ownership. VeriFast requires that this is expressed with fractional permissions in the specification. VeriFast furthermore requires that fractional permissions to the same memory location sum up to at most one. VeriFast checks that writing to a variable only occurs when the writer has permission 1, thus avoiding race conditions. We use fractional permissions on line 3, 4 and 14 in listing 19. The variable queueMaxSize and the ghost variable itemsGhost are each split into two fractions, both have a 1/2 fraction, for both variables the sum of the fractions equals to one.

   Deadlock freedom is achieved by the structure of the program, we do not have to verify this. Because we have only one binary semaphore it is impossible to produce deadlocks: a deadlock requires to have at least two locks/semaphores. In VeriFast it is possible to prove deadlock freedom in programs with multiple locks . Lock ordering is vital for this. Nested locks must always be obtained in the same order. This prevents the embrace deadlock. With additional specification this lock order can be incorporate into the specifications, and VeriFast tests if the acquiring of nested locks occurs in the correct order, thus preventing deadlock in the code.

2. The send and receive operations on the itemList should be the only ones that access the list.

   We formalize access to the itemList: itemList is added to the invariant of the semaphore, see line 3 in listing 19: channel.itemList |− > ?itemList This specifies that the implementation of the send has to be such, that the semaphore is acquired when the send accesses the itemList.

   In the implementation this is indeed the case: sem.acquire(), line 10 and 28 in listing 20. The constructor method is an exception to this, and we can access the itemList there because the semaphore, and therefore the invariant, is not yet created.

3. The atomicity described in the general specification is achieved in the implementation by having the same acquire/release structure.

4. To specify the functionality of send, we need to refer to the itemList. However, essential in Separation Logic is that such a list has only one owner, in this case the semaphore is the owner, and hence the list cannot be referred to at the call of send: the semaphore is only acquired inside send.

   Therefore a trick is necessary: a ghost variable itemsGhost is introduced, its permission is split into two fractions (1/2, 1/2), one of which is added to the invariant and the other

---

The support for proving the absence of deadlocks is there only for C programs, but the VeriFast libraries for Java can be extended with the same support.
used to specify the functionality of send. The invariant states that both lists have the same content. Adding one fraction to the invariant and requiring content equality is on line 3: `itemList.List(?items) &*\* [1/2]channel.itemsGhost |− > ?ghostItems &*\* items == ghostItems` in listing 19.

Before stating the functionality of the send in the implementation, we introduce two predicates, `Channel` and `ChannelState`. Those two predicates represent this class in the verification. See line 13 and 14 in listing 19 The `Channel` predicate holds the semaphore, and therefore also the semaphore invariant, while the `ChannelState` predicate holds a fraction of the `itemGhost` variable and `queueMaxSize`.

We use the introduced predicates to specify the functionality of the send in the implementation. On line 5 of listing 20 we require both predicates, we bind the `itemsGhost` and `queueMaxSize` of the `ChannelState` to `?items` and `?qms`. Those two bind variables can be used to implement the functionality specification of the send method in the postcondition, see lines 6 and 7 in listing 20. The Boolean result indicating that if the buffer is not full, message adding can be done, is specified as: `result == (length(items) < qms)`. The specification code that states that a message is appended to the list if there is room for it is specified as: `ChannelState(length(items) < qms ? append(items, cons(msg, nil)) : items, qms)`.

5. The last property to be translated into VeriFast notation is the `#itemList <= queueMaxSize`. As this is an invariant over the `itemList`, it can be added to the existing semaphore invariant. This is expressed as a read-fraction of the `queueMaxSize`, and a function that determines the number of messages in a list, the length function. This specification is written at line 4: `[1/2]channel.queueMaxSize |− ?qms &*\* length(items) <= qms` in listing 19.

The specification and the channel class without the methods are given in listing 19:

```
Listing 19: Channel class setup

*/@
predicate ctor channel_sema_inv(Channel channel) =
    channel.itemList |− > ?itemList &*\* itemList.List(?items) &*\* [1/2]channel.
    itemsGhost |− > ?ghostItems &*\* items == ghostItems &*\*
    [1/2]channel.queueMaxSize |− > ?qms &*\* length(items) <= qms;

/*@ public final class Channel {
    //.@ list<Object> itemsGhost;
    List itemList;
    Semaphore sema;
    int queueMaxSize;

    //@ predicate Channel() = sema |− > ?sema &*\* [_.]sema.Semaphore(  
    channel_sema_inv(this));
    //@ predicate ChannelState(list<Object> items, int qms) = [1/2]itemsGhost
    //.@ requires true;
    //.@ ensures Channel() &*\* ChannelState(nil, _);
    {  
        itemList = new ArrayList();
        sema = new Semaphore(1);
    }

    // ...
```

50
The specification of the methods and their bodies are given in listing 20.

Listing 20: Channel class specification

```java
public final class Channel {
    // ...

    boolean send(String msg)
    //@ requires Channel() & ChannelState(?items, ?qms);
    //@ ensures Channel() & result == (length(items) < qms ? append(items, cons(msg, nil)) : items , qms);
    {
        sema.acquire();
        boolean result;
        if (itemList.size() < queueMaxSize) {
            itemList.add(msg);
            result = true;
        } else {
            result = false;
        }
        sema.release();
        return result;
    }

    Object receive()
    //@ requires Channel() & ChannelState(?items, ?qms);
    //@ ensures Channel() & ChannelState(tail(items), qms) & result == (items != nil ? head(items) : null);
    {
        sema.acquire();
        String result;
        if (itemList.size() == 0) {
            result = null;
        } else {
            result = (String)itemList.remove(0);
        }
        sema.release();
        return result;
    }
}
```

7.2.3 Channel VeriFast annotation

VeriFast can not verify whether or not the postcondition holds from just the precondition and the source code: it requires additional annotations in the body of the method to reach a conclusion. In listing 21 we give the class including its specifications and annotations.

To explain the ideas behind the extra annotation we make a few remarks. The first one is that most ghost statements are open or close statements. A predicate is a named and parameterized assertion, enabling more concise contracts and data abstraction. Predicates can be be used to bundle several heap chunks (see section 6.1 into one heap chunks). To open such a bundle,
the ghost statement \texttt{open} is used. This statement consumes the abstract predicate and then produces its body, e.g., it creates the separate heap chunks from a bundle. The \texttt{close} ghost statement consumes the predicates body and then produces the abstract predicate, e.g., it creates the bundled heap chunk from the separate heap chunks.

Secondly, a ghost variable can not be updated by the regular source code, hence the three ghost field updates that appear in lines 10, 34 and 62.

The third remark is that there are some special ghost statements pertaining to the semaphore. The semaphore creation requires a special predicate that states that the invariant of the semaphore can be used, in our case, one time. The \texttt{one\_time} ghost statement, on line 12, consumes the semaphore invariant and generates this special predicate.

After semaphore creation, in the constructor, the semaphore usually is acquired there - in our case, this happens in a different method, the send method. Intuitively, to obtain the semaphore’s invariant in this method we need to “leak” the semaphore handle predicate: see line 14. The acquire in the send method, line 28, uses the semaphore handle predicate, the one that was leaked in the constructor, this handle is produced by the ghost statement on line 27.

The last remark is about line 64, the switch state introduced there is a VeriFast technicality, having to do with taking the tail of an empty list, which we do not further elaborate.

Listing 21: Channel annotation

```java
public final class Channel {
    // ...

    public Channel()
        //@ requires true;
        //@ ensures Channel() && ChannelState(nil, _);
    {
        itemList = new ArrayList();
        //queueMaxSize = 2;
        //@ itemsGhost = nil;
        //@ close channel_sema_inv(this)();
        //@ one_time(channel_sema_inv(this));
        sema = new Semaphore(1);
        //@ sema.leakHandle();
        //@ close Channel();
        //@ close ChannelState(nil, _);
    }

    boolean send(String msg)
        //@ requires Channel() && ChannelState(?items, ?qms);
        //@ ensures Channel() && result == (length(items) < qms) &&
                ChannelState(length(items) < qms ? append(items, cons(msg, nil)) : items, qms);
        @*
    {
        //@ open Channel();
        //@ open ChannelState(items, qms);
        //@ sema.makeHandle();
        sema.acquire();
        //@ open channel_sema_inv(this)();

        boolean result;
        if (itemList.size() < queueMaxSize) {
            itemList.add(msg);
```
With this annotation, the Channel class is verified by VeriFast as conforming to the specification. This means that no race conditions or deadlocks occur and that the methods respect their contracts.

Thus the Send method adds the msg parameter to the queue of messages if the queue is not full, and the Receive methods removes the first element of the message queue and returns this element if the message queue is not empty. Another important property that is proven is that the message queue, the itemList, is “protected” by the semaphore, meaning that threads who want to write to this variable should have acquired the semaphore, thus ensuring that no racing on this shared variable can occur.

7.3 Communication channel client ready

This first set-up of the communication channel, in section 7.2 was appropriate for stand alone verification of the channel class. This setup can not be used directly in a verification of a program with clients like we want, see 7.3.2. Clients are objects which use the channel class, in our example a program with two end point objects. The changes only affect the VeriFast specification and the
VeriFast annotation: the general specification as well as the Java source code for the channel remain the same. The new VeriFast specification/annotation for use with the clients is called “client ready”.

The preparation of the specification and annotation for use with clients is modular in the sense that the clients are only specified and not implemented: for the verification of the channel this suffices.

The channel specification will now be prepared for modular verification in the context of clients that send and receive messages. This preparation is necessary, because the specification of the channel in isolation does not take into account the setting with clients. VeriFast requires some assumptions about clients.

The ChannelState predicate in the specification of the channel in used in the, as yet unspecified, Client invariant of the Client specification. In the specification of the channel information from the Client invariant, about the queue content, is required. To preserve modularity, we specify (but not implement) four lemma functions in the channel specification, that provide the information from the Client invariant. Note, that these lemma functions are used as assumptions in the channel specification and, when they are implemented in the Client specifications, they have to satisfy these specifications.

Note, that the requirements that are specified about the context that the channel is used in are very mild: they do not specify the clients that use the channel, but only some of the properties they should have to properly function with a channel.

In the next section, 7.4, the client is specified much more precisely.

7.3.1 Channel specifications

The general description, given in 7.2.1 of the communication channel does not need to change.

7.3.2 Channel VeriFast specifications

The source code of the channel class does not need to change. The same holds for the channel invariant, ghost item list, Channel predicate and ChannelState predicate.

The changes required are due to the fact that in the client program there are requirements about the state of the content of the channel during the execution: #queue = #sent − #received. Because of this kind of requirements changes are required to the specification out of section 7.2. The client specification can not do this with the specifications of section 7.2 but it can with the specification of this section. To be able to deal with this kind of client requirements the ChannelState must be added to a client invariant. This is further complicated by the, in itself desirable, compositional approach.

Then two issues are to be dealt with.

First, the client invariant must be shared among threads. In that case, no thread has write permission to objects that are referenced in the client invariant. However the ChannelState needs to be updated, for example in the send method of the channel, so the send method requires write access. This conflicts with the loss of the write permission.

The remedy is the use of atomic spaces. Atomic spaces can be shared within a program, thus also among threads. On items in the atomic space we may (only) apply atomic actions. (Even if the atomic space is read only, we still may update an item in the atomic space with an atomic action.) Atomic actions are defined as lemma functions, with a precondition and a postcondition.
So the client invariant is put in an atomic space. Which is required by the send method. In the send method an atomic action will update the ChannelState in the client invariant.

There is a second problem with adding the ChannelState to a client invariant. This is a consequence of the modular approach, where first the client ready channel is verified without the client implementation. At this moment the knowledge about what this client invariant looks like does not yet exist (this will be shown in the next section, in the client specification). The atomic action in the send method requires possessing this knowledge, in the send method the ChannelState needs to be separated from the rest of the client invariant. This can not be done without this knowledge.

To remedy this, we state the existence of a lemma function that is able to separate the ChannelState from the rest. We emphasize that the rest has a unknown structure. We specify the lemma function with a pre and postcondition, but we do not give the body that does the actual separating (this code is not yet available, we have not yet made a client invariant!). This lemma function can be used to do the separating: remember that in VeriFast function calls asserts the precondition and then assumes the postcondition without executing the body. This body needs to be given in client verification to verify the steps. For similar purposes, the inverse of the separation lemma function is needed.

We start with the second issue. We specify the separation lemma functions, see listing 22, here the two lemma function type declarations are presented: The first is for the separating, the second is for its inverse. The separating predicate, channel sep, requires a inv predicate and a sepPred predicate, the inv is the client invariant. The lemma function will produce the ChannelState and the rest of the invariant. The rest is bundled into a parameterized unsepPred predicate.

```
/*@
typedef lemma void channel.sep (predicate() inv, Channel c, predicate() sepPred, predicate(list<Object>, int) unsepPred());
requires sepPred() & inv();
ensures c.ChannelState(? items, ?qms) & unsepPred(items, qms);
*/

typedef lemma void channel.unsep (predicate() inv, Channel c, predicate() sepPred, predicate(list<Object>, int) unsepPred());
requires unsepPred(? items, ?qms) & c.ChannelState(items, qms);
ensures sepPred() & inv();
@
*/
```

Still in the context of the second issue, we need two other lemma type declarations, beside the separation lemma type declaration and its inverse, namely: a send and receive lemma type declarations. Those lemmas are needed, because of the unsepPred(_,_) predicate. This predicate makes up the rest of the client invariant. This predicate also contains the items (see line 4 of listing 22). The unsepPred predicate is not yet defined which, like the client invariant, is not possible here. We need a way to update the items in the unsepPred predicate. For this we apply the same strategy as for the separating lemma function, by defining the lemma function with its pre and postcondition. In the client verification we need to give the body to verify this step.

In listing 23 the two type declarations for the send and receive lemma function are presented.

```
/*@
typedef lemma void channel.send (predicate() inv, Channel c, predicate(list<Object>, int) unsepPred, String msg, predicate() pre, predicate(boolean) post)(boolean r);
*/
```
The VeriFast concepts used here are atomic spaces and lemma function pointer predicates.

Now with those new lemmas, we can specify the channel; presented in listing 24. The Channel predicate is fractionized because multiple clients will use the channel class. The methods also need the client invariant, which is in the atomic space. In this method we require the separation lemma function, its inverse and the send function. The required specifications can be made because we have defined their types.

Listing 24: Channel specification

```java
public final class Channel {
    // ...
    
    public boolean send(String msg);  
    /**
     * requires
     * [fc] Channel() \&\& [fa] atomic_space(?inv) \&\&
     * is_channel_sep(?sep, inv, this, ?sepPred, ?unsepPred) \&\&
     * is_channel_unsep(?unsep, inv, this, sepPred, unsepPred) \&\&
     * is_channel_send(?sendClientOp, inv, this, sepPred, unsepPred, msg, ?pre, ?post) \&\&
     * sepPred() \&\& pre();
     */
    /\*
     
     * ensures
     * [fc] Channel() \&\& [fa] atomic_space(inv) \&\&
     * sepPred() \&\& post(result);
     */
    
    public String receive();  
    /**
     * requires
     * [fc] Channel() \&\& [fa] atomic_space(?inv) \&\&
     * is_channel_sep(?sep, inv, this, ?sepPred, ?unsepPred) \&\&
     * is_channel_unsep(?unsep, inv, this, sepPred, unsepPred) \&\&
     * is_channel_receive(?receiveClientOp, inv, this, unsepPred, ?pre, ?post) \&\&
     * sepPred() \&\& pre();
     */
    /\*
     
     * ensures
     * [fc] Channel() \&\& [fa] atomic_space(inv) \&\&
     * sepPred() \&\& post(result);
     */
}
```

In this verification we state the functionality requirement of the send method in an indirect way, via the unsepPred predicate and the channel_send and channel_unsep lemma type declarations.
In the channel_send we state the postcondition of the send. The function has a Boolean value \( r \), which is used to indicate if the Java update of the itemList was successful. If so the items in the unsepPred predicate also require an update (see line 4 of 23). The channel_unsep lemma function must be called in the send atomic action, to update the items in the unsepPred predicate. The channel_unsep lemma function requires that unsepPred items and the ChannelState items are the same to be able to produce the client invariant. This requirement in fact checks if we updated the state in the ChannelState (see line 7 of listing 22).

### 7.3.3 Channel VeriFast annotations

Modifications to the annotations in the bodies of the methods are also required. One change already mentioned is the atomic action on the ChannelState. In listing 25 we present\(^\text{10}\) the send method with its corresponding annotation.

The lemma function that performs the updating of the ChannelState is defined at lines 42 to 64. The pre- and postcondition of this lemma function are defined at lines 31 to 40. We bundle all required heap predicates into the precondition on line 21, and split them from the postcondition in line 24. With a special ghost statement we prepare for the atomic action, and the next ghost statement executes the atomic action, see line 22 and 23.

On line 47 we use the sep predicate to separate the ChannelState in the invariant, and on 62 its inverse. Between those two lines we have the full permission of the ChannelState, which we can open, to get the complete permission of itemsGhost. At this point we can do the update if needed, see lines 49 to 60.

<table>
<thead>
<tr>
<th>Listing 25: Channel send annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>28</td>
</tr>
</tbody>
</table>

\(^\text{10}\)The code is slightly rearranged for readability, in the appendix A.1 the original code is presented.
```c
return result;
*/@

predicate P_send() =
  is_channel_sep(sep, inv, this, sepPred, unsepPred) &
  is_channel_unsep(unsep, inv, this, sepPred, unsepPred) &
  is_channel_send(sendClientOp, inv, this, unsepPred, msg, pre, post) &
  [1/2]itemsGhost |-> items;

predicate Q_send() =
  [1/2]itemsGhost |-> (length(items) < qms ? append(items, cons(msg, nil)) : items) &
  sepPred() &
  post(length(items) < qms);

lemma void my_ghost_send_op()
  requires P_send() & inv();
  ensures Q_send() & inv();
{
  open P_send();
  sep();
  open ChannelState(., .);
  if (length(items) < qms)
  {
    itemsGhost = append(items, cons(msg, nil));
    close ChannelState(append(items, cons(msg, nil)), .);
    sendClientOp(true);
  }
  else
  {
    itemsGhost = items;
    close ChannelState(items, .);
    sendClientOp(false);
  }
  unsep();
  close Q_send();
} @*/
```

The quite similar treatment of the receive method is presented in appendix ??.

The channel class is verifiable against its specifications. Therefore there are no race conditions and it is deadlock free. Also proven are the correctness of functional properties like the message sending and receiving and the limited queue size.

### 7.4 Client program

Possessing a specification for the channel which should be usable by clients, a specification of those clients is given. An important issue is the context requirements that were assumed in the verification of the channel have to be satisfied by the clients, i.e., have to be put in the specification of the clients. They were formed by functions with a pre and postcondition, the implementation of those functions should be provided in this verification.

