Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Input-Output Verification in Viper

Master Thesis

Vytautas Astrauskas

Friday 30th September, 2016

Advisors: Prof. Dr. Peter Müller, Marco Eilers

Department of Computer Science, ETH Zürich

Abstract

While the Internet becomes more and more pervasive, it gets increasingly important to guarantee that server software works correctly. A possible way to ensure this is to formally verify that the server software implementation satisfies the desired properties. Two examples of properties that are important for servers are that an implementation performs only the allowed input-output (IO) operations and that it eventually performs the required ones. In this report, we present a methodology that allows verifying the IO behaviour of non-terminating processes such as servers. Our methodology is a combination of the work by Penninckx, Jacobs, and Piessens on IO verification with Petri nets [8], which allows verifying IO behaviour of terminating processes, and of the work by Boström and Müller on ensuring finite blocking in non-terminating processes [2]. We have designed an encoding of our methodology in the Viper [7] verification infrastructure's intermediate verification language, implemented it in Nagini, a Python front-end for Viper, and tested it on a large number of examples.

Acknowledgements

First and foremost, I would like to thank my thesis supervisor Marco Eilers for many interesting discussions, inspiring questions, and valuable feedback. I also would like to thank Prof. Dr. Peter Müller for giving me a chance to work on this exciting topic and for discussions about the finite blocking encoding in Chalice2Viper. Last but not least, I would like to thank Dr. Alex Summers and Malte Schwerhoff for fixing issues in Viper back-end verifiers and to the entire Programming Methodology group for helpful discussions and feedback.

Contents

Chapter 1

Introduction

While the Internet becomes more and more pervasive, it becomes increasingly important that its implementation provides strong security guarantees. The SCION (Scalability, Control, and Isolation on Next-Generation Networks) project aims to provide a network architecture that solves the main security issues of the current Internet implementation [1]. At the SCION core, are the protocols that describe how different nodes in a network should communicate. An important part of the project is a verification of the protocols and their implementation. To verify the server implementation, we need a methodology that allows specifying and verifying input-output (IO) behaviour. Moreover, the methodology must be scalable enough to handle the SCION code base.

A common approach to handling the scalability issue is to perform a modular verification where, for example, each procedure is verified independently of others. To verify each procedure separately, one needs to abstract the behaviour of each procedure by its precondition and postcondition. In this setup, the precondition usually denotes what the procedure is allowed to assume, and the postcondition specifies what it has to achieve. For example, a methodology for verifying IO behaviour proposed by Penninckx, Jacobs, and Piessens [8] uses preconditions to specify what IO operations the procedure is allowed to perform and postconditions to specify what state it has to reach. However, this approach does not work with non-terminating procedures, which can often be found in server code, because the execution never reaches the end of the procedure and, as a result, the postcondition does not get evaluated. In this thesis, we, therefore, explore how IO behaviour can be verified without relying on the postcondition being eventually reached. The solution presented in this thesis is a combination of work by Penninckx, Jacobs, and Piessens [8] on using Petri nets for verifying IO behaviour and of work by Boström and Müller [2] on ensuring finite blocking in non-terminating programs. The final methodology allows verifying the

IO behaviour of potentially non-terminating processes and is implemented in Nagini, a Python front-end for the Viper verification infrastructure.

1.1 Motivating Example

```python
class Server(Thread):
  def run():
    server_socket = create_server_socket()
    while True:
      client_socket = server_socket.accept()
      data = client_socket.read_all(timeout=1)
      if data:
        print(client_socket.address)
        client_socket.send("Hello!")
        client_socket.send(data)
      client_socket.close()
```

Listing 1: Echo server.

A simple example in Python of a non-terminating process that illustrates the properties we want to verify is the echo server that is given in Listing 1. When started, the server thread creates a server socket and enters an infinite loop. At the beginning of the loop, it calls `accept()` on the socket and blocks (potentially forever) until some client opens a connection. If the `accept()` call returns, then the server thread performs the needed communication with a client and then calls `accept()` again. On a high level, we would like to verify that the server continues to accept incoming requests. It should be possible to verify such property because all calls in this example except `accept()` can be assumed to be terminating in a finite number of steps. We, therefore, would like to be able to verify that if `accept()` terminates, then the server thread performs all needed communication with the client in a finite number of steps and calls `accept()` again.

More precisely, we would like to have a methodology that allows expressing the following five properties:

progress

An operation will be performed in a finite number of steps.

For example, `server_socket.accept()` will be called after a finite number of steps and if it terminates, then it will be called again after a finite number of steps.

IO obligation

A program is allowed to and has to to perform an IO operation with arguments that satisfy the given requirements expressed in terms of a program state and result values of other IO operations. The IO operation may return a value.

For example, if the implementation managed to receive the data successfully, it has to send it back.

IO credit
A program is allowed to perform an IO operation with arguments that satisfy the given requirements expressed in terms of a program state and result values of other IO operations. The IO operation may return a value.

For example, the implementation is allowed to print the client address to the standard output, but is not required to do so.

strict order
IO operations have to be performed in the specified order.

For example, the implementation has to send `"Hello!"` before it sends `data`.

arbitrary order
IO operations can be performed in an arbitrary order.

For example, the implementation is allowed to print a client address before or after it sends a reply.

Moreover, we would like to verify these properties under the assumptions that:

1. `create_server_socket`, `read_all`, `send`, `print`, and `close` are guaranteed to terminate.
2. The scheduler is fair.

1.2 Contributions and Outline

The contributions of this thesis are:

1. It presents a methodology that allows verifying all five properties given in the previous section, and which is a combination of the methodologies presented in [8] and [2].
2. It describes an implementation of the methodology in Nagini (a Python front-end for the Viper verification infrastructure [7]) that is based on ideas in [6] and [4].
3. It shows how obligations encoding from [2] implemented by Meier [6] in Chalice2Viper, a Chalice front-end for Viper, could be improved.

Obligations ensure that some specific action is eventually performed and together with deadlock freedom ensure finite blocking [2]. This thesis shows how the encoding can be made more expressive and presents a way to fix a discovered unsoundness. It also provides a performance evaluation of the encoding.

4. It presents how work on obligations [2] can be extended to allow transferring obligations between threads via channels, which would allow modeling a common server implementation pattern where one listener thread accepts incoming connections and gives them to the worker thread pull that handles them.

5. It also presents how the finite blocking encoding in Chalice2Viper [6] can be extended to add support for the technique from [2] that guarantees deadlock freedom to fully support finite blocking in Viper.

Chapter 2 presents the existing work on verifying IO [8] and progress [2], and presents our final methodology that is combination of the presented work and that allows verifying IO behaviour of non-terminating processes. Chapter 3 gives a brief introduction to Viper, presents existing ideas from [4] and [6] for encoding VeriFast predicates and obligations into Viper respectively, and describes what was implemented differently in Nagini. Chapter 4 evaluates the final methodology and its implementation in Nagini. Chapter 5 presents obligation channels and how a technique from [2] for ensuring deadlock freedom can be encoded into Viper. The last chapter concludes and presents possible future work.

Chapter 2

Methodology

This chapter describes the methodology that can be used to verify IO behaviour of non-terminating processes. The first section presents existing work: a methodology for verifying the IO behaviour by using Petri nets [8] and a methodology for verifying finite blocking in non-terminating processes [2]. Then, the second section explains how this work can be combined into a methodology that allows verifying the IO behaviour of non-terminating processes.

2.1 Existing work

This section presents the existing work on IO verification with Petri nets by Penninckx, Jacobs, and Piessens [8] and on ensuring finite blocking in non-terminating processes by Boström and Müller [2].

2.1.1 Verification of Input-Output with Petri Nets

This subsection aims to provide an intuition of using Petri nets for IO behaviour specification and verification from [8]. For a more detailed discussion, we refer the reader to [8].

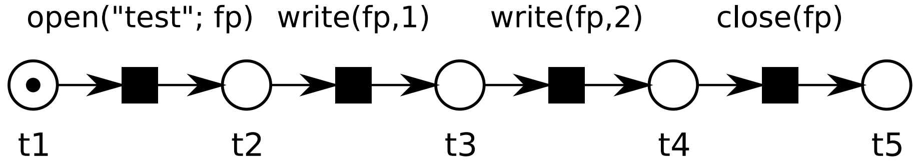

Figure 2.1: A Petri net that denotes a specification of opening a file `"test"`, writing the numbers 1 and 2 into it and closing it. Arguments are separated from results by a semicolon.

Figure 2.1 shows a simple Petri net that specifies that a program is allowed to write the numbers 1 and 2 into the file `"test"`. Transitions denoted by squares model IO operations while places denoted by circles mark states between IO operations. IO operations can have parameters such as the string `"test"` and results such as the file object `fp`. IO operation results can be used as arguments in subsequent operations. The black dot in the place `t1` is a token. Tokens represent at which execution state the program currently is. An IO operation can be executed only if each place in its preset (the set of places from which there is a connection to the transition) has at least one token. Therefore, in Figure 2.1 only the `open` operation could be executed.

Executing an IO operation removes a token from each place in its preset and adds a token to each place in its postset (the set of places to which there is a connection from the transition). In addition, executing a transition removes it from the Petri net. Removing executed transitions from the Petri net effectively prevents one from defining loops that otherwise could be defined by connecting transition's in and out edges to the same place. Instead, the presented approach allows defining composite transitions that can contain arbitrary Petri net fragments, which can include even the transition that is being defined, thus enabling recursion. For example, one could define an IO operation `write_12_to_test()` that contains the Petri net from Figure 2.1. Such composite IO operations are called non-basic while the primitive ones are called basic. An implementation can execute a composite IO operation by opening it and executing the Petri net fragment that defines the IO operation.

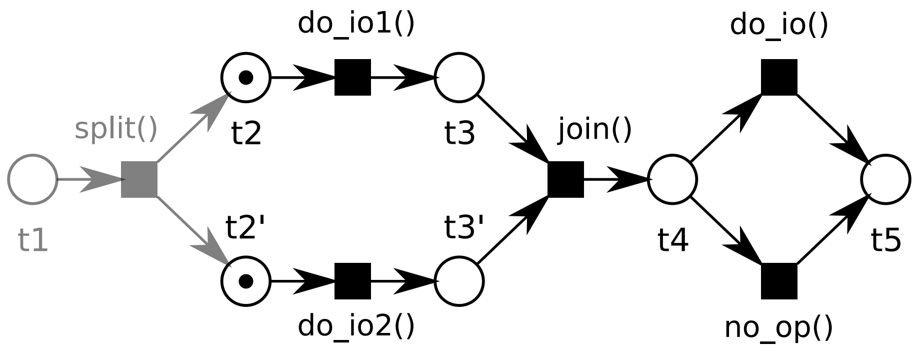

Figure 2.2: A Petri net that demonstrates a parallel execution of the operations `do_io1` and `do_io2`, as well as a non-deterministic choice between performing `do_io` and `no_op`.

Petri nets allow specifying interesting behavioural patterns such as a non-deterministic choice and parallel execution. Figure 2.2 shows an example Petri net that allows executing `do_io1` and `do_io2` in any order. After executing a ghost IO operation `split`, both places `t2` and `t2'` have tokens. Therefore, the implementation is allowed to execute both IO operations `do_io1` and `do_io2` and is free to decide which one to execute first. However, the ghost operation `join` can be executed only when both places `t3` and `t3'` have tokens. This ensures that both `do_io1` and `do_io2` are executed before `join`, as well as before all operations that come after `join`. Figure 2.2 also shows an example of a non-deterministic choice. When place `t4` has a token both `do_io` and `no_op` could be executed, and, as a result, an implementation if free to decide which operation to execute. In this example, one of the choices is a ghost operation `no_op` that means no-operation. Therefore, the specification essentially allows an implementation to choose to perform `do_io` or not.

This methodology for verifying IO behaviour was implemented in VeriFast, a verifier based on symbolic execution that supports C and Java programming languages. Listing 2 shows the contract of a C procedure that promises to perform the IO operation `write_12_to_test()`, whose contents are given in Figure 2.1, in VeriFast syntax. The authors of [8] model Petri net transitions by using VeriFast precise predicates, which, unlike regular predicates, have output parameters that are uniquely determined by the input parameters. The input places are modeled as explicit arguments and output places as explicit results of the predicate. Basic IO operations are modeled by using abstract predicates while predicates representing non-basic IO operations have the Petri net fragment as their body. The procedure that implements a non-basic IO can access its contents by opening the predicate. It is important to note that the correctness of the procedure that implements a basic IO operation cannot be verified and, therefore, the procedure's contract has to be assumed correct. Tokens are also modeled by using a predicate `token` that takes a place as its argument. The ghost IO operations (shown in Listing 3) are modeled in the same way as regular IO operations. However, they are implemented not by regular, but by ghost procedures.

2.1.2 Verification of Finite Blocking

This section aims to briefly explain the technique proposed by Boström and Müller in [2] that allows verifying finite blocking of non-terminating processes. For a more detailed discussion about finite blocking, we refer the reader to [2].

Finite blocking is the property that no thread is blocked forever [2, 6]. The verification technique from [2] that ensures finite blocking is based on two main ingredients: deadlock freedom and obligations. The former is ensured by establishing a global order among threads that guarantees that threads are not waiting for each other in a cycle and is discussed in more detail in Section 5.2.

```
/*@
predicate open(place t1, char *filename; FILE *file, place t2);
predicate write(place t1, FILE *file, int number; place t2);
predicate close(place t1, FILE *file; place t2);
predicate write_12_to_test(place t1; place t2) =
     open(t1, "test", ?fp, ?t2) &*&
     write(t2, fp, 1, ?t3) &*&
     write(t3, fp, 2, ?t4) &*&
     close(t4, fp, ?t5);
@*/
void write_two_ints()
//@ requires token(?t1) &*& write_12_to_test(t1, ?t2)
//@ ensures  token(t2)
{
    //@ open write_12_to_test(t1, t2)
    // Rest of the body omitted.
}
```

Listing 2: The VeriFast encoding of the IO operation `write_12_to_test` whose contents are given in Figure 2.1, and the procedure that implements it. `?x` is VeriFast syntax for expressing that there exists an `x` that satisfies the given conditions. For example, `token(?t1)` means that there exists a predicate `token` in the heap and we can use `t1` to refer to its argument.

