RIGOUROUS ANALYSIS OF COMBINED SOFTWARE

PROCESSES VIA RUNTIME VERIFICATION

by

Mark McDermott

A thesis submitted to the Graduate Council of

Texas State University in partial fulfillment

of the requirements for the degree of

Master of Science

with a Major in Software Engineering

August 2020

Committee Members:

Rodion Podorozhny, Chair

**COPYRIGHT**

by

Mark McDermott

2020

**FAIR USE AND AUTHOR’S PERMISSION STATEMENT**

**Fair Use**

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgment. Use of this material for financial gain without the author’s express written permission is not allowed.

**Duplication Permission**

As the copyright holder of this work I, Mark McDermott, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

**TABLE OF CONTENTS**

**Page**

ABSTRACT …………………………………………………………………………………………...

CHAPTER

1. INTRODUCTION ………………………………………………………………………….. 1

2. APPROACH …………. …………………………………………………………………….

3. PROBLEM DESCRIPTION ……………………………………………………………….

4. SOLUTION DESIGN ……...……………………………………………………………….

5. MY CONTRIBUTION ……………………………………………………………………..

6. EVALUATION & EXPERIMENTAL RESULTS ………………………………………….

7. FUTURE WORK .…………………………………………………………………………..

LITERATURE CITED ………………………………………………………………………………...

**ABSTRACT**

Applying model checking and rigorous analysis to parallel processes can help identify errors in formalized parallel processes where properties like safety and liveness do not hold. Modern parallel processes, both in computer science and in general industry, are very complex and pinpointing errors can be difficult, time consuming, and expensive. This paper explores the application of CTL model checking to the Little-JIL process language and some of the issues, namely scalability issues, that arise from the exponential blowup in the number of potential paths that may be taken in large parallel processes. The conclusion drawn in this paper is that modern personal computers are generally powerful enough to handle the interleavings’ exponential blowup quite well, with some caveats related to looping and extremely massive processes.

The example processes studied here are a banking program written in Java and a hospital’s process for treating a COVID-19 patient. We check for a safety property in the banking example, specifically that transfers and deposits only occur when the program is not reading from disk. In the COVID-19 example, we check for the safety property that someone with COVID-19 who is a high risk to others is in the hospital and not at home where they can infect others. We also check for the liveness property that if a ventilator is requested, it will always eventually be available for use. We hope that the system-critical and life or death nature of the errors found in paths specified by our analyzer suggest how valuable a tool model checking can be when designing parallel processes.

1. INTRODUCTION

Failures in real world parallel processes can cause expensive and sometimes life-threatening system failures. In large-scale parallel computing systems, failures have increased both response time and slowdown time [1]. In hospitals, failures in the parallelized processes of nurses’ assessment, planning, implementation, and evaluation of patients at times has kept supplies, medication, equipment, and information from being properly provided to patients [2]. Finding and eliminating errors in parallel systems prior to their occurrence is thus important. One approach to identifying errors in parallel processes is to use model checking on a process language.

We use CTL model checking and the Little-JIL process language to identify errors in parallel processes and to identify specific paths the error route takes. Since the different execution paths a process may take can be accurately modeled using a tree graph, CTL is an appropriate logic to use to identify whether certain properties hold on the graph since it checks all possible paths through a tree. Little-JIL is a flexible and adaptive process language that models how agents performing processes [3]. It has been chosen because it is easy to use, yet retains semantic depth [4].

When parallelizing processes, it is often difficult to control the order of execution. Process A may execute before Process B or vice versa. The number of possible execution paths can be calculated as a permutation of the number of executed in parallel. So the number of interleavings is *x!* where x is the number of processes. In modelling our process in a processing language like Little-JIL, which breaks down each state into further sub-states representing the state’s status, our graph becomes even larger.

Little-JIL represents process control structures such as *leaf*, *sequential*, *parallel*, *try* and *choice*. Each of these has its own template of status states. A leaf state, for example, consists of leaf posted, leaf started, leaf completed and leaf terminated [5]. A simple process of two leaves run sequentially, when expanded according to Little-JIL rules, becomes a graph of 20 nodes (See figure 1). Similarly, two leaves executed in parallel, expand to a graph of 12 nodes after translation and then to a graph of 20 nodes after the interleavings are represented (See figure 2). When more complicated real world processes are modelled the number of nodes quickly reaches to the thousands or more. In analyzing these larger graphs, we speak of the “topology” of the graph, abstracting out some of the smaller details.