In this verification the specification uses the channel specification given in the previous section. VeriFast only uses the function type declarations, the two channel predicates and the methods
contracts during the verification of the channel.

In the first version of the clients, there will be one sender and one receiver with a channel between them. The sender will send $k$ messages though the channel, and the receiver will receive $k$ messages. The sender and receiver both will do the execution of sending and receiving respectable in their own thread.

We only present the sender side here, the receiver will go in an analogue manner. The complete code for the verification is presented in appendix A.3.

7.4.1 Client specification

A number of properties should hold for the program, which follow from the goal of this iteration of the verification. The properties are:

1. Deadlock and race free.
2. One main program with a channel, one sender and one receiver, which work in their own thread.
3. After executions $k$ messages are sent and $k$ messages are received.
4. The program can not receive more messages than have been sent.
5. The number of messages in the communication channel is the number of sent messages minus the number of received messages.
6. No message loss, all sent messages will be received.

7.4.2 Client VeriFast specification

First the Java code is developed after which the general specifications of the program are taken and transformed into a form that VeriFast can handle. In this section only the sender party is presented; the receiver will operate in a similar manner.

In listing 26 we present the main program class, and in listing 27 we present the sender party. Their implementation is straightforward.

Listing 26: Channel client: program

```java
public class Program {
    public static int messageMaxCount; // k

    public static void main(String[] args) {
        Channel c = new Channel();
        SenderThread sender = new SenderThread(c);
        Thread threadS = new Thread(sender);
        threadS.start();

        ReceiverThread receiver = new ReceiverThread(c);
        Thread threadR = new Thread(receiver);
        threadR.start();

        threadS.join();
        threadR.join();
    }
}
```
We can now transform the general specification of the program into a form that VeriFast can handle. We do so by considering how the list of properties is reflected in the formalization.

1. **Deadlock and race free.**
   For race condition freedom we do not have to make any specification; in this program there is no shared variable to which multiple threads need to write. In the channel class there is such a variable, but the previous verification verified the use of it. Because there are no locks and semaphores in the client code, there can be no deadlocks.

2. **One main program with a channel and one sender and one receiver, which work in their own thread.**
   This specification holds because of the structure of the program. The main program creates a channel class and two threads that start the execution of the `SenderThread` and the `ReceiverThread` class. The `SenderThread` class has to implement the `Runnable` interface and implement a method `run` to be executable by a newly created thread.

3. **After executions \( k \) messages are sent and \( k \) messages are received.**
   To prove this, we introduce two new ghost variables for counting the number of messages (lines 2 and 3 of listing [28]). In the precondition of the main method we bound the `messageMaxCount` variable to the specification variable `mmc`, and have an additional requirement that they are at least 1: \( \text{Program} \cdot \text{messageMaxCount}(\text{?mmc}) \land \text{?mmc} < \text{mmc} \). In the post we bound the `sendCount` and `receiveCount` respectively to `sc` and `rc`, and check if they are equal to `mmc`: \( \text{[\_]Program} \cdot \text{sendCount}(\text{?sc}) \land [\_]\text{Program} \cdot \text{receiveCount}(\text{?rc}) \land \text{?mmc == sc \land ?mmc == rc} \).

Now it is up to the main method to achieve the postcondition. Shown in the annotation. Because we know the sending is not done in the main method, but in a separate thread, the `SenderThread` class. We have to specify the postcondition of the class. We do this by specifying two predicates in the `SenderThread` class: a `pre` for the condition before starting the thread, and a `post` for the condition after the thread has executed, see lines 4 and 5 of listing [29]. The constructor method has to end in a state that complies to the precondition.
4. *The program can not receive more messages than the program has sent.*

This is required to hold during the execution, thus this needs to be incorporated into the client invariant, see lines 6, 7 and 11 of listing 30. Fractions are used here because the other halves are used to specify the functionality in the pre and post predicates of the SenderThread.

This invariant also enforce that the send count is from 0 up to and including the `messagesMaxCount`, see lines 6 to 10 of listing 30.

5. *The number of messages in the communication channel is the number of sent messages minus the number of received messages.*

This requirement is slightly more difficult; it tells something about the content of the channel. Luckily we have foreseen this\(^{11}\), and we may use it in a client invariant because we have already taken measures to be able to separate this predicate from the rest of the invariant. This is needed for the send operation in the channel. We can add the `ChannelState` predicate to the client invariant, and use this predicate to reason about the state of the channel. This can be seen on lines 5 and 11 of listing 30. Line 5 states that there are no empty items in the `itemList`.

6. *No message loss, all sent messages will be received.*

This last requirement is achieved because the number of messages sent will equal the number received, and if all messages are received, this is as much as are sent, and we can conclude that no messages were lost.

A few more specification lines of listing 30 must be explained. Section 7.3 stated the need for a predicate to hold the residue of the separation of the `ChannelState` from the client invariant. This residue predicate, called `my_unsep_pred`, is seen on lines 16 to 24, and is exactly the same as `client_inv` except for the `ChannelState`. A second requirement derived from the channel specification is the implementation of the separating lemma function and its inverse. Those lemma functions were defined in the channel verification, and where assumed during the verification of channel. To be able to prove the correctness of those assumption, it is necessary to provide the body of the functions, with their contract. VeriFast will verify the body against the function contract. If VeriFast is able to verify this, it means that the assumption we did in the channel verification is correct. The function for the separating and its inverse are given on line 26 to 42.

---

Listing 28: Channel client: program specification

```java
public class Program {
    //@ static int sendCount;
    //@ static int receiveCount;
    public static int messageMaxCount; // k

    public static void main(String[] args)
        //@ requires class_init_token(Program.class) && Program_messageMaxCount(? mmc) && 0 < mmc;
        //@ ensures Program_messageMaxCount(mmc) && (_ | Program_sendCount(?sc) && (_ | Program_receiveCount(?rc) && mmc == sc && mmc == rc);
    {
        // ...
    }
}
```

\(^{11}\) With the help of Bart Jacobs.
Listing 29: Channel client: SenderThread class specification

```java
public class SenderThread implements Runnable {

    Channel c;

    //@ predicate pre() = this.c |-> c.Channel() &* [1/2]
    // atomic_space(client_inv(c)) &* [1/2] Program_sendCount(0) &* [1/2] Program_messageMaxCount(?mmc) &* 0 <= mmc;

    SenderThread(Channel c)
    //@ requires [1/2] atomic_space(client_inv(c)) &* [1/2] Program_sendCount(0) &* [1/2] Program_messageMaxCount(?mmc) &* 0 <= mmc;
    //@ ensures pre();
    {
        // ...
    }

    public void run()
    //@ requires pre();
    //@ ensures post();
    {
        // ...
    }
}
```

Listing 30: Channel client: client invariant and lemma functions

```java
/*@ */
fixpoint boolean non_null(Object o) { return o != null; }

_predicate_ctor client_inv(Channel c)() =
    c.ChannelState(?items , ?qms) &* & forall(items , non_null) == true &* &
    [1/2] Program_sendCount(?sc) &* &
    [1/2] Program_receiveCount(?rc) &* &
    [1/3] Program_messageMaxCount(?mmc) &* &
    0 <= sc &* & sc <= mmc &* &
    0 <= rc &* & rc <= mmc &* &
    length(items) == sc - rc &* &
    rc <= sc ;

_predicate_ctor my_unsep_pred(Channel c)(list<Object> items, int qms) =
forall(items , non_null) == true &* &
[1/2] Program_sendCount(?sc) &* &
[1/2] Program_receiveCount(?rc) &* &
[1/3] Program_messageMaxCount(?mmc) &* &
0 <= sc &* & sc <= mmc &* &
0 <= rc &* & rc <= mmc &* &
length(items) == sc - rc &* &
rc <= sc ;

lemma void my_sep()
requires exists<Channel>(?c) &* & my_unsep_pred() &* & client_inv(c)();
```
ensures c.ChannelState(?items, ?qms) &\& my_unsep_pred(c)(items, qms);
{
    open client_inv(c());
    assert c.ChannelState(?items, ?qms);
    close my_unsep_pred(c)(items, qms);
}

lemma void my_unsep()
requires exists<Channel>(?c) &\& my_unsep_pred(c)(?items, ?qms) &\& c.
    ChannelState(items, qms);
ensures my_sep_pred() &\& client_inv(c());
{
    open my_unsep_pred(c)(_, _);
    close client_inv(c());
    close my_sep_pred();
}
@*/

7.4.3 Client VeriFast annotation

For VeriFast to be able to conclude that the postcondition is reachable from its precondition it
requires additional annotation of the code.

It is important to note: the source code for threads work as is for Java, but VeriFast can not work
with it as it is. The source code for thread starting and joining needs changes. In the specifications
of the thread class of VeriFast the join method is not even defined. We will use an additional class,
the JoinableRunnable class, for the thread starting and joining. This class can join threads and
it has the capability to catch exceptions in threads, when an exception occurs in a thread it is
stored in a field of the JoinableRunnable class. In our verification this last capability is not used.
The JoinableRunnable class is supplied with VeriFast in the example folder.12

The source code changes are small, therefore the changes to the code are given together with the
annotations, see listing 31.

Listing 31: Channel client: program annotation

```java
public class Program {
    //@ static int sendCount;
    //@ static int receiveCount;
    public static int messageMaxCount; // k

    public static void main(String[] args)
    //@ requires class_init_token(Program.class) &\& Program_messageMaxCount(? mmc) &\& 0 < mmc;
    //@ ensures Program_messageMaxCount(mmc) &\& [_]Program_sendCount(?sc) &\&
    // [_]Program_receiveCount(?rc) &\& mmc == sc &\& mmc == rc;
    {
        //@ init_class();
        Channel c = new Channel();
        //@ close client_inv(c());
        //@ create_atomic_space(client_inv(c));
        SenderThread sender = new SenderThread(c);
```
The `SenderThread` class contains two loops. In VeriFast each loop requires a loop invariant. In which all required heap chunks need to be specified (lines 11 and 14 of listing 32).

The inner loop body has additional annotations: the first two ghost statements create special function pointers, see line 16 and 17 of listing 32. We have defined their type declaration in the channel verification, listing 22, and the implementation in listing 30. The ghost statement on line 19 in listing 32 also creates a function pointer predicate; the send was also declared in the channel verification, listing 25, but its implementation is on lines 33 to 43. The pre and postcondition of this atomic send action are specified as two predicates, \( P_{\text{clientSend}} \) and \( Q_{\text{clientSend}} \), see lines 30 and 31. The my\_clientSend lemma function provides the stated effects of the send operation on the client state, in this case it is the increment of the sendCounter. With this function an even more important property is proven, the send lemma function that was assumed in the channel verification. For the same reason as the separating function, if VeriFast is able to prove this lemma function, then the assumption made what this function could accomplish is proven. The same holds for the not given receiver lemma function.

The `senderThread` class is presented in listing 32; the receiver class is presented in the appendix A.3. For readability the code is rearranged, the last specification part (lines 29 to 44) originally belongs between lines 15 and 16.
The complete program with the specification and annotation is verifiable with VeriFast. Again this means no race conditions, no deadlocks and the code meets the specification.

A second conclusion that can be made from this verification is the correctness of the verification which was given in 7.3. There four functions where assumed in the channel specification. The functions where used in that verification, what they accomplish is specified via a pre and post-condition. The correctness of those functions is verified in this verification by presenting their implementation and verifying the implementation against its pre and postcondition.

There is a seemingly circular proof in the modular approach presented here. The proof in section 7.3 of the channel has made assumptions during the validation, the function declarations. Those assumptions are proven in this section, thus the channel verification depends on the client verification and the client verification depends on the channel verification. This kind of proofs seem to
have circular reasoning, which is harmful because it could result in proving unsound results. In our proof this is not the case: there is a bottom case which does not depend on any assumption, thereby allowing a well-founded proof. In our case this are the lemma function implementations, they can be verified without any assumptions. Stating it differently, they do not depend on the (verified) specification of the channel, the only requirement is that the pre and postconditions match. A similar kind of proof is explained in the paper [46], with a machine-checked proof for the correctness of this proof strategy. This argumentation holds also for the other iterations of the verification.

7.5 Client program with multiple senders and receivers

Having only one party on either side is a very limited the use of the channel. The channel should behave correctly in a setting with multiple parties on either side, this will be the next challenge in the verification. Note that we do not have to change the channel, only the client program. The requirement we want to verify is that if we have \( n \) senders and \( m \) receivers, and a sender sends \( k \) messages and a receiver receives \( l \) messages. For this program we will not have requirement that states something during the execution of threads, because this kind of requirements make the specification much harder. In the next section, this kind of requirements is added to the requirement of this section.

7.5.1 Client specification

A number of properties should hold for the program; they follow out of the goal of this iteration of the verification. The properties are:

1. Deadlock and race free.
2. One main program with a channel and \( n \) senders and \( m \) receivers, each working in their own thread.
3. After executions \( n \cdot k \) messages are sent and \( m \cdot l \) messages are received.

7.5.2 Client VeriFast specification

First we have to make the Java code after which we take general specifications of the program and transform it into a form that VeriFast can handle. In this section only the sender party is presented; the receiver is omitted, but could be presented in a similar manner. The complete code for the verification is presented in appendix A.4.

In listing 33 we present the main program class, and in listing 34 we present the sender party. The idea for the code is that we have two arrays in the main program, one with the senders and one with the receivers. The SenderThread class has two properties for holding the Thread and JoinableRunnable objects. There is a slightly different design with respect to where the channel object is referenced. Previously this was also done in the SenderThread and ReceiverThread class, now there is public static program variable for this, allowing all classes to access it. This change makes the specification more understandable.

Listing 33: Channel client: program

```java
public class Program {
    public static Channel channel;
}
```

\[13\] The same design as the previous client specification can be used, if one wants the Channel as a property of the SenderThread and ReceiverThread.
```java
public static int sendersCount; // n
public static int receiversCount; // m
public static int sendMaxCount; // k
public static int receiveMaxCount; // l

public static void main(String[] args)
{
    Program.channel = new Channel();
    SenderThread[] senders = new SenderThread[Program.sendersCount];
    for (int i = 0; i < Program.sendersCount; i++)
    {
        SenderThread s = new SenderThread();
        JoinableRunnable j = ThreadingHelper.createJoinableRunnable(s);
        Thread t = new Thread(j);
        s.thread = t;
        s.joinable = j;
        t.start();
        senders[i] = s;
    }
    // Receiver thread creation, and thread starting.
    for (int j = 0; j < Program.sendersCount; j++)
    {
        SenderThread sw = senders[j];
        ThreadingHelper.join(sw.thread, sw.joinable);
    }
    // Receiver thread joining
}
```

---

**Listing 34: Channel client: SenderThread class**

```java
class SenderThread implements Runnable {
    Thread thread;
    JoinableRunnable joinable;

    SenderThread(Channel c)
    {
        this.c = c;
    }

    public void run()
    {
        String m;
        for (int i = 0; i != Program.sendMaxCount; i++)
        {
            for (; ;)
            {
                boolean success = this.c.send(m);
                if (success) break;
            }
        }
    }
}
```
We can now transform the general specification of the program and translate it into a form that VeriFast can handle. Considering how the list of properties is reflected in the formalization.

1. **Deadlock and race free.**
   Holds for the same reasons as in section 7.4: no writes to shared variables and no locks.

2. **One main program with a channel and n senders and m receivers, each working in their own thread.**
   This specification is achieved by the structure of the program.

3. **After executions n \( \times k \) messages are sent and m \( \times l \) messages are received.**
   The first place we write this down in the code is as a postcondition of the main method, see line 17 of listing 35. Note that it is written in an abbreviated form. Also required is postcondition of the sender thread. Its postcondition requires it to have sent \( k \) messages.

   To be able to count the number of messages this sender has sent, a ghost variable is used, see line 2 of listing 36. We are now able to formulate the postcondition of this thread, see line 7 listing 36.

The new specification for the program is given in listing 35.

```
public class Program {
    //@ static int sendCount;
    //@ static int receiveCount;
    public static Channel channel;
    public static int sendersCount; // n
    public static int receiversCount; // m
    public static int sendMaxCount; // k
    public static int receiveMaxCount; // l

    public static void main(String[] args)
    /*@ requires class_init_token(Program.class) &
        Program_channel(?channel) &
        Program_sendCount(?psc) &
        Program_receiveCount(?prc) &
        psc==0 &
        pre(?c) = [ ] Program_channel(?c) &
        [ ] atomic_space(client_inv(c)) &
        Program_sendMaxCount(?smc) &
        Program_receiveMaxCount(?rmc) &
        rmc < smc &
        [ ] Program_sendMaxCount(?smc) &
        [ ] Program_receiveMaxCount(?rmc) &
        rmc < smc &
        Program_sendCount(smc * sendersCount) &
        Program_receiveCount(rmc * receiversCount); @*/
    {
        // ...
    }
}
```

The specification of the SenderThread is given in listing 36.

```
public class SenderThread implements Runnable {
    //@ int senderSendCount;
    Thread thread;
    JoinableRunnable joinable;

    //@ predicate pre() = [ ] Program_channel(?c) &
    // atomic_space(client_inv(c)) &
    // Program_sendMaxCount(?smc) &
```
senderSendCount |−> ?ssc & ssc==0 & ssc==smc;
//@ predicate post() = [.]Program_channel(?c) & & [.]c.Channel() & ssc & [.]atomicspace(client_inv(c)) & & [.]Program_sendMaxCount(?smc) & & this.
    senderSendCount |−> ?ssc & ssc==smc;

public void run()
//@ requires pre();
//@ ensures post();
{
    // ...
}

Because we omit the requirements that apply during execution we are left with a small client invariant, see listing 37. The my_sep and my_unsep lemmas of listing 30 remain the same.

Listing 37: Channel client: client invariant

//@
predicate ctor client_inv(Channel c)() =
c.ChannelState(?items, ?qms) & & forall(items, non_null) == true;
predicate ctor my_unsep_pred(Channel c)(list<Object> items, int qms) =
    forall(items, non_null) == true;
//@

7.5.3 VeriFast client annotation

Having the specification of the methods given, we can continue with the annotations. Each loop in the main method requires a loop invariant. In the loop invariant we can state predicates about the contents of arrays. We can do this with an array slice, an array slice is a predicate that holds for a number of array items in an array. Beside the array slice we need to use array slice deep, with this predicate we keep a predicate per array element. To use this array slice deep we need to specify a new predicate. This predicate will hold the SenderThreads which are stored in the array. In listing 38 this additional predicate is given together with the annotated main method of the program class.