Deadlock freedom is not enough to ensure finite blocking in non-terminating programs because a thread can try to acquire a lock that is held by a non-terminating thread that never releases that lock. In this case, the thread that tries to acquire the lock would be blocked forever. The key idea of [2, 3] is to associate an obligation with each action a thread must perform. The authors of [2] focus on using obligations for guaranteeing finite blocking, that is, for ensuring that each blocked thread will be eventually unblocked, for example, by releasing a lock. However, obligations are not limited to unblocking actions as we will see in the next section. The obligation types presented in [2] are:

MustTerminate: An obligation for a method or loop to terminate.

MustRelease: An obligation to release a lock.

MustSend: An obligation to send a message over a channel. It has a dual: a credit MayReceive that denotes a permission to receive from the channel. Each time a credit is created, a corresponding obligation is created, too. We do not cover credits in detail because our work does not need them.

The proposed verification technique guarantees that each obligation is even-

```
/*@
predicate split(place t1; place t2, place t3);
lemma void split();
    requires split(?t1, ?t2, ?t3) &*& token(t1);
    ensures token(t2) &*& token(t3);

predicate join(place t1, place t2; place t3);
lemma void join();
    requires join(?t1, ?t2, ?t3) &*& token(t1) &*& token(t2);
    ensures token(t3);

predicate no_op(place t1, place t2;);
lemma void no_op();
    requires no_op(?t1, ?t2) &*& token(t1);
    ensures token(t2);
@*/
```

Listing 3: The ghost operations from [8] and ghost procedures that implement them.

tually satisfied, which implies that:

1. Obligations are not kept forever – ensured by using a lifetime measure that must decrease with each procedure call and loop iteration.
2. Obligations cannot be leaked – ensured by leak checks.

The property that some action will be eventually performed is a liveness property. A liveness property can be converted into a correctness property by specifying a moment in time until when something good must happen. Therefore, it can be guaranteed that each obligation is eventually satisfied by associating a lifetime measure with each obligation and ensuring with a lifetime check that it always decreases. A measure can be any expression that evaluates to a member of a well-founded set. In this report, we use natural numbers for measures.

Listing 4 shows a Python procedure that recursively prints a sequence from `1` to `n` by using the built-in procedure `print`. Here `Requires` and `Invariant` are special Nagini functions used to specify preconditions and invariants, respectively. Nagini syntax is covered in detail in Section 3.1. `MustTerminate` in the precondition indicates that a procedure takes an obligation to terminate. The lifetime measure expression is provided as an argument to `MustTerminate`. To guarantee that the obligation to terminate is eventually satisfied we require that the measure decreases with each call level and that it is always positive. For example, if `print_seq` had `MustTerminate(n)` instead of `MustTerminate(n+1)`, it would not be able to call `print` when n == 1 because the measure would not be decreasing.

```
def print(value: int) -> None:
    Requires(MustTerminate(1))  # An obligation to terminate.
    # Implementation is assumed to be correct.

def print_seq(n: int) -> None:
    Requires(n > 0)
    Requires(MustTerminate(n+1))  # An obligation to terminate.
    if n > 1:
        print_seq(n - 1)
    print(n)
```

Listing 4: A procedure that prints a sequence of integers from 1 to n. MustTerminate in its precondition means that procedure takes an obligation to terminate.

Loops are handled in a similar way to method calls: with each loop iteration a measure must decrease. Note, however, that a lifetime measure mentioned in a loop invariant is independent of the lifetime measure of the same obligation mentioned in a precondition or the invariant of a different loop. Therefore, the procedure in Listing 5 can have MustTerminate(2) in its precondition.

```
def print_seq(n: int) -> None:
    Requires(n > 0)
    Requires(MustTerminate(2))
    i = 0
    while i < n:
        # Loop and method measures are not compared.
        Invariant(MustTerminate(n-i))
        i += 1
        print(i)
```

Listing 5: A procedure that prints a sequence of integers from 1 to n (like Listing 4), and which illustrates that loop and call measures are not compared.

What we presented so far would not allow verifying the example shown in Listing 6 because the lifetime measure of the MustRelease obligation is not decreasing with each iteration. However, the loop body releases the lock, thus allowing other threads to make progress. To account for cases like this, the authors introduced a concept of a fresh obligation [2, 5]. A fresh obligation is an obligation that was acquired in the current method execution or loop iteration. Fresh obligations are excepted from the lifetime checks, and they can be used everywhere where bounded (non-fresh) obligations are required. In Listing 6, after reacquiring a lock, a thread holds a fresh MustRelease obligation, therefore, the check in the invariant succeeds and the example can be successfully verified.

```python
def watch(l: Lock) -> None:
    Requires(MustRelease(l, 1))
    while True:
        Invariant(MustRelease(l, 1))
        old_value = get_value(l)
        l.release()
        l.acquire()
        new_value = get_value(l)
        if old_value != new_value:
            signal(l, old_value, new_value)
```

Listing 6: A simple watch dog implementation that calls a procedure `signal` when the value guarded by lock `l` changes. To allow other threads to change the value, it reacquires the lock in each loop iteration. Deadlock prevention specifications were omitted.

As was already mentioned, it is guaranteed that obligations are not leaked by performing leak checks. Leak checks can be grouped into two groups:

1. The leak checks that guarantee that a procedure / loop body either satisfies or transfers all obligations it has. For procedures, the leak check is performed after its postcondition was evaluated and all obligations mentioned in the postcondition were removed from the current context. Similarly, for loops, the leak check is performed after the loop invariant was evaluated after the loop body.

2. The leak checks that guarantee that the caller of a procedure or the procedure / loop body that contains a loop either transfers all obligations to the callee / loop, or is guaranteed to have a chance to satisfy them because the callee / loop promises to terminate. For procedure calls, the leak check is performed after the callee's precondition was evaluated and all obligations mentioned in the precondition were removed from the current context. Similarly, for loops, the leak check is performed after the loop invariant was evaluated before the loop.

Obligations are a mechanism to ensure that some action will be eventually performed. The next section discusses how they can be used for ensuring progress in the context of IO verification.

2.2 Verification of Input-Output in Non-Terminating Processes

This section explains why the Petri net approach ([8]) cannot handle the progress property and shows how obligations ([2]) can be used to improve it.

As exemplified in Listing 7, the Petri net approach from [8] handles four out of five properties we would like to verify:

IO credit: an implementation is allowed, but not obliged to perform output, because it can perform `no_op` instead of `perform_output_io` (lines 12 and 13).

IO obligation: an implementation has to read all data from the socket because there is no alternative operation for `read_all_io` (line 11).

arbitrary order: an implementation can either first print the data, or send it back to the client, because the operations are not ordered (lines 4-8).

strict order: an implementation has to send `"hello"` before `data` because `send_io` with `"hello"` precedes the one with `data` (lines 5 and 6).

However, the presented approach cannot ensure the progress property. An example that shows why progress is not guaranteed is given in Listing 8. Both loops are indistinguishable and successfully verify, even though the left one performs the desired IO and the right one does nothing. The following subsections present how the Petri net approach can be extended with ideas from [2] to also verify the progress property.

2.2.1 Combining Petri Nets with Obligations

The challenge in reasoning about progress of IO operations is that often the IO operation's termination depends on the environment. For example, the procedure `read` in Listing 9 terminates only if a user provides input. `write`, however, is not dependent on a user input and as a result it is safe to assume that it always terminates. We would, therefore, like to verify that if `read` terminates, then `write` is eventually called and it eventually terminates.

Our verification approach is an extension of the Petri net approach from [8] and is based on the idea of splitting progress into two parts:

1. **Operation invocation:** verify that if there is a token at some place t, then some IO operation, which has t in its preset, is eventually invoked.
2. **Operation termination:** keep track precisely of which IO operations are guaranteed to terminate under which conditions.

```
/*@
predicate perform_output_io(place t1, SOCKET *client_socket,
                            char *data; place t_end) =
  split(t1, ?t2, ?t6) &*&
  send_io(t2, client_socket, "hello", ?t3) &*&
  send_io(t4, client_socket, data, ?t5) &*&
  print_str_io(t6, data, ?t7) &*&
  join(t5, t7, t_end);
predicate handle_client_io(place t_pre, SOCKET *client_socket;
                           place t_end) =
  read_all_io(t_pre, client_socket, ?data, ?t2) &*&
  perform_output_io(t2, client_socket, data, ?t3) &*&
  no_op(t2, t3) &*&
  close_io(t3, client_socket, t_end);
@*/
void handle_client(SOCKET *client_socket)
/*@ requires token(?t1) &*& client_socket != NULL &*&
             handle_client_io(t1, client_socket, ?t_end)
@*/
//@ ensures token(t_end)
{
    //@ open handle_client_io(t1, client_socket, t_end);
    char *data = read_all(client_socket);
    //@ open perform_output_io(_, _, _, _);
    //@ split();
    print_str("hello");
    send(client_socket, data);
    print_str(data);
    //@ join();
    close(client_socket);
    //@ leak no_op(_, _);
}
```

Listing 7: The example that demonstrates the four properties that can be verified with the Petri net approach ([8]).

These two parts together allow reasoning about procedure's progress from its top-level contract. The following subsections cover each part in more detail.

2.2.2 Operation Invocation

A token leaves its place only when some Petri net transition is taken. Therefore, the requirement that the token leaves its current place is equivalent to

```
while true
/*@
invariant token(?t) &*&
          loop(t, fp)
@*/
{
    //@ open loop(t, fp)
    write(fp, 1)
}
```

(a) Loop that writes `1` in each iteration.

```
while true
/*@
invariant token(?t) &*&
          loop(t, fp)
@*/
{
    // Do nothing.

}
```

(b) Loop that does nothing.

Listing 8: An example where approach from [8] does not guarantee progress.

```
void echo_number()
{
    int x = read();
    write(x);
}
```

Listing 9: A procedure that terminates only if a user provides input.

the requirement to invoke an IO operation (take a transition) that has the token's current place in its preset. If we make a token an obligation to leave a place, then the methodology presented in [2] would ensure that the obligation is eventually satisfied and that some IO operation that has the token's current place in its preset is eventually invoked.

An alternative to making a token an obligation would be to have an obligation to invoke an operation. Then, however, it is not clear how to model the case shown in Figure 2.3 when we want to allow the implementation to choose which operation to perform. Our approach handles this situation trivially because invoking either operation would satisfy the obligation for the token to leave its place.

Always requiring to make progress is sometimes too restrictive. For example, Listing 10 shows a C program that checks if some other program always terminates. Halting problem is undecidable, therefore, we cannot guarantee that the program will eventually print the result if it successfully read the input. However, we would like to be sure that it will eventually print the result if the call to `check_if_halts` terminated. For this case we introduce a credit token `ctoken` that can be used everywhere where an obligation token `token` can be used but is not an obligation. However, we note that the term "credit" could be misunderstood here: `ctoken` is a token that can be used instead of the obligation token while a credit like MayReceive that is a dual of the obligation cannot be used instead of an obligation. We, also, introduce

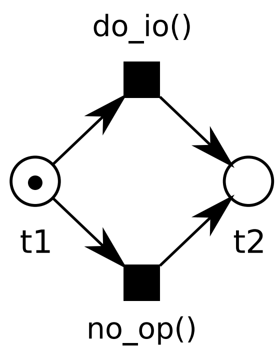

Figure 2.3: A choice between `do_io` and `no_op` operations.

a new ghost IO operation gap that allows converting an obligation token into a credit token as shown in Listing 10.

```
int main()
    /*@ requires token(?t1, 2) &*&
                 read_program_io(t1, ?p, ?t2) &*& gap(t2, ?t3) &*&
                 write_int_io(t3, (halts(p) ? 1 : 0), ?t4)
        ensures token(t4)
    @*/
{
    // At this point we have an obligation token.
    program p = read_program();
    //@ gap()
    // At this point we have a credit token, no requirement
    // to make progress.
    result = check_if_halts(p);
    if (result)
        write_int(1)
    else
        write_int(0)
    return 0;
}
```

Listing 10: A program that checks if a given program always halts. `token(?t1, 1)` is an obligation token at place `t1` with a lifetime measure `1`.

2.2.3 Operation Termination

To keep track of which IO operations are terminating and which are not, we require a programmer to annotate each IO operation with two attributes:

1. A termination condition – a condition under which the IO operation is guaranteed to terminate.
2. A termination measure – an integer expression (similar to an obligation's call measure).

As was already mentioned, our approach is an extension of the Petri net approach from [8]. As a result, we also have basic and non-basic IO operations. For basic IO operations we assume that the information provided by a programmer is correct while for non-basic operations we perform well-formedness checks that ensure that the termination condition implies that:

1. The termination measure is always positive.
2. The termination measure of each IO operation mentioned in the definition is strictly smaller than the termination measure of the enclosing operation or the condition that guards that IO operation is equivalent to `false`.
3. The termination condition of each IO operation mentioned in the definition is equivalent to `true` or the condition that guards that IO operation is equivalent to `false`.
4. The condition that guards each `gap` operation mentioned in the definition is equivalent to `false`.

The guarding condition here is the condition under which the IO operation can be executed. For example, in Listing 11, the operation `read_int_io` can be executed only if `number < 0`.

```
/*@
predicate terminating_io(place t_pre, int number; place t_post) =
    write_int_io(t_pre, number, t_post) &*&
    number < 0 ?
        read_int_io(t_pre, ?value, ?t2) &*&
        write_int_io(t2, value, t_post)
    : true;
@*/
```

Listing 11: An example where an IO operation is guarded by a condition.

This check covers all three sources of non-termination:

1. Non-terminating basic IO operations: with a termination condition check.
2. Infinite sequence of IO operations: with a termination measure checks.
3. Infinite computation, which is represented by the `gap` operation: with the `gap` check.

It is, therefore, guaranteed that if the execution context satisfies the IO operation's termination condition, the IO operation is guaranteed to terminate, assuming that the assumptions about basic IO operations and contracts of the procedures that implement them are correct.

2.2.4 Summary

The presented methodology guarantees that:

1. If there is a token at some place, then some IO operation that has it in its preset will be eventually invoked.
2. If the context satisfies the termination condition of the invoked operation, then the operation is guaranteed to eventually terminate.

Together they guarantee that if there is a path over a Petri net that terminates under some condition b, then the procedure that implements it is guaranteed to eventually reach the final place assuming that this condition b holds at the procedure call time.

Chapter 3

Implementation

This chapter presents the implementation of the combined methodology for verifying IO of non-terminating processes in Nagini, a Python front-end for the Viper verification infrastructure [7]. The first section describes the extensions to the Nagini contract language that are required to specify IO behaviour and obligations. The next section covers the features of the Viper intermediate verification language that are used in our methodology encoding. Section 3.3 describes the encoding of IO operations, and the last section covers the encoding of obligations.