Diagram

Description automatically generated

Figure x. Little-JIL expansion of two leaves executed sequentially

Diagram

Description automatically generated

Figure x. Little-JIL expansion of two leaves executed in parallel

2. APPROACH

The approach I take in this paper is to apply rigorous model checking algorithms to Kripke structures created through an abstract process language represented in code by an analyzer I’ve written. Model checking is an appropriate way to bring predictability into the unpredictable area of parallelized processes because it is one way to formally verify software. Using formal methods of mathematics, we can say with 100% certainty (as long as our Kripke structure properly represents our processes) whether a certain code block can ever execute before another code block. Our main challenge is thus to ensure that the Kripke structure we design properly represents our processes.

Our approach begins with processes. For this paper, a process is just a collection of steps which can have certain control structures like sequential, parallel and try. We use two processes in this paper, a computer program and a medical process. We first model the abstract process with a graphical process language. Then we output the graphical process into XML. Now that the process is represented by code, we run it through the translator. The translator takes in XML steps and outs a directed graph type of data structure that represents the steps. The translator also translates the control structure nodes of the graph into their substeps, according to the rules laid out by our graphical process language. At this point, we have a Kripke structure ready for model checking. We use a temporal logic on our Kripke structure to formally verify whether our combined parallel processes have the safety property of certain nodes never being executed before other nodes.

3. PROBLEM DESCRIPTION

When two processes run in parallel on a computer it can be difficult to predict when certain subprocesses will execute. Many libraries exist for creating “critical sections” of code that request that the section be executed in serial. This is helpful, but if we could know in a formal way that purely parallel process that a certain process that needs a certain property will never execute without that property would be very helpful.

Race conditions can be created in parallel programs which create many problems. For example, a certain variable is written to before it is defined, throwing an undefined error and halting the program execution. Even in real world parallel processes, like in a hospital, race conditions can exist. If a heart transplant procedure, in an extreme example, is started before the new heart has been received, it could cause the death of the patient while they are waiting on the operating table. In the real world many processes, even when explicitly defined, have a surprising amount of ambiguity which can lead to misinterpretations and, in some cases, race conditions.

The problem I address in this paper is how to prove if a certain step in one process which needs a certain property will never happen before another certain step with that property in another process is also executed. The statement we would like to prove is, “Whenever the ventilator is needed, it is available for use.” One process is the hospitalization treatment of a COVID-19 patient. The other process is the ordering of a ventilator from a supplier. If both process are started in parallel, we’d love to know for sure that when the ventilator is needed by the patient, it will be available for use. In CTL, this statement is AG(needed→available). In plain English this means, “For all possible states in the combined parallel process, if the ventilator is needed, then the ventilator is available right now.” So any state that has the process needed, will also have the property available, if this model holds. This is a global property (*not a liveness property*).

The second problem I address in this paper is determining if the code of a bank system which both deposits checks and processes transfers will always deposit the checks first. This is code that would be run in the morning, after mobile check deposits and online transfers were requested the previous night after business hours. First thing in the morning, the bank wants to run this program to deposit the checks and then process the transfers. Since the check deposits in this case will all add to the bank accounts’ balances, they want to make sure all the checks are processed first to ensure the transfers will have the maximum available funds to minimize the amount of overdrafts. Since there is a certain amount of reading from disk in both cases, we would ideally like to run both programs in parallel. We would like to make sure though, that the actual processing of transfers never happens before all the checks are deposited. We can use the CTL model A[¬transfer U deposit],

A[¬transfer U deposit] “or exists a path where there is no transfer and no deposit”

“On all paths if a state is marked with a transfer then on all paths leading to it, there must have been a state marked with a deposit”

May not be possible to look back in time like that

Maybe demorgans law may allow this?

Safety: “On no path should we have a transfer without a deposit before it”

On all paths, elevator does not move unless button was hit beforehand”

which in plain English means, “as long as deposits have not been processed, transfers will not be processed”. This is a safety property, ensuring that the potentially dangerous error state where transfers are processed before deposits are processed.

4. SOLUTION DESIGN

I have decided to use Little-JIL to model processes for this paper. Little-JIL is a graphical language that models processes [3]. Little-JIL was developed by Leon Osterweil at University of Massachusets in 1998 and was further developed over the next sixteen years. The tools Little-JIL provide allow for the modelling of processes followed by autonomous agents and the different possible paths a process is allowed to take. It introduces types of states, like sequential, parallel and leaf, as well as the statuses of those states, such as started, completed or terminated. The parallel node will be especially useful in our modelling of parallel processes.

The Little-JIL status states help bring specificity to our processes and help eliminate ambiguity. For example, say a state just “perform heart transplant”. If it’s running in parallel with another state and the two alternate switching back and forth for which is running at any moment, we won’t know if the transplant state is just ready, or if it has just begun. Bringing status states like posted, started terminated and completed into the process model help us get a more granular status of where each state is and when. This specific can help avoid errors caused by ambiguity.

