Input-Output Verification in Viper

Master Thesis Project Description

Vytautas Astrauskas
Supervised by Prof. Dr. Peter Müller, Marco Eilers
Department of Computer Science
ETH Zürich
Zürich, Switzerland

SCION is a network architecture that aims to resolve main security issues of the current Internet implementation [1]. To have reliable security guarantees, not only the protocols must be verified, but also their implementation. An important aspect of the entities participating in the protocol is their input-output (IO) behavior. The primary goal of this project is to find a methodology and implement support in the Viper [2] frontend py2viper for verifying IO properties of non-terminating programs. IO verification of programs that are assumed to terminate was already explored in [3] while the methodology for verifying properties of non-terminating programs was shown in [4] and implemented in the Viper frontend Chalice2Silver [5]. However, the way to combine these two approaches still needs to be discovered.

The first section explains the used property and assumption types. The second section gives three example programs and a list of properties we want to verify. The following section presents subgoals and the predicted workload. The final section lists the project deliverables.

I. Property and Assumption Types

This section presents property and assumption types that were introduced to simplify goal descriptions. Verifying property types under some assumption types means verifying all properties that belong to given property types under all assumptions that belong to given assumption types.

All property types we want to verify in this project and their explanations:

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

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

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

arbitrary order
: IO operations are performed in arbitrary order.

strict order
: IO operations are performed in the specified order.

periodic IO obligation
: A thread periodically gets an IO obligation or an IO credit.

IO obligation transfer
: If one thread fulfills an IO obligation by transferring it to some other thread, it is still guaranteed that the transferred obligation will be eventually fulfilled.

All assumptions used in this project and their explanations:

call termination
: A call is guaranteed to terminate.

loop iteration termination
: A loop iteration is guaranteed to terminate.

lock fairness
: Locks are fair.

scheduler fairness
: A scheduler is fair.

infinite memory
: Memory size is unbounded.

II. Motivating Examples

This section presents three examples we want to verify.

Example 1. If a server thread receives a specific input, then it performs a specific output in a finite number of steps.

A code example is given in Listing 1. For this example, we would like to verify some subset of the following properties:

1) **P {progress}:** A thread creates a server socket in a finite number of steps.
2) **P {progress, IO obligation}:** The thread calls `accept` on the server socket in a finite number of steps.

3) **P {progress}:** If the `accept` call returns, then the thread in a finite number of steps:
   a) **P {IO obligation}:** Tries to read all data from the socket.
   b) If succeeds in reading all data:
      - **P {IO obligation}:** Sends it back via the same socket. The sent data is the same data that was received.
      - **P {IO obligation}:** Prints the client address to the standard output.
   c) If fails to read all data:
      - **P {IO credit}:** Does not send any information.
      - **P {IO credit}:** Does not print any information.
   d) **P {IO obligation}:** Closes the socket.
   e) **P {IO obligation}:** Calls `server_socket.accept`.

   Also:
   a) **P {strict order}:** The data is sent after it is read.
   b) **P {strict order}:** The address is printed after the data is read.
   c) **P {strict order}:** The socket is not closed before the data is sent.
   d) **P {arbitrary order}:** Address printing and sending data can be performed in an arbitrary order.

under some subset of these assumptions:

- **A {call termination}:** `read_all`, `send` and `close` are guaranteed to terminate.
- **A {loop iteration termination}:** A loop iteration is guaranteed to terminate.

```
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(data)
      client_socket.close()
```

Listing 1: Echo server.

Example 2. A thread periodically outputs a value that depends on a configuration parameter that can be modified by some other thread.

A code example is given in Listing 2. For this example, we would like to verify the following properties:

1) **P {periodic IO obligation}:** Each second a thread gets an obligation to print a message.
2) **P {progress}:** The thread fulfills each obligation in a finite number of steps by printing a message.
3) **P {IO obligation}:** The printed message is the same as stored in a shared data structure.

under assumptions:

- **A {call termination}:** `print` and `sleep` are guaranteed to terminate.
- **A {lock fairness}:** Locks are fair.
- **A {scheduler fairness}:** The scheduler is fair.

```
class Server(Thread):
  def run():
    while True:
      mutex.lock()
      msg = mutex.data
      print(msg)
      mutex.unlock()
      sleep(1)
```

Listing 2: Periodical output.

Example 3. A server thread puts received input into a concurrent queue. A worker thread picks input from the queue and performs specific output in a finite number of steps.

A code example is given in Listing 3. For this example, we would like to verify the following properties of the `Worker.run` method:

1) **P {progress}:** A thread calls `self.queue.pop` in a finite number of steps.
2) **P {progress}:** If the `self.queue.pop` call returns, then the thread in a finite number of steps:
   a) **P {IO obligation}:** Tries to read all data from the socket.
   b) **P {IO obligation}:** If succeeds in reading all data, sends it back via the same socket. The sent data is the same data that was received. Otherwise, does nothing.
   c) **P {IO obligation}:** Closes the socket.
   d) **P {progress}:** Calls `self.queue.pop` in a finite number of steps.

   Also:
   a) **P {strict order}:** The data is sent after it is read.
   b) **P {strict order}:** The socket is not closed before the data is sent.

under assumptions:

- **A {call termination}:** `read_all`, `send` and `close` are guaranteed to terminate.

Also, the following properties of the `Listener.run` method:

1) **P {progress}:** A thread creates a server socket in a finite number of steps.
2) **P {progress, IO obligation}:** The thread calls `accept` on the server socket in a finite number of steps.