Listing 38: Channel client: program annotation

//@
predicate senderWorker(unit u, SenderThread s; unit u2) =
s.thread |−> ?t & & s.joinable |−> ?j & &
t.Thread(j, true) & & j.JoinableRunnable(s, true) & & s.getClass() ==
    SenderThread.class & &
u2 == unit;
//@

public class Program {
    // ...
    public static void main(String[] args)
//@ requires class_init_token(Program.class) & &
    Program_channel(?channel) & &
    Program_sendCount(?psc) & & Program_receiveCount(?prc) & & psc==0 & & prc
    ==0 & &

69
Program sendersCount (sendersCount) &< 0 < sendersCount &< Program_sendMaxCount(smc) &< 0 < smc &< Program_receiveMaxCount(rmc) &< 0 < rmc; @
/*@ ensures Program_sendCount(smc * sendersCount) &< Program_receiveCount(rmc * receiversCount); @*/
{
  Program.channel = new Channel();
  //@ close client_inv(Program.channel());
  //@ create_atomic_space(client_inv(Program.channel));

  SenderThread[] senders = new SenderThread[Program.sendersCount];
  //@ array_slice_empty_close(senders, 0, senderWorker, unit);
  for (int i = 0; i < Program.sendersCount; i++)
  { 
    Program.channel(c) &< [c].Channel() &< [client_inv(c)] 
    Program_sendMaxCount(smc) &< 0 < sendersCount &< 
    Program_sendersCount(sendersCount) &< 0 < sendersCount &< 
    array_slice(senders, i, sendersCount, ?elems) &< all_eq(elems, null) == 
    true &< 
    array_slice_deep(senders, 0, i, senderWorker, unit, _, _);
  }
  
  SenderThread s = new SenderThread();
  JoinableRunnable j = ThreadingHelper.createJoinableRunnable(s);
  //@ close s.pre();
  //@ j.closeIt();
  Thread t = new Thread(j);
  s.thread = t;
  s.joinable = j;
  t.start();

  senders[i] = s;
  //@ array_slice_split(senders, i, i + 1);
  //@ close senderWorker(unit, senders[i], unit);
  //@ array_slice_deep_close(senders, i, senderWorker, unit);
}

  // Receiver thread creation, and thread starting.
  for (int j = 0; j < Program.sendersCount; j++)
  { 
    Program_sendCount(smc) &< 
    Program_sendersCount(sendersCount) &< 0 < sendersCount &< 
    Program_sendCount(j * smc) &< 
    array_slice_deep(senders, j, sendersCount, senderWorker, unit, _, _);
  }
  
  SenderThread sw = senders[j];
  ThreadingHelper.join(sw.thread, sw.joinable);
  //@ open sw.post();
  //@ Program.sendCount += sw.senderSendCount;
}
//@ assert Program.sendCount == Program.sendMaxCount * sendersCount;

  // Receiver thread joining
In the invariant of the first loops we have to state two things, one that the array up to the index holds SenderThread elements and from the index to the end is empty. The second statement is easy specified (line 31 of listing 38). For the non-empty elements we use the array_slice_deep predicate, with this we can state that the elements are of a certain predicate (the senderWorker predicate). When the loop body is executed the loop invariant must be maintained. This means that the array slices need to be updated. The last element of the array_slice needs to be split off because that element is just updated. Therefore not empty anymore. This is done at line 45. The array_slice_deep also needs to be updated because element i needs to be added. As can been seen in lines 46 and 47.

The last two loops can do with less annotation. Because the arrays are not used after the loops, it is fine if we leak their information. Important in this loop is the update of the Program.sendCount with the send count of the SenderThread.

This specification gives rise to a problem within VeriFast. VeriFast was unable to conclude the assertion of the sendCount and the multiplication of sendMaxCount and sendersCount. Because of the the interenal definitions of multiplications in VeriFast. A suggested fix by Bart Jacobs was to help the SAT solver by additional multiplication lemmas. Regardless of the additional lemmas to help the SAT solver, they failed to produce the desired result; a second suggested fix was to use a different SAT solver, VeriFast comes with two SAT solvers. The Redux SAT solver did not have a problem with the multiplication, consequently the Redux SAT solver is used going forward.

Having the main class annotated, the annotated version of the SenderThread class in listing 39 is presented next. We have seen how this works and no new concepts are used here, therefore the code and annotation is understandable without any explanations.

```
Listing 39: Channel client: SenderThread class annotation

public class SenderThread implements Runnable {

    public void run() {
        // @ requires pre();
        // @ ensures post();

        String m;
        for(int i=0; i < Program.sendMaxCount; i++)
            //@ invariant [.] Program_channel(?c) &@& [.] c.Channel() &@& [.] atomic_space (client_inv(c)) &@& this.senderSendCount |-> ?ssc &@& [.]
            Program_sendMaxCount(?smc) &@& i == ssc &@& i <= smc;

        for (;;)
            //@ invariant [.] Program_channel(c) &@& [.] c.Channel() &@& [.]
            atomic_space(client_inv(c)) &@& this.senderSendCount |-> ssc &@& [.]
            Program_sendMaxCount(smc) &@& i < smc &@& i == ssc;

            //@ produce_lemma function_pointer_chunk(my_sep) : channel_sep( client_inv(c), c, my_sep_pred, my_unsep_pred(c)()) { close_exists(c ); call(); };

            //@ produce_lemma function_pointer_chunk(my_unsep) : channel_unsep( client_inv(c), c, my_sep_pred, my_unsep_pred(c)()) { close_exists(c ); call(); };

            //@ close my_sep_pred();
```

The complete code is verifiable with VeriFast, meaning there are no race conditions and no deadlocks in the program, and the program meets its specifications.

7.6 Client program with multiple senders and receivers and richer specifications

The prior verification left out some desirable verification properties because adding them increased the complexity a lot for one iteration. In this section they are added in the verification again.

7.6.1 Client specification

We specify a number of properties that should hold for the program, they follow out of the goal of this iteration of the verification. The properties are:

1. Deadlock and race free.
2. One main program with a channel and one sender and one receiver, which work in their own thread.
3. After executions $n \times k$ messages are sent and $m \times l$ messages are received.
4. The program can not receive more messages than the program has sent.
5. The number of messages in the communication channel is the number of sent messages minus
the number of received messages.

6. No message loss, all sent messages will be received.

### 7.6.2 Client VeriFast specification

The general specification of the program is now transformed and translated into a form that
VeriFast can handle. This was accomplished by considering how the list of properties is reflected
in the formalization. This section only presents the sender party; the receiver will go in a similar
manner, see appendix A.5.

1. **Deadlock and race free.**
   
   Holds for the same reasons as in section 7.5; no writes to shared variables and no locks.

2. **One main program with a channel and a m senders and n receivers, each working in their
   own thread.**
   
   Holds for the same reasons as in section 7.5; the specification is achieved by the structure of
   the program.

3. **After executions m * k messages are sent and n * l messages are received.**
   
   Holds for the same reasons as in section 7.5. As done previous, the postcondition of the
   main method, see line 20 of listing 41. As we did before we need to specify a postcondition of the
   sender thread. Its postcondition need to be that it has sent k messages. The postcondition
   of this thread is formulate at lines 14 to 20 of listing 42.

4. **The program can not receive more messages than the program has sent.**
   
   To be able to conclude this, VeriFast needs to know the number of send and received messages
   at anytime during the execution. In the client program of section 7.4 ghost variables were
   used to hold those numbers, and they were maintained during the loop of the threads. The
   write permission to the variable is given to the thread. In section 7.5 each thread had its
   own counter, the thread maintained its counter, and after thread termination those counters
   where summed up. Each thread has write access to his own counter, and after completion it
   was returned from that thread to the main thread for the summation. With multiple senders
   and a client invariant, both strategies will not work. The counters need to be maintained
   by the threads during threads executions and also be readable by the client invariant for the
   summation of the send count or receive count.

   The count needs to be maintained in a place where the thread has write permission, but at
   other moments the client invariant has read permission to. We will do this via the sender
   thread array and `array_slice_deep`, in the `array_slice_deep` we use the value property to
   store in each individual array slice the count of the corresponding sender thread. The sender
   thread will need write permission to the corresponding single slice in the array, thus it will
   be able to update that variable. A consequence of this is that the thread needs to know its
   own position, or index, in the array.

   The arrays of the senders and receivers need to be at a place accessible by threads. Therefore
   we store them as public static variables of the main program. This is a minor change of the
   Java code.

   The VeriFast specification of this requirement is on lines 6 to 13 of listing 40. On line 6 the
   sender thread array is accessed. Line 8 states that the array elements are of a certain new
   predicate, `senderWorkerInv` defined at line 15 and 16. We also state that we have a list of
   send counts, `?lstSc`. The receiver is similar, and this requirement is specified at line 13. A
   fixpoint lemma function sums the count lists, lines 18 to 23.
5. The number of messages in the communication channel is the number of sent messages minus the number of received messages.
This can be easily expressed, because of the work required for the previous property. We need to check whether the number of messages in the buffer is the number of send messages (the sum of sender counts) minus the number of received messages (the sum of the sender counts). This is expressed at line 12 of listing 10.

6. No message loss, all sent messages will be received.
This property is at this moment not proving in the verification. Only when we assume \( n \cdot k = m \cdot l \) in the precondition of the main, it is easily proven with VeriFast.

For this we need to give a proof by hand. With the current standard VeriFast Java libraries and the way we have made this verification this is not possible. We present the prove by hand first, then we will explain why this could not be proving with VeriFast. In the verification we have a client invariant which hold during and after the execution of the sender and receiver threads. By this we have three properties \( \alpha : \# received \leq \# send \), \( \beta : \# queue \equiv \# send - \# received \) and \( \gamma : \) no empty items in the queue. We use those three properties in the prove for no message loss. For this we have to make a case distinction on \( n \cdot k \) and \( m \cdot l \).

- If \( n \cdot k \leq m \cdot l \). By the post condition of the sender thread we know that each sender has send \( k \) messages, having \( n \) senders means \( \# send \equiv n \cdot k \). By the post condition of the receiver thread we know that each receiver has received \( l \) messages, having \( m \) receivers means \( \# received \equiv m \cdot l \). By \( \alpha \) and substitution we have \( m \cdot l \leq n \cdot k \). By the case distinction we get \( n \cdot k \equiv m \cdot l \). Which means all send messages are received, and there can not be message loss.

- If \( n \cdot k > m \cdot k \). By the post condition of the sender thread we know that each sender has send \( k \) messages, having \( n \) senders means \( \# send \equiv n \cdot k \). By the post condition of the receiver thread we know that each receiver has received \( l \) messages, having \( m \) receivers means \( \# received \equiv m \cdot l \). By \( \beta \) and substitution we have \( \# queue \equiv n \cdot k - m \cdot l \). By the case distinction and \( \beta \) we know the queue has to hold at least one message. Having \( m \cdot l \) messages received from the \( n \cdot k \) send messages and that the remaining \( n \cdot k - m \cdot l \) messages are still in the queue, and by \( \gamma \) we are certain that there are no empty elements in the queue. We can conclude that if all send messages are received or still in the queue, then no message is lost (up to that moment).

The issue is that after thread joining, we have multiple fractional heap chunks of the atomic space with the client invariant. The way we designed the verification, VeriFast can not conclude that it is only one atomic space with the same client invariant. Because of this VeriFast can not “open” the client invariant to use the properties stated in this invariant.

Thus we can not use the properties in a reasoning like we did by hand. Modification to the VeriFast Java libraries or a different verification design could make the properties of the invariant accessible, but we have not tried to accomplish this.

Next the client invariant is presented.

```java
/*@
predicate_cotr _client_inv(Channel c) () = 
  c.ChannelState(?items, ?qms) &
  forall(items, non_null) == true &
  [1/2] Program_sendMaxCount(?smc) &
   0 < smc &
  [1/2] Program_receiveMaxCount(?rmc) &
   0 < rmc &
  [1/3] Program_senders(?s) &
  [1/3] Program_sendersCount(?sc) &
  [1/3] array_slice_deep(s, 0, sc, senderWorkerInv, unit, _, ?lstSc) &
*/
```

Listing 40: Channel client: client invariant
Note that the `senderWorkerInv` has one output parameter (the last parameter) which outputs the sender send count of that array element.

An interesting detail that stands out in line 16 in listing 40, is the presence of a fractional permission greater than one. This is allowed (sound), but usage is tedious. This is sound because the predicate is only used in an array slice for which there is only a fraction. Getting a predicate out of the fractional array slice results in also having a fraction of that predicate. Making mistakes in the fractions can result in crashing VeriFast. In this case it is correct and will not crash because the fraction is in fact a half, $\frac{3}{2} + \frac{1}{3}$, the $\frac{1}{3}$ comes from the fraction of the `array_slice_deep` on line 8.

The new program class, with sender threads is given as a static parameter is presented in listing 41. The `SenderThread` class with the new ghost variable `myIndex`, and in the pre and postcondition information about the sender thread array.

Listing 41: Channel client: program specification

```java
public class Program {
    public static void main(String[] args) {
        // requires class_init_token(Program.class) &
        // Program_channel(?channel) &
        // Program_senders(?) &
        // Program_receivers(?) &
        // Program_sendCount(?)sendersCount) &
        // Program_receiversCount(?)receiversCount) &
        // 0 < sendersCount &
        // 0 < receiversCount &
        // Program_sendCount(?)psc) &
        // Program_receiveCount(?)prc) &
        // psc==0 &
        // Program_sendMaxCount(?)smc) &
        // 0 < smc &
        // Program_receiveMaxCount(?)rmc) &
        // 0 < rmc; @
    }
}
```

14 this is a bug in the current version.
20 /*@ ensures Program_sendCount(smc * sendersCount) &
  Program_receiveCount(rmc * receiversCount); @*/
21 {
22   // ...
23 }

Listing 42: Channel client: SenderThread class specification

public class SenderThread implements Runnable {
  //@ int senderSendCount;
  //@ int myIndex;
  Thread thread;
  JoinableRunnable joinable;

  /*@
  predicate pre() =
  [.] Program_channel(?c) &
  [.] c.Channel() &
  [.] atomic_space(client_inv( c )) &
  [.] Program_sendMaxCount(?smc) &
  0 < smc &
  [.] Program_senders(?s) &
  [.] this.myIndex |--?myIndex &
  [.] Program_sendersCount(?sc) &
  myIndex < sc &
  [1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil), cons(0, nil));
  @*/

  /*@
  predicate post() =
  [.] Program_channel(?c) &
  [.] c.Channel() &
  [.] atomic_space(client_inv( c )) &
  [.] Program_sendMaxCount(?smc) &
  [.] Program_senders(?s) &
  [.] this.myIndex |--?myIndex &
  [.] Program_sendersCount(?sc) &
  myIndex < sc &
  [1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil), cons(smc, nil));
  @*/

  public void run()
  //@ requires pre();
  //@ ensures post();
  {
    // ...
  }
}

7.6.3 Client VeriFast annotation

For VeriFast to be able to conclude that the postcondition is reachable from its precondition it requires the annotation of the code.

A second code change is required, namely in thread creation and starting. A problem is that when the thread is started, the client invariant needs to hold. In this client invariant all elements of the array are used to get the send count, while some of the elements are still empty. The creation and starting of the threads needs to separated. Because this is easier than changing the specification, but it requires a second source code change.

In VeriFast there was insufficient support for fractional array slices. New lemmas where, with the help of Bart Jacobs, added to the VeriFast library to support this verification, those new lemma will be given in appendix A.6.
Because to no new concepts of VeriFast are used, we give the annotation without further description.

Listing 43: Channel client: program annotation

```java
public class Program {
    // ...

    public static void main(String[] args)
    {
        //@ requires class_init_token(Program.class) &
        Program_channel(?channel) &
        Program_senders(., .) &
        Program_receivers(., .) &
        Program_sendersCount(?sendersCount) &
        Program_receiversCount(?receiversCount) &
        0 < sendersCount &
        0 < receiversCount &
        Program_sendCount(?psc) &
        Program_receiveCount(?prc) &
        psc == 0 &
        prc == 0 &
        Program_sendMaxCount(?smc) &
        0 < smc &
        Program_receiveMaxCount(?rmc) &
        0 < rmc; @
        //@ ensures Program_sendCount(smc * sendersCount) &
        Program_receiveCount(rmc * receiversCount); @

        Program_senders = new SenderThread[Program.sendersCount];
        //@ SenderThread[] s = Program_senders;
        //@ leak Program_senders(s);
        for (int i = 0; i < Program.sendersCount; i++)
        { //@ invariant 0 <= i &
            //@ Program_senders(s) &
            //@ Program_sendersCount(sendersCount) &
            //@ array_slice_deep_empty_close(s, 0, senderWorker, unit);
            Program_senders[i] = new SenderThread();
            //@ Program_senders[i].myIndex = i;
            //@ array_slice_split(senders, i, i + 1);
            //@ close [2/3]senderWorkerNull(unit, senders[i], unit);
            //@ close [1/3]senderWorkerInv(unit, senders[i], unit);
            //@ array_slice_deep_close_precise(2/3, senders, i, senderWorkerNull, unit);
            //@ array_slice_deep_close_precise(1/3, senders, i, senderWorkerInv, unit);
        }

        // ... (Receivers: ReceiverThread creation and array filling) ...

        Program_channel = new Channel();
        //@ close client_inv(Program.channel);()
        //@ create_atomic_space(client_inv(Program.channel));
        //@ close foreach_i(0, nil, senderWorkerIndex);
        for (int i = 0; i < Program.sendersCount; i++)
        { //@ invariant 0 <= i &
            //@ Program_senders[i].myIndex = i;
            //@ array_slice_split(senders, i, i + 1);
            //@ close [2/3]senderWorkerNull(unit, senders[i], unit);
            //@ close [1/3]senderWorkerInv(unit, senders[i], unit);
            //@ array_slice_deep_close_precise(2/3, senders, i, senderWorkerNull, unit);
            //@ array_slice_deep_close_precise(1/3, senders, i, senderWorkerInv, unit);
        }
    }
```
[@] Program_sendersCount (sendersCount) & \& [\_] Program_senders(s) & \&
[\_] Program_channel(?c) & \& [\_] c.Channel() & \& [\_] atomic_space(
client_inv(c)) & \& [\_] Program_sendMaxCount(smc) & \& 0 < smc & \&
[2/3] array_slice_deep(s, i, sendersCount, senderWorkerNull, unit, _)
& \&
[1/3] array_slice_deep(s, 0, i, senderWorker, unit, ?elements, _)
& \& foreach_i(0, elements, senderWorkerIndex);
@*/
{
// 0 array_slice_deep_split(s, i, i + 1);
// 0 array_slice_deep_open_purpose(2/3, s, i);
SenderThread sender = senders[i];
// 0 close [1/3]senderWorkerInv(unit, sender, _);
// 0 array_slice_deep_close_purpose(1/3, s, i, senderWorkerInv, unit);
JoinableRunnable j = ThreadingHelper.createJoinableRunnable(sender);
// 0 sender.myIndex = i;
// 0 leak sender.myIndex ~> i;
// 0 close sender.pre();
// 0 j.closeit();
Thread t = new Thread(j);
t.start();

sender.thread = t;
sender.joinable = j;
// 0 close [1/3]senderWorker(unit, sender, unit);
// 0 array_slice_deep_close_purpose(1/3, senders, i, senderWorker, unit);
// 0 close foreach_i(i + 1, nil, senderWorkerIndex);
// 0 close senderWorkerIndex(i, sender);
// 0 close foreach_i(i, cons(sender, nil), senderWorkerIndex);
// 0 foreach_i_append(0, elements, cons(sender, nil));
}
//@ asserts ( Receivers: ReceiverThread thread starting ) ...
for (int j = 0; j < Program_sendersCount; j++)
/*@ invariant 0 <= j & \& j <= sendersCount & \&
[\_] Program_sendersCount(sendersCount) & \& [\_] Program_senders(s) & \&
[\_] Program_channel(?c) & \& [\_] c.Channel() & \& [\_] atomic_space(
client_inv(c)) & \& Program_sendCount(j * smc) & \& [\_] Program_sendMaxCount(smc) & \&
[1/3] array_slice_deep(s, j, sendersCount, senderWorker, unit, ?
elements, _) & \& foreach_i(j, elements, senderWorkerIndex);
@*/
{
SenderThread sw = senders[j];
ThreadingHelper.join(sw.thread, sw.joinable);
//@ open sw.post();
//@ open foreach_i(j, elements, senderWorkerIndex);
//@ open senderWorkerIndex(j, sw);
//@ int jj = sw.myIndex;
//@ assert j == jj;
//@ array_slice_deep_open_purpose(1/3, s, j);
//@ open senderWorkerInv(unit, sw, _);
//@ Program.sendCount += sw.senderSendCount;
//@ assert Program.sendCount == Program.sendMaxCount * sendersCount;
Listing 44: Channel client: SenderThread class annotation

```java
public class SenderThread implements Runnable {

    public void run()
    {
        String m;
        for(int i=0; i < Program.sendMaxCount; i++)
            //@ invariant
            [1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil), cons(i, nil));
        }
    }

    for (;;) {
        //@ invariant
        [1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil), cons(i, nil));
        @*/
    }

    /@ produce_lemma_function_pointer_chunk(my_sep) : channel_sep( client_inv(c), c, my_sep_pred, my_unsep_pred(c))() { close exists(c); call(); }
}
```

// ... (Receivers: ReceiverThread thread joining) ...