Little-JIL has a helpful implementation which uses the Eclipse IDE [4]. In this I’ve modelled our processes visually in a UML-type diagram. This Little-JIL Eclipse extension has an “export to XML” feature. I have used this to get my initial XML files. These XML files are the representation of my processes that I feed into the translators. I have pared down and cleaned up the XML code which Eclipse exports to just the process steps. Originally the Eclipse XML includes the XML prolog, some metadata and some connector tags which are a bit extraneous to our task.

I’ve written the translator in Java. The initial translator takes in the XML file of the Little-JIL processes and then translates them using algorithms I’ve written that correspond to the specifications of the Little-JIL language. The programmatic output of the algorithms are Kripke structure graphs of nodes of the different paths the process can take. The visual output of the translation are graphs displayed using Java Swing [5] and the Jung [6] graphing library. To run model checking CTL algorithms, I’ve used the Antlr [7] compiler library. Antlr is a modern compiler writing library, which can handle the potentially infinite nesting aspect of CTL formulas well. The analyzer I’ve written can process looping structures and also self-referential structures.

Two real life processes are presented in this paper to demonstrate the benefits of using translation algorithms on a process language. The first process consists of two COVID-19 related processes running in parallel: the treating of a patient with COVID-19 and the ordering of a ventilator for the patient. The goal in this example is to see if it can be determined from the steps of the processes if the ordered ventilator will always be available by the time the patient treatment has reached the mechanical ventilation step where the ventilator will be needed. The second real life process in this paper is some banking related code written in Java.

The second real life process example in this paper is a banking program written in Java that runs two processes in parallel: the processing of check deposits and the processing of bank transfers. The processes are modelled using Little-JIL states. Even though we want to ideally run these two processes in parallel, it would be helpful to know if all check deposits will necessarily be finished by the time the step of the actually moving the money of the transfers is executed.

The COVID-19 hospitalization process is fairly straightforward. It is a sequential process where the first step is a parallel step (diagnosis) and the second step is another sequential step (treatment). For diagnosis, the nurse taking tests, the doctor examination and the CT scan can all happen in parallel. The nurse has two sequential tests to take, the COVID-19 test and a blood test. The doctor examination has two sequential steps, listening to the lungs and checking the oxygen level with a finger clip. For the treatment step, the steps of antibiotics, isolation room, high flow nasal oxygen and mechanical ventilation must happen in that order.

COVID-19 Hospitalization [8] [9] [10] [11] [12] [13] [14] [15]

