RIGOROUS ANALYSIS OF COMBINED SOFTWARE PROCESSES VIA MODEL CHECKING

by

Mark McDermott, B.A.

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 2021

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my thesis advisor, Dr. Rodion Podorozhny, for his invaluable guidance. I am also indebted to Meredith Poor, whose advice, tutoring and friendship have helped me in my computer science pursuits from the beginning. I wish to thank my wife, Lisa McDermott, for her thankless patience and support during my many long hours of study. I also wish to thank Thomson Tan and William Olson for each helping me immensely when I really needed it. Lastly, I wish to thank my mother, Karen Halbasch, for her love and support and for teaching me how to use a Commodore 64 and my late father, Ron McDermott, whose love, patience and insight have helped me throughout both my studies and my life in general.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The ability to automatically combine and analyze multiple concurrent processes, perhaps written by different people, becomes increasingly important in the modern mobile, distributed, and ad-hoc computational environments. For instance, medical processes that guide medical procedures in hospitals are written by different people, yet they are performed concurrently on the same patient. There are critical properties that such combined, concurrent processes must adhere to. Failure to adhere to such properties may result in the loss of life or serious disability. This work presents an automatic verification system written from scratch (about 32,000 lines of Java) that takes in rigorous descriptions of individual processes in Little-JIL, translates them semantically, combines them into an ad-hoc concurrent process, and performs static verification against specified critical properties via CTL model checking algorithms.

I. 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. In the modern age of mobile, distributed and ad hoc computing, it has become even more important to be able to automatically find errors in processes written by different people which are being run concurrently. One approach to identifying errors in parallel processes is to use model checking on a process language.

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 processes executed in parallel. So, the number of interleavings is $x!$ where x is the number of processes. There can thus be an exponential blowup in the number of interleavings. In modelling our processes in a process language like Little-JIL, which breaks down each state into further sub-states representing the state's status, our graphs become even larger.

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 a parallel process with a certain subprocess that needs a certain property will never execute without that property, it would be very helpful.

Race conditions accidentally introduced in parallel programs 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. One approach to reducing ambiguity in processes is to model the process explicitly in a process language, such as Little-JIL.

Little-JIL models contain control structures such as *leaf*, *sequential*, *parallel*, *try*, and *choice*. Each of these then have their own translation into a template of states. A leaf state's translation, as shown in Figure 1, starts with a leaf control structure. This is fed into our program in an XML (extensible markup language) representation. The leaf control structure is then converted to its semantic translation consisting of *leaf posted*, *leaf started*, *leaf completed* and *leaf terminated* [3]. The *leaf posted* transitions into the *leaf started* through its connecting edge. The *leaf started* then has edges going to both *leaf completed* and *leaf terminated*. These multiple nodes which both connect to the *leaf* process successfully or terminating the program due to failure or error.

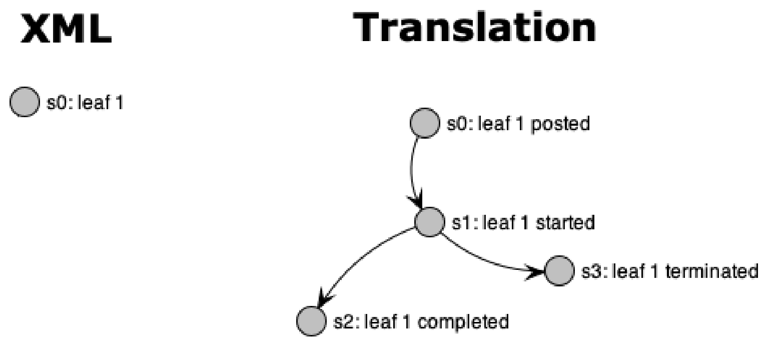


Figure 1. Little-JIL Leaf State Translation

Little-JIL's sequential control structure's translation is shown in Figure 2. The initial control structure shown on the left represents a process path where first *leaf 1* is traversed and then *leaf 2*. It is sequential because the *s2* is never started until the *s1* is finished. The sequential translated template on the right starts at *sequential posted*, then traverses to *sequential started*. *Sequential started* then traverses to *leaf 1 posted*, which then goes to *leaf 1 started*. *Leaf 1 started* then connects to both *leaf 1 completed* and *leaf 1 terminated*. Note that the leaf is being translated exactly how it was in Figure 1, but now because it is a child of the sequential node, the *leaf 1 posted* is transitioned into from *sequential 1 started* and *leaf 1 completed* then continues on to *leaf 2 posted,* while *leaf 1 terminated* continues on to *sequential 1 terminated*. In general, completed nodes continue on to the next control structure, if there is one, while terminated nodes continue on to the next terminated node, if there is one.

In Figure 2 we see that after *leaf 1* completes, the process continues to *leaf 2 posted*, which then continues to *leaf 2 started*. *Leaf 2 started* connects to both *leaf 2 completed* and *leaf 2 terminated*. *Leaf 2 completed* continues to *sequential 1 completed*, which *leaf 2 terminated* continues on to *sequential 1 terminated*. So, we see how a sequential node will transition through its first child, then through its second child and so on, before returning to end on either its own *sequential completed* or *sequential terminated* node.

