Testing and Verification of Concurrent and Distributed Software

Bruno Dias Gião, João Luís Cruz Pereira, and Maria Filipa Rodrigues

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a96544,a95375,a97536}@alunos.uminho.pt

Abstract. With this article, we intend to present a library and a tool for visualizing distributed programs written in the Harmony language, an executable model checker with Python-like syntax that natively allows for concurrent and distributed programs [7]. By analyzing common implementations of distributed algorithms in Harmony, we developed a set of APIs that translate topology- independent networks defined by either order-independent mailboxes or FIFOs. These networks operate under an unicast model and account for potential faults, such as message duplication or loss. Our visualization tool processes failing programs that use this library and generate Lamport diagrams [6] to depict each sent, received and pending messages, including those that could be lost or are yet to be received.[1]

Keywords: Distributed Programs · Concurrent Programs · Visualization · Formal Verification

1 Introduction

The Harmony programming language is a Python-like language with a built-in model-checker for concurrent and distributed programs.

Harmony programs are executed in the command line or a GUI, be it Visual Studio Code or Harmony-GUI. The command line Harmony will produce an HTML page with a trace of a failing execution showing variable, thread and bytecode information, as well as the current macro-instruction. If the program doesn't fail, Harmony presents a DFA with each possible print as edges and program states in which these prints occur as states.

Harmony generates some auxiliary JSON files, of note are the ".hco" and ".hvm" files, The ".hco" file presents in detail the user's code, the generated assembly and the full trace of the failing execution to generate the interactive HTML file. The ".hvm" file only contains the program bytecode and is used by Harmony's model checker, *Charm*. The Model Checker stops either when the entire state space has been visited or when any of the following errors occur:

[1] https://github.com/greybrunix/Testing-and-Verification-of-Concurrent-and-Distributed-Software

– Invariant Violation: If there exists a state where any given invariant is violated, be it user or Harmony-command line as safety and fairness[3][5];
– Assertion Violation: If, when executing an assertion statement, the given condition is violated;
– Non-Terminating State: Representing infinite cycles, deadlocks or active locks.

1.1 Failing Concurrent Programs

Let's consider the following failing implementations of Dekker's Solution[3][2] and Peterson's Algorithm[1][4]:

```
sequential wants, turn
wants = (False, False)
turn = 0

def dekker p_q:
  while choose {False, True}:
    wants p_q = True

    while wants (1 - p_q):
      if turn == (1 - p_q):
        wants p_q = False
        #await turn == p_q
        wants p_q = True

    print("Entered", p_q);
    print("Left", p_q);
    turn = 1 - p_q
    wants p_q = False

spawn dekker 0
spawn dekker 1
```

```
sequential flags, turn
flags = ( False, False )
turn  = choose { 0, 1 }

def peterson self:
  while choose {False, True}:
    flags self = True
    turn = 1 - self
    await (flags[1-self])
           or (turn==self)
    (*await (not flags[1-self])
           or (turn==self)
    *)

    print("Entered", self);
    print("Left", self);
    flags self = False

spawn peterson 0
spawn peterson 1
```

Despite being extremely simple examples, it may be tricky to perceive what is the problem with these solutions. Harmony correctly finds an active busy wait and a blocked thread, respectively.

1.2 Implementing Distributed Algorithms in Harmony

Mailboxes in Harmony are usually represented using a global channel, list, multiset, from hereon referred to as bag, or set. In a unicast model, a client sending a message is, thus, represented as an insertion to the global structure. Receiving a message is translated to code as indexing an element of the mailbox. In a broadcast model, however, there are no message duplications, i.e., it's topology-independent.

Let us thus consider two algorithms that represent the two philosophies adapted in our library: bags and FIFOs.

Bags and Leader Election Originally, the leader election algorithm, as presented in the Harmony Github repository, was written using a set for a mailbox, however, let's just consider a bag implementation and just disregard the multiplicity of the message as it shall always be 1, regardless, message order is irrelevant and duplications are not considered.

Receiving a message is done in the client code itself by constantly atomically polling the mailbox for new messages for the client. When the client receives a message, they check if the sender claimed to be the leader or not and proceed to send from the sender to the successor the result.

This is common for unicast bag algorithm implementations.

FIFOs and Replicated State Machines Harmony's implementation of the State Machine Replication algorithm simply appends a message to the end of a list. This FIFO algorithm only has 1 node capable of receiving messages, as such, the destination is implicit and, to receive messages, we just index the first message in the queue, with the help of a counter for delivered messages.

As such we print the client that sent the message and a 2-uple corresponding to the receiver and the payload of the message.

Our RSM will be the output of all possible combinations of prints in this program.

2 Library

The user is given four public functions: a "constructor" that changes any of three boolean flags for introducing faults or changing the behaviour of the mailbox itself, a "getter" for accessing the current state of the network, and send and receive functions.

To ease code legibility, sends have explicit arguments: a source, a destination, and a payload, whereas receives have only an identifier for who is receiving the message.

Messages are structured as a dictionary composed of a source, a destination, a payload, and an identifier. The identifier allows the identification and synchronization between receive and send when parsing the output.

Only receives and the "getter" for the network return values, the received message, and the current network, respectively.

Receiving a message in our library is a matter of filtering all messages and choosing a message per the data structure used.

Duplication adds a copy of the received message to the network, facilitating the testing of duplicate handling. Dropping a message removes it from the network and the function's output, simulating packet loss. Both of these events occur randomly and are conditioned by the parameters of the "constructor" function.