A picture containing clock

Description automatically generated

Figure 1. COVID-19 Hospitalization Little-JIL Diagram

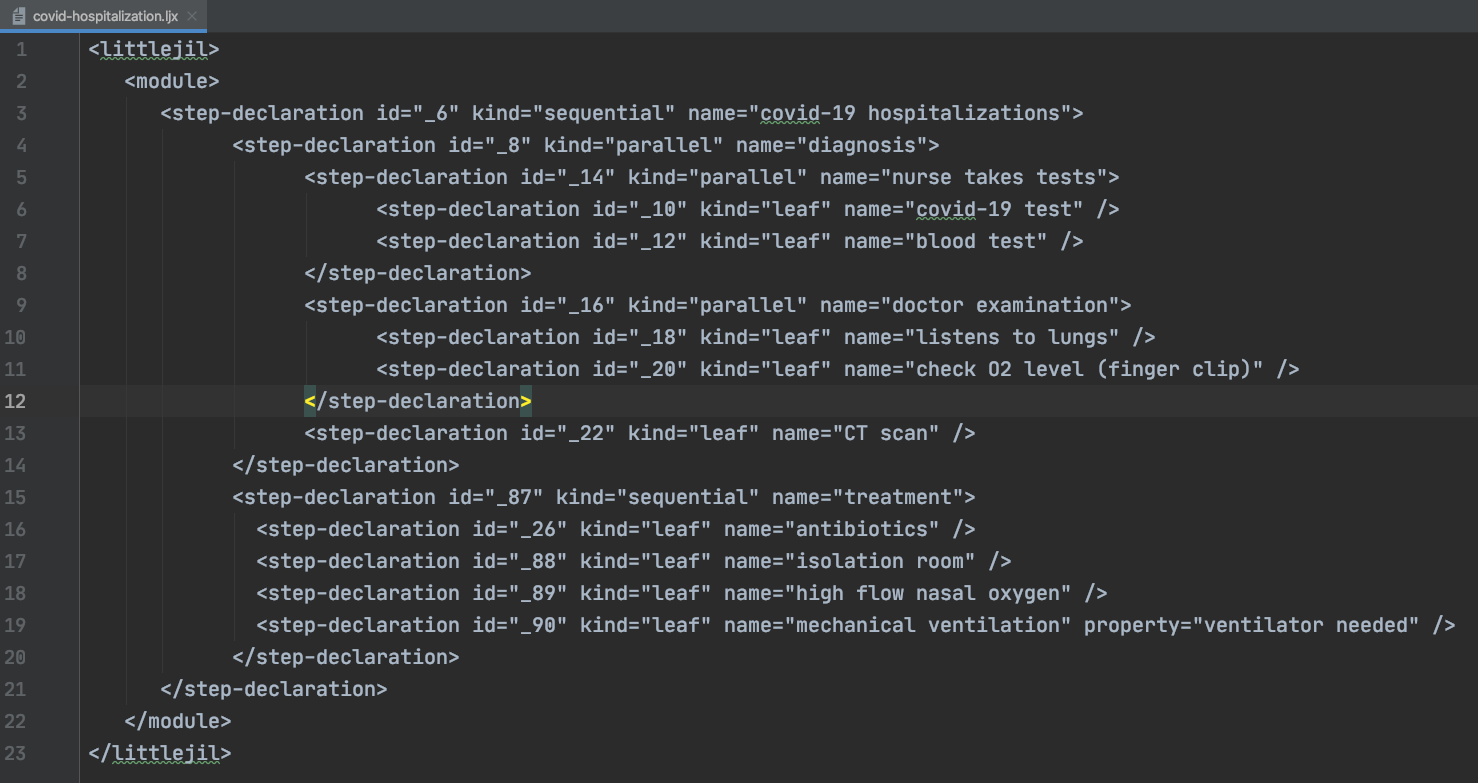


Figure 2. covid-hospitalization.ljx File

A close up of a necklace

Description automatically generated

Figure 3. covid-hospitalization XML graph

The COVID-19 ventilator order process is composed of seven steps which must happen in that order. First the department head asks management for a new ventilator, the management orders the ventilator from the supplier, the supplier finds it in their warehouse, ships it to the hospital, then the hospital receives the ventilator on their loading docks, stores in in their storage area and finally an orderly brings the ventilator to the hospital room where the patient is staying.