3) **P {progress}:** If the `accept` call returns, then the listener thread in a finite number of steps:
   a) **P {IO obligation transfer}:** Puts the client socket in the concurrent queue and in this way transfers all IO obligations and IO credits related to that socket to the thread that will take the socket from the queue.
   b) **P {IO obligation}:** Calls `server_socket.accept`.

under assumptions:

- **A {infinite memory}:** The `ConcurrentQueue` size is unbounded.

and prove obligations for the context that creates the listener thread:

- **P {IO obligation transfer}:** The worker thread is running.
- **P {IO obligation transfer}:** For each socket put into `self.queue`, the worker thread in a finite number of steps:
  1) **P {IO obligation transfer}:** Will take the inserted client socket from the `self.queue`.
  2) **P {IO obligation}:** Will try to read all data from the socket.
  3) **P {IO obligation}:** If succeeds in reading all data, then will send it back via the same socket. The sent data is the same data that was received. Otherwise, it will do nothing.
  4) **P {IO obligation}:** Will close the socket.

  Also:
  1) **P {strict order}:** The data is sent after it is read.
  2) **P {strict order}:** The socket is not closed before the data is sent.

III. Goals

This section presents the subgoals that support the primary goal of this project.

A. Core Goals

The core goals of this project that should be achieved in the first 4 months of the project:

Goal 1. Create a methodology that allows verifying the following property types of the motivating Example 1:

1) progress
2) IO credit
3) IO obligation
4) arbitrary order
5) strict order

under assumptions:

1) call termination
2) loop iteration termination

This includes:

```
def main():
  queue = ConcurrentQueue()
  worker = Worker(queue)
  worker.wait_until_started()
  listener = Listener(queue)
class Listener(Thread):
  def __init__(self, queue):
    self.queue = queue
  def run(self):
    server_socket = create_server_socket()
    while True:
      client_socket = server_socket.accept()
      self.queue.put(client_socket)
class Worker(Thread):
  def __init__(self, queue):
    self.queue = queue
  def run(self):
    while True:
      client_socket = self.queue.pop()
      data = client_socket.read_all(timeout=1)
      if data:
        client_socket.send(data)
      client_socket.close()
```

Listing 3: Echo server with worker thread.

1) Creating rules that allow verifying the Example 1 by using a pen and paper.
2) Demonstrating how Example 1 can be verified by using the created rules.
3) Demonstrating how variations of the Example 1 that require showing properties that belong to types listed in this goal under assumptions that belong to types listed in this goal can be verified by using created rules.
4) Providing an argument why properties verified with the methodology hold.

Planned workload: 1 month.

Goal 2. Define how rules from Goal 1 can be encoded in the Viper language.

This includes extending the Viper language with new constructs if rules cannot be reasonably expressed with existing ones, or if new constructs are needed to achieve a reasonable verification time.

Planned workload: 1 month.

Goal 3. Define extensions for the py2viper contract specification language that allow to automatically apply rules from Goal 1.

Planned workload: 0.25 month.

Goal 4. Implement support for the methodology (Goal 1) in Viper tools.

This includes:

1) Implementing support for py2viper contract specification language extensions (Goal 3).
2) Implementing encoding of py2viper contract specification language extensions to Viper language (Goals 3 and 2).
3) If needed, implementing new constructs in the Viper language (Goal 2).
4) If needed, implementing support for new Viper language constructs in at least one of the backends (Goal 2).

Planned workload: 1.5 months.

B. Extension Goals

Below are the extension goals of this project. Some of them should be completed in the last 2 months of the project.

Goal 5. Create a methodology that allows verifying the following property types of Example 1:

1) progress

under assumptions:

1) call termination

and is compatible with the methodology from Goal 1. Also, extend deliverables from Goals 2, 3 and 4 to support it.

Goal 6. Extend the methodology (Goal 1) and Viper tools (Goals 2, 3 and 4) to support the verification of these property types of motivating Example 2:

1) periodic IO obligation
2) progress
3) IO obligation

under assumptions:

1) call termination
2) lock fairness
3) scheduler fairness

Goal 7. Extend the methodology (Goal 1) and Viper tools (Goals 2, 3 and 4) to support the verification of these property types of motivating Example 3:

1) IO credit
2) IO obligation
3) arbitrary order
4) strict order
5) IO obligation transfer

under assumptions:

1) call termination
2) infinite memory

C. Additional Workload

An additional workload that does not belong to any particular goal:

1) Report finalization: 0.25 month.

IV. Deliverables

The deliverables of this project are:

1) A project report that includes:
   a) The methodology description from Goal 1 with rules, examples and the soundness argument.
   b) A description of how rules from Goal 1 are encoded in Viper language (Goal 2).
   c) A description of py2viper contract specification language extensions (Goal 3).
   d) If done, the methodology description from Goal 5 with rules, examples and the soundness argument.
   e) If done, the methodology extension description from Goal 6 with rules, examples and the soundness argument.
   f) If done, the methodology extension description from Goal 7 with rules, examples and the soundness argument.
2) The implementation of py2viper extensions from Goal 4 and (if done) Goals 5, 6 and 7.
3) If needed, the implementation of new Viper language constructs and support for them in at least one of the backends for Goal 4 and (if done) Goals 5, 6 and 7.

References

[1] D. Barrera, R. M. Reischuk, P. Szalachowski, and A. Perrig, "The scion internet architecture."

[2] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, ser. LNCS, B. Jobstmann and K. R. M. Leino, Eds., vol. 9583. Springer-Verlag, 2016, pp. 41–62.

[3] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *Programming Languages and Systems*. Springer, 2015, pp. 158–182.

[4] P. Boström and P. Müller, "Modular Verification of Finite Blocking in Non-terminating Programs," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LIPIcs, J. T. Boyland, Ed., vol. 37. Schloss Dagstuhl, 2015, pp. 639–663.

[5] R. Meier, "Verification of finite blocking in chalice," Master's thesis, ETH Zürich, 2015.