3.1 Syntax

One of the goals of this project was to implement the methodology in Nagini, a Python front-end for the Viper verification infrastructure. We, therefore, had to extend the Nagini specification language to support obligations and IO operations. Nagini currently supports a subset of the Python language, which beside other things includes procedures, classes, single inheritance, methods, and fields. Listing 12 shows a Python procedure that computes a factorial. Preconditions, postconditions and loop invariants are defined by calling the special functions `Requires`, `Ensures`, and `Invariant`. Ghost code, such as unfolding a predicate, is written like regular Python code. Predicates and pure functions are defined by using Python functions annotated with the `@Predicate` and `@Pure` decorators, respectively. In order to verify a Python program, Nagini encodes it into the Viper intermediate verification language and invokes one of the Viper back-end verifiers.

The existing Nagini specification language had the following properties:

1. All contracts were specified in valid Python syntax.
2. All special constructs like `Requires`, `Invariant` and `Implies` were defined in a Python library as regular Python functions and decorators.

```python
@Predicate
def valid(n: int) -> bool:
    return n >= 1

@Pure
def fac(n: int) -> int:
    Requires(n >= 1)
    return n * fac(n-1) if n > 1 else 1

def factorial(n: int) -> int:
    Requires(valid(n))
    Ensures(valid(n) and Unfolding(valid(n), Result() == fac(n)))
    Unfold(valid(n))
    i = 1
    res = 1
    while i != n:
        Invariant(i >= 1 and i <= n)
        Invariant(res == fac(i))
        i += 1
        res *= i
    Fold(valid(n))
    return res
```

Listing 12: A Python procedure that computes a factorial. Specifications are written in syntax supported by Nagini.

3. The verified program had to be well-typed in a nominal type system based on PEP 484[1]. Nagini uses unmodified Mypy[2] to type-check the program and to infer all missing types.

We, therefore, aimed that the extended specification language preserves these properties.

Listing 13 and Listing 14 show a small example that illustrates the final syntax. Most elements have a straightforward syntax with the exception of the `IOExists` construct that is explained in detail below. Listing 14 shows a verified implementation of the `echo` procedure that uses a library to perform IO. The library code is not verified and in Listing 13 only the contracts are provided that are assumed to be correct (this is indicated by the `@ContractOnly` decorator). IO operations are defined in a similar way to predicates: by using a Python function annotated with the `@IOOperation` decorator. IO operation input and output parameters are specified as Python function parameters. The output parameters are distinguished from the input parameters by the

[1]https://www.python.org/dev/peps/pep-0484/
[2]http://mypy-lang.org/

use of `Result()` as a default value. IO operation attributes can be specified by using the functions `Terminates` and `TerminationMeasure`. The default value for `Terminates` is `False`, and the default value for `TerminationMeasure` is `1`. The body of a non-basic IO operation (like `echo_io` in Listing 14) is provided as a single expression in the `return` statement. In order to access the contents of a non-basic IO operation, it has to be opened, which can be done by using the `Open` operation.

As mentioned before, a `token` in Nagini is an obligation. It, therefore, takes a measure as an additional optional argument. Omitting the measure indicates that the obligation is fresh. Another obligation type shown in the example is the `MustTerminate` obligation. A `MustTerminate` obligation is always bounded, therefore, the measure is a mandatory argument.

```
@IOOperation
def read_str_io(t1: Place, val: str = Result(),
                t2: Place = Result()) -> bool:
    Terminates(False)
@ContractOnly
def read_str(t1: Place) -> Tuple[Place, str]:
    IOExists2(Place, str)(
        lambda t2, val: (
            Requires(token(t1, 1) and read_str_io(t1, val, t2)),
            Ensures(token(t2) and Result()[0] == t2 and
                    Result()[1] is val)))
@IOOperation
def write_str_io(t1: Place, val: str, t2: Place = Result()) -> bool:
    Terminates(True)
    TerminationMeasure(1)
@ContractOnly
def write_str(t1: Place, val: str) -> Place:
    IOExists1(Place)(
        lambda t2: (
            Requires(token(t1, 1) and write_str_io(t1, val, t2) and
                     MustTerminate(1)),
            Ensures(token(t2) and Result() == t2)))
```

Listing 13: Unverified library (stubs) that are used in Listing 14.

The main problem with designing a syntax was finding a way to represent Petri nets. The syntax used by VeriFast is intuitive; however, it uses the construct `?x` that serves two use cases in our context:

1. Existential matching of predicates: `token(?t)` means that there exists a predicate `token` in the current heap and `t` is its argument.

```python
@IOOperation
def echo_io(t1: Place, t3: Place = Result()) -> bool:
    Terminates(False)
    return IOExists2(Place, str)(lambda t2, val:
        read_str_io(t1, val, t2) and write_str_io(t2, val, t3))
def echo(t1: Place) -> Place:
    IOExists1(Place)(
        lambda t3: (
            Requires(token(t1, 2) and echo_io(t1, t3)),
            Ensures(token(t3) and Result() == t3)))
    Open(echo_io(t1))
    t2, value = read_str(t1)
    t3 = write_str(t2, value)
    return t3
```

Listing 14: A verified procedure `echo` that uses the library stubs shown in Listing 13.

2. Getting IO operation's result: `no_op(t1, ?t2)` means that `t2` is the output of the operation `no_op`.

Viper does not allow existentially matching predicates. Therefore, in Nagini, a programmer has to provide concrete values as predicate arguments. As a result, Nagini requires that all ghost arguments and results are also normal arguments and results of the Python procedure. For example, instead of existentially matching `t1` in the `read_str` procedure's precondition in Listing 13, `t1` is a formal argument of the procedure.

Supporting `?x` for getting IO operation's result is more complicated because `?t2` in `no_op(t1, ?t2)` is essentially an assignment expression and Python has only assignment statements. If Python had assignment expressions, we could use them for getting an IO operation's result as follows:

```python
(t2 = no_op(t1))
```

Prof. Dr. Peter Müller proposed to use the trick shown in Listing 15. Here `t2` and `value` are existential variables. While the code shown in Listing 15 is valid Python syntax, Mypy fails to infer types of existential variables and, as a result, does not perform the type checking. Moreover, Mypy does not have syntax for specifying types of lambda function arguments. On the other hand, it successfully infers argument types if the type of the whole lambda is given, which includes argument and result types. We, therefore, used a class shown in Listing 16 whose constructor `__init__` takes a sequence of types and whose `__call__` method, which is called when object is called as a function, takes a lambda that takes a sequence of arguments of the provided types. This leads to the final syntax that is shown in Listing 17.

This approach has two drawbacks:

1. We need to define a new class for each number of existential variables.
2. Existential variables are defined only within a `lambda` body, whereas in VeriFast they can be used in the entire procedure. We, therefore, introduced a ghost function `GetGhostOutput` that allows binding the IO operation result to a variable as shown in Listing 18. This is needed in cases where we need to preserve the place which we are trying to reach in the loop invariant.

```
IOExists(
    lambda value, t2: (
        Requires(
            read_int_io(t1, value, t2) and
            write_int_io(t2, value, t3)),
    )
)
```

Listing 15: An initial idea of `IOExists`.

```
class IOExists1(Generic[T1]):
    """``IOExists`` for defining 1 IO existential variable."""

    def __init__(
            self,
            t1: Type[T1]) -> None:
        pass

    def __call__(
            self,
            expr: Callable[[T1], Any]) -> bool:
        pass
```

Listing 16: A definition of `IOExists` that allows defining one existential variable. The method `__init__` is a constructor, and the method `__call__` is a method that is called when a class instance is called as a function.

3.2 Viper

Viper is a verification infrastructure for permission-based reasoning [7]. It consists of an intermediate verification language and two back-end verifiers. One verifier is based on symbolic execution and uses Z3 from Microsoft Research as a theorem prover. The other verifier verifies Viper programs by

```
IOExists2(int, Place)(
    lambda value, t2: (
        Requires(
            read_int_io(t1, value, t2) and
            write_int_io(t2, value, t3)),
    )
)
```

Listing 17: The final `IOExists` syntax.

```
def print_sequence(t1: Place, n: int) -> Place:
    IOExists1(Place)(
        lambda t2: (
            Requires(
                n > 0 and token(t1, 2) and
                print_sequence_io(t1, n, t2) and MustTerminate(2)),
            Ensures(
                token(t2) and t2 == Result())))
    t = t1
    t2 = GetGhostOutput(print_sequence_io(t, n), 't_post')  # type: Place
    while n > 1:
        Invariant(
            token(t, 1) and print_sequence_io(t, n, t2) and
            MustTerminate(n)
        )
        Open(print_sequence_io(t, n))
        t = print_int(t, n)
        n -= 1
    Open(print_sequence_io(t, n))
    t = print_int(t, 1)
    return t
```

Listing 18: An example usage of `GetGhostOutput`. Variable `t2` is assigned the output value `t_post` of the operation `print_sequence_io`. Without this the verifier would not be able to show that the place we reached in the loop is the place we had to reach.

encoding them into Boogie, a verifier from Microsoft Research based on verification condition generation. Both back-end verifiers perform verification in a method modular way. This section presents the Viper language features that are later used for the implementation of the methodology. For a more in depth description of Viper, we refer the reader to [7].

The Viper language is object based, however, it does not support classes. Methods, pure functions, fields, predicates, and mathematical domains are top level language constructs. A method in the Viper language is essentially an impure procedure: it is neither associated with a class or an object, nor does it have an implicit parameter. Also, each object in Viper has all fields defined in the program. However, a method or function is allowed to access only the fields to which it has a permission. Permissions are explained in more detail in the following subsection.

3.2.1 Permissions

As was already mentioned, Viper is a verification infrastructure for permission-based reasoning. Therefore, permissions is a core concept in Viper and they are used to address two problems:

1. **Framing:** if a method has an access permission to a memory location, then it can be sure that the value stored in that location will not be changed by other methods as long as it keeps the permission.
2. **Data race freedom:** a thread is allowed to write to a memory location only if it has a full permission to it, and having a partial permission allows reading the memory location. The underlying logic in Viper relies on the property that the sum of all permissions to a memory location is equal to at most one full permission. Therefore, it is guaranteed that a thread can write to a memory location only when it has an exclusive access to it, which implies the data race freedom.

As already mentioned, fields in Viper are defined globally and permissions are used for denoting which fields can be accessed by using a reference. In the example in Listing 19, two fields `f` and `g` are defined, both of type integer. The method `get_f` has a parameter `arg` of the reference type and requires access to its field `f`. The verifier guarantees that the program accesses only the memory locations to which it has permission. As a result, the assignment to the return value `res2` on line 8 cannot be verified because the permission to access `arg.g` is missing. For it to pass, one would have to add the assertion `acc(arg.g)` to the precondition.

The assertion `acc(arg.f)` denotes a full permission to `arg.f` and is syntactic sugar for `acc(arg.f, write)`. A full permission gives write access to a field while any positive amount of permission gives read access. Our encoding uses only full permissions, therefore, we will not cover read permissions in more detail.

3.2.2 Inhale and Exhale

`inhale` and `exhale` are two basic statements for manipulating permissions. The following `inhale` statement:

```
inhale acc(x.f, p)
```

```
field f: Int
field g: Int
method get_f(arg: Ref) returns (res1: Int, res2: Int)
    requires acc(arg.f)
    ensures acc(arg.f) && res1 == arg.f
{
    res1 := arg.f
  //res2 := arg.g   // Verification error: might not have
                    // permission to access arg.g.
}
```

Listing 19: A simple getter method. The assignment to the result variable `res2` fails to verify because the method does not have a permission to access field `arg.g`.

increases the currently held permission amount to `x.f` by the amount `p`. Similarly, the `exhale` statement:

```
exhale acc(x.f, p)
```

checks that the current permission amount to `x.f` is at least `p`. If the check fails, a verification error is reported. Otherwise, the current permission amount to `x.f` is reduced by `p`. If after the exhale the context holds no permission to `x.f`, all assumptions about the value stored in this memory location are havocked. For pure assertions, `inhale` and `exhale` behave in the same way as `assume` and `assert`, respectively. `inhale` and `exhale` statements are primitive in the sense that they can be used to encode preconditions, postconditions, and loop invariants. For example, calling a method is equivalent to exhaling its precondition and inhaling its postcondition.

Related to `inhale` and `exhale` statements is the inhale-exhale expression:

```
[inhale_part, exhale_part]
```

When an inhale-exhale expression is inhaled, only the `inhale_part` is inhaled. Similarly, when it is exhaled, only the `exhale_part` is exhaled. The inhale-exhale expression is, for example, useful for adding assertions to the method precondition that should be checked on the caller side, but should not be assumed on the method definition side.

3.2.3 Predicates

A predicate in Viper [7, 4] is a language construct that:

1. Allows abstracting over a concrete assertion like equality:

   ```
   predicate Equal(x: Int, y: Int) { x == y }
   ```

2. Allows writing recursive specifications of heap data structures such as a linked list:

   ```
   predicate ListNode(ref: Ref, tail: Seq[Int]) {
       ref != null ==> (
           acc(ref.value) && ref.value == tail[0] &&
           acc(ref.next) && ListNode(ref.next, tail[1..]
       )}
   ```

Similarly to VeriFast predicates, Viper predicates can be abstract:

```
predicate A(r: Ref, v: Int)
```

However, Viper does not have equivalents for VeriFast precise predicates that can have output parameters and coinductive predicates that can be infinitely recursive. An important property of Viper predicates is that we can have more than a full permission to a predicate (which is not possible with fields).

In order to the get permissions and other assertions stored in the predicate body, the predicate must be unfolded, which can be done by using the `unfold` statement:

```
unfold acc(ListNode(x, Seq(3, 2, 1)))
```

To put permissions back into a predicate instance it can be folded by using the `fold` statement:

```
fold acc(ListNode(x, Seq(3, 2, 1)))
```

The `unfold` statement is essentially an exhale of the predicate followed by an inhale of its body, while `fold` is an exhale of the body followed by an inhale of the predicate, except that the verifier does not perform havocking.

3.2.4 `perm` and `forperm`

The permission amount that is currently held to a field `f` can be retrieved by using a `perm` expression:

```
permission_amount := perm(x.f)
```

An interesting property of the `perm` expression is that in an exhale it is evaluated in a non-standard way. All pure expressions, except `perm` and `forperm` (presented below), are evaluated in the heap before the exhale, while `perm` and `forperm` are evaluated in the current heap. As a result, the following exhale would verify in a context that has a full permission to `x.f`:

```
exhale perm(x.f) == write && acc(x.f) && perm(x.f) == none
```

Here `write` denotes a full permission and `none` denotes no permission.