// predicate senderWorker(unit u, SenderThread SenderWorker; unit u2) =
//  SenderWorker.getClass() == SenderThread.class &\&
//  u2 == unit;

// predicate senderWorkerNull(unit u, SenderThread SenderWorker; unit u2) =
//  SenderWorker.getClass() == SenderThread.class &\& [3/4] SenderWorker.
//  senderSendCount |-> 0 &\&
//  u2 == unit;

// predicate senderWorkerIndex(int i, SenderThread t) =
//  [1] t.myIndex |-> i;
```
/@ produce lemma function_pointer_chunk(my_unsep) : channel_unsep cI(c), c, my_unsep_pred, my_unsep_pred(c)() { close exists(c); call(); }; 
//@ produce lemma function_pointer_chunk(my_send) : channel_send cI(c), c, my_unsep_pred(c), m, P_clientSend, Q_clientSend(r) { call(); }; 
//@ close P_clientSend(); 

boolean success = Program.channel.send(m); 
//@ open Q_clientSend(success); 
if (success) break; 
//@ close post(); 

predicate P_clientSend() = 
[1/3]array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil)); 

predicate Q_clientSend(boolean r) = 
[1/3]array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons(this, nil)); 

lemma void my_send(boolean r) 
requires i < smc & my_unsep_pred(c)(?items, ?qms); 
ensures Q_clientSend(r) & my_unsep_pred(c)(r ? append(items, cons(m, nil)) : items, qms); 
{ open P_clientSend(); 
open my_unsep_pred(c)(items, qms); 
SenderThread[] senders = Program.senders; 
int sendersCount = Program.sendersCount; 
int k = myIndex; 
assert [1/3]array_slice_deep(senders, 0, sendersCount, senderWorkerInv, unit ._.?oldSenderCounts); 
array_slice_deep_split(Program.senders, 0, myIndex); 
array_slice_deep_split_precise(1/3, Program.senders, myIndex, Program. 
sendersCount, senderWorkerInv, unit, myIndex + 1); 
array_slice_deep_open_precise(2/3, Program.senders, myIndex); 
int myOldSendCount = sendersCount; 
if (r) { 
sendersCount++; 
} 
int myNewSendCount = sendersCount; 
array_slice_deep_close(Program.senders, myIndex, senderWorkerInv, unit); 
array_slice_deep_join_precise(1/3, Program.senders, myIndex, myIndex + 1, 
senderWorkerInv, unit, Program.sendersCount); 
array_slice_deep_join_precise(1/3, Program.senders, 0, myIndex, 
senderWorkerInv, unit, Program.sendersCount); 
forall_append(items, cons(m, nil), non_null);
VeriFast is not able to update an item in the middle of the list, therefore we need lemmas to split the list, take the head of the second list, update the head, put the head back on top of the second list and merge the first list and the second list. This strategy including the updating of the send count is presented at lines 62 to 82 of listing 44. Because not all required functionality existed, some additional lemmas and predicates are required, they are presented in listing 45.

Listing 45: Channel client: some additional lemmas

```plaintext
/*@
lemma auto void sum_all_eq(list<int> vs)
  requires all_eq(vs, 0) == true;
  ensures sum(vs) == 0;
{
  switch(vs) {
    case nil :
    case cons(x0, xs0): sum_all_eq(xs0);
  }
}

lemma void sum_take_cons_drop(int k, list<int> xs, int x)
  requires 0 <= k & k < length(xs);
  ensures sum(xs) == sum(append(take(k, xs), cons(x, drop(k + 1, xs)))) - x +
       head(take(1, drop(k, xs)));
{
  switch (xs) {
    case nil:
    case cons(x0, xs0):
      if (k == 0) {
        } else {
        sum_take_cons_drop(k - 1, xs0, x);
      }
    }
}

predicate foreach_i<t>(int i, list<t> xs, predicate(int, t) p) =
  switch (xs) {
    case nil: return true;
    case cons(x0, xs0): return p(i, x0) & foreach_i(i + 1, xs0, p);
  };
*/
```
The code is verifiable against the specification. At this point we know using the communication channel will not lead to any deadlocks, there are no race conditions, and the communication channel behaves as specified.

### 7.7 Remark on verification

We have to make a few small remarks on this verification.

The verification is partial, we do not prove termination. If fact, we have proven a program with the possibility to never terminate. For example, if $n \times k < m \times l$ the program will try to receive more messages then will be send, therefore at least one receiver thread will indefinitely wait until he has received $l$ messages.

The modular proving strategy works fine, but one has to be careful about the conclusions. For example, in the verification of the channel we have proven that the queue is limited. In the verification of the clients this detail is abstracted away from, only the notion that sending could not succeeds is used. This has as a result that proving a program with a limited queue and sending more messages then the queue could hold can be proven correctly. In our verification this is not a problem, but when termination needs to be proven this detail is crucial.

### 7.8 Source code improvements

During the verification process we made an observation about the quality and/or understandability of the programming code.  

During the translation of the C# code to Java code we thought about its specifications. The code fragments we translated needed to have a clear specification, this appeared to be harder than expected. The code seemed to function correctly, as is claimed in the Capita Selecta work, but formulating the functionality of the code was difficult. The code is developed without the idea of correctly specifying each method, and this seemed to be a weak point. Especially methods that rely on synchronization constructs for correct behavior where hard to specify. The verification can be done on the original source code, but it was clear that changes to the code would give benefits, easier specifications, to the verification.

For example, the original `send` method in the channel class had a locking strategy the worked but it was hard to specify, let even stand to verify. In listing 48 the original send method in C# code is presented. In listing 47 we show how this method is used in a method that represents the action of a transition.

---

15 This may be outside the scope of verification, but due its importance and was made during the channel abstraction phase we present it here at the end of this chapter.

16 A small alternation is done for readability, for example some comments and all logging commands are left out.
Listing 46: Original Send method

```csharp
public bool Send(SignalMessage msg, StateMachine sender, bool isFirstActionOfTransition)
{
    Monitor.Enter(_lockObjPort); // acquire lock such that no other sm can use
    // the channel at the same time
    while (_portAtoB.Count() >= queueLength) // while the queue is full we stay
        in this loop.
    {
        if (isFirstActionOfTransition) // if the buffer is full and the send is
            // the first action of a transition we need to return false.
        {
            Monitor.Exit(_lockObjPort); // release the lock on the ports, such
                // that other sm’s can send.
            return false; // send can not be done because of full buffer, return
                // false (the action is false and the transition is thereby not
                // enabled)
        }
        else
        {
            _WaitForSendingGate.Reset(); // close the senders gate (queue is
                // full so all senders have to wait)
            Monitor.Exit(_lockObjPort); // release the lock on channel, let
                // other sm’s work on channel
            sender.EndAtomicStep(); // end the atomic step of the sender’s
                // sm
            _WaitForSendingGate.Wait(); // wait at this gate until it is opened
                // again (...)
            sender.BeginAtomicStep(); // new atomic step try again
            if (sender.IsDestructing) return false; // abort the sending, the sm
                // is destructing
            Monitor.Enter(_lockObjPort); // get the lock again on channel
        }
    }
    _portAtoB.Enqueue(msg); // the SENDING ;)
    Monitor.Exit(_lockObjPort); // exit the lock on the channel
    if (_objectB != null) _objectB.SignalStateMachines(); // signal all sm’s on
        // the receivers side
    return true;
}
```

Listing 47: Usages of original Send method

```csharp
private bool ActionA2B(object sender, smActionEventArgs args)
{
    bool result = true;

    result = Parent.Ports["Q"].Send(new SignalMessage("msg1", new object[] { } ),
        this, true);
    if (!result) return false;
    EndAtomicStep();
    BeginAtomicStep();
    result = Parent.Ports["Q"].Send(new SignalMessage("msg2", new object[] { } ),
        this, false);
}
```
return result;
}

The main problem with specifying are the line 13 to 19 of listing 48, here we need to specify operations on synchronization constructs. The operations, e.g., locking and releasing of locks and semaphores, influence the execution of the current thread and another threads. We where not able to make a nice understandable specification for this.

We made the observation that a redesign of the code would improve the the Send method. The synchronization statements should be there to make sure the execution behaves like it is specified, not there because we specify the locking in listing ?? we presented a new design of the Send method. In this design the synchronization constructs must be their to let the method execute like we expects it does. The method is not responsible anymore for retrying the send if it has to, that we have moved to the caller method, in listing 49 we see this. The redesign changed the way the state machine worked with the channel, but eventually it had the same result in functionality.

### Listing 48: Redesigned Send method

```csharp
public bool Send(SignalMessage msg, StateMachine sender)
{
    Monitor.Enter(_lockObjPort); // acquire lock such that no other sm can use the channel at the same time
    if (_portAtoB.Count() >= queueLength) //IF the queue is full we ...
    {
        _WaitForSendingGate.Reset(); // – close the senders gate (let all senders wait)
        Monitor.Exit(_lockObjPort); // – exit the lock on the channel
        return false; // – and return false
    }
    _portAtoB.Enqueue(msg); // the SENDING ;)
    Monitor.Exit(_lockObjPort); // exit the lock on the channel
    if (_objectB != null) _objectB.SignalStateMachines(); //signal all sm’s in the on the receivers side
    return true;
}
```

### Listing 49: Usages of redesigned Send method