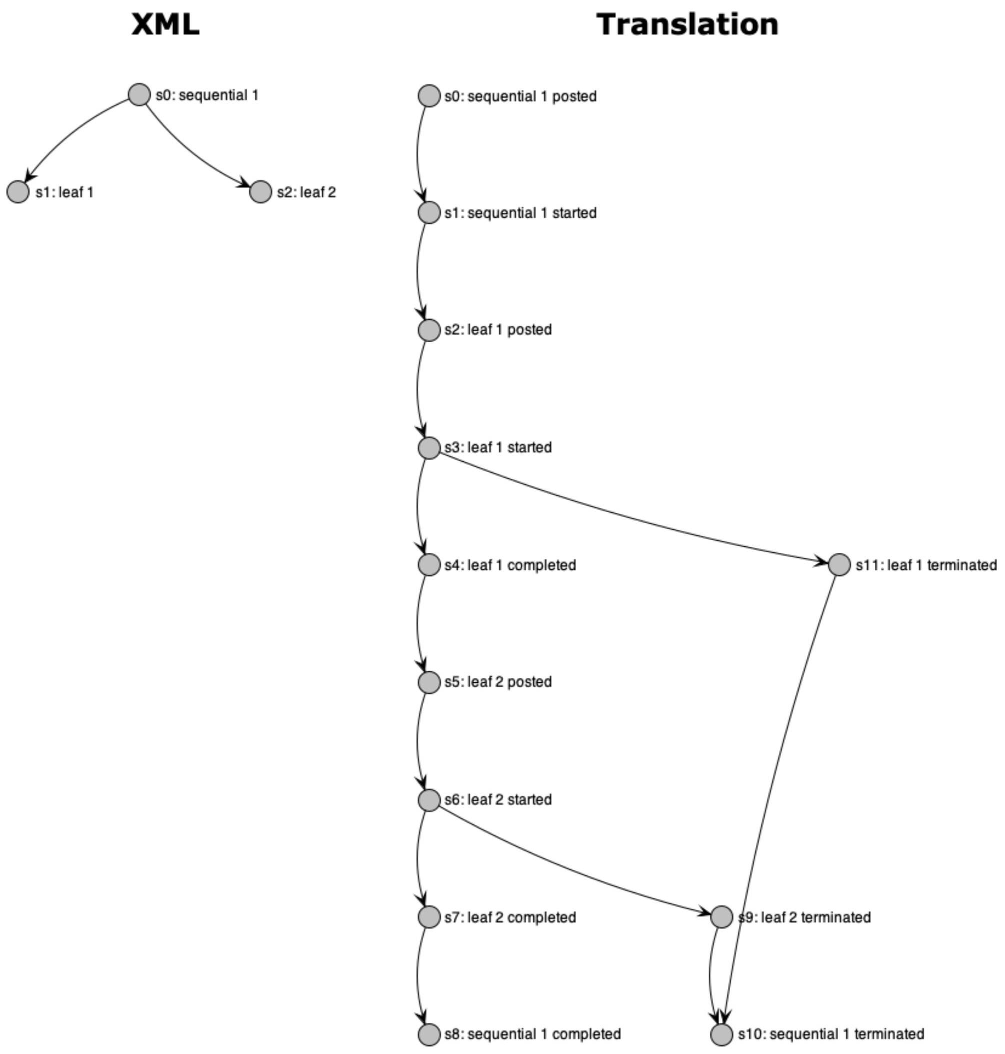


Figure 2. Little-JIL Sequential State Translation

In Figure 3 we see how a Little-JIL parallel node is translated. The initial control structure format is shown on the left, which would come from the XML file. Here *parallel 1* connects to both *leaf 1* and *leaf 2*. We can see this is exactly the initial form of the sequential with two leaf children in its control structure XML form in Figure 2. The initial Little-JIL structures show the different paths a process can take, but the order in which the process will traverse its nodes is only inferred by the name of the control structures. The order is not clearly marked out by the structure's directed edges, as it is in the accompanying translated templates.

On the right of Figure 3 the translation template of the parallel node with two leaf nodes is shown. It begins at *parallel 1 posted*, which transitions to *parallel 1 started*. Then *parallel 1 started* connects to both *leaf 1 posted* and *leaf 2 posted*. This could represent the different ways parallel process are actually executed in practice in an unthreaded, single core situation or this could just be a sequential representation of two actual concurrent processes. One child is executed first, then the other child, but we generally do not know which one will go first because it could be either. Both *leaf posted* nodes connect to their respective *leaf started* nodes. Both *leaf started* nodes then connect to their respective *leaf completed* nodes as well as their respective *leaf terminated* nodes. Then each *leaf completed* node connects to parallel 1 completed as well as to the other leaf's *posted* node. So, *leaf 1 completed* connects to *parallel 1* and also to *leaf 2 posted*. Each leaf's *terminated* node connects to *parallel 1 terminated*. There is some ambiguity in this translation template in the sense that although the first executed *leaf completed* node has an edge pointing to *parallel 1 completed*, that flow will never actually happen. The *leaf completed* node of first *leaf executed* will always transition next to the other leaf's *posted* node. The edge connecting that initial *leaf completed* to *parallel 1 completed* really only represents the path the second executed leaf will take after it's *completed* node is finished. This ambiguity is removed when the translation's explicit interleavings are shown, as in Figure 4.