2.1 Translations

We can translate the Leader Election algorithm into the following program:

```
from bag_net import *
construct(0,0,0)
const NIDS = 3   # number of identifiers
leader = 0
def processor(self, succ):
    send(self,succ,False)
    var working = True
    while working:
        atomically:
        var req = receive(self)
            if (len(req) > 0):
                var id, found = req[0].src,req[0].payload
                if id == self:
                    assert self == leader
                    send(id,succ, True)
                elif id > self:
                    assert self != leader
                    send(id,succ, found)
                if found:
                    working = False
ids, nprocs, procs = { 1 .. NIDS }, choose({ 1 .. NIDS }), []
for i in { 0 .. nprocs - 1 }:
    let next = choose(ids):
        ids -= { next }
        procs += [ next, ]
        if next > leader:
            leader = next
for i in { 0 .. nprocs - 1 }:
    spawn processor(procs[i], procs[(i + 1) % nprocs])
```

Let's also implement the State Machine Replication algorithm in our library:

```
from list_net import *
construct(0,0,1)
const NREPLICAS = 3
const NOPS = 2
whois = 0
def crash():
    stop()
```

```
def replica(self, immortal):
    if not immortal:
        trap crash()
    var delivered = 0
whois = self
    while True:
        atomically when len(get_net()) > delivered:
            let msg = receive(self):
                if msg != {:}:
                    print(self, msg.payload)
                    delivered += 1
def client(self):
    print(self)
    send(self, whois, self)
let immortal = choose {1..NREPLICAS}:
    for i in {1..NREPLICAS}:
        spawn eternal replica(i, i == immortal)
for i in {1..NOPS}:
    spawn client(i)
```

2.2 Results

Considering only code, we find the following results representative of the increase in code complexity from the introduction of our library:

Program	Bytecode Instructions
rsmLib	981
rsm	124
leader	227
leaderLib	729

However, it is also important to consider the resulting Non-Deterministic Automaton generated by the Model Checker:

Program	Edges	Computations	States
rsmLib	2146	56314	13472
rsm	145	8602	1952
leader	39385	92758	33005
leaderLib	6355	5905	3638

Of note are the list implementation results that are easily attributed to the lack of a `cons` operator in harmony, thus forcing either a $O(N \log N)$ or a $O(N)$ insertion at the head of the list.

3 Visualization

To visualize the transfer of messages between processes, a program was created for such an objective, resulting in a temporal diagram [6]. The tools of choice were Python and “D3.js”. When running Harmony code through this program, the user can obtain an HTML file with all the transferred messages. The program can capture all the messages by parsing the “.hco” file and its content, regex was used to find and store the messages in a “JSON” file that is then processed by a JavaScript script, using “D3.js” [8], in the resulting HTML file.
The user will be able to see the following types of messages:

- Message sent and received
- Message sent and not received
- Duplicated Message
- Dropped Message

The order in which the messages are shown, follows the appearance order in the “.hco” file, meaning that it follows the natural flow of the program. Each message sent has its unique ID, duplicated messages use the same ID, this way a receive is matched to one message only, in the case of duplicated messages the first one is the one that is received.

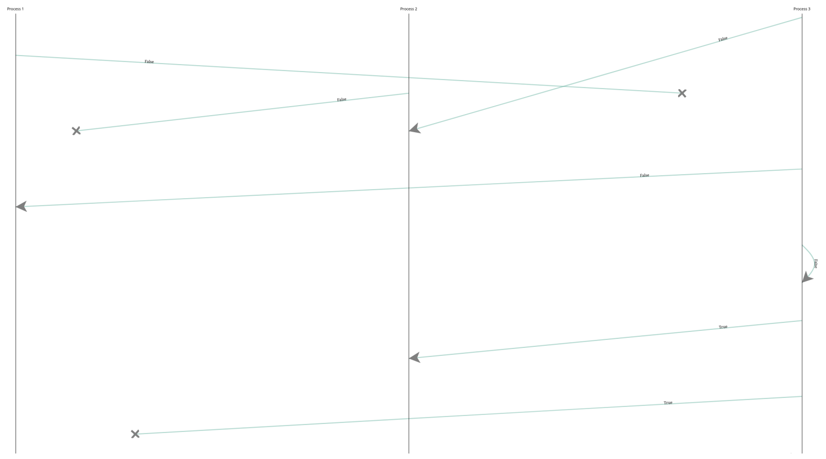

Fig. 1. Leader Election Time Diagram

4 Conclusion and Future Work

We have successfully implemented a visualizer and a library in Harmony for unicast networks in a sufficiently efficient manner, resulting in a simpler and easier way of visualizing distributed program flow in the case of failures. Of note, however, is how the implementation of our FIFO-based network could be optimized using, for example, Functional Programming-like lists, i.e. a pair of message and network. It would also be of interest to have our visualizer colour code messages by their payload type.

References

1. Ben-Ari, M.: Principles of concurrent programming. Prentice-Hall International, Englewood Cliffs, N.J (1982)
2. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569 (Sep 1965). https://doi.org/10.1145/365559.365617, https://dl.acm.org/doi/10.1145/365559.365617
3. Djikstra, E.: About the sequentiality of process descriptions (1962), http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF, http://www.cs.utexas.edu/users/EWD/translations/EWD35-English.html
4. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann, Amsterdam, revised first edition edn. (2012)
5. Knuth, D.E.: Additional comments on a problem in concurrent programming control. Commun. ACM **9**(5), 321–322 (may 1966). https://doi.org/10.1145/355592.365595, https://doi.org/10.1145/355592.365595
6. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (Jul 1978). https://doi.org/10.1145/359545.359563, https://dl.acm.org/doi/10.1145/359545.359563
7. van Renesse, R.: Concurrent Programming in Harmony (2020), https://harmonylang.dev/book.pdf
8. Yan, H.: Most basic parallel coordinates chart in d3.js, https://d3-graph-gallery.com/graph/parallel_basic.html