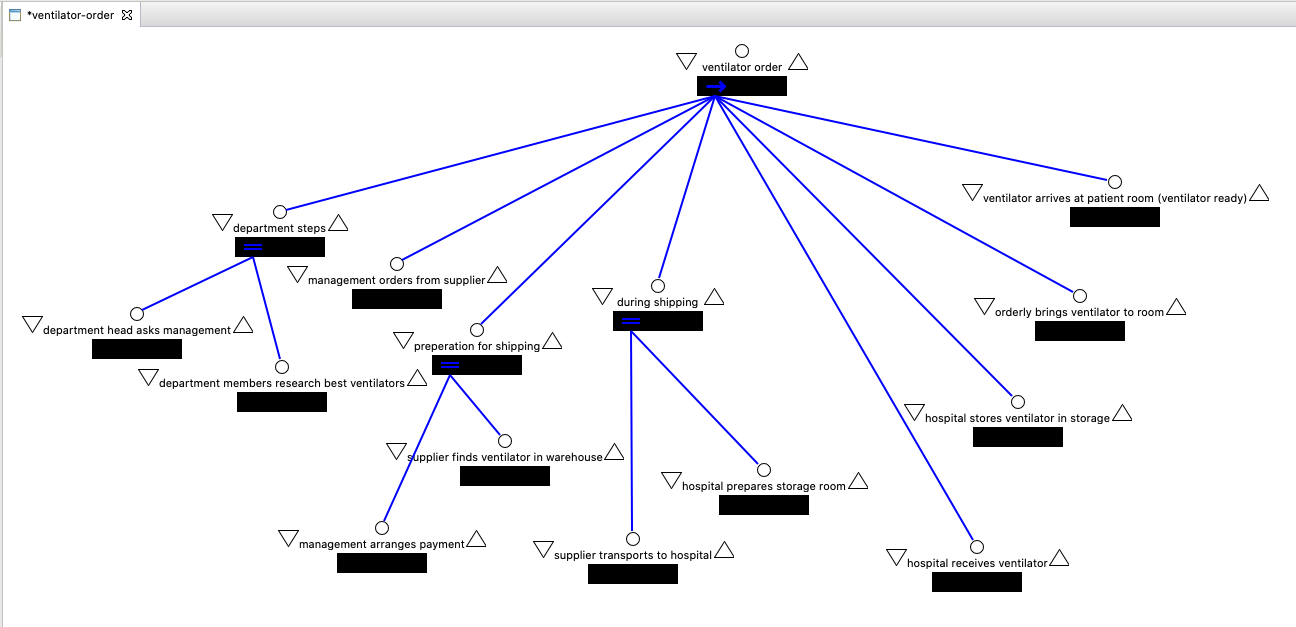


Figure 4. Ventilator Order Little-JIL Process

A screenshot of a cell phone

Description automatically generated

Figure 3. ventilator-order.ljx

A picture containing crane

Description automatically generated

Figure 4. ventilator-order XML graph

In the problem of proving whether a COVID-19 patient’s mechanical ventilator will always be available when it is needed, I use CTL and the AG(p→q) model. p in this case is “ventilator is needed” and q is “ventilator is ready”. Using the AG(p→q) model, we can know that p implies q or if p exists then q also exists. Applied to our real life situation, this would means that the ventilator would always be available when it is needed.

The banking example in this paper is a bit more involved than the COVID-19 example. The code for the ProcessChecks.java class file and for the ProcessTransfers.java class file is about 60 lines of code for each.

A screenshot of a cell phone

Description automatically generated

Figure 5. ProcessChecks.java

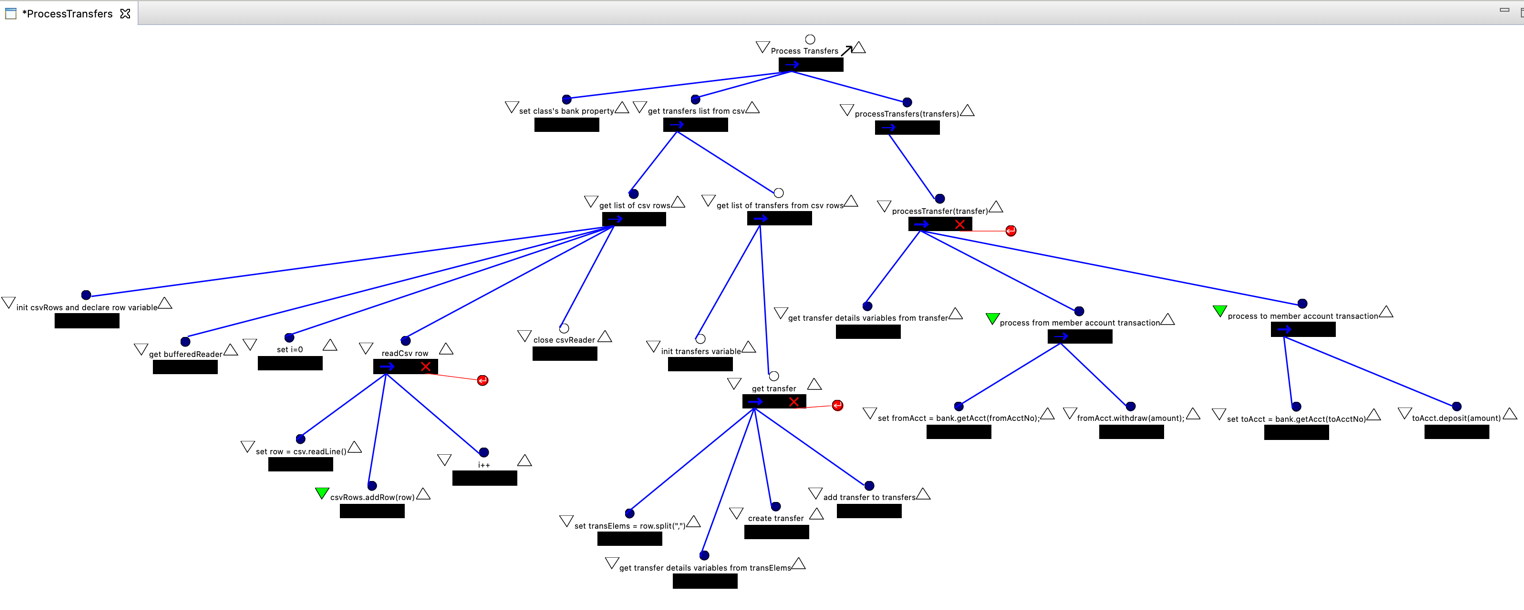


Figure 6. ProcessChecks.java

Both class files were then represented in Little-JIL.