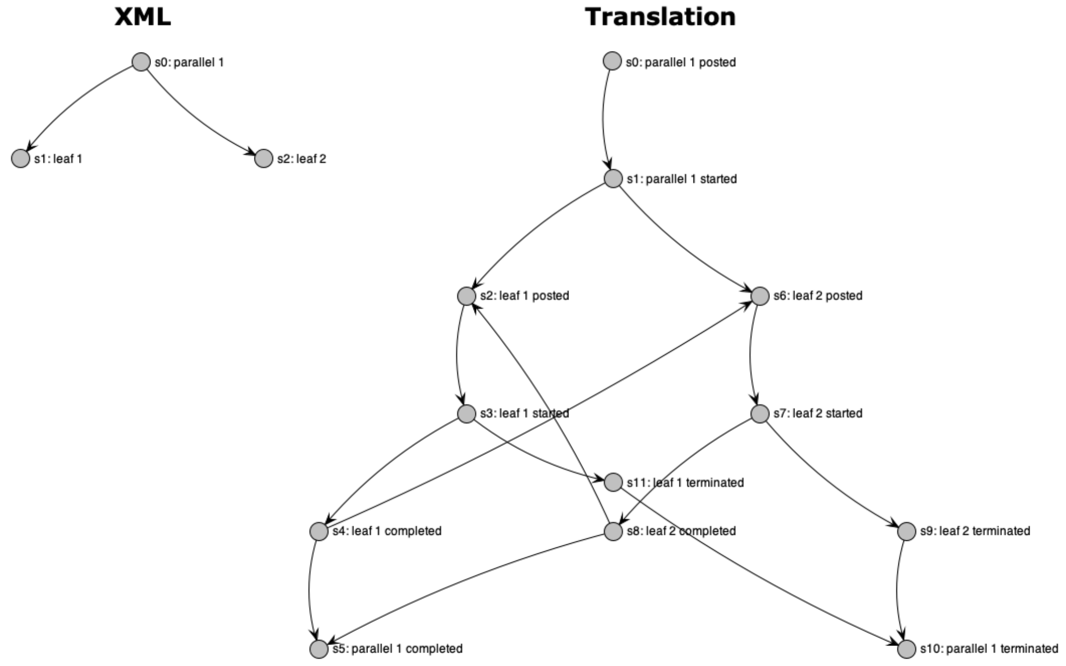


Figure 3. Little-JIL Parallel State Translation

The structure on the right side of Figure 3 and the structure shown in Figure 4 are semantically equivalent. The parallel template in Figure 3 shows the different possible interleavings by having both *leaf completed* nodes point to both *parallel 1 completed* and the other leaf's *posted* node. This is slightly ambiguous because the reader has to infer which leaf executes first and which second and understand that only the second actually transitions to *parallel 1 completed*, while the first *leaf completed* actually only transitions to the other leaf's *posted* node. These kinds of edges are called "may immediately precede" edges because their existence in the actual chosen path is not necessarily a given. If the reader did not understand this, they might incorrectly infer that the diagram is even suggesting a loop might be possible with one leaf transitioning to the other leaf and then back to the first leaf. Yet the small size and simplicity in the calculation of this graph are the benefits this trade off.

Figure 4 shows the semantically equivalent version of the parallel template translation on the right of Figure 3, but with the ambiguity removed by showing the leaves' explicit interleavings and what exactly they will transition into. We see that it begins with *parallel 1 posted* and that node transitions into *parallel 1 started*. *Parallel 1 started* then connects to *leaf 1 posted* and *leaf 2 posted*. *Leaf 1 posted* connects to *leaf 1 started* and *leaf 1 started* connects to *leaf 1 completed* and *leaf 1 terminated* as we would expect after learning the leaf's translation template in Figure 1. *Leaf 1 completed* connects to *leaf 2 posted*. Here is one interleaving shown. *Leaf 2 posted* connects to *leaf 2 started*, which connects to *leaf 2 completed* and *leaf 2 terminated*. *Leaf 2* connects to *parallel 1 completed* and *leaf 2 terminated* connects to *parallel 1 terminated*.