```csharp
private bool ActionA2B(object sender, smActionEventArgs args)
{
    bool result = true;
    result = Parent.Ports["Q"].Send(new SignalMessage("msg1", new object[] { } ), this);
    if (!result) return false;
    EndAtomicStep();
}
```

We not claim this is always possible, but in this case it is.
The verification process has led to an improvement of the source code, mainly to the understandability of the code but this also influence the quality of the code in a positive sense. Also the effort for verification decreases because the specification that needs to be formulated are less complex.
8 Conclusions and future work

We have investigated the current state of formal verification tools for proving correctness of multi-threaded code, this is concluded in section 8.1. In section 8.2 we present the conclusions of the verification of a non-trivial essential part of the TOOSM framework, the communication channel, which was performed with VeriFast. This process gave insights into formal verification and verification with VeriFast in section 8.3. We address a number of threats to validity of this research in section 8.4. Finally we present some future work in section 8.5.

8.1 Verification techniques and tools

In this section, we present the conclusions of the research question A: What is the current state of art in tool-based verification of concurrent, object-oriented code?

Verification of programming code is an actively researched area in computer science. There are a number of techniques and tools available for verification of programs. In order to find a technique and tool, usable for verification of the communication channel, we list some preferences: (1) Since it is our goal to verify by means of tools, the technique needs to have tool support. (2) The technique and tool must support concurrency. (3) The technique and tool must be familiar with the object-oriented programming paradigm and (4) preferably operate on source code in C#. (5) Preferably, the tool and technique are relatively easy to understand and use and is actively maintained and supported.

Based on these preferences, we looked into the following techniques: Hoare logic, Owicki-Gries logic, the Rely/Guarantee method, and Separation Logic. All techniques, besides Hoare logic are capable of describing concurrent programs. The techniques that support concurrency it is possible to use for the verification. Mainly due to the tool choice we use Separation logic.

Separation logic is a technique that has been getting a lot of attention in the last couple of years. Separation logic is focussed around the heap and resources. This technique is very well suited for verifying programming languages which supports pointers and a number of verification tools implement this technique. Separation logic lends itself extremely well for verification of concurrent programs because of the notion of local reasoning. Since we use C# which is pointer based with concurrency via threads, this technique looks promising for our verification.

Based on the preferences stated above, we investigated the following tools: Spec#, Code Contracts, Dafny, Chalice, VeriFast, and ProPar. We investigated the tools by working through the available tutorials and other documentation and by implementing and verifying the simple summation example in multiple tools.

Spec# is not maintained anymore and the lessons learned were implemented in Code Contracts. Unfortunately, concurrency was left out of Code Contracts making both tools unsuitable for our verification. Also Dafny does not support concurrency well. Chalice lacks the object oriented paradigm and ProPar on the other hand used its own language and paradigm. VeriFast scored, see table 2, well on all but one of our preferences: VeriFast can work with Java but not with C#. Fortunately, because of the straightforward programming constructs used, the translation of C# to Java is not to difficult, making VeriFast the best tool for this project.

Most tools support verification of programs with concurrency, but it is not always easy to do this verification. We also note that in the last Software Verification Competition [47] of the “Verified Software: Theories, Tools and Experiments” international conference, all the problems where non-concurrent. Most tools support only one or two languages. From tool development view this choose is understandable, but if the language is fixed for a user it reduces the tools available to use drastically.
Table 2: Tool comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Concurrency</th>
<th>Object-Oriented</th>
<th>Language</th>
<th>Availability and ease of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpecSharp</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Code Contracts</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dafny</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>Chalice</td>
<td>1</td>
<td>1/2</td>
<td>1/2</td>
<td>0</td>
</tr>
<tr>
<td>VeriFast</td>
<td>1</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>ProPar</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
</tr>
</tbody>
</table>

8.2 Verification of the communication channel

In this section, the conclusion of research question B is presented: *Can one of the tools verify the correctness of concurrent object-oriented code, especially the generic code of Toosm?*

The goal was to investigate if and to what extent we could verify the generic code of Toosm using a verification tool. Based on the overview of techniques and tools, we chose to use VeriFast to do the verification of a generic part of Toosm, namely the communication channel.

To make the verification feasible, we made abstractions to the channel and took an iterative approach, adding functionality with every iteration. To be precise, we abstracted away from the object that is transmitted, have no peek method for looking at the content of the channel and only looked at the communication channel and no other artifacts. The first iteration started with the verification of the channel in isolation. Here we proved race condition freedom, deadlock freedom, and a number of functional requirements, e.g., the functionality of the send method. In the next iteration the channel specifications where modified to be usable with clients. The verification properties remained the same. We extended the verification to a setting with one sender and one receiver and a parameterized number of messages sent and received. The verification showed that no race conditions and deadlocks where present in the code and that the code fulfilled a number of functional requirement, e.g., no more messages could have been received then where send. The next verification iteration verified a parameterized number of senders and receivers and messages sent and received. In this verification we only verified for no racing and no deadlocks. In the last verification we extended the verification properties with functional requirement, e.g., program can not receive more messages then have been send.

In a total of 6 iterations, we managed to verify, with a tool, the correctness of the abstracted communication channel with regard to the requirements shown above.

It important to remark that race condition freedom is easily achieved, because all that is required is a correct VeriFast specification. For achieving deadlock freedom, the developer has to make a lock order and specify via additional annotations the checks on the order. Because we did not need multiple synchronization constructs in the verification, deadlock freedom was also relatively easy to prove. Verification of purely functional requirements (without race conditions and deadlocks) is relatively straightforward. It is the combination of the race condition freedom and functional requirements that is hard. The combination of these properties causes the specification to becomes much more complex. For example see, in section 7.3, how we have proven that no racing is possible on the shared variable itemList of the channel and specify functionality of the send method on the list. Since we only needed a single synchronization construct, proving deadlock freedom in combination with race conditions and functional requirements did not add much complexity. When multiple synchronization constructs are used, we expect the verification to be even more difficult.

In order to successfully specify and annotate the source code, some source code modifications were necessary. In some cases, synchronization constructs were moved or removed in the code, this
made the specification easier to write. An additional benefit was a better understandability of the source code itself, see section 7.8.

8.3 Practicability and usefulness of tool-based approach

To conclude the last research question (question C): What lessons can be learned from this approach?, we will first look back at the verification with VeriFast and later at tool-based verification in general. In this section we have also used our opinion which is based on the experience of this project.

8.3.1 VeriFast experience

The design of specifications and annotations for VeriFast heavily depend on the experience of the user. The entry level of simple non-concurrent specifications is low. It takes only a small amount of effort and time to master the basics of VeriFast, and therefore also Separation logic. For an example see the banking example which was presented in section 6.3. When the complexity of the requirements grow, the required experience with VeriFast also increases. Working with arrays and lists is not as easy as in programming languages. With limited experience, the step to concurrency can be with VeriFast, but when the developer combines concurrency with functional requirements, it becomes very complex and the necessary knowledge and experience required grows rapidly.

The following verification concepts require in-depth knowledge and experience in using VeriFast:

- **Arrays and lists**: the `array_slice` predicate is not very complicated because the documentation on this subject is sufficiently clear. A more difficult concept for arrays, however, is the `array_slice_deep` predicate. Using this predicate is hard. The concept is not easy to understand and the use of this predicate is not well documented;
- **Thread starting/joining**: In contrast to thread starting, which is rather simple, the joining of threads is rather hard. The VeriFast Java library does not support the `join` method of the thread class by default;
- **Atomic spaces and lemma function pointer chunks** These concepts are difficult to understand for beginning users. They are explained in a number of scientific papers [44] [45], which are hard to read for non-scientific developers. Fortunately the use of those concepts becomes easier if one knows in which situations they can be used.

8.3.2 VeriFast effort

The required effort needed for verification naturally depends on the experience of the user. An expert on verification with VeriFast has significant less effort need to make the specification. Still, the effort needed for the verification is rather large. Writing verification specifications and annotations is not trivial and can therefore be a time consuming task. Also, the amount of specifications and annotations needed can become very large for complex cases.

For a very simple program like the example in section 6.3 the VeriFast “overhead” is 125%. This means 25% more lines of annotation than lines of source code. Based on the investigation into different verification tools, this overhead seems to be the norm, rather than the exception. For the final verification in section 7 the amount of annotation needed is large, even though the program under verification is still rather small and simple with only basic functionality and specification. Regardless of the simple and basic functionality, there seems to be some kind of minimum amount of annotations required for formal verification. Larger and more complex systems will only increase
the annotation effort. The required annotation overhead needed for the verifications done in this thesis are presented in table 3. For the three client verification versions of chapter 7, only the channel and client files are taken into account.

Table 3: Code vs annotations comparison

<table>
<thead>
<tr>
<th>Version</th>
<th>LOC</th>
<th>LOA</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank account example</td>
<td>20</td>
<td>25</td>
<td>125%</td>
</tr>
<tr>
<td>First client version</td>
<td>72</td>
<td>245</td>
<td>340%</td>
</tr>
<tr>
<td>Channel</td>
<td>35</td>
<td>145</td>
<td>414%</td>
</tr>
<tr>
<td>Client</td>
<td>37</td>
<td>100</td>
<td>340%</td>
</tr>
<tr>
<td>Second client version</td>
<td>93</td>
<td>286</td>
<td>308%</td>
</tr>
<tr>
<td>Channel</td>
<td>35</td>
<td>145</td>
<td>414%</td>
</tr>
<tr>
<td>Client</td>
<td>58</td>
<td>141</td>
<td>243%</td>
</tr>
<tr>
<td>Third client version</td>
<td>98</td>
<td>536</td>
<td>547%</td>
</tr>
<tr>
<td>Channel</td>
<td>35</td>
<td>145</td>
<td>414%</td>
</tr>
<tr>
<td>Client</td>
<td>63</td>
<td>391</td>
<td>621%</td>
</tr>
</tbody>
</table>

Those findings seem to be confirmed by the “Solution verbosity metrics” of [48] in which the verification results in a software verification competition are reported/compared against each other.

We make the remark the many required annotation are rather simple, like the open and close annotation commands. Also the annotations for using atomic spaces is almost the same each time.

In our case, the effort we spend on giving a formal verification of the communication channel is defendable because it is done on generic code that is used with every transformation. Some effort however will be needed to translate the results from the abstracted channel to the full Toosm communication channel.

8.3.3 VeriFast capabilities

VeriFast has some strong points which where for us important, namely:

- **Parameterized verification**: Because VeriFast uses symbols instead of actual values for variables, it is capable of performing parameterized verification. This means that VeriFast can reason over not only one but for all possible values of a certain variable. This makes it possible to consider an unlimited number of threads communicating with the channel. We can also perform a verification with an unlimited memory use. This means we can verify programs with an infinite state space, which is very nice feature.

- **Concurrency via threads**: VeriFast uses the Java concurrency model. This makes it very practicable to use in verification of Java like languages. Our C# code is very close to this. Not all synchronization constructs are implemented but this can be added if required.

- **Object-Oriented**: Not all tools support the complete object oriented approach, but VeriFast does.

- **Fast verification results**: VeriFast can, in most cases, perform a verification and present the results within a seconds. Other verification tools we found while answering research question A could take up to a few minutes to produce results, even with simple programs.

---

18 Channel consists of the files ‘Channel.jasvaspec’ and ‘Channel.java’.
- **Symbolic debugging**: The symbolic debug capability of the VeriFast IDE makes the verification process much developer friendly.

We have seen verifications where VeriFast is used to prove safety requirements, like race conditions, in programs with concurrency, e.g., [49]. VeriFast is a good tool for these kind of verifications. We have also seen good results on functionality verifications on non-concurrent programs, see both Software Verification Competition enterings/participations of VSTTE 2010 [45] and VSTTE 2012 [47]. When this project started, no effort in verifications of functionality and safety requirements in programs with concurrency were known to us. During this project however, a verification of Linux’s USB BP keyboard driver was performed [50], with unbounded number of threads, unbounded number of keyboards, and an unbounded amount of time the driver is running.

How to make good specification in VeriFast depends on experience and on the desired verification properties. A good pointer is to use encapsulation of smaller predicates into larger predicates. This results in smaller method contracts and the ability to abstract away from data. This abstraction abstracts away the internals the original class which makes the verification more modular and understandable.

### 8.3.4 Formal verification experience

We see a common high knowledge and experience requirement for all the formal verification tools, which makes it difficult to use in the software development process. However, in some cases, formal verification can be beneficial. In those cases, formal verification will be a dedicated task for an experienced user. The benefits of formal verification can outweigh testing, but not in the majority of the cases. Formal verification tools are becoming more mature, and therefore their applicability and usability increase. However, this does not decrease the required knowledge and experience level of the user.

### 8.3.5 Verification approach

The iterative approach resulted in a successful verification. In the last iteration we managed to verify the goal set at the start of the project. The modular approach of first verifying the communication channel and then the client was successful, and coupling between the two verification was kept at a minimum. A consequence is that the verification has become more complex.

Even though it is possible to verify source code that was not developed for formal verification, keeping formal verification in mind during the design and implementation of the software, makes the verification a lot easier. This is because the developer has to think carefully about the goals and responsibilities of the classes and methods. Especially in multi threaded code, where synchronization constructs are introduced, their purpose needs to be completely clear in order to do formal verification. In separation logic this is even mandatory because synchronization constructs have invariants.

### 8.4 Threats to validity

In this project, we used a tool to do the verification. If this tool contains errors or faulty reasoning, it may have an impact on the results. In the case of VeriFast, we do not think this is the case because of the tool’s reputation, contest results [45][47] and the fact that the tool is still actively maintained. When a specification is incorrect, the results of VeriFast may be faulty. Because making a specification can become very complex this is a threat.

Another threat to validity lies in two necessary translations: from hard-to-read specifications in SLCO to our own requirements and from that to VeriFast specifications for the communication
channel. Most requirements are self-evident and can be easily traced back to the specification in SLCO. However, because the translation is done manually and no proof is given, there are no guarantees that the SLCO specifications and the VeriFast specifications describe exactly the same behavior. But because of careful inspection we are highly confident that no errors or alternative behavior were introduced in the translations.

We cannot claim that we have proven the implemented communication channel of Toosm since we made an abstraction for verification. Even so we believe that the general results are valid for the real implementation, some effort is needed to ensure the correctness of the implemented channel.

A small issue is the programming languages used. Because VeriFast works with Java and Toosm works with C#, we had to translate the latter into the former. Because .NET and Java are different platforms, their implementation is different, especially the working of threads and synchronization constructs. During the translation we tried to find constructs with the same behavior in both languages and we are confident this was done successfully.

A remark has to be made about the use of integer values as parameters for the verification. Having no relation between the number of send and received messages, \( n \times k \) and \( m \times l \), it can be in the last iteration that \( n \times k < m \times l \) holds but, that we have proven \( m \times l \) messages are received. This clearly can not be the case in the real execution. During verification the client invariant still requires that \( \#send \leq \#received \). This fact is not incorporated into the postcondition \( m \times l \) messages are received. This weak point in the verification can be overcome by modeling the verification differently, but this also would require some modifications to the standard Java library of VeriFast.

8.5 Future work

This project leads to a number of possible project for the future. We describe the most promising ones.

A fully validated Toosm framework is would be a nice project, but this is only worth if more use of Toosm is expected, otherwise the effort required for this is undefendable. A project like this may also require reinvestigating which tool is best suited for the task, because the complete code is significant larger and more complex. A project where the verification is extended for more complex client behavior, e.g., the state machines, is already very useful. This would also require that the peek method in the channel, which was abstracted away from, needs to be included in the new verification. The implementation of the state machines and the peek method results in having multiple synchronization constructs. Therefore, checking for deadlocks is required, making it an interesting project. In my opinion, VeriFast meets all the requirements for this kind of verification.

With the techniques and tools evaluated and the lessons learned during this project, possible future work would be to reevaluate and re-implement Toosm. By revisiting the design, Toosm can be modified to be better verifiable. This may include a look at the design of SLCO. In addition, it is also possible to investigate the atomicity used in Toosm. One observation made in the Capita Selecta project was that using the atomicity of SLCO during transformation resulted in a very strict interleaving in the resulting code. We can ask whether a different atomicity would allow for better performance, i.e. more concurrent activities, yet still remain functionally correct with regard to the model in SLCO. A new project can use the results of the Capita Selecta and Masters project to investigate this question.

This investigation into the the atomicity can be seen in a broader perspective. An investigation into the granularity of multi-threaded code generated in a MDE chain: coarse granularity vs fine granularity of code. In my opinion, there is a fine line between generating large blocks of code that operate in critical sections vs smaller atomicity almost on machine level instructions. The
first will properly have a performance penalty, but is easier to generate and formally verify, while
the later will have a much bigger development time and much more complex verification, but it
can have a much better performance on execution time. The best choose depends probably on the
type of system developed, but other issues also play a part, i.e., reusability.

A new MDE chain from SLCO to Java source code would be a nice project. A similar design of
the code such as the design of the C# code is achievable with a small effort. If the project includes
a redesign of the code, the project grows in complexity and effort.
References


[38] G. Stewart, L. Beringer, and A.W. Appel. Verified heap theorem prover by paramodulation.


[40] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs.


A Appendix: Channel verification

All verification are done with version 12.2 of VeriFast. All verification require the REDUX SAT solver, i.e., “vfide.exe -prover redux client.jarsrc”.

A.1 Communication Channel (Ia)

The VeriFast files required for the verification are:

- **Channels.jarsrc**: implementation of the “Channels” module, this file contains information about which files are required for the channel source code.
- **Channel.java**: source file belonging to the “Channel” module implementation, the file with the java code with the required annotation.
- **Channels.jarspec**: specification of the “Channels” module, this file contains information about which files are required for the channel specification.
- **Channels.javaspec**: source file belonging to the “Channels” module specification, this file contains the specification of the channel class, this file is used by the client for its verification.

The content of the files are presented in the next four listings.

---

### Listing 50: Channels.jarsrc

```java
Listing 51: Channel.java

```
```java
itemList = new ArrayList();
//queueMaxSize = 1;
//@ itemsGhost = nil;
//@ close channel_sema_inv(this());
//@ one_time(channel_sema_inv(this()));
sema = new Semaphore(1);
//@ sema.leakHandle();
//@ close Channel();
//@ close ChannelState(nil, _);
}

boolean send(String msg)
/*@ requires */
//@ Channel() &
//@ ChannelState(?items, ?qms);
/*@ ensures */
//@ Channel() &
//@ ChannelState(length(items) < qms ? append(items, cons(msg, nil)) : items,
//@ qms) &
//@ result == (length(items) < qms);
/*@ */
{//@ open Channel();
//@ open ChannelState(items, qms);
//@ sema.makeHandle();
sema.acquire();
//@ open channel_sema_inv(this());
boolean result;
if (itemList.size() < queueMaxSize) {
    itemList.add(msg);
    //@ itemsGhost = append(items, cons(msg, nil));
    result = true;
} else {
    result = false;
}
//@ close channel_sema_inv(this());
sema.release();
//@ close ChannelState(_, _);
//@ close Channel();
return result;
}

Object receive()
/*@ requires */
//@ Channel() &
//@ ChannelState(?items, ?qms);
/*@ ensures */
//@ Channel() &
//@ ChannelState(tail(items), qms) &
//@ result == (items != nil ? head(items) : null);
/*@ */
{//@ open Channel();
//@ open ChannelState(items, qms);
//@ sema.makeHandle();
sema.acquire();
//@ open channel_sema_inv(this());
String result;
if (itemList.size() == 0) {
```
result = null;
} else {
    result = (String)itemList.remove(0);
    //@ itemsGhost = tail(itemsGhost);
}
//@ switch (items) { case nil: case cons(i0, is0): }
//@ close_channel_sema_inv(this());
sema.release();
//@ close ChannelState(., _);
//@ close Channel();
return result;
}
A.2 Communication Channel - client ready (Ib)

The VeriFast files required for the verification are:

- **Channels.jarsrc**: implementation of the “Channels” module (presented in [A.1]).
- **Channel.java**: source file belonging to the “Channel” module implementation.
- **Channels.jarpec**: specification of the “Channels” module (presented in [A.1]).
- **Channels.javaspec**: source file belonging to the “Channels” module specification.
- **verifast.jarpec**: Specification of the “verifast” module, supplied with VeriFast in the example folder.
- **verifast.javaspec**: source file belonging to the “verifast” module specification, supplied with VeriFast in the example folder.

Because only “Channel.java” and “Channels.javaspec” are modified, only the content of those files are presented in the next two listings.

Listing 54: Channel.java

```java
package channels;

import java.util.concurrent.*;
import java.*;

//@ predicatector channel_sema_inv(Channel channel) =
channel.itemList -> ?itemList && itemList.List(?items) && [1/2]channel.itemsGhost
|-> ?ghostItems && items == ghostItems &&
[1/2]channel.queueMaxSize |-> ?qms & & length(items) <= qms;

public final class Channel {

  List<Object> itemsGhost;
  List itemList;
  Semaphore sema;
  int queueMaxSize;

  //@ predicate Channel() = sema |-> ?sema & & (.)sema.Semaphore(channel_sema_inv(this));
  //@ predicate ChannelState(list <Object> items, int qms) = [1/2]itemsGhost |-> items
& & [1/2]queueMaxSize |-> qms;

  public Channel() {
    //@ requires true;
    //@ ensures Channel() & & ChannelState(nil, _);
    { 
      itemList = new ArrayList();
    //@ itemsGhost = nil;
    //@ close channel_sema_inv(this());
    //@ one_time(channel_sema_inv(this));
    sema = new Semaphore(1);
    //@ sema.leakHandle();
    //@ close Channel();
    //@ close ChannelState(nil, _);
    }

  boolean send(String msg)
    //@ requires
    [?fc] Channel() & &
    [?fa] atomic_space(?inv) & &
```
is_channel_sep(?sep, inv, this, ?sepPred, ?unsepPred) &&
is_channel_unsep(?unsep, inv, this, sepPred, unsepPred) &&
is_channel_send(?sendClientOp, inv, this, unsepPred, msg, ?pre, ?post) &&
pre();

/*@*/
/*@ ensures */
@fc|Channel() &&
@fa|atomic_space(inv) &&
sepPred() &&
post(result);
@*/
{
//@ open [fc]Channel();
//@ sema.makeHandle();
sema.acquire();
//@ open channel_sema_inv(this());
//@ assert itemList |-> ?l && l.List(?items);
//@ assert [1/2]queueMaxSize |-> ?qms;

boolean result;
if (itemList.size() < queueMaxSize) {
  resultList.add(msg);
  result = true;
} else {
  result = false;
}

//@
/*@ predicate P_send() = */
is_channel_sep(sep, inv, this, sepPred, unsepPred) &&
is_channel_unsep(unsep, inv, this, sepPred, unsepPred) &&
is_channel_send(sendClientOp, inv, this, unsepPred, msg, pre, post) &&
sepPred() &&
pre() &&
[1/2]itemsGhost |-> items;

predicate Q_send() =
[1/2]itemsGhost |-> (length(items) < qms ? append(items, cons(msg, nil)) : items) &&
sepPred() &&
post(length(items) < qms);

lemma void my_ghost_send_op()
  requires P_send() && inv();
  ensures Q_send() && inv();
{ open P_send();
  sep();
  open ChannelState(_, _);
  if (length(items) < qms) {
    itemsGhost = append(items, cons(msg, nil));
    close ChannelState(append(items, cons(msg, nil)), _);
    sendClientOp(true);
  } else {
    itemsGhost = items;
    close ChannelState(items, _);
    sendClientOp(false);
  }
  unsep();
  close Q_send();
}/*@*/
//@ close P_send();
//@ produce_pointer_chunk(my_ghost_send_op) :
atomic_space_ghost_operation(inv, P_send, Q_send) { call(); };
//@ perform_atomic_space_ghost_operation();
//@ close channel_sema_inv(this());
sema_release();
//@ close [fc] Channel();
return result;
}

String receive()
//@ requires
[? fc] Channel() &
[? fa] atomic_space(? inv) &
is_channel_sep(? sep, inv, this, ? sepPred, ? unsepPred) &
is_channel_unsep(? unsep, inv, this, sepPred, unsepPred) &
is_channel_receive(? receiveClientOp, inv, this, unsepPred, ? pre, ? post) &
sepPred() &
pre();
//@ sema.makeHandle();
sema.acquire();
//@ open channel_sema_inv(this());
String result;
if (itemList.size() == 0) {
    result = null;
} else {
    result = (String)itemList.remove(0);
}
//@ list<Object> items = itemsGhost;
{
//@ predicate P() =
is_channel_sep(sep, inv, this, sepPred, unsepPred) &
is_channel_unsep(unsep, inv, this, sepPred, unsepPred) &
is_channel_receive(receiveClientOp, inv, this, unsepPred, pre, post) &
sepPred() & pre() &
[1/2]itemsGhost |-> items;
//@ predicate Q() =
is_channel_sep(sep, inv, this, sepPred, unsepPred) &
is_channel_unsep(unsep, inv, this, sepPred, unsepPred) &
is_channel_receive(receiveClientOp, inv, this, unsepPred, pre, post) &
sepPred() & pre() &
post(items != nil ? head(items) : null);

lemma void my_ghost_op()
    requires P() & inv();
enforces Q() & inv();
{
    open P();
    sep();
    open ChannelState(_, _);
    itemsGhost = tail(items);
    close ChannelState(tail(items), _);
    receiveClientOp();
    unsep();
    close Q();
}
typedef lemma void channel_sep(predicate() inv, Channel c, predicate() sepPred,
    predicate(list<Object>, int) unsepPred());
requires sepPred() & & inv();
ensures c.ChannelState(?items, ?qms) & & unsepPred(items, qms);

typedef lemma void channel_unsep(predicate() inv, Channel c, predicate() sepPred,
    predicate(list<Object>, int) unsepPred());
requires unsepPred(?items, ?qms) & & c.ChannelState(items, qms);
ensures sepPred() & & inv();

typedef lemma void channel_send(predicate() inv, Channel c, predicate(list<Object>, int)
    unsepPred, String msg, predicate() pre, predicate(boolean) post)(boolean r);
requires pre() & & unsepPred(?items, ?qms);
ensures post(r) & & unsepPred(r ? append(items, cons(msg, nil)) : items, qms);

typedef lemma void channel_receive(predicate() inv, Channel c, predicate(list<Object>,
    int) unsepPred, predicate() pre, predicate(Object) post)();
requires pre() & & unsepPred(?items, ?qms);
ensures post(items != nil ? head(items) : null) & & unsepPred(tail(items), qms);

public final class Channel {
    // @ predicate Channel();
    // @ predicate ChannelState(list<Object> items, int queueSizeMax);
    public Channel();
    // @ requires true;
    // @ ensures Channel() & & ChannelState(nil, _);
    public boolean send(String msg);
    // @ requires
    | [fc] Channel() & &
    | [fa] atomic_space(?inv) & &
    | is_channel_sep(?sep, inv, this, ?sepPred, ?unsepPred) & &
    | is_channel_unsep(?unsep, inv, this, ?sepPred, unsepPred) & &
    | is_channel_send(?send, inv, this, unsepPred, msg, ?pre, ?post) & &
    | sepPred() & & pre();
    // @ ensures
    | [fc] Channel() & &
    | [fa] atomic_space(inv) & &
    | sepPred() & &
    | post(result);
}

Listing 55: Channels.javaspec
A.3 Client - version 1 (II)

The VeriFast files required for this verification are:

- **Client.java**: source file belonging to the “Client” module implementation.
- **Client.jarsrc**: implementation of the “Client” module.
- **Channels.jarspec**: specification of the “Channels” module (presented in A.1).
- **Channels.javaspec**: source file belonging to the “Channels” module specification (presented in A.2).
- **verifast.jarspec**: Specification of the “verifast” module, supplied with VeriFast in the example folder.
- **verifast.javaspec**: source file belonging to the “verifast” module specification, supplied with VeriFast in the example folder.

Only the content of the new files are presented in the next two listings.

Listing 56: Client.javasrc