Another expression that allows reasoning about the currently held permission amount is `forperm`. It allows quantifying over all references that have access to the specified field. For example, the following assertion:

```
assert forperm[f] r :: r.f > 0
```

checks that for all references `r` for which we have some positive permission amount to the field `f`, `r.f > 0` holds. In addition to fields, `forperm` also supports predicates that take exactly one argument of type `Ref`.

3.2.5 Functions

Unlike methods, Viper functions are pure. That is, they have no side effects and can be used in expressions. Viper functions can be:

1. Abstract:

   ```
   function get_measure(r: Ref): Int
   ```

2. Concrete:

   ```
   function add(x: Int, y: Int): Int { x + y }
   ```

3. As well as heap dependent:

   ```
   function add(x: Ref, y: Ref): Int
       requires acc(x.value) && acc(y.value)
       ensures result == x.value + y.value
   { x.value + y.value }
   ```

3.2.6 Custom Domains

Viper has a built-in support for sequences, sets, and multisets. It also allows declaring mathematical theories by using a domain construct. Listing 20 shows a domain that defines a new type `Pair` – a pair of references. The declaration uses three uninterpreted functions, one constructor and two getters, whose meaning is defined by two axioms.

3.3 IO Operation Encoding

This section explains how IO operation definitions and their uses are encoded in Viper.

As already mentioned, an IO operation in VeriFast is modeled by using a precise predicate where the input places and the IO operation's input parameters are the input parameters of the predicate, and the output places together with the IO operation's output parameters are the output parameters of the predicate. Modeling IO operations with predicates has the benefit that all bookkeeping is done automatically by the verifier. Viper predicates

```
domain Pair {
    function Pair$create(first: Ref, second: Ref): Pair
    function Pair$first(pair: Pair): Ref
    function Pair$second(pair: Pair): Ref
    axiom Pair$create_first {
        (forall f: Ref, s: Ref ::
            Pair$first(Pair$create(f, s)) == f)
    }
    axiom Pair$create_second {
        (forall f: Ref, s: Ref ::
            Pair$second(Pair$create(f, s)) == s)
    }
}
```

Listing 20: A custom domain that defines a pair of references.

are very similar to VeriFast ones, we, therefore, chose to also model IO operations by using them. However, Viper predicates, unlike VeriFast ones, cannot have output parameters. This raises the question how to translate `IOExists` shown in Listing 21, where `t2` and `data` are existential variables that are assigned the result values of the operation `read_io`.

```
def echo(t1: Place, t3: Place) -> None:
    IOExists2(Place, str)(
        lambda t2, data: (
            Requires(read_io(t1, data, t2) and
                     write_io(t2, data, t3))
        )
    )
    # Method body.
```

Listing 21: Example use of `IOExists`.

The authors of [4] investigated how VeriFast predicates can be encoded in Chalice, a simple object oriented verification language, which also has a Viper front-end. Chalice predicates, unlike Viper and VeriFast ones, support only the implicit `this` parameter. The authors' idea was to model the parameters of a VeriFast predicate with a heap dependent getter functions that take the predicate in their precondition. In our encoding we use a slightly simplified approach because Viper predicates, unlike Chalice ones, can have input parameters. The following subsections provide a detailed description of how we encode various constructs.

3.3.1 IO Operation Definition

Listing 22 shows an example of a non-basic IO operation and Listing 23 shows its encoding in Viper. As can be seen in Listing 23, an IO operation definition encoding has three parts:

1. An abstract predicate.
2. A getter function for each output parameter.
3. A method that performs the termination check (only for non-basic operations).

Viper has no equivalent to VeriFast's coinductive predicate and Viper predicates have inductive semantics. As a result, we cannot use regular Viper predicates to model non-basic IO operations and use `unfold` statement for opening them. Therefore, we decided to encode non-basic IO operations as bodyless predicates. Bodyless predicates cannot be unfolded, therefore, we exhale the predicate corresponding to the IO operation and inhale IO operation's contents. We will explain the reason why havoc does not cause any problems later. As was mentioned before, Viper predicates do not support output parameters (unlike VeriFast ones). Therefore, the emitted predicate has only the input parameters of the IO operation, and output parameters are modeled by using getter functions. Unlike in [4] where the authors are translating arbitrary VeriFast predicates, in our case we do not need a getter function to have a predicate in its precondition. The reason is that the predicate denotes a permission to execute an IO operation once, and, as a result, the IO operation can have only one result. The advantage of having heap independent getters is that a predicate exhale does not havoc knowledge about them. This is the reason why we can translate an IO operation open as an exhale of the predicate and an inhale of its contents. In addition to predicates and getter functions, for non-basic IO operations we also emit a method that has an assertion for each termination check. For example, in Listing 23 on line 6 we check that the termination condition implies that the termination measure is positive.

3.3.2 IO Operation Use in a Contract

The translation of the IO operation use is the same as the translation described in [4] of the VeriFast predicate use: we emit a predicate access and equalities between all getters representing output parameters and actual result expressions. An example of an IO operation use encoding is shown in Listing 24.

```
@IOOperation
def read_write_io(
        t_pre: Place,
        number: int,
        t_post: Place = Result(),
        ) -> bool:
    Terminates(number >= 0)
    TerminationMeasure(2)
    return IOExists2(Place, int)(
        lambda t2, value: (
            (
                read_int_io(t_pre, value, t2)
                if number < 0 else
                (value == number and no_op_io(t_pre, t2)))
            and write_int_io(t2, value, t_post)
        )
    )
```

Listing 22: A non-basic IO operation that writes the given number if it is non-negative. Otherwise, it reads a number and writes it.

3.3.3 IOExists

As presented in Section 3.1, `IOExists` is a construct that allows defining existential variables that can be used to link inputs and outputs of different IO operations. One possibility to encode existential variables would be to use `exists` expressions. These, however, have a poor support in SMT solvers. We, therefore, decided to follow the VeriFast approach[3] and make the first use of the existential variable defining. More precisely, when translating an occurrence of an existential variable we perform the following steps:

1. Check if we already know the variable's definition. If yes, emit the definition and stop.
2. If the variable is mentioned in a result position of an IO operation, then remember the corresponding getter as its definition and stop.
3. If variable is mentioned on the left side of an equality, then set its definition to the right side of the equality and stop.
4. Report an error that the used existential variable is undefined.

An example encoding of existential variables is shown in Listing 25.

[3]In VeriFast, unlike in Nagini, the defining use is indicated syntactically with a question mark preceding the variable name. For example, in `no_op(t1, ?t2)`, `t2` is defined as `no_op` output.

```
predicate read_write_io(t_pre: Ref, number: Int)
function get__read_write_io__t_post(t_pre: Ref, number: Int): Ref
method read_write_io__termination_check(t_pre: Ref, number: Int)
{
  // Termination measure must be positive.
  assert (number >= 0) ==> (2 > 0)
  // Termination condition of read_int_io.
  assert ((number >= 0) && (number < 0)) ==> false
  // Termination measure of read_int_io.
  assert ((number >= 0) && (number < 0)) ==> (2 > 1)
  // Termination condition of no_op_io.
  assert ((number >= 0) && !(number < 0)) ==> true
  // Termination measure of no_op_io.
  assert ((number >= 0) && !(number < 0)) ==> (2 > 1)
  // Termination condition of write_int_io.
  assert (number >= 0) ==> true
  // Termination measure of write_int_io.
  assert (number >= 0) ==> (2 > 1)
}
```

Listing 23: An encoding of the definition of the IO operation from Listing 22.

```
def do_no_op(t1: Place, t2: Place) -> None:
    Requires(no_op_io(t1, t2))
```

(a) A method contract that requires an operation that leads to a specific place.

```
method do_no_op(t1: Ref, t2: Ref)
    requires acc(no_op_io(t1)) &&
             get__no_op_io__t2(t1) == t2
{
}
```

(b) The do_no_op procedure translated into Viper. The argument in the output parameter position was replaced by an equality between the argument and getter.

Listing 24: Encoding of the IO operation use in the method precondition.

```
def write_1_twice(t1: Place) -> Place:
    IOExists3(Place, Place, int)(
        lambda t2, t3, value: (
            Requires(value == 1 and
                     write_io(t1, value, t2) and
                     write_io(t2, value, t3)),
            Ensures(t3 == Result())))
```

(a) An example use of existential variables.

```
method write_1_twice(t1: Ref) returns (_res: Ref)
  requires acc(write_io(t1, 1)) &&
           acc(write_io(get__write_io__t2(t1, 1), 1))
  ensures get__write_io__t2(get__write_io__t2(t1, 1), 1) == _res
```

(b) The `write_1_twice` method contract translated into Viper. The use of the existential variable `value` was replaced with its definition `1`, and the uses of existential variables `t2` and `t3` were replaced with defining getters.

Listing 25: Encoding of existential variables.

3.4 Obligation Encoding

This section describes the main ideas of the obligation encoding in Chalice2Viper [6], a Chalice front-end for the Viper verification infrastructure, and explains the places where we chose a different encoding. For a more detailed discussion on obligation encoding we refer the interested reader to [6].

3.4.1 Modeling Obligations

We model obligations in the same way as described in [6, 29].

The authors of [6, 29] distinguish two types of obligations:

Boolean – we either have, or we do not have an obligation. An example would be an obligation to release a lock (assuming that locks are not reentrant). A boolean obligation is modeled by using permissions to a special field where a full permission denotes having and no permission denotes not having an obligation.

Numeric – we can have zero or more obligations. For example, we can have more than one token at the same place. A numeric obligation is encoded by using permissions to a special predicate where the amount of full permissions denotes the number of obligations.

The following table shows all three obligation types supported in Nagini together with the Python syntax and the Viper field or predicate that is used for modeling the obligation:

	Type	Nagini Obligation	Viper Field / Predicate
Termination	numeric	MustTerminate(m)	MustTerminate
IO	numeric	token(t, m)	MustInvokeBounded
		token(t)	MustInvokeUnbounded
		ctoken(t)	MustInvokeCredit
Locks	boolean	MustRelease(l, m)	MustReleaseBounded
		MustRelease(l)	MustReleaseUnbounded

Even though an obligation to terminate is boolean in nature, we model it as a numeric one to simplify the encoding in the case where a procedure or loop needs several termination measures that are used in different conditions. For example, the procedure in Listing 26 terminates in no more than 5 if the parameter `a` is `True`, and in no more than 3 if the parameter `b` is `True`. As a result, if both `a` and `b` are `True`, then the procedure would require more than one full access to the MustTerminate predicate. We could model MustTerminate as a field and add a check that guarantees that we do not inhale more than one full permission, but we decided that it is nicer to omit the check. Note, however, that even though the example for motivating a numeric type is taken from [6, 39], the underlying reason is different – in our case it is motivated only by a nicer encoding. We use a different encoding for the termination obligation because the one implemented in Chalice2Viper [6] turned out to be unsound. Section 3.4.4 discusses the problem in detail and explains the new encoding.

```python
def potentially_terminating(a: bool, b: bool) -> None:
    Requires(Implies(a, MustTerminate(5)))
    Requires(Implies(b, MustTerminate(3)))
    # Method body.
```

Listing 26: An example method from [6, 39] that requires more than one full access to MustTerminate in its precondition.

3.4.2 Obligation Transfer

When one procedure calls another, or when we enter a loop, the obligations mentioned in the called procedure precondition or the entered loop invariant have to be transfered from one context to another. The encoding of the obligation transfer (except of the termination obligation) is almost the same as described in [6, 30]. The encoding used in Nagini is slightly simpler because we do not support obligation types that have a dual of credit. An example of such an obligation would be an obligation to send a message over a channel with a dual of a credit to receive over the channel. Note that in this sense, the `ctoken` is not a dual of `token`.

As was mentioned in Section 3.2, inhale and exhale are primitives that can be used to encode preconditions, postconditions and loop invariants. We, therefore, explain the obligation transfer in terms of inhale and exhale. The obligation inhale is encoded by simply inhaling a full permission to the field or predicate that represents an obligation. This is true for all obligation types, including the termination obligation.

We encode the exhale of a bounded obligation to release a lock as follows:

1. If we have a bounded obligation and the lifetime check succeeds, we exhale it and stop. Note that sometimes the lifetime check is not performed, for example, when we are exhaling a method postcondition.

2. We exhale an unbounded obligation. Note that if the context does not have the unbounded obligation, then the exhale will fail, which will result in a verification error.

The encoding of the transfer of the `token` is almost the same, but if the second step fails, we do not report an error, but instead try to exhale a credit token `ctoken`. Only if this step fails, we report a verification error. The presented exhale order ensures that at the beginning we try to give away the obligations that have to be satisfied earlier. If the exhaled obligation (either `token` or `MustRelease`) is fresh, we just skip the first step.

In Chalice2Viper [6] all obligation types, including the termination obligation, are encoded in a uniform way. In our case, the termination obligation is treated in a special way: we encode the exhale of a MustTerminate as `true`, which means that the exhale has no effect. Our MustTerminate encoding is discussed in detail in Section 3.4.4.

The problem with encoding obligation inhale and exhale differently is that contracts such as preconditions, postconditions, and loop invariants can be both inhaled and exhaled. Therefore, as described in [6, 44], we use an inhale-exhale expression – a Viper construct that allows specifying a different behaviour for inhale and exhale. Listing 27 shows the contract of a procedure that takes an obligation to release a lock and returns it. Its simplified encoding is shown in Listing 28.

In addition, we do not have to perform the lifetime check before entering a loop, but we have to perform it after the loop body. We solve this problem by introducing a local variable that is set to `true` before the loop and to `false` at the end of the loop body [6, 48].

Unlike [6], we do support fresh obligations in method preconditions and loop invariants. We, therefore, need to perform an additional step: convert these fresh obligations to bounded ones. We do that by exhaling the field / predicate corresponding to a fresh obligation and inhaling the bounded one at the beginning of a method / loop body.

```python
def must_release_example(l: Lock) -> None:
    Requires(MustRelease(l, 1))
    Ensures(MustRelease(l, 1))
```

Listing 27: The contract of the procedure that takes an obligation to release a lock and returns it.

3.4.3 Lifetime

This subsection explains the incompleteness in the obligation lifetime encoding from [6, 48] and presents an improved version.