The second interleaving shown in Figure 4 is when *parallel 1* connects to *leaf 2 posted*, which then connects to *leaf 2 started* and *leaf 2 started* connects to *leaf 2 completed* and *leaf 2 terminated*, while *leaf 2 completed* connects to *leaf 1 posted*. The first interleaving showed *leaf 1* being executed first and then transitioning into *leaf 2* (or into *parallel 1 terminated*), while this interleaving is showing *leaf 2* executing first and then transitioning into *leaf 1* (or into *parallel 1 terminated* as well). This interleaving mirrors the first. In this interleaving, *leaf 1 posted* connects to *leaf 1 started* which connects to *leaf 1 completed* and *leaf 1 terminated*. *Leaf 1 completed* connects to *parallel 1 completed* and *leaf 1 terminated* connects to *parallel 1 terminated*.

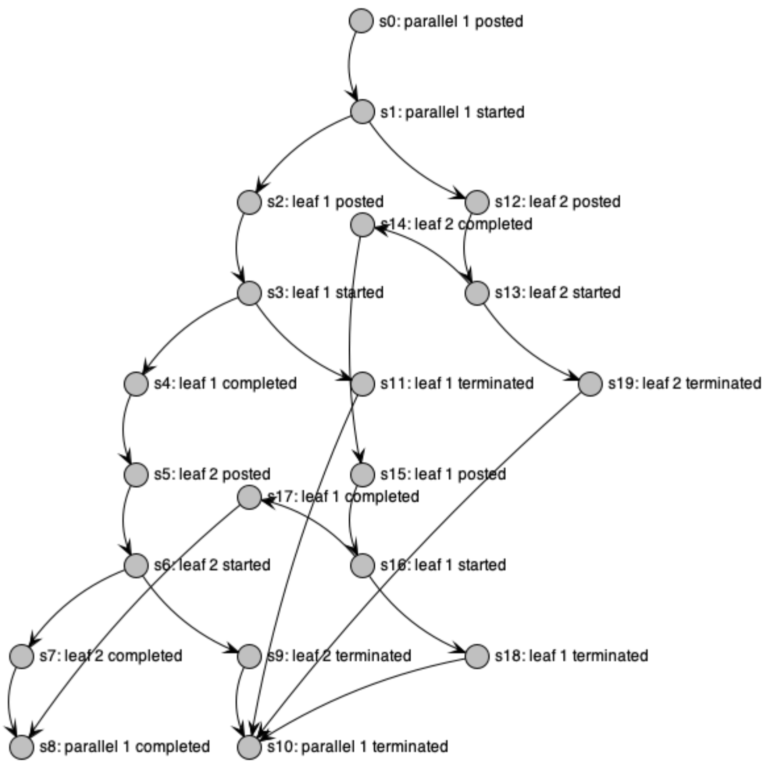


Figure 4. Translation Of The XML In Figure 3, Including The Leaves' Explicit Interleavings

A simple process of two leaves run sequentially, when expanded according to Little-JIL rules, becomes a graph of 12 nodes (see Figure 2). Similarly, two leaves executed in parallel, expand to a graph of 12 nodes after initial translation and then to a graph of 20 nodes after the explicit interleavings are modeled (see figures 3 and 4). 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 begin speaking of the "topology" of the graph, abstracting out some of the less significant details because there are so many nodes that the diagram is zoomed out to include them all, which makes the individual nodes and their names so small that they are hard or impossible to read.

II. APPROACH

This paper focuses on the ability to automatically combine multiple processes, possibly written by different people, so they run concurrently and can then be checked for certain properties. This is achieved through translation of Little-JIL control structures to their Little-JIL template states and by modelling the interleavings of parallel processes. The inputs to the translation program are XML files representing the processes modelled in Little-JIL syntax. The outputs of the translation program are Kripke structures, which are then ready for model checking.

I use CTL (computation tree logic) model checking and the Little-JIL process language to identify errors in parallel processes and to identify specific paths the error route takes. Since there are different execution paths a process may take, a process 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 perform processes [5]. It has been chosen because it is easy to use, yet powerful enough to model a wide variety of real-world processes [6].

A process is just a collection of steps. A formalized process language like Little-JIL has control structures like *sequential*, *parallel* and *try*. I use two processes in this paper (each consisting of two combined processes), a computer program and a medical process. I first model the abstract process with the graphical process language. Then I output the graphical process into XML. Now that the process is represented by code, I run it through the translator. The translator takes in one or more files of XML steps and outputs a directed graph data structure that represents the steps. The translator also translates the control structure nodes of the graph into their semantic substeps, according to the rules laid out by our graphical process language. At this point, I have a Kripke structure ready for model checking. I use a temporal logic on our Kripke structure to formally verify whether our combined parallel processes have certain safety or liveness properties.

III. PROBLEM DESCRIPTION

The problem this paper tries to solve is how to automatically combine two or more processes, potentially written by different people or organizations, into a single process with fine granularity and then check for various properties on the combined graph. This could make quick, painless, and potentially lifesaving fixes to errors introduced when processes are combined. In the modern mobile, distributed, and ad hoc computing environments, there is more need than ever to be able to quickly and easily combine processes and make sure they work together as intended.