A picture containing text

Description automatically generated  
Figure 7. Process Checks Little-JIL diagram

  
Figure 8. Process Transfers Little-JIL diagram

The graphs representing the xml files exported from the Little-JIL diagrams look as follows:  
  
A close up of a device

Description automatically generated  
Figure 9. Process Checks XML graph

A close up of a device

Description automatically generated

Figure 10. Process Transfers XML graph

4. MY CONTRIBUTION

Below are the translation algorithms I have written, based on Jamieson Cobleigh’s diagrams of Little-JIL states in *Verifying Properties of Process Definitions* [16]. These algorithms take in a sequence of Little-JIL control structure steps and output a sequence of Little-JIL steps with all their possible statuses, such as posted, started terminated and completed. For parallel states, I have added algorithms to find all the different possible permutations, as steps may happen in a somewhat arbitrary order in a parallel process.

Translation Algorithms

SET root to the first element in originalStates

SET states to a new array of nodes

SET stateNum to 0

IF root has children THEN

FOR each child

add child to queueToTranslate

END FOR

END IF

IF queueToTranslate is not empty THEN

WHILE queueToTranslate is not empty

DEQUEUE node

IF node is in originalStates THEN

SWITCH (node.kind)  
 CASE leaf

states = leafTemplate(node)

END CASE  
 CASE sequential

states = sequentialTemplate(node)

END CASE

CASE parallel

states = parallelTemplate(node, originalStates)

END CASE

END SWITCH  
 END IF

END WHILE

END IF

DEFINE leafTemplate(nodeToReplace)

# create template nodes

SET leafPosted to new node(stateNum++)

SET leafStarted to new node(stateNum++)

SET leafCompleted to new node(stateNum++)

SET leafTerminated to new node(stateNum++)

# hook up template nodes to each other

setParentAndChild(leafPosted, leafStarted)

setParentAndChild(leafStarted, leafTerminated)

setParentAndChild(leafStarted, leafCompleted)

# set template nodes’ children

SET children to nodeToReplace.getChildren()

FOR child IN children

setParentAndChild(leafCompleted, child)

setParentAndChild(leafTerminated, child)

unsetParentAndChild(vertexToReplace, child)

END FOR

# set first template nodes’ parents

SET parents to nodeToReplace.getParents()

FOR parent IN parents

setParentAndChild(parent, leafPosted)

unsetParentAndChild(parent, vertexToReplace)

END FOR

SET leafStates to new array(leafPosted, leafStarted, leafCompleted, leafTerminated)

RETURN leafStates

END DEFINE leafTemplate

DEFINE sequentialTemplate(nodeToReplace)

# create template nodes

SET seqPosted to new node(stateNum++)

SET seqStarted to new node(stateNum++)

SET seqCompleted to new node(stateNum++)

SET seqTerminated to new node(stateNum++)

# hook up first two template nodes to each other

setParentAndChild(seqPosted, seqStarted)

# set template nodes’ children

SET children to nodeToReplace.getChildren()

FOR X = 1 to the number of children

SET child to the Xth node of children

IF X = 1 THEN

setParentAndChild(seqStarted, first child of children)

ELSE

setParentAndChild(Xth – 1 child of children, child)

END IF

IF X = number of nodes in children THEN

setParentAndChild(child, seqCompleted)

END IF

unsetParentAndChild(nodeToReplace, child)

END FOR

# set first template nodes’ parents

SET parents to nodeToReplace.getParents()

FOR parent IN parents

setParentAndChild(parent, seqPosted)

unsetParentAndChild(parent, vertexToReplace)

END FOR

SET seqStates to new array(seqPosted, seqStarted, seqCompleted, seqTerminated)

RETURN seqStates

END DEFINE sequentialTemplate

DEFINE parallelTemplate(nodeToReplace, originalStates)

# create template nodes

SET parPosted to new node(stateNum++)

SET parStarted to new node(stateNum++)

SET parCompleted to new node(stateNum++)

SET parTerminated to new node(stateNum++)

# hook up first two template nodes to each other

setParentAndChild(parPosted, parStarted)

# set template nodes’ children (interleavings permutations)

SET interleavings to a new array of array of nodes

permuteStatesRecursive(nodeToReplace.getChildren(), 0, number of states)

FOR interleaving in interleavings   
 linkArrayNodes(interleavings)

removeVertexToRemoveFromArrayElementsParents(interleavings)

setParentAndChild(parStarted, interleavings[1])

addArrayToArray(states, interleaving)  
 END FOR

# remove original children

SET children to nodeToReplace.getChildren()

FOR child in children

unsetParentAndChild(vertexToReplace, child)

END FOR

# attach

# set template nodes’ parents

SET parents to nodeToReplace.getParents()

FOR parent IN parents

setParentAndChild(parent, parPosted)

END FOR

SET parStates to new array(seqPosted, seqStarted, seqCompleted, seqTerminated)

RETURN parStates

END DEFINE parallelTemplate

DEFINE addArrayToArray(targetArray, arrayToAdd)