Original Encoding Incompleteness

In the finite blocking encoding into Boogie that was presented in [2], the default lifetime measure of an obligation that is used when no concrete lifetime measure is provided is $\top$, a value that is larger than any concrete measure.

```
method must_release_example(l: Ref)
  requires [
    acc(l.MustReleaseBounded),
    (perm(l.MustReleaseBounded) > none && /* lifetime check for l */) ?
     acc(l.MustReleaseBounded) : acc(l.MustReleaseUnbounded)]
  ensures [
    acc(l.MustReleaseBounded),
    perm(l.MustReleaseBounded) > none ?
    acc(l.MustReleaseBounded) : acc(l.MustReleaseUnbounded)]
```

Listing 28: A simplified encoding of the procedure's `must_release_example` contract from Listing 31 that shows only the parts relevant for the MustRelease obligation transfer.

However, in the encoding implemented in Chalice2Viper that was presented in [6, 48], the default lifetime measure is unknown because the lifetime is modeled by using an uninterpreted function from a reference, to which an obligation is attached, to a lifetime measure, which is not known unless explicitly specified. This has the consequence that a fresh obligation can be converted into a bounded one only when a concrete measure is provided. An example would be calling a procedure while holding a fresh obligation as shown in Listing 29. However, not allowing to convert a fresh obligation to a bounded one without an explicit measure prevents at least two important use cases:

1. It is not possible to have fresh obligations in method preconditions and loop invariants.
2. If an obligation was created in a procedure and bounded by a loop, it cannot be transferred to a callee. An example is given in Listing 30.

Our new encoding fixes both issues.

```
def callee(l: Lock) -> None:
    Requires(MustRelease(l, 1))
    # ...
def caller() -> None:
    l = Lock()
    l.acquire()      # We get a fresh obligation.
    callee(l)        # Fresh obligation is converted to a bounded
                     # one with measure 1.
```

Listing 29: An example where a fresh obligation is converted to a bounded one with an explicit lifetime measure.

```
def check(l: Lock) -> bool:
    Requires(MustRelease(l, 1))
    # ...
def await() -> None:
    l = Lock()
    l.acquire()     # We get a fresh obligation.
    while True:
        Invariant(MustRelease(l, 1))
        # Here the obligation is bounded, but it has only the loop
        # measure. As a result, the following call fails with an
        # error that the measure is not decreased.
        b = check(l)
```

Listing 30: An example where a call fails because the call measure is unknown.

New Encoding

One approach to keep track of lifetime measures would be to follow the encoding of finite blocking in Boogie presented in [2]. Boogie supports mutable maps and the authors modeled lifetime by using a map from obligations to lifetime measures. When verifying a procedure's body, at first all values in the map are initialized to $\top$. Then, the precondition is evaluated and the map is updated with the minimum of the measure specified in the obligation instance and the value currently stored in the map for that obligation. When a procedure is called, it is checked that each obligation's lifetime measure specified in the called procedure's precondition is strictly smaller than the one stored in the map for that obligation.

Viper supports user defined domains that can be used to axiomatize a map from references to which obligations are attached to measures. However, we would need to also axiomatize our own integers, or use a special value to support $\top$. We can avoid this by observing that in [2], the measure map (either P_{method}, or P_{loop}) has $\top$ for all obligations, except for the bounded ones that are explicitly mentioned in the method precondition / loop invariant. We therefore can model $\top$ as an absence of a key in the map. Then, the check if the measure of some obligation o decreased could be:

$$lifetime\_check(o, map) := o \in map.keys \Rightarrow o.measure < map(o)$$

Here, o is an obligation mentioned in the contract and $o.measure$ is its lifetime measure.

However, the problem with directly modeling the approach from [2] is performance. We need to axiomatize a map that is mutable and always returns the smallest value from all values that were assigned to the specific key. Such an axiomatization would involve many complex quantifiers that are likely to cause performance problems. However, we can observe that the measure map in [2] is initialized at the beginning of the method / loop body, and never changed afterward. Therefore, instead of a map, we can have a local variable, an immutable sequence, that contains all measures mentioned in the contract, which is initialized at the beginning of the method / loop body. Then, the lifetime check would be:

$$\begin{aligned}&lifetime\_check(o, measures) := \\ &\quad (\forall m \in measures : (m.guard \wedge m.obl == o) \Rightarrow m.value > o.measure)\end{aligned}$$

Here, *measures* is a sequence of all measures mentioned in the contract, *m.guard* is a boolean expression that is true iff the obligation is going to be inhaled / exhaled, *m.obl* is the obligation to which this measure belongs to, and *m.value* is the actual value.

Viper has native support for sequences, but it does not support tuples. We, therefore, use a custom domain to define a new type M that is a triple $< reference : Ref, guard : Bool, measure : Int >$ and use it as the type parameter for the measure sequence. Listing 32 shows a simplified encoding of the procedure `caller` from Listing 31. In the method precondition, we cannot access local variables, therefore, each procedure has an additional ghost parameter `_caller_measures` that is the measure sequence of its caller. This parameter is used only on the caller side to check if the measure has decreased and is ignored on the method definition side. The method's measure sequence is constructed by the front-end and stored in a local variable, which is never modified afterward, as shown on line 10 in Listing 32. Storing the sequence in a local variable that is never modified has the advantage that its value can be read inside a loop body. This is not the case, for example, for heap locations: to access the information stored in the field we would have to add a permission to that field to the loop invariant. The measure sequence used in a loop invariant is initialized in exactly the same way. The only difference is that we use a different variable for each loop.

```
def callee(l: Lock) -> None:
    Requires(MustRelease(l, 1))
    l.release()
def caller(l: Lock) -> None:
    Requires(MustRelease(l, 2))
    callee(l)
```

Listing 31: An example call that takes an obligation.

```
method caller(_caller_measures: Seq[M], l: Ref)
    // ...
    requires [/* ... */,
              /* ... */
              lifetime_check(_caller_measures, l, 2)
              /* ... */ ]
    // ...
{
    var _method_measures: Seq[M]
    _method_measures := Seq(M(l, true, 2))
    callee(_method_measures, l)
}
```

Listing 32: A simplified encoding of procedure `caller` from Listing 31.

One interesting difference between our encoding and the one presented in [6, 48] is that in the former, all references can be aliases and the verifier needs to be able to show that they are not aliases, while in the latter, possible aliasing combinations have to be explicitly encoded.

3.4.4 The Termination Obligation and Leak Checks

This subsection explains the unsoundness of the termination obligation encoding from [6] and presents the fixed encoding together with the obligation leak check encoding.

One consequence of modeling obligations as access permissions to either fields or predicates is that we need to have a reference to which they belong. In Chalice2Viper, a termination obligation is associated with the `this` reference. However, Python unlike Chalice allows procedures that are not declared inside a class. We therefore cannot assume that we always have a `self` reference, which is always bound and accessible by both the caller and receiver. As a result, we decided to add an additional argument `_cthread` – the reference to a current thread – to all procedures and methods, and associate the termination obligation with it.

Unsoundness of the Original Encoding

The encoding of the termination obligation implemented in Chalice2Viper [6] turned out to be unsound. Listing 33 shows an example that verifies, but should not.

The problem is that the termination check is performed after the method's postcondition was inhaled. So, if the postcondition is `false`, then this check succeeds.

```python
def non_terminating() -> None:
    Ensures(False)
    while True:
        pass
def terminating_caller() -> None:
    Requires(MustTerminate(2))
    non_terminating()
```

Listing 33: An example illustrating the unsoundness in the termination obligation encoding implemented in [6]. Procedure `terminating_caller` should not verify because it has an obligation to terminate and calls a procedure that does not promise to terminate. However, it verifies.

Promise to Terminate

As mentioned in Section 3.4.2, we only inhale permissions to MustTerminate, but never exhale. We, therefore, use a different mechanism for detecting if a method call / loop promised to terminate. The idea is to generate a boolean expression that yields `true` iff the called method / loop promises to terminate, that is, if there is at least one MustTerminate on the Python level that should be exhaled in the current context. More precisely, we define the termination condition as:

$$tcond := \vee\{guard(o) \wedge lifetime\_check(o) : o \in term\_obligations\}$$

where:

$guard(o)$ is a condition that needs to be true for the specific obligation instance to be inhaled / exhaled (on the Python level);

$lifetime\_check(o)$ is a lifetime check for the specific obligation instance;

$term\_obligations$ is all MustTerminate mentions in the analysed method precondition / loop invariant.

Here we implicitly assume that guarding conditions do not include elements such as `forperm` and `perm` that can yield different values depending on their location in the assertion. This assumption holds for all contracts currently expressible in Nagini.

We also define a variation of the termination condition that ignores the lifetime check:

$$tcond\_ignore\_lifetime := \vee\{guard(o) : o \in term\_obligations\}$$

In the following subsections, $tcond$ and $tcond\_ignore\_lifetime$ are used as macros. That is, their uses are replaced with their definitions.

Leak Check

We define the leak check for all obligations except termination in the same way as [6, 40]:

$\lambda_1 :=$ `forperm [MustInvokeBounded] r1 :: false`
$\lambda_2 :=$ `forperm [MustInvokeUnbounded] r2 :: false`
$\lambda_3 :=$ `forperm [MustReleaseBounded] r3 :: false`
$\lambda_4 :=$ `forperm [MustReleaseUnbounded] r4 :: false`

$$\lambda = \lambda_1 \land \lambda_2 \land \lambda_3 \land \lambda_4$$

We also define a leak check for the termination obligation:

$\lambda_t :=$ `perm(MustTerminate(_cthread)) == none`

Here, we use `perm` instead of `forperm` because the termination obligation can be associated only with `_cthread`, and `perm` should have a better performance.

Method Encoding

We encode methods in almost the same way as described in [6, 45]:

1. We add a leak check at the end of the precondition that checks that either the called procedure promises to terminate, or the caller has no obligations left:
$$[true, tcond \lor (\lambda_t \land \lambda)]$$
   The check is wrapped in an inhale-exhale expression to make sure that it only applies to the call site.

   The main differences from the encoding described in [6, 45] are that we also check for the leak of the termination obligation and have a different way of checking if the method terminates.

2. We also add a leak check (identical to the one described in [6, 45]) at the end of the postcondition that ensures that a method body does not leak obligations:
$$[true, \lambda]$$

The method definition encoding is summarized in Listing 34. Unlike [6, 46], we do not perform any special actions when translating a method call.

Loop Encoding

We encode loops in a similar way to [6, 47]:

```
method m(_cthread: Ref, _caller_measures: Seq[M], /* ... */)
    requires /* Method precondition. */
    requires [true, /* tcond ∨ (λ_t ∧ λ) */]
    ensures /* Method postcondition. */
    ensures [true, /* λ */]
{
    // Method body.
}
```

Listing 34: Summary of the method definition encoding.

1. Similarly to the loop encoding in Chalice2Viper [6, 47], we save the current amount of MustTerminate permission before the loop and restore it after the loop. This is needed to prevent a loop from generating termination obligations. The problem is that after the loop, a loop invariant is inhaled together with all obligations mentioned in it. Listing 35 shows a problematic example.
2. Just before the loop, we save the loop's promise to terminate into a local variable:

   $$termination\_flag := tcond\_ignore\_lifetime.$$

   Using a local variable that is not assigned to inside a loop, has the benefit that the variable's value is known inside the loop.
3. Like in [6, 47], to distinguish if we are exhaling before the loop or after the loop, we use a boolean local variable `exhale_before_loop`. We set its value to `true` before the loop, and to `false` at the end of the loop body.
4. At the end of the loop invariant (similarly to [6, 47]), we add two leak checks:

   a) One guarantees that either the loop promised to terminate, or that the context does not have any obligations:

   $$[true, exhale\_before\_loop \Rightarrow \\ \neg loop\_condition \vee termination\_flag \vee (\lambda_t \wedge \lambda)]$$

   b) The other guarantees that the loop body does not leak obligations:

   $$[true, \neg exhale\_before\_loop \Rightarrow \lambda]$$

   Both leak checks are wrapped into inhale-exhale pairs to ensure that they are only exhaled.

5. At the end of the loop body, we add a check that the loop upholds its promise to terminate:

$$termination\_flag \Rightarrow (tcond \lor \neg loop\_condition)$$

The loop encoding is summarized in Listing 36.

```python
def non_terminating() -> None:
    b = True
    while b:
        Invariant(Implies(not b, MustTerminate(1)))
        b = False
    # If MustTerminate was not reset, the verification would
    # fail because the following loop does not promise to
    # terminate and we inhaled the obligation to terminate from
    # the previous invariant.
    while True:
        pass
```

Listing 35: An example that illustrates why we need to reset the MustTerminate obligation to its previous level.

```
org_amount := perm(MustTerminate(_cthread))
termination_flag := /* tcond_ignore_lifetime */
exhale_before_loop := true
while (b)
    invariant /* Loop invariant. */
    invariant [true,
                exhale_before_loop ==>
                  !b || termination_flag || /* λ_t ∧ λ */]
    invariant [true, !exhale_before_loop ==> /* λ */]
{
    // Loop body.
    exhale_before_loop := false
    assert termination_flag ==> /* tcond */ || !b
}
exhale acc(MustTerminate(_cthread),
           perm(MustTerminate(_cthread)) - org_amount)
```

Listing 36: Summary of the loop encoding.

3.4.5 Acquire and Release

In Python, unlike in Chalice, `acquire` and `release` are not statements, but method calls on the `Lock` object. We, therefore, translate them as normal calls. The contracts that we attached to the `acquire` and `release` methods are shown in Listing 37. The postcondition of the method `acquire` returns a fresh MustRelease obligation, and the precondition of the method `release` takes either a fresh, or a bounded MustRelease obligation. Like [6, 50], we allow the precondition of `release` to have a non-positive measure 0, so that releasing a lock is always possible. The specified contracts are assumed to be correct – the bodies of `acquire` and `release` are not verified. Like [6, 50], Nagini currently does not support deadlock prevention. Therefore, if some thread tries to acquire a lock twice, it will deadlock. However, a possible encoding of the deadlock prevention from [2] is described in Section 5.2.

```
class Lock:
    def acquire() -> None:
        Ensures(MustRelease(self))
    def release() -> None:
        Requires(MustRelease(self, 0))
```

Listing 37: Contracts of the `acquire` and `release` methods.

Chapter 4

Evaluation

This chapter presents the evaluation of our work. The main goal of this thesis was to create a methodology that allows verifying the IO behaviour of non-terminating processes. We succeeded in creating such a methodology and successfully verified all five properties of the motivational example presented in Section 1.1 with Nagini. The following sections evaluates the methodology and the implementation in Nagini in more detail.