The first example problem I look at is the combined parallel processes of a patient hospitalized with COVID-19 (coronavirus disease 2019) and the hospital's ordering of a ventilator from their supplier. We would like to know two things here. The first is, "Is it true that someone who is a high COVID-19 risk to others will never be at home?" In this question, the hope is that perhaps low risk individuals will quarantine at home, but that high-risk individuals would hopefully be in the hospital where their chance of widely spreading the virus would be much lower. The second question we would like to know is, "If a ventilator is requested, will it always eventually be available to use?" The concerns here are whether the ventilator will be shipped and if the supplier has any ventilators currently in stock.

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.

IV. SOLUTION DESIGN

I decided to use Little-JIL to model processes for this paper for its combination of expressiveness and simplicity. Little-JIL is a graphical language that models processes, which was developed by Leon Osterweil at University of Massachusetts in 1998 and was further developed over the next sixteen years [7]. The tools Little-JIL provides allow for the modelling of processes followed by autonomous agents and the modelling of all the different possible paths a process is allowed to take. It introduces types of states, like *sequential*, *parallel* and *leaf*, as well as semantic translations of those each of those states, into nodes such as *started*, *completed* or *terminated*. The parallel node will be the core of our modelling of parallel processes.

The Little-JIL semantic translation states help bring specificity to our processes and help eliminate ambiguity. For example, say a state is just “perform heart transplant”. If it’s running in parallel with another state and the two alternate switching back and forth, we won’t know if the heart 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 specificity can help avoid errors caused by ambiguity. This is an example of Little-JIL’s expressive power.

Little-JIL has a helpful implementation which uses the Eclipse IDE (integrated development environment) [8]. In Eclipse, I’ve modelled our processes visually in Little-JIL diagram, which has vague parallels to a UML-type (Unified Modeling Language) diagram. The Little-JIL Eclipse extension has an “export to XML” feature. I have used this to get my initial XML files. When Little-JIL uses XML, it uses the file extension “ljx” instead of “xml”, but the file is actually just an XML file. In this paper, I refer to these files as just XML files.

These XML files are the representation of my processes that I feed into the translator. 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 have also removed the connector XML tags to simplify the XML reading module.

I’ve written the translator in Java. The entire program, including the translator and the analyzer, is about 32,000 lines of code. 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 outputs of the algorithms are Kripke structure graphs of nodes of the different paths the process can take. The visual outputs of the translation are graphs displayed using Java Swing [9] and the JUNG (Java Universal Network/Graph Framework) [10] graphing library. To run model checking CTL algorithms, I’ve used the ANTLR (Another Tool for Language Recognition) [11] 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.

Little-JIL diagrams use certain symbols to represent different control structures. As seen in Figure 5 below, there are various symbols in blue on the black step bars: an equals sign for a parallel step, an arrow for a sequential step and no symbol if it’s a leaf step. The circles above the steps are interface badges and the downward triangle and upward triangle are pre-requisite badge and post-requisite badge, respectively. Although these are helpful features of Little-JIL, we need not concern ourselves with the interface badges and pre- and post-requisites for the purposes of this paper. [12]

The COVID-19 problem is fairly straightforward, as shown below in Figure 5. It is two processes run in parallel: an individual's experience with COVID-19 and how a hospital orders a ventilator. There is a parallel root step added to connect the two separate processes and to execute them concurrently. The COVID-19 process consists of two parallel processes run in sequence. First is the initial response and then is hospitalization. The initial response consists of isolating at home, monitoring symptoms, researching and planning how to get a swab test, and then researching and planning a hospital visit. The hospitalization consists of a RT-PCR (reverse transcription polymerase chain reaction) test, a lung ultrasound, a chest CT (computed tomography) scan and ventilation. The hospital orders ventilation process consists of five steps run sequentially: medical team requesting the ventilator from management, the hospital requesting the ventilator from their vendor, the vendor shipping the ventilator, the hospital receiving the ventilator, and the hospital's orderly setting up the ventilator in the room. The first of those, the medical team requesting the ventilator from management is further broken down into three steps run sequentially: the team filling out the purchase order, the team requesting approval from management, and the team receiving feedback from management. These were both processes I put together myself after researching online the different processes hospitals were using for COVID-19 and for procuring medical equipment. [13] [14] [15] [16] [17] [18] [19] [20]