FOR element in arrayToAdd

add element to targetArray

END FOR  
END DEFINTE

DEFINE removeVertexToRemoveFromArrayElementsParents(states)

FOR state in states

SET parents to state.getParents()

IF parents contains vertexToRemove

remove vertexToRemove from parents

END IF

END FOR  
END DEFINE

DEFINE permuteStatesRecursive(states, left, right)

SET numStates = number of states

IF left = right THEN

SET permutation to new array of nodes

FOR X=0 to numStates

SET permutation[X] = copy of Xth state

END FOR

Add permutation as new element in interleavings

ELSE

SET X=left

FOR X to right

swap(states, left, X)

permuteStatesRecursive(states, left+1, right)

swap(states, left, X)

END FOR

END IF

END DEFINE

DEFINE swap(array, element1, element2)

SET tempElement1 = element1

SET array[index of element1] = element2

SET array[index of element2] = tempElement1  
END DEFINE

DEFINE linkArrayNodes(array)

FOR X=1 to array length – 1

setParentAndChild(array[X], array[X+1])

END DEFINE

EVALUATION & EXPERIMENTAL RESULTS

When combining covid-hospitalization-simple.ljx and ventilator-order.ljx in parallel, I found the result was that the ventilator will always be available when it is needed. Three of the expanded mechanical ventilation nodes (posted, started and completed) of the covid-hospitalization-simple.ljx hold for A[qUp]. Since the ventilator is always available until it is needed, we know it won’t be needed before it’s available. By modelling the processes and graphing them in parallel, we have results which could potentially be live saving, in this theoretical example.

The combined covid-hospitalization-simple.ljx and ventilator-order.ljx processes, when translated with interleavings, have 1,892 nodes. I had not anticipated nearly this many nodes, but it turned out the large number of interleavings nodes made the number of nodes very large. Checking the A[qUp] model took the analyzer program 0.18 seconds, which was much faster than I had anticipated. I had feared the program would crash if the number of nodes became too big, but it actually handed them quite easily in this example.

Below are the translated graphs created by my algorithms modelling the Little-JIL translation specification. First are the translated graphs without interleavings and then the translated graphs with interleavings. The graphs with interleavings really blow up in size exponentially, compared to the graphs without interleavings.

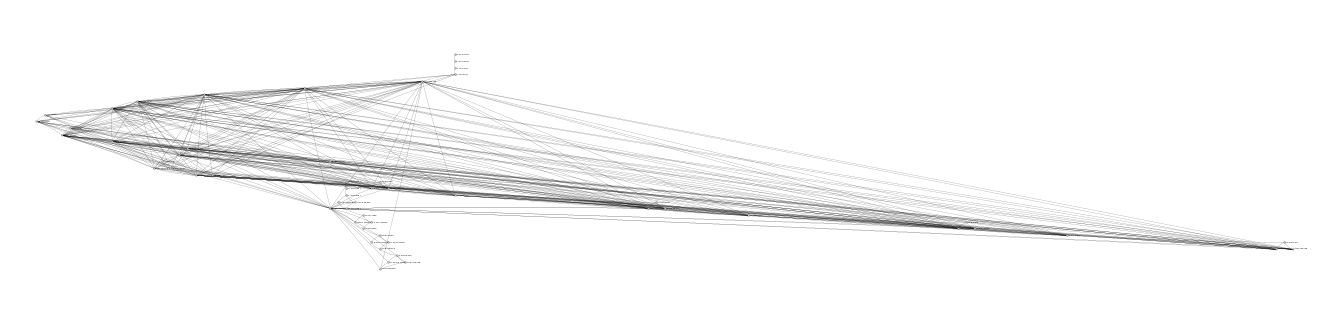


Figure 5. covid-hospitalization translation graph (without interleavings)



Figure 6. covid-hospitalization translation graph (with interleavings)

A picture containing snow, person, skiing, bed

Description automatically generated

Figure 7. ventilator-order translation graph (without interleavings)



Figure 8. ventilator-order translation graph (with interleavings)

Here are the graphs for the process checks XML graph and the process transfers XML graph, combined and running in parallel:

A picture containing bicycle

Description automatically generated  
Figure 13. combined ProcessChecks and ProcessTransfers graphs in parallel

Here is are the translated graphs without interleavings:

A close up of a logo

Description automatically generated

Figure 14. ProcessChecks graph translated (without interleavings)

A close up of a logo

Description automatically generated

Figure 15. ProcessTransfers graph translated (without interleavings)

The combined ProcessChecks and ProcessTransfers graphs translated and running in parallel with interleavings has 196 nodes. The analyzer checked the A[pUq] model in 0.00416 seconds. There are three states that hold for A[pUq]: s34, s149, and s15. These three nodes are part of the expansion of the initial sequential node of ProcessTransfers. Since p is depositing checks and q is processing transfers, when A[pUq] holds it means that checks are always deposited before the transfers are processed. This is very helpful information and could keep accounts from being incorrectly marked as overdrawn.