4.1 Methodology

Our approach for verifying Input-Output of non-terminating programs is a combination of the methodologies described in [2] and [8]. As a result, it inherits all assumptions that are required for each of these techniques. From [2, 20] it inherits the assumptions that:

1. The thread scheduler ensures strong fairness.
2. Locks are fair.
3. The number of threads is finite.

From [8] we get the assumptions that:

1. Basic IO specifications are correct.
2. The contract provided at the program's entry point is not contradictory.

Our combined approach does not require any additional assumptions, but it has two interesting limitations:

1. It is not possible to verify IO properties without dealing with progress. This is the case because `token` is an obligation to leave a place, which must be eventually satisfied. As a result, once the implementation has a `token`, it is forced to make progress and invoke an IO operation.

2. The combined approach is not expressive enough to capture thread communication via IO. As a result, it cannot ensure their progress. All IO therefore must be modeled as communication with the environment. This limitation in relevant neither for VeriFast, nor for Chalice2Viper, because the former does not guarantee progress and the latter does not support IO.

To summarize, with our methodology that combines finite-blocking with IO it should be possible to verify:

1. All programs that can be verified with [2].
2. All programs that can be verified with [8] and for which it is possible to ensure progress.

4.2 Implementation

This section evaluates the implementation of our approach in Nagini from the perspectives of annotation convenience, completeness, and performance.

4.2.1 Syntax

The syntax for obligations used in Nagini is almost identical to the one used in Chalice2Viper. The two main differences are different capitalization (`MustTerminate` in Nagini and `mustTerminate` in Chalice2Viper) and that in Nagini `MustTerminate` is a special function while in Chalice2Viper it is a keyword.

On the other hand, the syntax used in Nagini for IO specifications is considerably different from the one that is used in VeriFast. As was mentioned in Section 3.1, the main reasons for this are that Python does not support assignment expressions and that Viper does not allow existentially matching predicates. Nagini, therefore, requires more verbose annotations. As can be seen in Listing 39 and Listing 38, Nagini requires to use the `IOExists1` construct to express that there exists such variable `t2` that is the output of the `write_io` operation while VeriFast supports the more elegant syntax `?t2`. Moreover, in Nagini, the input place is an explicit parameter of the procedure while in VeriFast it is not required. However, the Nagini specifications are valid Python code which can be type-checked by unmodified Mypy.

```
void write_int(FILE *fp, int number)
//@ requires token(?t1) &*& write_io(t1, fp, number, ?t2)
//@ ensures  token(t2)
```

Listing 38: An example contract in VeriFast of a procedure that implements a `write` operation.

```
def write_int(fp: File, number: int, t1: Place) -> Place:
    IOExists1(Place)(
        lambda t2: (
            Requires(
                token(t1) and write_io(t1, fp, number, t2)
            ),
            Ensures(
                token(t2) and t2 == Result()
            )
        )
    )
```

Listing 39: An example contract in Nagini of a procedure that implements a `write` operation.

4.2.2 Completeness

This subsection discusses the completeness of our IO verification methodology implementation in Nagini. The implementation of both obligations and IO works with all features supported by Nagini like while and for loops, classes and behavioural subtyping, exceptions, and Viper level error translation back to Python level errors. There are, however, some inconveniences. For example, it is a common pattern that the procedure that implements an IO operation returns the reached place with a token to a caller. However, if the procedure throws an exception, then the exception object must have the reached place as its field. As a result, if the raised exception is not defined by a user, it has to be wrapped as shown in Listing 40. Below follows a more detailed comparison between each original implementation and Nagini.

Obligations: Chalice2Viper

To evaluate the completeness of the obligation encoding, we tried to encode all Chalice2Viper [6] obligation tests to Nagini. We managed to successfully transfer all Chalice2Viper termination and lock tests that do not use any `fork` statements. We had to skip tests with `fork` statements because currently Nagini does not support any mechanism for creating threads.

As was mentioned in Subsection 3.4.4 and Subsection 3.4.3, the Chalice2Viper [6] implementation has known soundness and completeness issues. We fixed all of them, and we are not aware of any soundness or completeness issues in the current obligation implementation.

```python
class OSErrorWrapper(Exception):
    def __init__(self, exception: OSError, place: Place) -> None:
        # ...
        self.exception = exception      # type: OSError
        self.place = place              # type: Place
def mkdir(t1: Place, path: str) -> Place:
    IOExists2(Place, OSErrorWrapper)(
        lambda t2, ex: (
            Requires(
                path is not None and
                token(t1, 1) and
                mkdir_io(t1, path, ex, t2) and
                MustTerminate(1)
            ),
            Ensures(
                token(t2) and
                t2 == Result() and
                ex is None
            ),
            Exsures(OSErrorWrapper,
                ex == RaisedException() and
                Acc(ex.place) and ex.place == t2 and token(t2) and
                Acc(ex.exception) and
                isinstance(ex.exception, OSError)
            ),
        )
    )
    #...
```

Listing 40: An example that shows why we need to wrap exceptions thrown by procedures that perform IO. If the thrown exception does not have the field `place`, the caller would not know in which place the `token` is and would not be able to satisfy an obligation to move from that place.

IO: VeriFast

Similarly to the obligation implementation evaluation, we tried to encode all examples from the `examples/io` folder of the VeriFast distribution[1] into Nagini. We omitted examples from the `work-in-progress` and `template_method` folders from our comparison. Table 4.1 and Table 4.2 summarize the status of the encoding. As can be seen from the tables, the 8 (out of 18) tests we failed to encode are using VeriFast features not related to IO that are currently not

[1]Downloaded from https://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast-15.11-x64.tar.gz.

supported in Nagini.

Encoded	Comment
`buffering_in_library/putchar_with_buffer.c`	
Yes	Additionally verified that the program terminates.
`hello_world/hello_world.c`	
Yes	Additionally verified that the program terminates.
`matching_brackets/matching_brackets_checker.c`	
Yes	Additionally verified that the program will make progress as long as reading succeeds.
`matching_brackets/matching_brackets_input.c`	
No	Uses an unsupported feature: lemmas.
`matching_brackets/matching_brackets_output.c`	
Yes	Additionally verified that the program terminates.
`never_ending_program/infinite_counter.c`	
Yes	Additionally verified that the program will print numbers continuously.
`never_ending_program/yes.c`	
Yes	Additionally verified that the program will print "yes" continuously.
`output_anything/output_any_char.c`	
Yes	The Nagini encoding uses a user defined IO operation instead of `BigStar`. Additionally verified that the program terminates.
`output_anything/output_any_string.c`	
No	Uses an unsupported feature: lists.
`read_files_of_file/read_files_of_file.c`	
No	Uses an unsupported feature: lists.

Table 4.1: The list of VeriFast IO tests and the status of their encoding in Nagini (Part 1).

4.2.3 Performance

This subsection describes how we evaluated the implementation's performance. We decided not to compare Nagini's performance neither with Chalice2Viper, nor with VeriFast. The reason is that most tests are verified in less than 5 seconds and runtime state such as a cold or warm JVM can cause a significant difference in verification time. We instead focused on comparing different Nagini configurations with these goals in mind:

1. Measure the overhead of supporting obligations, which is important to know because even methods that do not use obligations on the Python level still can suffer from the performance degradation. This

Encoded	Comment
`tee/tee_buffered_recursive.c`	
No	Uses an unsupported feature: lists.
`tee/tee_buffered_while.c`	
No	Uses an unsupported feature: lists. However, this test was encoded manually into Viper.
`tee/tee_out.c`	
Yes	Additionally verified termination.
`tee/tee_unbuffered.c`	
Yes	Changed specification to not use lists. Additionally verified that the program will make progress as long as reading succeeds.
`turing_complete/turing_complete.c`	
No	Uses an unsupported features: inductive data types, lists.
`turing_complete/turing_complete_lowtech.c`	
No	Uses an unsupported features: lists.
`turing_complete/turing_complete_underspec_lowtech.c`	
No	Uses an unsupported features: lists.
`user_sets_contract/untrusted_implementation.c`	
Yes	Implemented by using Python inheritance. Additionally verified that the program terminates.

Table 4.2: The list of VeriFast IO tests and the status of their encoding in Nagini (Part 2).

can happen because code for handling termination and obligation leak checks are always emitted. Moreover, such a performance test was not performed as part of the obligations' implementation in Chalice2Viper [6].

2. Measure how well the new termination obligation encoding that fixes an unsoundness performs compared to the one described in [6]. We do not compare the old lifetime encoding with the new one because the old one cannot handle most IO tests and we have not implemented it.

For experiments, we used the platform described in Table 4.3 and the configurations shown in Table 4.4. The **NO** configuration uses the same code as the **NEW** configuration, we just pass an additional flag to Nagini that instructs to:

1. Remove all obligation related elements such as leak checks and lifetime handling from the encoding.
2. Translate all uses of MustTerminate as `true`.
3. Translate all uses of `token` as uses of `ctoken`.

Property	Value
Processor	`Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz`
Memory	`16 GB`
OS	`Ubuntu 16.04.1 LTS`
Java	`OpenJDK 64-Bit 1.8.0_91`
Mono	`4.2.1`
Z3	`4.4.0`
Boogie commit	`56916c9d12f608dc580f4da03ef3dcbe35f42ef8`
SE	Project name: Silicon Changeset: `bf63c344c47a`
VCG	Project name: Carbon Changeset: `63e01a20b4f5`
Language	Project name: Silver Changeset: `f49a3481e780`

Table 4.3: Platform used for performance evaluation. SE is a Viper back-end verifier based on symbolic execution, and VCG is a back-end verifier based on verification condition generation.

ID	Description	Translation Changeset	Contracts Changeset
NO	Disabled obligations.	`9f6cb4449d2a`	`0c7917661ab2`
OLD	Implementation with termination handling from [6].	`2bfd4df21cb3`	`0c7917661ab2`
NEW	The final obligation implementation with fixed termination handling.	`9f6cb4449d2a`	`0c7917661ab2`

Table 4.4: Three configurations used in the performance evaluation.

We conducted two experiments:

1. Run Nagini with each configuration on a subset of its test suite and measure the overall running time.
2. Run Nagini with each configuration on a subset of its test suite and measure the running time of each test. We used almost the same test suite as in the first experiment, we just inserted four large tests at the beginning of the test suite to warm up the JVM.

We used the tests from the changeset `9f6cb4449d2a`, from which we omitted:

1. Translation tests because they do not test verification.
2. Obligation tests (folder `obligations`) because almost all of them would

	NO		OLD			NEW		
Verifier	T (s)	σ (s)	T (s)	σ (s)	S (%)	T (s)	σ (s)	S (%)
SE	152	3.47	180	6.72	19	169	2.67	11
VCG	251	0.98	406	1.32	61	294	1.06	17

Table 4.5: Results of the first experiment: run time of the test suite with different configurations. T – average run time in seconds, σ – standard deviation, S – slowdown in percent compared to **NO** configuration.

fail with the **NO** configuration.

3. Tests that were failing in the **OLD** configuration (five in total) because they relied on bug fixes that were made between the **OLD** and **NEW** configurations.

We also modified four tests to work in all configurations. The final test set had 83 tests. In both experiments we ran Nagini 10 times with each configuration. For measuring time, we used a built-in feature of the py.test[2] framework, which is used for testing Nagini.

Table 4.5 shows the results of the first experiment. As can be seen from the table, obligations cause a moderate overhead with the back-end verifier based on symbolic execution (SE). The slowdown with the **OLD** configuration is 19%, with **NEW** only 11%. However, the back-end verifier based on verification condition generation (VCG) is more sensitive to obligations: with the **NEW** configuration it has 17% overhead, and with **OLD** 61%. To sum up, the performance decrease with the **NEW** configuration over the entire test suite does not seem to be worrisome and is significantly better compared to **OLD** configuration.

The second experiment results for SE verifier also do not show any potential problems. Table 4.6 shows the five tests that experienced the largest slowdown with the **NEW** configuration compared to the **NO** configuration when using a SE back-end verifier. As expected from the first experiment results, the **NEW** configuration performed slightly better than the **OLD** one. The increase in verification time of these five tests can be explained by two reasons. The test `terminating.py`, as its name indicates, tests termination and uses MustTerminate a lot. It is, therefore, expected that such a test would perform worse with enabled obligations. The other four tests contain many small methods and, as a result, the additional obligation handling code constitutes a larger proportion of the emitted code. This additional code therefore causes a more significant growth in verification time. Moreover, this increase should be a constant factor per method and should not cause prob-

[2] http://docs.pytest.org/en/latest/

	NO		OLD			NEW		
Test	T (s)	σ (s)	T (s)	σ (s)	S (%)	T (s)	σ (s)	S (%)
`.../test_behavioural_subtyping.py`								
	4.45	0.06	8.21	0.08	84	6.70	0.07	51
`.../io/master/terminating.py`								
	1.64	0.02	2.36	0.03	44	2.04	0.02	25
`.../test_while.py`								
	1.18	0.02	1.47	0.01	25	1.45	0.01	23
`.../issues/00001.py`								
	0.95	0.01	1.32	0.02	40	1.16	0.01	23
`.../test_super.py`								
	1.03	0.06	1.45	0.01	41	1.24	0.01	21

Table 4.6: Results of the second experiment with the Viper back-end verifier based on symbolic execution: five tests that had the largest slowdown in configuration **NEW** compared to configuration **NO**. T – average run time in seconds, σ – standard deviation, S – slowdown in percent compared to **NO** configuration.

lems for using Nagini with the SE back-end verifier for verification of larger pieces of code.

However, the second experiment results reveal some performance problems with the VCG back-end verifier. Similarly to Table 4.6, Table 4.7 shows the five tests that had the largest slowdown when using the VCG back-end. Unlike the SE verifiers's five tests with the most significant performance degradation, the VCG verifiers's ones are the tests that have only a few, but very large methods. Moreover, most tests were significantly slower with the **OLD** configuration (the slowest test was almost 5 times slower with the **OLD** than the with **NEW** configuration) that emits additional statements for handling termination. It therefore seems that VCG verifier has problems with handling methods that inhale and exhale permissions many times. This is likely to be a problem for using Nagini with the VCG back-end verifier in larger projects, even though, it performs significantly better with the **NEW** configuration.