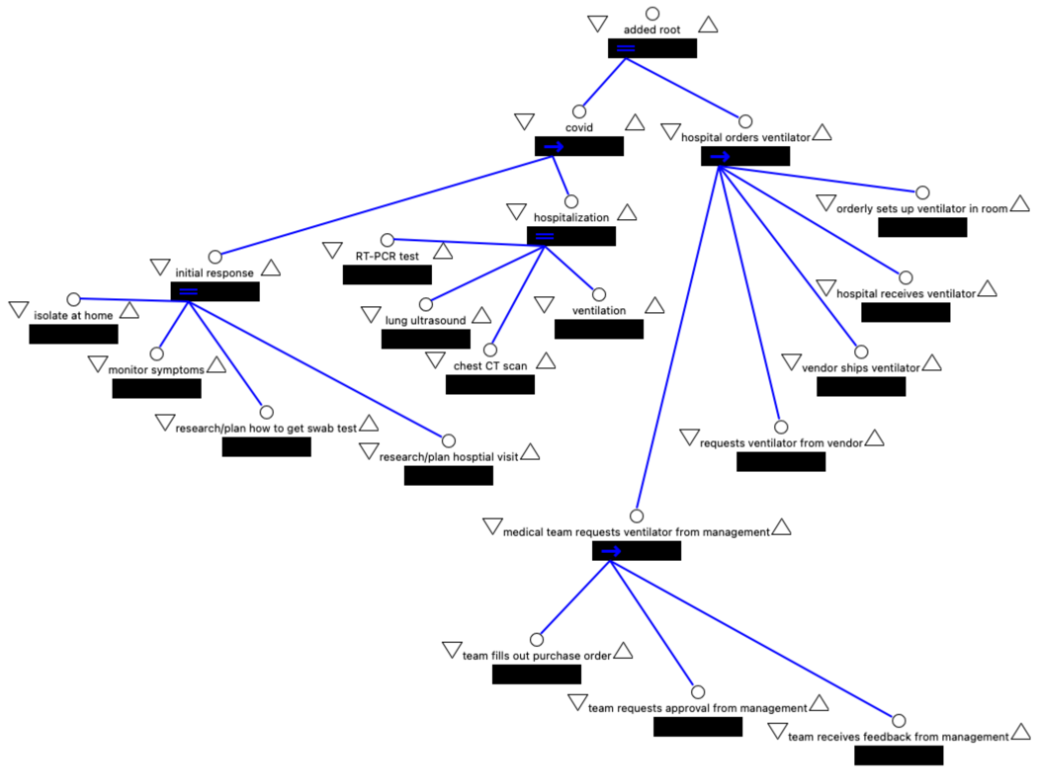

added root
covid
hospital orders ventilator
hospitalization
orderly sets up ventilator in room
initial response
RT-PCR test
hospital receives ventilator
isolate at home
lung ultrasound
ventilation
chest CT scan
vendor ships ventilator
monitor symptoms
research/plan how to get swab test
requests ventilator from vendor
research/plan hosptial visit
medical team requests ventilator from management
team fills out purchase order
team requests approval from management
team receives feedback from management

Figure 5. Little-JIL Diagram Of COVID-19 Problem

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. The code for both is included here in Appendix B. Both *ProcessChecks* and *ProcessTransfers* is its own class. There is also a *Bank* class with utility methods for parsing csv files and there are some additional class files for the data structures of account, check and transfer. Both the *ProcessChecks* and *ProcessTransfers* files are implemented with Java callable so they run asynchronously to demonstrate parallel execution. *ProcessChecks* takes in a string variable for the file path to the checks csv file. It first calls a *getChecksList* function, which splits the csv file by line and saves each line as an element in an array. Then it creates a check object for each csv row and creates a list of checks. It next calls a *depositChecks* function, which loops through the list of checks and for each check gets the account number, gets the account for that account number and then deposits the check in the account. The actual deposit method is a method on the account data type. The deposit method just adds the amount of the check to the account balance.

The *ProcessTransfers* file is very similar to the *ProcessChecks* file. It mirrors each method but with transfers instead of checks. First it gets each line of the transfers csv file, then it gets the transfers from each line. Then it processes the transfer. In the *processTransfer* method, it gets the from account and the to account for the transfer, as well as the transfer amount. There are some if statements creating some logic checking if the from and to accounts are both accounts at the bank. If they are, then the transfer amount is subtracted from the from account and added to the to account.

Both class files were then represented in Little-JIL. Figure 6 shows the *ProcessChecks.java* Little-JIL representation. The root node, *process checks*, is a sequential control structure. *Process checks* has three children: *set class's bank property*, *get checks list from csv*, and *depositChecks(checks)*. *Set class's bank property* is a leaf, while both *get checks list from csv* and *depositChecks(checks)* are sequential steps. *Get checks list from csv* has two children: *get list of csv rows* and *get list of checks from csv rows*. Both of these are sequential steps. Get list of csv rows has five children: *init stringList*, *get bufferedReader*, *set i=0*, *readCsv row*, and *close csvReader*. All of these are leaf steps except for *readCsv row*, which is sequential. *ReadCsv* row has three children, *set row = csv.readLine()*, *csvRows.addRow(row)*, and *i++*. *Get list of checks from csv rows* has two children: *init checks var*, a leaf, and *add check to checks*, a sequential step. *Add check to checks* has four leaf children: *checkElems = row.split*, *get check details var*, *create check*, and *add check to checks*. *DepositChecks(checks)* is a sequential with one sequential child, *depositCheck*. *DepositCheck* has two children: *acct = bank.getAcct()* and *set balance = bal + amt*.