A close up of a logo

Description automatically generated

Figure 16. ProcessChecks and ProcessTransfers graph translated and running in parallel (without interleavings)

A picture containing crane

Description automatically generated

Figure 17. ProcessChecks and ProcessTransfers translated with interleavings and combined in parallel

FUTURE WORK

More work that could be done in this area is improving the readabililty and visual layout of the types of large graphs parallel Little-JIL process create. Improved visualization algorithms, while not affecting the logical results of CTL, could be helpful in quickly expressing the results clearly.

# Works Cited

|  |  |
| --- | --- |
| [1] | Y. Zhang, M. S. Squillante, A. Sivasubramaniam and R. K. Sahoo, "Performance Implications of Failures in Large-Scale Cluster Scheduling," *Lecture Notes in Computer Science,* vol. 3277, 2005. |
| [2] | A. L. T. a. S. J. Spear, "US National Library of Medicine," National Institutes of Health, Jun 2006. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1713207/. [Accessed 13 October 2020]. |
| [3] | Laboratory for Advanced Software Engineering Research, University of Massachusetts Amherst, "Little-JIL," 2006. [Online]. Available: http://laser.cs.umass.edu/tools/littlejil.shtml. [Accessed 2 August 2020]. |
| [4] | Laboratory for Advanced Software Engineering Research (LASER), University of Massachusetts Amherst, "Installing Visual-JIL," 2009. [Online]. Available: http://laser.cs.umass.edu/documentation/vjinstall/. [Accessed 2 August 2020]. |
| [5] | Oracle, "About the JFC and Swing," 2019. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/start/about.html. [Accessed 2 August 2020]. |
| [6] | SourceForge, "JUNG Java Universal Network/Graph Framework," 2010. [Online]. Available: http://jung.sourceforge.net/. [Accessed 2 August 2020]. |
| [7] | T. Parr, "Antlr," 2014. [Online]. Available: https://www.antlr.org/. [Accessed 2 August 2020]. |
| [8] | M. A. Gibbs, "COVID-19 a 3-step approach to intubation," (Covid-3-step-to-intubation-b.png in resources/misc/png folder in repo), 23 March 2020. [Online]. Available: https://litfl.com/covid-3-step-approach-to-intubation/. [Accessed 1 August 2020]. |
| [9] | UCSF COVID-19 Clinical Working Group, "UCSF Inpatient Adult COVID-19 Interim Management Guidelines," UCSF Health, 3 March 2020. [Online]. Available: https://infectioncontrol.ucsfmedicalcenter.org/sites/g/files/tkssra4681/f/UCSF%20Adult%20COVID%20draft%20management%20guidelines.pdf. [Accessed 1 August 2020]. |
| [10] | R. B. Jennifer L.W. Fink, "How Hospitals Treat COVID-19 Patients," 16 June 2020. [Online]. Available: https://www.healthgrades.com/right-care/coronavirus/how-hospitals-treat-covid-19-patients. [Accessed 18 August 2020]. |
| [11] | Walter Reed Bethesda, "COVID 19 Code Blue," 12 April 2020. [Online]. Available: https://www.youtube.com/watch?v=n\_JGBpdc\_-U. [Accessed 18 August 2020]. |
| [12] | D. M. Hansen, "How to Treat Coronavirus Patients in the ICU (Intensive Care Unit) | Covid-19," 30 March 2020. [Online]. Available: https://www.youtube.com/watch?v=i7U2pkeysXI. [Accessed 18 August 2020]. |
| [13] | What If, "What Happens If You Were Hospitalized With COVID-19?," 5 May 2020. [Online]. Available: https://www.youtube.com/watch?v=oSHXOlnmNXI. [Accessed 18 August 2020]. |
| [14] | New York State Task Force on Life and the Law New York State Department of Health, "Ventilator Allocation Guidelines," New York State Department of Health, 2015. |
| [15] | K. AM, "Who buys medical devices for hospitals?," 24 July 2015. [Online]. Available: https://www.quora.com/Who-buys-medical-devices-for-hospitals. [Accessed 18 August 2020]. |
| [16] | J. M. Cobleigh, "Verifying Properties of Process Definitions," University of Massachusetts, Amherst, 2000. |
| [17] | M. R. a. A. Gillis, "Static Analysis (Static Code Analysis)," 1 August 2020. [Online]. Available: https://searchsoftwarequality.techtarget.com/definition/static-analysis-static-code-analysis. |