```java
import channels.*;
import verifast.*;

/*@*/
public String receive();
/*@*/
requires
[?fc] Channel() &&
[?fa] atomic_space(?inv) &&
is_channel_sep(?sep, inv, this, ?sepPred, ?unsepPred) &&
is_channel_unsep(?unsep, inv, this, sepPred, unsepPred) &&
is_channel_receive(?receive, inv, this, unsepPred, ?pre, ?post) &&
sepPred() && pre();
/*@*/
/*@*/
ensures
[fc] Channel() &&
[fa] atomic_space(inv) &&
sepPred() &&
post(result);
/*@*/
}
```

Listing 57: Channels.java

```java
import channels.*;
import verifast.*;

/*@*/
fixpoint boolean non_null(Object o) { return o != null; }

predicate ctor_client_inv(Channel c)() =
c.ChannelState(?items, ?qms) && forall(items, non_null) == true &&
[1/2] Program_sendCount(?sc) &&
[1/2] Program_receiveCount(?rc) &&
```
predicate my_sep_pred() = true;
predicate my_unsep_pred(Channel c) (list<Object> items, int qms) =
  forall(items, non_null) == true &
  [1/2] Program_sendCount(0) &
  [1/2] Program_receiveCount(rc) &
  [1/3] Program_messageMaxCount(mmc) &
  0 <= sc &
  sc <= mmc &
  rc <= sc &
  length(items) == sc - rc &
  length(items) <= qms;
lemma void my_sep()
  requires exists<Channel>(c) & my_sep_pred() &
  client_inv(c);
  ensures c.ChannelState(items, qms) &
  my_unsep_pred(c)(items, qms);
  {
    open client_inv(c);
    assert c.ChannelState(items, qms);
    close my_unsep_pred(c)(items, qms);
  }
lemma void my_unsep()
  requires exists<Channel>(c) & my_unsep_pred(c)(items, qms) &
  c.ChannelState(items, qms);
  ensures my_sep_pred() &
  client_inv(c);
  {
    open my_unsep_pred(c)(_. _);
    close client_inv(c);
    close my_sep_pred();
  }

class SenderThread implements Runnable {
   Channel c;
   //@ predicate pre() = this.c |-> ?c &
   //  [1/2] atomic_space( c.Channel() ) &
   //  [1/2] Program_sendCount(0) &
   //  [1/3] Program_messageMaxCount(mmc)
   //  0 <= mmc;
   //@ predicate post() = this.c |-> ?c &
   //  [1/2] atomic_space( c.Channel() ) &
   //  [1/2] Program_sendCount(0) &
   //  [1/3] Program_messageMaxCount(mmc) &
   //  0 <= mmc;
   SenderThread(Channel c)
   //@ requires [1/2] atomic_space( client_inv(c) ) &
   //  Program_sendCount(0) &
   //  Program_messageMaxCount(mmc) &
   //  0 <= mmc;
   {
     this.c = c;
   }
   public void run()
   //@ requires pre();
   //@ ensures post();
   {
     String m = "Hello";
     int i;
     for (i = 0; i != Program.messageMaxCount; i++)
       //@ invariant this.c |-> ?c &
       //  [1/2] atomic_space( c.Channel() ) &
       //  [1/2] Program_sendCount(i) &
       //  [1/3] Program_messageMaxCount(i) &
       //  0 <= i;
Program_messageMaxCount(?mmc) &\& 1 <= mmc;

```java
for (;;) {
    // invariant this.c |-> c &\& [\_].c.Channel() &\& [1/2] atomic_space(
    //     client_inv(c)) &\& [1/2] Program_sendCount(i) &\& [1/3]
    //     Program_messageMaxCount(?mmc);

    /**
     * predicate P_clientSend() = [1/2] Program_sendCount(i) &\& [1/3]
     * Program_messageMaxCount(?mmc); // &\& i < mmc &\& 0 <= mmc &\& 0 <= i;
     */

    predicate Q_clientSend(boolean r) = [1/2] Program_sendCount(r ? i + 1 : i)
        &\& [1/3] Program_messageMaxCount(?mmc);

    lemma void my_clientSend(boolean r)
        requires i < mmc &\& P_clientSend() &\& my_unsep_pred(c)(?items, ?qms);
    ensures Q_clientSend(r) &\& my_unsep_pred(c)(r ? append(items, cons(m, nil)) : items, qms);
    {
        open P_clientSend();
        open my_unsep_pred(c)(items, qms);
        if (r) Program_sendCount++;
        close Q_clientSend(r);
        forall_append(items, cons(m, nil), non_null);
        close my_unsep_pred(c)(r ? append(items, cons(m, nil)) : items, qms);
    }

    //@ produce lemma function_pointer_chunk(my_unsep) : channel_unsep(
    //     client_inv(c), c, my_unsep_pred, my_unsep_pred(c)) {
    //     close exists(c);
    //     call(); }
    //}@/
    //@ produce lemma function_pointer_chunk(my_clientSend) : channel_send(
    //     client_inv(c), c, my_unsep_pred(c), m, P_clientSend, Q_clientSend)(r)
    //     {
    //     call(); }
    //}@ close post();
```

```java
class ReceiverThread implements Runnable {

    Channel c;

    //@ predicate pre() = this.c |-> ?c &\& [\_].c.Channel() &\& [1/2] atomic_space(
    //     client_inv(c)) &\& [1/2] Program_receiveCount(0) &\& [1/3] Program_messageMaxCount(?mmc) &\& 0 <= mmc;
    //@ predicate post() = this.c |-> ?c &\& [\_].c.Channel() &\& [1/2] atomic_space(
    //     client_inv(c)) &\& [1/3] Program_messageMaxCount(?mmc) &\& [1/2]
    //     Program_receiveCount(?mmc);

    ReceiverThread(Channel c)
        requires [\_].c.Channel() &\& [1/2] atomic_space(client_inv(c)) &\& [1/2]
        Program_receiveCount(0) &\& [1/3] Program_messageMaxCount(?mmc) &\& 0 <= mmc;
    //@ ensures pre();
```
```java
this.c = c;
}

public void run()
//@ requires pre();
//@ ensures post();
{
  for (int i = 0; i < Program.messageMaxCount; i++)
    //@ invariant this.c |-> c &\& [1/2] atomic_space(client_inv(c)) &\& [1/2] Program.receiveCount(i) &\& [1/3] Program.messageMaxCount(?mmc) &\& i <= mmc;
  
  for (; ; )
    //@ invariant this.c |-> c &\& [1/2] atomic_space(client_inv(c)) &\& [1/2] Program.receiveCount(i) &\& [1/3] Program.messageMaxCount(mm);
  
    //@ predicate P_clientReceive() = [1/2] Program.receiveCount(i) &\& [1/3] Program.messageMaxCount(?mmc);
  
    //@ predicate Q_clientReceive(Object r) = [1/2] Program.receiveCount(r == null ? i : i + 1) &\& [1/3] Program.messageMaxCount(?mmc);
  
    //@ lemma void my_clientReceive()
    //@ requires i < mmc &\& P_clientReceive() &\& my_unsep_pred(c)(?items, ?qms);
    //@ ensures Q_clientReceive(items != nil ? head(items) : null) &\& my_unsep_pred(c)(tail(items), qms);
    
    open P_clientReceive();
    open my_unsep_pred(c)(items, qms);
    switch (items) {
      case nil:
        close Q_clientReceive(null);
        close my_unsep_pred(c)(items, qms);
        Program.receiveCount++;
        close Q_clientReceive(head);
        close my_unsep_pred(c)(tail, qms);
      }
    }
    //@
    //@ produce lemma function_pointer_chunk(my_sep) : channel_sep( client_inv(c), c, my_sep_pred, my_unsep_pred(c))()
    //@ requires exist(c);
    //@ exists(c);
    //@ call(); 
    //@
    //@ produce lemma function_pointer_chunk(my_unsep) : channel_unsep( client_inv(c), c, my_sep_pred, my_unsep_pred(c))()
    //@ requires exist(c);
    //@ exists(c);
    //@ call(); 
    //@
    //@ produce lemma function_pointer_chunk(my_clientReceive) : channel_receive(client_inv(c), c, my_unsep_pred(c), P_clientReceive, Q_clientReceive)()
    //@ requires exist(c);
    //@ exists(c);
    //@ call(); 
    //@
    String m = this.c.receive();
    //@ open Q_clientReceive(m);
    if (m != null) break;
  }
  //@ close post();
}
A.4 Client - version 2 (IIIa)

The VeriFast files required for this verification are:

- Client.java: source file belonging to the “Client” module implementation.
- Client.jarsrc: implementation of the “Client” module (presented in A.4).
- Channels.jarspec: specification of the “Channels” module (presented in A.1).
- Channels.javaspec: source file belonging to the “Channels” module specification (presented in A.2).
- verifast.jarspec: Specification of the “verifast” module, supplied with VeriFast in the example folder.
- verifast.javaspec: source file belonging to the “verifast” module specification, supplied with VeriFast in the example folder.

Only the content of the updated file is presented in the next listing.

Listing 58: Channels.java
```java
import channels.*;
import verifast.*;

/*
fixpoint boolean non_null(Object o) { return o != null; }
*/
predicate_ctor client_inv(Channel c) =
    c.ChannelState(?items, ?qms) &&
    for_all(items, non_null) == true;
predicate_ctor my_sep_pred() = true;
predicate_ctor my_unsep_pred(Channel c)(list<Object> items, int qms) =
    for_all(items, non_null) == true;

lemma void my_sep()
    requires exists<Channel>(?c) && my_sep_pred() &&
    client_inv(c);
    ensures c.ChannelState(?items, ?qms) &&
    my_sep_pred(c)(items, qms);
{
    open client_inv(c);
    assert c.ChannelState(?items, ?qms);
    close my_unsep_pred(c)(items, qms);
}

lemma void my_unsep()
    requires exists<Channel>(?c) && my_unsep_pred(c)(?items, ?qms) &&
    c.ChannelState(items, qms);
    ensures my_sep_pred() &&
    client_inv(c);
{
    open my_unsep_pred(c)(?, ?);
    close client_inv(c);
    close my_sep_pred;
}

lemma void mul(int x, int y)
    requires y == z;
    ensures x * y == x * z;
{
    note(mul(x, y) == mul(x, z));
    note(mul(x, y) == x * y);
    note(mul(x, z) == x * z);
}

public class SenderThread implements Runnable {
    //@ int senderSendCount;
    Thread thread;
    JoinableRunnable joinable;

    //@ predicate pre() = [ _ ]Program_channel(?c) &&
    [ _ ]c.Channel() &&
    [ _ ]atomic_space
    (client_inv(c)) &&
    [ _ ]Program_sendMaxCount(?smc) &&
    this.senderSendCount |−> ?
    ssc && ssc==0 && 0 < smc;
    //@ predicate post() = [ _ ]Program_channel(?c) &&
    [ _ ]c.Channel() &&
    [ _ ]atomic_space
    (client_inv(c)) &&
    [ _ ]Program_sendMaxCount(?smc) &&
    this.senderSendCount |−> ?
    ssc && ssc==smc;

    public void run()
    //@ requires pre();
```
String m = "Hello";
for(int i=0; i < Program.sendMaxCount; i++)
//@ ensures post ( ) ;
{
//@ invariant [.] Program_channel(c) &\& [.] c.Channel() &\& [.] atomic_space( client_inv(c) ) &\& this.senderSendCount |-> ?ssc &\& [.]
Program_sendMaxCount(?smc) &\& i == ssc &\& i <= ssc ;
{
for (;;) {
//@ invariant [.] Program_channel(c) &\& [.] c.Channel() &\& [.]
atomic_space(client_inv(c)) &\& this.senderSendCount |-> ssc &\& [.]
Program_sendMaxCount(?smc) &\& i < smc &\& i == ssc ;
{
/*
predicate P_clientSend() = this.senderSendCount |-> ssc &\& [.]
Program_sendMaxCount(?smc) &\& ssc < smc;
predicate Q_clientSend(boolean r) = this.senderSendCount |-> ?ssc2 &\&
ssc2 == ( r ? ssc + 1 : ssc ) &\& [.]Program_sendMaxCount(?smc);
*/
lemma void my_clientSend(boolean r)
requires i < smc &\& P_clientSend() &\& my_unsep_pred(c)(?items , ?qms);
ensures Q_clientSend(r) &\& my_unsep_pred(c)(r ? append(items , cons(m, nil)) : items , qms);
{
open P_clientSend();
open my_unsep_pred(c)(items , qms);
if (r) senderSendCount++;
forall(append(items , cons(m, nil), non_null);
close my_unsep_pred(c)(r ? append(items , cons(m, nil)) : items , qms);
close Q_clientSend(r);
}@*
//@ produce_lemma_function_pointer_chunk(my_sep): channel_sep(
client_inv(c), c, my_sep_pred , my_unsep_pred(c))( ) { close exists(c)
: call(); };}
//@ produce_lemma_function_pointer_chunk(my_unsep): channel_unsep(
client_inv(c), c, my_sep_pred , my_unsep_pred(c))( ) { close exists(c)
: call(); };}
//@ close my_sep_pred();
//@ produce_lemma_function_pointer_chunk(my_clientSend): channel_send(
client_inv(c), c, my_unsep_pred(c), m, P_clientSend , Q_clientSend)(r ) { call(); };}
//@ close P_clientSend();
boolean success = Program.channel.send(m);
//@ open Q_clientSend(success);
if (success) break;
}
//@ assert senderSendCount == Program.sendMaxCount;
//@ close post ( ) ;
}
*/
predicate senderWorker(unit u, SenderThread s; unit u2) =
s.thread |-> ?t &\& s.joinable |-> ?) &\&
t.Thread(j, true) &\& j.JoinableRunnable(s, true) &\& s.getClass() == SenderThread.
class &\&
u2 == unit;
*/
public class Program {
//@ static int sendCount;
//@ static int receiveCount;
}
public static void main(String[] args) {
  //@ requires class_init_token(Program.class) &&
  Program_sendCount(0) && Program_receiveCount(0) &&
  Program_sendMaxCount(0) && Program_receiveMaxCount(0)
  &
  (_) Program_sendCount(0) &< sendersCount & && (_)
  Program_receiveCount(0) &< receiversCount & &&
  Program_sendMaxCount(0) &< smc & && Program_receiveMaxCount(0)
  & && 0 < rmc; @*
  //@ ensures Program_sendCount(smc * sendersCount) & && Program_receiveCount(rmc * receiversCount); @*
  {
    Program_sendCount = 1000;
    Program_receiveCount = 1000;
    Program_sendMaxCount = 2000;
    Program_receiveMaxCount = 2000;
    Program.work();
    //@ assert Program.sendCount == 2000000;
    //@ assert Program.receiveCount == 2000000;
  }
}

public static void work() {
  //@ requires class_init_token(Program.class) &&
  Program_channel(0) & &
  Program_sendCount(0) & & Program_receiveCount(0) & &
  Program_sendMaxCount(0) & & Program_receiveMaxCount(0) & &
  (._) Program_sendCount(0) &< sendersCount & && (._)
  Program_receiveCount(0) &< receiversCount & &&
  Program_sendMaxCount(0) &< smc & && Program_receiveMaxCount(0)
  & && 0 < rmc; @*
  //@ ensures Program_sendCount(smc * sendersCount) & && Program_receiveCount(rmc * receiversCount); @*
  {
    Program_channel = new Channel();
    //@ close client_inv(Program.channel)();
    //@ create_atomic_space(client_inv(Program.channel));
    SenderThread[] senders = new SenderThread[Program_sendCount];
    //@ array_slice_empty_close(senders, 0, senderWorker, unit);
    for (int i = 0; i < Program_sendCount; i++)
      //@ invariant 0 <= i && i < sendersCount &&
      (@) Program_channel(c) & && (._) c.Channel() & && (._) atomic_space(client_inv(0)) & &&
      (._) Program_sendMaxCount(smc) & && 0 < smc &&
      (._) Program_sendCount(sendersCount) & && 0 < sendersCount &&
      array_slice(senders, i, sendersCount, ?elems) & && all_eq(elems, null) ==
      true &&
      array_slice_deep(senders, 0, i, senderWorker, unit, _, _);
    @*
  }