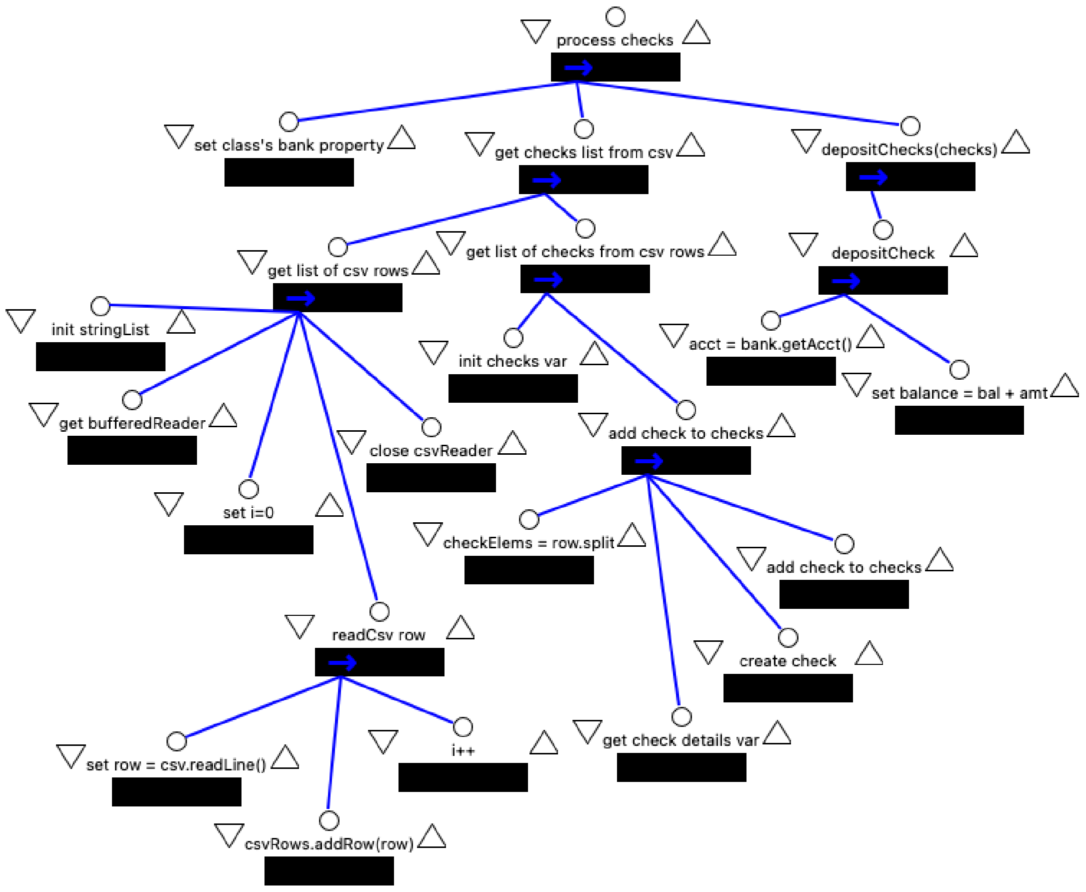


Figure 6. Process Checks Little-JIL Diagram

In many ways, the process transfers diagram in Figure 7 mirrors the process checks diagram, since the code of the two files has quite a similar control flow. The root node, *process transfers*, is a sequential step with three children, *set class bank property*, *get transfers list from csv*, and *processTransfers*. *Set class bank property* is a leaf, while the other two are sequential steps. *Get transfers list from csv* has two children, both sequential: *get list of csv rows* and *get transfers from rows*. *Get list of csv rows* has five children: *init csvRows & declare row var*, *get bufferedReader*, *i=0*, *readCsv row*, and *close csvReader*. All five are leaves except for *readCsv row*, which is sequential. *ReadCsv row* has three leaf children, *row = csv.readLine*, *csvRows.addRow*, and *i++*. *Get transfers from rows* has two leaf children, *init transfers var* and *get transfer*. *ProcessTransfers* has one sequential child, *processTransfer*. *ProcessTranfer* has three children, *get transfer details var*, *process from acct transaction*, and *process to acc transaction*. *Get transfer details var* is a leaf while the other two are sequential. *Process from acct transaction* has two leaf children: *fromAcct = bank.getAcct* and *fromAcct.withdraw*. *Process to acct transaction* has two leaf children: *toAcct = bank.getAcct* and *toAcct.deposit*.