4.2.4 Conclusion

In this section, we evaluated the implementation of our methodology for verifying IO of non-terminating processes in Nagini. We discovered several issues, from which the most notable is that the verification condition generation (VCG) back-end verifier does not scale well with our encoding, despite a significant performance improvement between the old and the fixed ter-

	NO		OLD			NEW		
Test	T (s)	σ (s)	T (s)	σ (s)	S (%)	T (s)	σ (s)	S (%)
`.../io/master/example1.py`								
	5.04	0.05	70.92	1.35	1308	14.65	0.09	191
`.../io/master/example1_fork.py`								
	4.86	0.04	30.78	0.29	534	9.56	0.30	97
`.../io/test_builtins.py`								
	3.41	0.05	9.97	0.05	193	5.88	0.05	72
`.../io/verifast/never_ending_program/infinite_counter.py`								
	4.26	0.07	5.82	0.07	37	6.38	0.07	50
`.../io/verifast/matching_brackets/matching_brackets_output.py`								
	4.22	0.09	8.21	0.08	94	5.70	0.06	35

Table 4.7: Results of the second experiment with the Viper back-end verifier based on verification condition generation: five tests that had the largest slowdown in configuration **NEW** compared to configuration **NO**. T – average run time in seconds, σ – standard deviation, S – slowdown in percent compared to **NO** configuration.

mination obligation encoding. We, nevertheless, believe that Nagini, as it becomes more mature, could be used for verifying complex programs with IO when used with the symbolic execution back-end verifier.

Chapter 5

Extensions

This chapter presents two extensions to the work on which our work is based. The first one is an extension to the finite blocking technique [2] that allows sending an obligation over a channel. If this extension was implemented in Nagini, it would be possible to verify a common server implementation pattern where one listener thread accepts incoming connections and forwards them to the pool of worker threads that handle them. The second one is an extension to the encoding of finite blocking in Viper [6] that describes how the technique that guarantees deadlock freedom can be encoded into Viper. Implementing deadlock freedom in Nagini would allow completely verifying the finite blocking property.

5.1 Obligation Channels

This section describes an extension to [2] that allows sending obligations over a channel.

5.1.1 Motivation

A common pattern for writing servers is to have one listener thread that accepts incoming connections and puts them into a queue from which they are picked by worker threads. This allows having a larger throughput on the server because several clients can be handled simultaneously. Listing 41 shows a variation of the echo server example (Listing 1), which we would like to verify. The main problem here is how to transfer an obligation from one thread to another in such a way that it is still guaranteed that the obligation will eventually be satisfied.

```python
def main():
    queue = ConcurrentQueue()
    worker = Worker()
    worker.start(queue)
    listener = Listener()
    listener.start(queue)
class Listener(Thread):
    def run(self, queue):
        server_socket = create_server_socket()
        while True:
            client_socket = server_socket.accept()
            queue.put(client_socket)
class Worker(Thread):
    def run(self, queue):
        while True:
            client_socket = queue.pop()
            handle_client(client_socket)
```

Listing 41: A modified version of the echo server example from Listing 1, where a client is handled in a separate thread. Here, `handle_client` is a procedure that performs all required communication with a client.

5.1.2 Credit Channels

Chalice [5] has a channel construct (which we will call a credit channel) that can be used for sending data from one thread to another. A credit channel has an invariant which is exhaled on the sender side and inhaled on the receiver side, and which can contain pure assertions, permissions, and credits. An example of a credit channel in possible Python syntax is shown in Listing 42. The channel transfers a reference to a data object with permission to access its field `x` and guarantee that this field's value is larger than five. The channel type is defined by inheriting from the generic class `Channel` that takes the type of the transfered object as a parameter. A channel invariant can be specified by overriding a ghost method `Invariant` and providing the actual invariant in the `return` statement. The `receive` operation blocks the receiving thread until there is a message on the channel. To prevent deadlocking, it is therefore required for the thread to have a credit to receive from the channel, which means that some thread has an obligation to send a message over that channel.

5.1.3 Obligations in Channel Invariants

A channel that allows obligations in its invariant could be used instead of the `queue` in Listing 41. However, it is unsound to allow obligations in the credit

```python
class Data:
    def __init__(self) -> None:
        Requires(MustTerminate(1))
        Ensures(Acc(self.x))
        self.x = 0      # type: int
class C(Channel[Data]):
    def Invariant(self, value: Data) -> bool:
        return Acc(value.x) and value.x > 5
def receiver(c: C) -> Data:
    Requires(MayReceive(c, 1))  # 1 credit to receive from
                                # channel.
    Ensures(Acc(Result().x) and Result().x > 5)
    return c.receive()
def sender(c: C) -> None:
    Requires(MustSend(c, 1, 1)) # 1 obligation to send over a
                                # channel.
    d = Data()
    d.x = 6
    c.send(d)
def main() -> None:
    c = C()
    sender(c)
    receiver(c)
```

Listing 42: A credit channel example in possible Python syntax.

channel invariants because there is no guarantee that any thread will ever read from the channel. Therefore, there is no guarantee that an obligation sent over a credit channel would be eventually satisfied. As a result, in order to be able to send obligations over a channel, we must guarantee that some thread will eventually read from it. However, requiring that some thread will eventually send a message over a channel and that some thread will eventually read from it is in many cases too restrictive. For example, it would not allow verifying our motivating example because `server_socket.accept()` is not guaranteed to terminate. We, therefore, define a new type of channel: an obligation channel that allows transferring obligations and has send as a credit (MaySend) and receive as an obligation (MustReceive). As a result, the method `receive` of the obligation channel, unlike the one of the credit channel, is not guaranteed to terminate. The receiver must not, therefore, have any obligations except the one to receive from the channel when it is calling the `receive` method.

To allow a sound sending of obligations over the channel, it is not enough to guarantee that a message is always received. We also need to make sure that the obligation lifetime cannot be interpreted as larger on the receiver side than on the sender side, and that the obligation cannot be sent indefinitely, otherwise it might never get satisfied. The problem with ensuring the former is that the lifetime measures in different loops are not related. We therefore only allow transferring fresh obligations. In order to guarantee the latter, we associate each fresh obligation with a lifetime measure in the same way as we do for bounded obligations. Then, each time the sender exhales the channel invariant, we check that the measure decreases.

For simplicity, we use a boolean lifetime measure, which indicates if a fresh obligation was already transfered, or not. However, one could use any well-founded set as for the regular obligations' lifetime. Of the three obligation types supported by Nagini, only an obligation to leave a place can be transfered to another thread, for which we encode the lifetime in Viper by replacing `MustInvokeUnbounded` with `MustInvokeUnboundedTransferred` and `MustInvokeUnboundedNotTransferred`. To differentiate between the two on the Python level, we allow passing `NotTransfered()` as a lifetime measure, which would indicate that the obligation was not yet transfered. The default value for an obligation measure would be *fresh and transfered*.

Listing 43 shows a modified version of the example from Listing 41 that uses an obligation channel to transfer a client socket and an obligation to handle it. The obligation channel is defined in a similar way to the credit channel – by inheriting from the generic class `ObligationChannel` and by providing a channel invariant in the body of the method `Invariant`. The invariant is written from the receiver's perspective; therefore, the obligation `token(value[0])` has a measure *fresh and transfered* and the invariant has a `MustReceive` obligation. When the listener exhales the invariant (line 32 in Listing 43), it exhales the `token` with a *fresh and not-transfered* lifetime measure (because the measure has to decrease) and inhales a `MaySend` obligation.

5.1.4 Conclusion and Future Work

This section presented a simple modification of Chalice channels that allows transferring obligations between threads by dealing with all three cases where an obligation that was sent over a channel might be never satisfied:

1. A MustReceive obligation guarantees that the message is going to be eventually received.
2. Allowing to send only fresh obligations guarantees that the bounded obligation lifetime cannot be interpreted as larger on the receiver side.
3. Associating each fresh obligation with a measure and checking that each transfer decreases it prevents the obligation from being transfered forever.

Like Chalice, we also assume that the send operation always succeeds, which implies that the execution environment has an infinite memory. Nagini does not support threads yet, therefore, we checked our approach by manually encoding the example in Listing 43 in Viper. An advantage of the presented obligation channels is that it is possible to have multiple worker threads that listen on the same channel because additional MaySend credits can be leaked. The main limitation we are aware of is the requirement for the transfered obligation to be fresh. Due to this requirement it is not possible to transfer an obligation in a called method. We leave lifting this restriction to future work.

```python
class EchoChannel(ObligationChannel[Tuple[Place, ClientSocket]]):
    def Invariant(self, value: Tuple[Place, ClientSocket]) -> bool:
        return (
            token(value[0]) and     # Default value for the
                                    # lifetime measure is fresh
                                    # and transfered.
            handle_client_io(value[0], value[1]) and
            MustReceive(self, count=1)
        )
def main(t1: Place) -> None:
    Requires(token(t1, 3) and listener_io(t1))
    channel = EchoChannel()
    worker = Worker()
    worker.start(channel)
    listener = Listener()
    listener.start(channel)
class Listener(Thread):
    def run(self, channel: EchoChannel):
        Requires(
            token(t1, 2) and listener_io(t1) and
            MaySend(channel, count=1)
        )
        Open(listener_io(t1))
        server_socket, t_loop = create_server_socket(t1)
        while True:
            Invariant(
                token(t_loop, 1) and
                listener_loop_io(t_loop, server_socket) and
                MaySend(channel, count=1)
            )
            Open(listener_loop_io(t_loop, server_socket))
            client_socket, t3 = accept(t_loop, server_socket)
            t4, t_loop = Split(t3)
            channel.send((t4, client_socket))
class Worker(Thread):
    def run(self, channel: EchoChannel):
        Requires(
            MustReceive(channel, count=1, lifetime=1)
        )
        while True:
            Invariant(MustReceive(channel, count=1, lifetime=1))
            t1, client_socket = channel.receive()
            handle_client(t1, client_socket)
```

Listing 43: An example from Listing 41 with obligation channels.

5.2 Deadlock Freedom

The methodology for proving finite blocking that was presented in [2] is a combination of two techniques: obligations ensure that a thread will eventually execute an unblocking operation and the global wait order guarantees deadlock freedom. While [6] showed how to encode the former in Viper, it did not show how to do it for the latter. This section, therefore, describes a possible encoding of the global wait order from [2] in Viper.

5.2.1 Global Wait Order

[2, 6] introduced a global fixed strict partial order on obligations and used it to establish a global order on threads. A global strict partial order on threads together with the property that a thread can have an obligation to unblock another thread only if that thread is below it in the order guarantees deadlock freedom. A thread's position in the global order is not fixed and is equal to the "largest" obligation it holds.

Listing 44 shows an example that creates a lock at a specific position in the order. We associate the order not with an obligation, but with a reference with which the obligation is associated. This works as long as the same reference cannot have two types of obligations associated with it (this is the case with all obligations supported by Nagini). [2, 8] encodes the obligation's position in the global order by assigning a wait level to it, which is a value in a dense lattice with a strict order and a bottom element. Similarly, we assign a wait level to each reference, which can be referred to on the Python level by calling a `Level` function. The function `WaitLevel` can be used to refer to the thread's wait level, which is the maximum level of all references that are associated with obligations it has. When creating a lock, a programmer can pass two optional parameters `above` and `below` that indicate that the new lock's level should be between `above` and `below`. Deadlocks are prevented by requiring that an acquired lock's level is strictly above the thread's wait level (`WaitLevel() < Level(l)`). This is not the case for the created lock, therefore, the `l.acquire()` on line 13 would fail to verify.

5.2.2 Encoding in Viper

This subsection presents the encoding in Viper. The encoding of `Level` is straightforward, we model it as an uninterpreted function from references to rationals:

```
function Level(r: Ref): Rational
```

This allows us to compare levels of two locks by using the less than operator on the Viper level:

```
Level(l1) < Level(l2)
```

```
1  def create_lock_between(l1: Lock, l2: Lock) -> Lock:
2      Requires(
3          Level(l1) < Level(l2) and      # l1 is below l2.
4          WaitLevel() < Level(l2)        # l2 level is above any
5                                          # obligation this thread
6                                          # currently has.
7      )
8      Ensures(
9          Level(l1) < Level(Result()) and
10         Level(Result()) < Level(l2)
11     )
12     l = Lock(above=Level(l1), below=Level(l2))
13     # l.acquire()    # should fail because it is not know if
14                      # WaitLevel() < Level(l).
15     return l
```

Listing 44: A procedure that creates a lock, which is in the order between `l1` and `l2`. The commented out statement `l.acquire()` on line 13 would fail to verify because it is not known if created lock is above any obligation that is currently held by a thread.

`WaitLevel() < e` in [2, 13] is encoded by using a $levelBelow(Residue, e_v)$ macro, which can be translated into Viper as:

$levelBelow(Residue, e_v) :=$
`forperm [MustReleaseBounded] r1 :: Level(r1)` $< e_v \wedge$
`forperm [MustReleaseUnbounded] r2 :: Level(r2)` $< e_v \wedge$
$Residue < e_v$

Here:

1. `Residue` corresponds to the maximum level of all obligations that are held by the current thread, but not in the current method execution or loop iteration [2, 13].
2. e_v is `e` translated into Viper.

The are two problems with using this `levelBelow` macro in method preconditions and postconditions:

1. The macro has to be exhaled after all other expressions and inhaled before them [2, 13]. The problem with inhale is that we cannot move the macro to the beginning of the contract because e_v would become non-framed if it is heap-dependent.
2. For encoding `Residue`, [2, 13] used a fresh variable for each method and loop body. We, however, cannot refer to local variables in method preconditions and postconditions.

```
def m(l: Lock) -> Lock:
    Requires(WaitLevel() < Level(l))
    Ensures(WaitLevel() < Level(Result))
    return Lock()
```

Listing 45: The procedure whose first parameter is a lock above the wait level and that returns a new lock, which is also above the wait level.

```
method m(l: Ref, methodResidue: Rational, callerResidue: Rational)
        returns (_res: Ref, currentWaitLevel: Rational)
    requires [methodResidue < Level(l), true]
    requires [true, levelBelow(callerResidue, Level(l))]
    ensures [levelBelowEqual(callerResidue, currentWaitLevel), true]
    ensures [true, levelBelow(methodResidue, Level(_res))]
{
    _res := new()
    Lock__init__(_res, methodResidue)
}
```

Listing 46: Encoding of the procedure `m` from Listing 45.