  SenderThread s = new SenderThread();
  JoinableRunnable j = ThreadingHelper.createJoinableRunnable(s);
  //@ close s.pre();
  //@ j.closeIt();
  Thread t = new Thread(j);
  s.thread = t;
  s.joinable = j;
  t.start();
  senders[i] = s;
  //@ array_slice_split(senders, i, i + 1);
ReceiverThread [] receivers = new ReceiverThread[Program.receiversCount];
for (int i = 0; i < Program.receiversCount; i++)
    array_slice_deep_close(receivers, i, receiverWorker, unit);

ReceiverThread r = new ReceiverThread();
JoinableRunnable j = ThreadingHelper.createJoinableRunnable(r);
for (int i = 0; i < Program.sendersCount; i++)
    for (int j = 0; j < Program.receiversCount; j++)
        array_slice_deep(receivers, i, j, unit, 0, receiversCount, receiversWorker);
    receivers[i] = r;
for (int j = 0; j < Program.sendersCount; j++)
    for (int i = 0; i < Program.receiversCount; i++)
        array_slice_deep(receivers, i, j, unit, 0, sendersCount, sendersWorker);
}
class ReceiverThread implements Runnable {
    // @ i n t receiverReceiveCount;
        Thread thread;
        JoinableRunnable joinable;
        //@ pre ( ) = [ ] Program_channel( ? c ) & c . Channel() & atomic_space( client_inv( c ) ) & this . receiverReceiveCount -> ? rrc & & rrc == 0 & 0 < rmc;
        //@ post ( ) = [ ] Program_channel( ? c ) & c . Channel() & atomic_space( client_inv( c ) ) & this . receiverReceiveCount -> ? rrc & & rrc == rmc;
    public void run() {
        // @ requires pre();
        // @ ensures post();
        for ( int i = 0; i < Program_receiveMaxCount(); i++)
            Program_receiveMaxCount( rmc ) & & i <= rmc & & i == rcc;
        for (;;) {
            // @ invariant [ ] Program_channel( c ) & c . Channel() & atomic_space( client_inv( c ) ) & & this . receiverReceiveCount -> rrc & & Program_receiveMaxCount( rmc ) & & i < rmc & & i == rcc;
            predicate P() = this . receiverReceiveCount -> rrc & & Program_receiveMaxCount( rmc ) & & rcc < rmc;
            predicate Q( Object r ) = this . receiverReceiveCount -> ? rrc2 & & rrc2 == ( r == null ? rrc : rrc + 1 ) & & Program_receiveMaxCount( rmc );
            lemma void my_channel_receive() {
                requires i < rmc & & P() & & my_unsep_pred( c )(? items , ? qms);
                ensures Q( items != nil ? head( items ) : null ) & & my_unsep_pred( c ) ( tail( items ) , qms );
                open P();
                open my_unsep_pred( c ) ( items , qms);
                switch ( items ) {
                    case nil:
                        close my_unsep_pred( c ) ( items , qms );
                        close Q( null );
                    case cons( head , tail ):
                        receiverReceiveCount++;
                        close my_unsep_pred( c ) ( tail , qms );
                        close Q( head );
                }
            }
            //@ produce lemma_function_pointer_chunk( my_sep ) : channel_sep( client_inv( c ) , c , my_sep_pred , my_unsep_pred( c ) )();
            //@ produce lemma_function_pointer_chunk( my_unsep ) : channel_unsep( client_inv( c ) , c , my_sep_pred , my_unsep_pred( c ) )();
            //@ close my_sep_pred();
            //@ close my_channel_receive();
            String m = Program_channel.receive();
            if ( m != null ) break;
        }
    }
}
A.5 Client - version 3 (IIIb)

The VeriFast files required for this verification are:

- **Client.java**: source file belonging to the “Client” module implementation.
- **Client.jarsrc**: implementation of the “Client” module (presented in A.4).
- **Channels.jarspec**: specification of the “Channels” module (presented in A.1).
- ** Channels.javaspec**: source file belonging to the “Channels” module specification (presented in A.2).
- **verifast.jarspec**: Specification of the “verifast” module, supplied with VeriFast in the example folder.
- **verifast.javaspec**: source file belonging to the “verifast” module specification, supplied with VeriFast in the example folder.

Only the content of the updated file is presented in the next listing.

Listing 59: Channels.java

```java
import channels.*;
import verifast.*;
//@ fixpoint boolean non_null(Object o) { return o != null; }

predicate c_ctor_client_inv(Channel c) () =
    c.ChannelState( ?items , ?qms ) &&
    for_all( items , non_null ) == true &&
    [1/2] Program_sendMaxCount( ?smc ) &&
    0 < smc &&
    [1/2] Program_receiveMaxCount( ?rmc ) &&
    0 < rmc &&
    [1/3] array_slice_deep( s , 0 , sc , senderWorkerInv , unit , .. , ?lstSc ) &&
    [1/3] array_slice_deep( r , 0 , rc , receiverWorkerInv , unit , .. , ?lstRc ) &&
    length( items ) == sum( lstSc ) - sum( lstRc ) &&
    sum( lstSc ) <= sum( lstSc )

predicate senderWorkerInv(unit u, SenderThread sw; int senderCount) =
    [3/2] sw.senderSendCount => senderCount;
    [3/2] sw.senderSendCount => senderCount1 && senderCount == senderCount1 + 1;

predicate receiverWorkerInv(unit u, ReceiverThread rw; int receiverCount) =
    [3/2] rw.receiverReceiveCount => receiverCount;

fixpoint int sum(list<int> vs) {
```
switch (vs) {
    case nil: return 0;
    case cons(h, t): return h + sum(t);
}

lemma auto void sum_all_eq(list<int> vs)
    requires all_eq(vs, 0) == true;
    ensures sum(vs) == 0;
{
    switch (vs) {
        case nil:
        case cons (x0, xs0): sum_all_eq(xs0);
    }
}

lemma void sum_take_cons_drop(int k, list<int> xs, int x)
    requires 0 <= k && k < length(xs);
    ensures sum(xs) == sum(append(take(k, xs), cons(x, drop(k + 1, xs)))) - x + head(take(1, drop(k, xs)));
{
    switch (xs) {
        case nil:
        case cons(x0, xs0):
            if (k == 0) {
            } else {
                sum_take_cons_drop(k - 1, xs0, x);
            }
    }
}

predicate my_sep_pred () = true;

predicate_ctor my_unsep_pred(Chanvel c)(list<Object> items, int qms) =
forall (items, non_null) == true &&
  [1/2] Program_sendMaxCount(?smc) && 0 < smc &&
  [1/2] Program_receiveMaxCount(?rmc) && 0 < rmc &&
  Program_senders(?sc) &&
  Program_sendersCount(?sc) &&
  [1/3] array_slice_deep(s, 0, sc, senderWorkerInv, unit, ?, ?sc) &&
  [1/3] array_sendersCount(?rc) &&
  [1/3] array_slice_deep(r, 0, rc, receiverWorkerInv, unit, ?, ?rc) &&
  //sum(lstSc) <= smc && sc && sum(lstSc) <= rmc && rc &&
  length(items) <= sum(lstSc) && sc &&
  sum(lstSc) <= sum(lstSc);

lemma void my_sep()
    requires exists<Chanvel>(c) && my_sep_pred() && client_inv(c());
    ensures c.ChannelState(?items, ?qms) && my_unsep_pred(c)(items, qms);
{
    open client_inv(c());
    assert c.ChannelState(?items, ?qms);
    close my_unsep_pred(c)(items, qms);
}

lemma void my_unsep()
    requires exists<Chanvel>(c) && my_unsep_pred(c)(items, qms) && c.ChannelState(items, qms);
    ensures my_sep_pred() && client_inv(c());
{
    open my_unsep_pred(c)(?, ?);
    close client_inv(c());
    close my_sep_pred();
}

/*
fixpoint int mul(int x, int y) { return x * y; }

lemma void note(boolean b) {
    requires b;
    ensures b;
}

lemma void mul_lemma(int x, int y, int z)
    requires y == z;
    ensures x * y == x * z;
{
    note(mul(x, y) == mul(x, z));
    note(mul(x, y) == x * y);
    note(mul(x, z) == x * z);
}

predicate foreach_i<t>(int i, list<t> xs, predicate(int, t) p) =
    switch (xs) {
        case nil: return true;
        case cons(x0, xs0): return p(i, x0) &
        foreach_i(i + 1, xs0, p);
    };

lemma void foreach_i_append<t>(int i, list<t> xs, list<t> ys)
    requires foreach_i(i, xs, ?p) &
    foreach_i(i + length(xs), ys, p);
    ensures foreach_i(i, append(xs, ys), p);
{
    switch (xs) {
        case nil: return p(i, append(xs, ys), p);
        case cons(x0, xs0):
            open foreach_i(i, xs0, p);
            foreach_i_append(i + 1, xs0, ys);
            close foreach_i(i, append(xs, ys), p);
    }
}

public class SenderThread implements Runnable {
    //@ int senderSendCount;
    //@ int myIndex;
    Thread thread;
    JoinableRunnable joinable;

    @*/

    public void run()
        //@ requires pre();
        //@ ensures post();
        {
            String m = "Hello";
        }
int i;
for(i=0; i < Program.sendMaxCount; i++)

/* invariants */
[.] Program_channel() & & [.] c.Channel() & & [.] atomic_space(client_inv(c)) & &
[.] Program_sendMaxCount() & & i <= smc & &
[.] Program_senders(s) & & [.] this.myIndex -> myIndex & &
[.] Program_sendersCount(sc) & & myIndex < sc & &
[1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons( this, nil), cons(i, nil));
*/
{

 /* invariants */
[.] Program_channel(c) & & [.] c.Channel() & & [.] atomic_space(client_inv(c)) & &
[.] Program_sendMaxCount(smc) & & i < smc & &
[.] Program_senders(s) & & [.] this.myIndex -> myIndex & &
[.] Program_sendersCount(sc) & & myIndex < sc & &
[1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons( this, nil), cons(i, nil));
*/
{

/* predicate P_clientSend() = */
[.] Program_sendMaxCount(smc) & &
[.] Program_senders(s) & & [.] this.myIndex -> myIndex & &
[.] Program_sendersCount(sc) & & myIndex < sc & &
[1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons( this, nil), cons(i, nil));

/* predicate Q_clientSend(boolean r) = */
[.] Program_sendMaxCount(smc) & &
[.] Program_senders(s) & & [.] this.myIndex -> myIndex & &
[.] Program_sendersCount(sc) & & myIndex < sc & &
[1/3] array_slice_deep(s, myIndex, myIndex + 1, senderWorkerInv, unit, cons( this, nil), (r ? cons(i+1, nil) : cons(i, nil)));

lemma void mySend(boolean r)
requires i < smc & & P_clientSend() & & my_unsep_pred(c)(?items, ?qms);
ensures Q_clientSend(r) & & my_unsep_pred(c)(?items, ?qms);
{
open P_clientSend();
open my_unsep_pred(c)(?items, ?qms);
SenderThread [] senders = Program.senders;
int sendersCount = Program.sendersCount;
int k = myIndex;
assert [1/3] array_slice_deep(senders, 0, sendersCount,
senderWorkerInv, unit, _, ?oldSenderCounts);
array_slice_deep_split(Program.senders, 0, myIndex);
array_slice_deep_split_promise(1/3, Program.senders, myIndex,
Program.sendersCount, senderWorkerInv, unit, myIndex + 1);
array_slice_deep_open_promise(2/3, Program.senders, myIndex);
int myOldSendCount = senderSendCount;
if (r) {
  senderSendCount++;
}
int myNewSendCount = senderSendCount;
array_slice_deep_close(Program.senders, myIndex, senderWorkerInv, unit);
array_slice_deep_join_promise(1/3, Program.senders, myIndex, myIndex + 1, senderWorkerInv, unit, Program.sendersCount);
array_slice_deep_join_promise(1/3, Program.senders, 0, myIndex, senderWorkerInv, unit, Program.sendersCount);
forall_append(items, cons(m, nil), non_null);
assert 1/3 array_slice_deep(senders, 0, sendersCount, senderWorkerInv, unit, _, ?newSenderCounts);
assert newSenderCounts == append(take(k, oldSenderCounts), cons(myNewSendCount, drop(1, drop(k, oldSenderCounts))));
assert length(oldSenderCounts) == sendersCount;
drop_drop(1, k, oldSenderCounts);
assert newSenderCounts == append(take(k, oldSenderCounts), cons(myNewSendCount, drop(k + 1, oldSenderCounts)));
sum_take_cons_drop(k, oldSenderCounts, newSenderCounts);
assert newSenderCounts == append(take(k, oldSenderCounts), cons(myNewSendCount, drop(k + 1, oldSenderCounts)));
sum(take, cons, drop(k, oldSenderCounts, myNewSendCount);
assert sum(newSenderCounts) == sum(oldSenderCounts) + myNewSendCount - myOldSendCount;
close my_unsep_pred(c)(r ? append(items, cons(m, nil)) : items, qms);

close Q_clientSend(r);
}
//@ produce lemma_function_pointer_chunk(my_sep) : channel_sep(
client_inv(c), c, my_sep_pred, my_unsep_pred(c)) { close exists(c);
call(); };
//@ produce lemma_function_pointer_chunk(my_unsep) : channel_unsep(
client_inv(c), c, my_sep_pred, my_unsep_pred(c)) { close exists(c);
call(); };
//@ close my_sep_pred();
//@ produce lemma_function_pointer_chunk(my_send) : channel_send(
client_inv(c), c, my_unsep_pred(c), m, P_clientSend, Q_clientSend)(r)
{ call(); };
//@ close P_clientSend();
//@ open Q_clientSend(success);
if (success) break;
}
//@ assert i == Program.sendMaxCount;
//@ close post();
}
/*@
predicate senderWorker(unit u, SenderThread SenderWorker; unit u2) =
[3] t.Thread() . true & t & [3].JoinableRunnable(SenderWorker, true) & t &
SenderWorker.getClass() == SenderThread.class & t &
u2 == unit;
*/
predicate senderWorkerNull(unit u, SenderThread SenderWorker; unit u2) =
[3/2] SenderWorker.thread |-? t & t & [3/2] SenderWorker.joinable |-? j & t &
[3/2] SenderWorker.myIndex |-? i & t &
SenderWorker.getClass() == SenderThread.class & t &
u2 == unit;
//@ predicate senderWorkerIndex(int i, SenderThread t) =
[3/4] t.myIndex |-? i & t &
//@
public class Program {
//@ static int sendCount;
//@ static int receiveCount;
}/
public static int sendMaxCount;
public static int receiveMaxCount;
public static Channel channel;
public static int sendersCount;
public static int receiversCount;

public static SenderThread[] senders;
public static ReceiverThread[] receivers;

public static void main(String[] args)
{
    // Requires class_init_token(Program.class) &&
    Program_channel(?channel) &&&
    Program_senders(._) &&& Program_receivers(._) &&&
    [.]Program_sendersCount(?sendersCount) &&& 0 < sendersCount &&&
    Program_sendCount(?rmc) &&& psc==0 &&&
    Program_receiversCount(?rc) &&& prc==0 &&&
    Program_sendMaxCount(?smc) &&& 0 < smc &&&
    Program_receiversMaxCount(?rmc) &&& 0 < rmc &&& rmc==smc &&& sendersCount==
    receiversCount; @&/
    // Ensures Program.sendCount(?sc) &&& Program_receiversCount(?rc)
    &&& sc == (smc * sendersCount) &&& rc == (rmc * receiversCount)
    @&

    @&
    for (int i = 0; i < Program.sendersCount; i++)
    {
        Program.senders[i] = new SenderThread();
        // @& Program.senders[i].myIndex = i;
        // @& array_slice_deep_close(s, 0, senderWorker, unit);
        // @& close (i, Program.senders[i]);
        // @& Program_sendersCount(sendersCount) &&&
        // [1/3]Program_sendMaxCount(smc) &&& 0 < smc &&&
        // array_slice(s, i, sendersCount, ?elems) &&& all_eq(elems, null) == true
        // &&&
        // [1/3]array_slice_deep(s, 0, i, senderWorkerInv, unit, _, ?v) &&& all_eq(
        // v, 0) == true &&&
        // [2/3]array_slice_deep(s, 0, i, senderWorkerNull, unit, _, _);
        @&
    }
    // @& Program.receivers = new ReceiverThread[Program.receiversCount];
    // @& ReceiverThread[] r = Program.receivers;
    // @& leak Program_receivers(r);
    // @& array_slice_deep_empty_close(r, 0, receiverWorker, unit);
    // @& close (i, Program.receivers[i]);
    // @& Program_receiversCount(receiversCount) &&&
    // [1/3]Program_receiveMaxCount(rmc) &&& 0 < rmc &&&
    // array_slice(r, i, receiversCount, ?elems) &&& all_eq(elems, null) ==
    // true &&&
    // [1/3]array_slice_deep(r, 0, i, receiverWorkerInv, unit, _, ?v) &&&
    // all_eq(v, 0) == true &&&
    // [2/3]array_slice_deep(r, 0, i, receiverWorkerNull, unit, _, _);
    @&

    @&
    for (int i = 0; i < Program.receiversCount; i++)
    {
        receivers[i] = new ReceiverThread();
        // @& Program.receivers[i].myIndex = i;
        // @& array_slice_split(receivers, i, i + 1);
        // @& close [2/3]receiverWorkerNull(unit, receivers[i], unit);
        // @& close [1/3]receiverWorkerInv(unit, receivers[i], unit);
        @&
    }
}
//@ array_slice_deep_close_precise(2/3, receivers, i, receiverWorkerNull, unit);
//@ array_slice_deep_close_precise(1/3, receivers, i, receiverWorkerInv, unit);
}

Program.channel = new Channel();
//@ close client_inv(Program.channel)();
//@ create_atomic_space(client_inv(Program.channel));
//@ close foreach_i(0, nil, senderWorkerIndex);

for (int i = 0; i < Program.sendersCount; i++)
//@ invariant 0 <= i && i <= sendersCount &
[@] Program_sendersCount(sendersCount) &
[@] Program_channel(channel) &
[@] atomic_space(client_inv(channel)) &
[2/3] array_slice_deep(s, 0, i, senderWorker, unit, ?elements, _) &
[1/3] array_slice_deep(0, elements, senderWorkerIndex);
  @as/
{
//@ array_slice_deep_split(s, i, i + 1);
//@ array_slice_deep_open_precise(2/3, s, i);
SenderThread sender = senders[i];
//@ close [1/3]senderWorkerInv(0, nil, senderWorkerIndex);
//@ array_slice_deep_close_precise(1/3, senders, i, senderWorker, unit);
JoinableRunnable j = ThreadingHelper.createJoinableRunnable(sender);
//@ sender.myIndex = i;
//@ leak sender.myIndex |= i;
//@ close sender.pre();
//@ i.close();
Thread t = new Thread(j);
t.start();

sender_thread = t;
sender_joinable = j;
//@ close [1/3]senderWorker(unit, sender, unit);
//@ array_slice_deep_close_precise(1/3, senders, i, senderWorker, unit);
//@ close foreach_i(i + 1, nil, senderWorkerIndex);
//@ close senderWorkerIndex(i, sender);
//@ close foreach_i(i, cons(sender, nil), senderWorkerIndex);
//@ foreach_i(0, elements, cons(sender, nil));
}
//@ close foreach_i(0, nil, receiverWorkerIndex);
for (int i = 0; i < Program.receiversCount; i++)
//@ invariant 0 <= i && i <= receiversCount &
[@] Program_receiversCount(receiversCount) &
[@] Program_channel(channel) &
[@] atomic_space(client_inv(channel)) &
[2/3] array_slice_deep(r, 0, i, receiverWorker, unit, ?elements, _) &
[1/3] array_slice_deep(r, 0, i, receiverWorker, unit, ?elements, _) &
  @as/
{
//@ array_slice_deep_split(r, i, i + 1);
//@ array_slice_deep_open_precise(2/3, r, i);
ReceiverThread receiver = receivers[i];
//@ close [1/3]receiverWorkerInv(0, receiver, _);
//@ array_slice_deep_close_precise(1/3, r, i, receiverWorkerInv, unit);
JoinableRunnable j = ThreadingHelper.createJoinableRunnable(receiver);
//@ receiver.myIndex = i;
//@ leak receiver.myIndex |= i;
//@ close receiver.pre();
// j . closeIt ( ) ;
Thread t = new Thread( j ) ;
t . start ( ) ;
receiver . thread = t ;
receiver . joinable = j ;
//@ close [ 1 / 3 ] receiverWorker ( unit , receiver , unit ) ;
//@ array_slice_d e e p _ close _ precise ( 1 / 3 , receivers , i , receiverWorker , unit ) ;
//@ close foreach _ i ( i + 1 , nil , receiverWorkerIndex ) ;
//@ close foreach _ i ( i , cons ( receiver , nil ) , receiverWorkerIndex ) ;
//@ foreach _ i _ append ( 0 , elements , cons ( receiver , nil ) ) ;
}
for ( int j = 0 ; j < Program . sendersCount ; j ++ )
//@ invariant 0 <= j && j <= sendersCount && [ ] . Program_sendersCount ( sendersCount ) && [ ] . Program_senders ( s ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
for ( int j = 0 ; j < Program . receiversCount ; j ++ )
//@ invariant 0 <= j && j <= receiversCount && [ ] . Program_receiversCount ( receiversCount ) && [ ] . Program_receivers ( r ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
for ( int j = 0 ; j < Program . sendersCount ; j ++ )
//@ invariant 0 <= j && j <= sendersCount && [ ] . Program_sendersCount ( sendersCount ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
for ( int j = 0 ; j < Program . receiversCount ; j ++ )
//@ invariant 0 <= j && j <= receiversCount && [ ] . Program_receiversCount ( receiversCount ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
for ( int j = 0 ; j < Program . sendersCount ; j ++ )
//@ invariant 0 <= j && j <= sendersCount && [ ] . Program_sendersCount ( sendersCount ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
//@ invariant 0 <= j && j <= sendersCount && Program_sendCount ( j * smc ) &&
for ( int j = 0 ; j < Program . receiversCount ; j ++ )
//@ invariant 0 <= j && j <= receiversCount && [ ] . Program_receiversCount ( receiversCount ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
//@ invariant 0 <= j && j <= receiversCount && Program_receiveCount ( j * rmc ) &&
ReceiverThread rw = receivers [ j ] ;
ThreadingHelper . join ( rw . thread , rw . joinable ) ;
//@ assert j == jj ;
//@ array_slice_d e e p _ open _ precise ( 1 / 3 , s , j ) ;
//@ open senderWorkerInv ( unit , sw , _ ) ;
//@ Program . sendCount += sw . senderSendCount ;
}
//@ assert Program . sendCount == Program . sendMaxCount * sendersCount ;
ReceiverThread sw = senders [ j ] ;
ThreadingHelper . join ( sw . thread , sw . joinable ) ;
//@ open sw . post ( ) ;
//@ open foreach _ i ( j , elements , senderWorkerIndex ) ;
//@ open senderWorkerIndex ( j , sw ) ;
//@ int jj = sw . myIndex ;
//@ assert jj == jj ;
//@ array_slice_d e e p _ open _ precise ( 1 / 3 , s , j ) ;
//@ open senderWorkerInv ( unit , sw , _ ) ;
//@ Program . sendCount += sw . senderSendCount ;
}
//@ assert Program . sendCount == Program . sendMaxCount * sendersCount ;
ReceiverThread rw = receivers [ j ] ;
ThreadingHelper . join ( rw . thread , rw . joinable ) ;
//@ open rw . post ( ) ;
//@ open foreach _ i ( j , elements , receiverWorkerIndex ) ;
//@ open receiverWorkerIndex ( j , rw ) ;
//@ int jj = rw . myIndex ;
//@ assert jj == jj ;
//@ array_slice_d e e p _ open _ precise ( 1 / 3 , r , j ) ;
//@ open receiverWorkerInv ( unit , rw , _ ) ;
//@ Program . receiveCount += rw . receiverReceiveCount ;
}
//@ assert Program . receiveCount == Program . receiveMaxCount * receiversCount ;
}
//@ assert Program . receiveCount == Program . receiveMaxCount * receiversCount ;
}
//@ assert Program . receiveCount == Program . receiveMaxCount * receiversCount ;
}
//@ assert Program . receiveCount == Program . receiveMaxCount * receiversCount ;
}
//@ assert Program . receiveCount == Program . receiveMaxCount * receiversCount ;
}
public void run()
{
    int i;
    for (i=0; i < Program.receiveMaxCount; i++)
    {
        @ invariant
        [Program_channel] &< [c].channel() &< [atomicspace(client_inv(c))]
        &< [Program_receiveMaxCount] &< [rmc] &< i <= rmc &<&
        [Program_receiveMaxCount] &< [rmc] &< i < rmc &<&
        [Program_receiveCount] &< [rc] &< myIndex < rc &<&
        [array_slice_deep] (s, myIndex, myIndex + 1, receiverWorkerInv, unit, cons( this, nil), cons( i, nil)));
    }
    @ invariant
    [array_slice_deep] (r, myIndex, myIndex + 1, receiverWorkerInv, unit, cons( this, nil), cons( i, nil));
    @ predicate P()
    { [Program_receiveMaxCount] &<&
      [Program_receiveCount] &< [rc] &< myIndex < rc &<&
      [array_slice_deep] (r, myIndex, myIndex + 1, receiverWorkerInv, unit, cons( this, nil), cons( i, nil));
    }
    @ predicate Q(Object result) =
    { [Program_receiveMaxCount] &<&
      [Program_receiveCount] &< [rc] &< myIndex < rc &<&
      [array_slice_deep] (r, myIndex, myIndex + 1, receiverWorkerInv, unit, cons( this, nil), (result != nil ? cons( i + 1, nil) : cons( i, nil)));
    }
    lemma void my_channel_receive()
    requires i < rmc &<& P() &<& my_unsep_pred(c)(?items, ?qms);
    ensures Q(items != nil ? head(items) : nil) &<& my_unsep_pred(c)( tail(items), qms);
```java
{  
    open P();  
    open my_unsep_pred(c)(items, qms);  
    ReceiverThread[] receivers = Program RECEIVERS;  
    int receiversCount = Program RECEIVERSCOUNT;  
    int k = myIndex;  
    assert [1/3] array_slice_deep(receivers, 0, receiversCount,  
        receiverWorkerInv, unit, ?, ?oldReceiverCounts);  
    array_slice_deep_split(1/3, Program RECEIVERS, 0, myIndex);  
    array_slice_deep_open_precise(2/3, Program RECEIVERS, myIndex);  
    array_slice_open_precise(1/3, Program RECEIVERS, 0, myIndex);  
    int myOldReceiveCount = receiverReceiveCount;  
    switch (items) {  
      case nil:  
        case cons(head, tail):  
          receiverReceiveCount++;  
        }  
    int myNewReceiverCount = receiverReceiveCount;  
    array_slice_deep_close(Program RECEIVERS, myIndex, receiverWorkerInv  
        unit());  
    array_slice_deep_join_precise(1/3, Program RECEIVERS, myIndex,  
        myIndex + 1, receiverWorkerInv, unit, Program RECEIVERSCOUNT);  
    array_slice_deep_join_precise(1/3, Program RECEIVERS, 0, myIndex,  
        receiverWorkerInv, unit, Program RECEIVERSCOUNT);  
    //forall append(items, cons(m, nil), non_null);  
    assert [1/3] array_slice_deep(receivers, 0, receiversCount,  
        receiverWorkerInv, unit, ?, ?newReceiverCounts);  
    assert newReceiverCounts == append(take(k, oldReceiverCounts), cons(  
        myNewReceiverCount, drop(1, drop(k, oldReceiverCounts))));  
    assert length(oldReceiverCounts) == receiversCount;  
    assert 0 <= 1 k & 0 <= k & 1 + k <= length(oldReceiverCounts);  
    drop_drop(1, k, oldReceiverCounts):  
    assert newReceiverCounts == append(take(k, oldReceiverCounts), cons(  
        myNewReceiverCount, drop(k + 1, oldReceiverCounts)));  
    sum_take_cons_drop(k, oldReceiverCounts, myNewReceiverCount);  
    assert sum(newReceiverCounts) == sum(oldReceiverCounts) +  
        myNewReceiverCount - myOldReceiveCount;  
    switch (items) {  
      case nil:  
        close my_unsep_pred(c)(items, qms);  
        close Q(null);  
        case cons(head, tail):  
          close my_unsep_pred(c)(tail, qms);  
          close Q(head);  
    }  
    }  
  }  
//@ produce lemma function pointer chunk (my sep) : channel sep(  
    client_inv(c), c, my sep pred, my unsep pred(c))() { close exists(c)  
        ; call(); };  
//@ produce lemma function pointer chunk (my unsep) : channel unsep(  
    client_inv(c), c, my sep pred, my unsep pred(c))() { close exists(c)  
        ; call(); };  
//@ close my sep pred();  
//@ produce lemma function pointer chunk (my channel receive) :  
    channel receive(client_inv(c), c, my unsep pred(c), P, Q)() { call()  
        ; };  
//@ close P();  
String m = Program.channel.receive();  
//@ open Q[m];  
if (m != null) break;  
}  
//@ close post();  
```
A.6 Modifications to the VeriFast libraries

There were some changes made to the VeriFast Java libraries for the verification in \[7,6\]. The file that is modified is "bin\java.lang.javaspec". The following lemmas are modified: the first, second and fifth. The other lemmas are added: the third, fourth, sixed and seventh.

Listing 60: java.lang.javaspec

```java
lemma auto void array_slice_deep_inv<T, A, V>() {
    requires ![f] array_slice_deep<T, A, V>((array, start, end, p, info, elems, vs) =>
    array != null & p <= start & p <= end & length(elems) >= start + length(elems) == end;}

lemma void array_slice_deep_close<T, A, V>(T[], array, int start, predicate(A, T; V) p, A a);
    requires ![f] array_slice<T>((array, start, start + 1, ?elems) & ![f] p(a, head(elems)), ?v);
    ensures ![f] array_slice<T>((array, start, start + 1, p, a, elems, cons(v, nil)));

lemma void array_slice_deep_close_precise<T, A, V>(real f, T[] array, int start, predicate(A, T; V) p, A a);
    requires ![f] array_slice<T>((array, start, start + 1, ?elems) & ![f] p(a, head(elems)), ?v);
    ensures ![f] array_slice<T>((array, start, start + 1, p, a, elems, cons(v, nil)));

lemma void array_slice_open_precise<T, A, V>(real f, T[] array, int start);
    requires ![f] array_slice<T>((array, start, start + 1, ?p, ?a, ?elems, ?vs);
    ensures ![f] array_slice<T>((array, start, start + 1, elems) & ![f] p(a, head(elems), head(vs)) & ![f] head(elems) == cons(head(elems), tail(elems)) & ![f] vs == cons(head(vs), tail(vs)));

lemma void array_slice_split<T, A, V>(T[], array, int start, int start1);
    requires ![f] array_slice<T>((array, start, start1, p, a, take(cons(), start, elem)) & ![f] p(a, head(elems), head(vs)) & ![f] a == cons(head(elems), tail(elems)) & ![f] vs == cons(head(vs), tail(vs));
    ensures ![f] array_slice<T>((array, start, start1, p, a, take(cons(), start, elem)) & ![f] p(a, head(elems), head(vs)) & ![f] head(elems) == cons(head(elems), tail(elems)) & ![f] vs == cons(head(vs), tail(vs));
```
lemma void array_slice_deep_split_precise<T, A, V>(real f, T [] array, int start, int end,
   predicate(A, T; V) p, A a, int start1);
  requires [f]array_slice_deep<T, A, V>(array, start, end, p, a, ?elems, ?vs) &
      start <= start1 &
      start1 <= end;
  ensures [f]array_slice_deep<T, A, V>(array, start, start1, p, a, take(start1 - start, 
      elems), take(start1 - start, vs)) &
      [f]array_slice_deep<T, A, V>(array, start1, end, p, a, drop(start1 - start, 
      elems), drop(start1 - start, vs));

lemma void array_slice_deep_join_precise<T, A, V>(real f, T [] array, int start, int
   start1, predicate(A, T; V) p, A a, int end);
  requires [f]array_slice_deep<T, A, V>(array, start, start1, p, a, ?elems1, ?vs1) &
      [f]array_slice_deep<T, A, V>(array, start1, end, p, a, ?elems2, ?vs2);
  ensures [f]array_slice_deep<T, A, V>(array, start, end, p, a, append(elems1, elems2)
      , append(vs1, vs2));
B Appendix: Spec# - multithread example

In section 4.2.1 a non-concurrent Spec# example was shown. In this verification a multithreaded example is treated. The goal is to verify a simple program with multithreading, locking and shared variables. First we study the problem treated in Owicki and Gries. Consider an integer array `a` and a constant \( N \geq 1 \). The task of the method `FIND` is to give the smallest index \( k \in 1, \ldots, N \) with \( a[k] > 0 \), if such an element of `a` exists, otherwise \( k = N + 1 \) should hold. To add parallel computation in a later stage we split up the finding into two parts, the first part searches for an odd index `k` and the second for an even one. We have an additional variable `i` to keep track of the progress checked odd indexes and a variable `oddtop` to mark the end of the search. The `addtop` variable is also used to store the first found odd index. The first part has to increase `i` by two as long `i < oddtop`, while increasing we check for `a[i] > 0` and if we find such an element we set `oddtop` to the value of `i`. This results in that we loop though array `i` and check for each odd index for a positive value, and we stop when we have found it (and stored in `oddtop`) of we are at the end of the array. For the even indexes it is analogous.

Listing 61: SpecSharp Find example.

```csharp
// 7.4 (Owicki and Gries, simplified for non-shared variable)
public class FIND1
{
    public int k = 0;
    public int N = 0;
    public int topeven = 0;
    public int topodd = 0;
    public int[] a;
    public int i = 0;
    public int j = 0;
    invariant a != null;

    public FIND1(int[] array, int n)
    requires array != null;
    requires 0 <= n;
    ensures a != null;
    ensures 0 <= N;
    {
        this.a = array;
        N = n;
    }

    public int FIND()
    requires a != null;
    requires 0 <= N;
    modifies i, topodd;
    modifies j, topeven;
    modifies k;
    ensures 1 <= k && k <= N + 1;
    ensures forall{int l in 1:k; (l % 2 == 1) == (a[l] <= 0)};
    ensures (k <= N) ==> a[k] > 0;
    ensures k == result;
    ensures (result <= N) ==> a[result] > 0;
    {
        i = 1;
        j = 2;
        topodd = N + 1;
        topeven = N + 1;
        S1();
        //assert forall{int l in 1:topodd; (l % 2 == 1) ==> a[l] <= 0}; //holds!
        S2();
        //assert forall{int l in 1:topodd; (l % 2 == 1) ==> a[l] <= 0};
```
assert j
110
109
{invariant
108
107
invariant forall
invariant
invariant
104
invariant
103
while
102
// ensures a == old(a) && forall
ensures
ensures forall
topodd
ensures
modifies requires j
93
requires a
private void S1() requires a != null; requires 0 <= N; requires i==1 && topodd == N + 1; modifies this.0, i, topodd; ensures 1 <= topodd && topodd <= N + 1; ensures forall(int 1 in (1: topodd); (1 % 2 == 1) => a[1] <= 0); ensures (topodd <= N) => a[topodd] > 0; // ensures a == old(a) && forall{int 1 in (1: N); a[1] == old(a[1])};
while (i < topodd) invariant (1 <= topodd && topodd <= N + 1); invariant (i % 2 == 1); invariant (1 <= i && i <= topodd + 1); invariant forall(int 1 in (1: i); (1 % 2 == 1) => a[1] <= 0); invariant (topodd <= N) => a[topodd] > 0;
assert i < topodd;
assume i < a.Length; /* additional assumption, we know that we are not outside the array */
if (a[i] > 0)
assert a[i] > 0;
topodd = i;
else
assert a[i] <= 0;
i = i + 2;
assert topodd <= i;
}
private void S2() requires a != null; requires 0 <= N; requires j==2; requires topeven == N + 1; modifies this.0, j, topeven; ensures 1 <= topeven;
ensures topeven <= N + 1;
ensures forall(int 1 in (1: topeven); (1 % 2 == 0) => a[1] <= 0);
ensures (topeven <= N) => a[topeven] > 0; // ensures a == old(a) && forall{int 1 in (1: N); a[1] == old(a[1])};
while (j < topeven) invariant (1 <= topeven); invariant (topeven <= N + 1); invariant (j % 2 == 0); invariant (2 <= j && j <= topeven + 1);
assert j < topeven;
assume j < a.Length; /* additional assumption, we know that we are not outside
the array */

if (a[j] > 0)
{
    assert a[j] > 0;
    topeven = j;
}
else
{
    assert a[j] <= 0;
    j = j + 2;
}
assert topeven <= j;

In this example the postconditions of FIND can not be proven, the specification of S1 and S2 can
be proven.