Listing 45 contains an example procedure that uses the `WaitLevel` in its precondition and postcondition. Its encoding is shown in Listing 46. We solve the local variable problem on the method definition side by introducing a ghost parameter `methodResidue`. This parameter is used only for encoding the method body and a caller is, therefore, allowed to pass any value. On the definition side, the precondition is inhaled and the postcondition is exhaled. We observe that the macro $levelBelow(Residue, e_v)$ when evaluated at the beginning of the precondition is equivalent to $Residue < e_v$ because no obligations have been inhaled yet. We, therefore, avoid the need to move the `WaitLevel() < e` to the beginning of the inhale and encode it for the method precondition inhale as:

```
[methodResidue < e_v, true]
```

When exhaling the postcondition, we have no problem of moving the `WaitLevel() < e` to the end of the contract. We, therefore, encode it by extracting the `guard`, a condition under which the `WaitLevel() < e` is going to be exhaled, and adding the following expression to the end of the contract:

```
[true, guard ==> levelBelow(methodResidue, e_v)]
```

On the method call side, the precondition is exhaled and the postcondition is inhaled. When the precondition is exhaled, the `Residue` in the $levelBelow(Residue, e_v)$ macro refers to the residue level variable of the enclosing context, which is either a loop or a method. We, therefore, add a ghost parameter `callerResidue` to each method and encode `WaitLevel() < e` in the same way as in the exhale of the method postcondition, just by using `callerResidue` instead of the `methodResidue`. The caller passes a residue level variable of the enclosing context, which is either `methodResidue` or `loopResidue` (described later), as a `callerResidue` argument. Note that `methodResidue` and `callerResidue` uses do not overlap. Therefore, it would be sufficient to have only one additional parameter. However, we describe the encoding with two variables because it is clearer.

When the postcondition is inhaled on the call side, we might already have some obligations. As a result, we cannot simplify the $levelBelow(Residue, e_v)$ macro into $Residue < e_v$. We note, however, that all assertions of the form `WaitLevel() < e` in the postcondition are evaluated in the same context [2, 14]. As a result, $levelBelow$ will yield the same set of lower bounds for all its uses in the postcondition. We can, therefore, encode `WaitLevel() < e` as $l < e_v$ where l is the largest lower bound that can be found by using a $levelBelowEqual(Residue, l)$ macro that is identical to the $levelBelow(Residue, e_v)$ macro, but uses $\leq$ instead of $<$:

```
levelBelowEqual(Residue, u) :=
  forperm [MustReleaseBounded] r1 :: Level(r1) ≤ u∧
  forperm [MustReleaseUnbounded] r2 :: Level(r2) ≤ u∧
  Residue ≤ u
```

As was already mentioned, we cannot refer to a local variable inside a method postcondition. We, therefore, add a ghost return value `currentWaitLevel` to each method and use it as l. More concretely, at the beginning of the postcondition, we add this assertion:

```
[levelBelowEqual(callerResidue, currentWaitLevel), true]
```

and encode `WaitLevel() < e` as $currentWaitLevel < e_v$.

```
l = Lock()
i = 0
while i < 5:
    Invariant(WaitLevel() < Level(l))
    l = Lock()
    i += 1
```

Listing 47: The loop that creates a new lock above its wait level in each iteration.

The encoding of `WaitLevel() < e` in loop invariants combines ideas from both method definition and method call encodings. Listing 47 shows an example loop, which has a `WaitLevel()` in its invariant. The encoding of the loop is given in Listing 48. The main difference between the `WaitLevel()`

```
l := new()
Lock__init__(l, methodResidue)
i := 0
exhale_before_loop := true
while (i < 5)
    invariant [
        levelBelowEqual(methodResidue, loopCurrentWaitLevel), true]
    invariant exhale_before_loop ==>
        [true, levelBelow(methodResidue, Level(l))]
    invariant !exhale_before_loop ==>
        [true, levelBelow(loopResidue, Level(l))]
{
    inhale loopResidue < Level(l)
    l := new()
    Lock__init__(l, loopResidue)
    i := i + 1
    exhale_before_loop := false
}
inhale loopCurrentWaitLevel < Level(l)
```

Listing 48: Encoding of the loop from Listing 47.

encoding in method contracts and loop invariants is that we can use local variables in loop invariants. We, therefore, define two variables for each loop:

1. `loopResidue` is used as the residue level variable of the loop body in the same way as `methodResidue` is used in the method body.
2. `loopCurrentWaitLevel` is used for inhaling the loop invariant after the loop in the same way as `currentWaitLevel` is used for inhaling the method postcondition.

We encode the exhale of `WaitLevel() < e` before the loop and after the loop body in the same way as for the method postcondition exhale. However, instead of `callerResidue` in the exhale before the loop we use the residue level variable of the context that encloses the loop, and in the exhale after the loop body we use `loopResidue`. To differentiate between the two exhales we reuse a boolean local variable `exhale_before_loop` (from Section 3.4.4) that is set to `true` before the loop and to `false` at the end of the loop body.

When inhaling the loop invariant at the beginning of a loop body, we need to use a different value for `Residue` than when we inhale it after the loop. Unfortunately, we cannot use the same trick for differentiating inhales, which we use for differentiating exhales. We therefore take out expressions from invariants and put them into explicit `inhale` statements before the loop body and immediately after the loop. Then, similarly to the method precondition inhale, we use loopResidue as an encoding for WaitLevel() in the inhale before the loop body. For the inhale after the loop, we use the same trick as for the method postcondition inhale by adding an assertion at the beginning of the loop invariant:

```
[levelBelowEqual(contextResidue, loopCurrentWaitLevel), true]
```

where contextResidue is the residue level variable of the enclosing context. Then, we use loopCurrentWaitLevel as an encoding of WaitLevel() in the inhale statements after the loop.

5.2.3 Conclusion

We presented the possible encoding of the technique from [2] that guarantees deadlock freedom. If this encoding were implemented in Nagini, then Nagini would fully support finite blocking. However, due to lack of time, we leave the implementation to future work.

Chapter 6

Conclusion

Penninckx, Jacobs, and Piessens [8] presented a methodology for verifying IO behaviour based on the idea to model IO operations as Petri net transitions. However, the presented approach guarantees that a program had performed the specified IO only if it terminated. In this report, we showed how the work can be combined with a technique for ensuring finite blocking in non-terminating programs by Boström and Müller [2]. The final methodology allows expressing and verifying all five properties mentioned in Section 1.1 that together allow reasoning about the IO behaviour of non-terminating programs such as servers. We achieved this by making the Petri net token an obligation to leave the place, which guarantees that some IO operation will be eventually invoked, and explicitly tracking the conditions under which each IO operation terminates, which makes it clear which parts of the Petri net are guaranteed to make progress.

The combined methodology, in addition to non-terminating programs that perform IO such as servers, can handle all programs that can be handled by the combined methodologies under joined assumptions with only one additional requirement: a programmer is forced to specify the progress guarantees. We implemented the methodology in Nagini, a Python front-end for the Viper verification infrastructure [7]. Our implementation is based on ideas from Meier work [6] on encoding obligations in Viper and from Jost and Summers work [4] on encoding VeriFast predicates into Chalice. However, our obligations' implementation is more expressive and has no known soundness issues. Our methodology implementation works with all Python constructs that are currently supported by Nagini. We evaluated it by trying to encode all related tests from the Chalice2Viper [6] and VeriFast [8] test suites. We successfully encoded all related tests from Chalice2Viper, and all tests from VeriFast that we failed to encode use some VeriFast features not related to IO that currently have no equivalents in Nagini. We also successfully verified the motivating example with all five properties from Section 1.1 and several other challenging examples. The performance evaluation revealed that the implementation has performance problems with the back-end verifier based on verification condition generation. However, the evaluation results of the back-end verifier based on symbolic execution look promising and do not indicate any potential scalability issues. We, therefore, believe that our work will be usable for verifying the SCION [1] implementation as soon as Nagini becomes expressive enough to handle its code.

In addition, the report describes two extensions to the work on which our methodology is based. The first one is obligation channels: a modification of channels presented in [2] that allows to transfer obligations between threads. The second one is an encoding of deadlock freedom verification from [2] into Viper. Both extensions were only designed, but not implemented in Nagini because of the lack of time.

6.1 Future Work

This section presents possible future work.

6.1.1 Well-Formedness Checks

Similarly to [8], we just assume that the information provided by a programmer about a basic IO operation behaviour, as well as the contract provided at the program's entry point, are correct. However, to make Nagini more usable for programmers who are not experts in verification, it is desirable to catch problems as early as possible.

One likely source of errors is the termination specification. Currently, the termination of a basic IO operation is not linked to the termination of the procedure that implements it. For example, it can happen that an IO operation is marked as terminating, but the procedure implementing it does not take a termination obligation, or vice-versa. If the procedure has only a single IO operation in its precondition, then checking that termination annotations are consistent is trivial. However, it is not clear how to ensure termination consistency when the procedure takes an arbitrary Petri net in its precondition.

Another likely source of errors is place comparison and reuse. Listing 49 shows a possible way to write an IO operation for reading an infinite sequence of integers. However, such a specification implies that all `read_int_io` operations read the same integer because in our encoding (as well as in VeriFast) IO operations have the semantics that their outputs are uniquely determined by their inputs and all instances of the `read_int_io` operation get the same `t` as input. In our encoding, such semantics is a result of encoding IO operation output by getter functions that depend only on input parameters. A possible solution would be to forbid reusing / comparing places in certain situations, but it is not clear when doing so would not impede completeness.

```python
@IOOperation
def loop_io(t: Place) -> bool:
    return IOExists1(int)(
        lambda x: read_int_io(t, x, t) and loop_io(t)
    )
```

Listing 49: An IO operation that denotes an infinite sequence of IO operations that read the same integer.

6.1.2 Contract Inference

Modular verifiers scale better than non-modular ones, but at the same time they have a significantly larger annotation overhead. One possible way to reduce the overhead for the programmer is to use contract inference. Viper already has a component based on abstract interpretation for the contract inference that can infer missing permissions. Moreover, we model obligations by using permissions. The inference, therefore, might successfully infer the missing permissions that represent obligations. However, the problem is that the inference is not aware of the obligation semantics, and the inferred permissions would not be the correct encoding of the respective obligations. For example, the inferred obligations would not have lifetime checks, which is unsound. It would, therefore, be interesting to investigate if the existing inference component could be used in Nagini, for example, by adding the missing lifetime checks after the inference.

6.1.3 Reducing Ghost Code

Nagini, unlike VeriFast, requires that the procedure that implements an IO operation takes the input place as an explicit argument and returns the output place as an explicit return value as shown in Listing 50. As a result, the caller has to use many ghost variables to link procedures together (Listing 51). However, often a procedure performs a sequence of operations (like the one in Listing 51) where each subsequent operation takes its immediate predecesor's return value as its input. In this case, it should be possible to reduce the ghost code the programmer needs to write by automatically chaining places. One posibility would be to introduce a Python level function `LastPlace()` that would refer to the last returned value of the type `Place` and allow using this new function as a default value for procedure parameters. As shown in Listing 52, this would also work nicely with Python constructs such as `with` blocks that might perform some IO implicitly.

```
def write_int(fp: File, number: int, t1: Place) -> Place:
    IOExists1(Place)(
        lambda t2: (
            Requires(
                token(t1, 1) and write_io(t1, fp, number, t2)
            ),
            Ensures(
                token(t2) and t2 == Result()
            )
        )
    )
    # Body omitted.
```

Listing 50: A procedure that writes an integer into a file.

```
def main(t1: Place) -> Place:
    IOExists4(Place, Place, Place, File)(
        lambda t2, t3, t4, fp: (
            Requires(
                token(t1, 2) and
                open_io(t1, 'test', fp, t2) and
                write_io(t2, fp, 5, t3) and
                close_io(t3, fp, t4)
            ),
            Ensures(
                token(t4) and t4 == Result()
            )
        )
    )
    fp, t2 = open(t1, 'test')
    t3 = write_int(fp, 5, t2)
    t4 = close(fp, t3)
    return t4
```

Listing 51: A procedure that writes an integer 5 to a file `"test"`.

6.1.4 Verification of Protocols

In our work we assumed that the process that we are trying to verify communicates only with the environment. However, in the context of computer networks, one might want to verify the behaviour of several processes that communicate via the environment by using some protocol. Moreover, it would be good to take into account a Dolev-Yao adversary [3] who has the capability of blocking, reordering and creating messages. Our methodology has the property that it is possible to reason about the implementation's behaviour

```python
def open(path: str,
         t1: Place = LastPlace()) -> Tuple[File, Place]:
    # The procedure's contract and body were omitted.
def write_int(fp: File, number: int,
              t1: Place = LastPlace()) -> t2: Place:
    # The procedure's contract and body were omitted.
def close(fp: File, t1: Place = LastPlace()) -> Place:
    # The procedure's contract and body were omitted.
def main(t1: Place) -> Place:
    IOExists4(Place, Place, Place, File)(
        lambda t2, t3, t4, fp: (
            Requires(
                token(t1, 2) and
                open_io(t1, 'test', fp, t2) and
                write_io(t2, fp, 5, t3) and
                close_io(t3, fp, t4)
            ),
            Ensures(
                token(t4) and t4 == Result()
            )
        )
    )
    with open('test') as (fp, _):
        write_int(fp, 5)
    return LastPlace()  # Here LastPlace() refers to the place
                        # returned by close.
```

Listing 52: An example how `LastPlace()` can be used to get hold of the place that was returned by an implicitly executed procedure `close`.

solely from the top level contract. Therefore, it should be possible to convert each process's Petri net into the role specification that can be used by a protocol verifier. In this way, it would be possible to verify the correctness of the protocol and its implementation.

Bibliography

[1] David Barrera, Raphael M Reischuk, Pawel Szalachowski, and Adrian Perrig. The scion internet architecture. 2016.

[2] P. Boström and P. Müller. Modular Verification of Finite Blocking in Non-terminating Programs. In J. T. Boyland, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 37 of *LIPIcs*, pages 639–663. Schloss Dagstuhl, 2015.

[3] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[4] D. Jost and A. J. Summers. An automatic encoding from verifast predicates into implicit dynamic frames. In E. Cohen and A. Rybalchenko, editors, *Verified Software: Theories, Tools and Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*, pages 202–221. Springer, 2013.

[5] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, 2010.

[6] Robert Meier. Verification of finite blocking in chalice. Master's thesis, ETH Zürich, 2015.

[7] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[8] Willem Penninckx, Bart Jacobs, and Frank Piessens. Sound, modular and compositional verification of the input/output behavior of programs. In *Programming Languages and Systems*, pages 158–182. Springer, 2015.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

INPUT-OUTPUT VERIFICATION IN VIPER

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
ASTRAUSKAS	VYTAUTAS

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 30.09.2016	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.