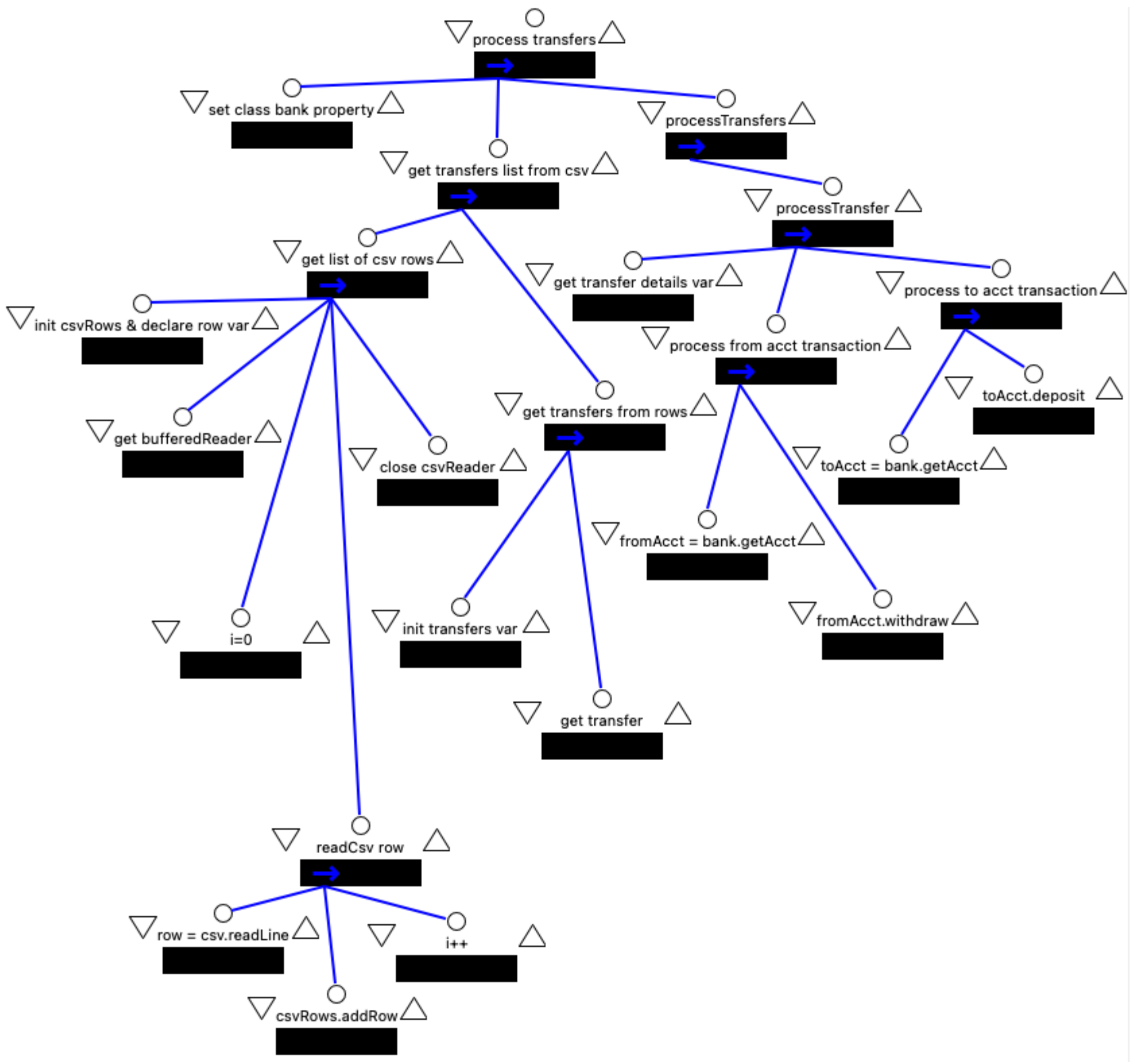


Figure 7. Process Transfers Little-JIL Diagram

The first step in translation was to get the manually created XML file (based on the manually created Little-JIL diagrams above) into a Kripke structure. This is the initial iteration of the Kripke structure which will later be replaced by the translated Kripke structure and after that by the translated Kripke structure which includes all the parallel interleavings. To get the Kripke structure representation of the XML files, I wrote a Java translator which iterates over the lines of the XML file, creates a vertex data structure for each XML tag and then put all the created vertices in an array. Quite a lot of information is stored for each vertex in the vertex data structure, including node number, node name, Little-JIL control structure type, what parents and children the vertex has, and if it has siblings, what sibling number it is and so on.

You can see the XML graph of the COVID-19 parallel process in Figure 8. The process that is the individual's experience of COVID-19 is on the left side of the root node and the hospital ordering the ventilator process is on the right side of the root node. The root node is an added parallel node so the process becomes the a parallel process, executing both processes in parallel. The left side of the process starts with a sequential step that is simple the individual's experience of COVID. The two children of that node are the initial response node, a parallel node, and the hospitalization node, also parallel node. The initial response has four leaf children: isolate at home, monitor symptoms, research/plan how to get swab test, and research/plan a hospital visit. The hospitalization node also has four leaf children: the RT-PCR test, a lung ultrasound, a chest CT scan, and ventilation. On the right side, hospital orders ventilator has one sequential step, medical team requests ventilator from management, and four leaf children: hospital requests ventilator from vendor, vendor ships ventilator, hospital receives ventilator, and orderly sets up ventilator in room. The medical team requests ventilator from management step has three leaf steps: team fills out purchase order, team requests approval form from management, and team receives feedback from management.

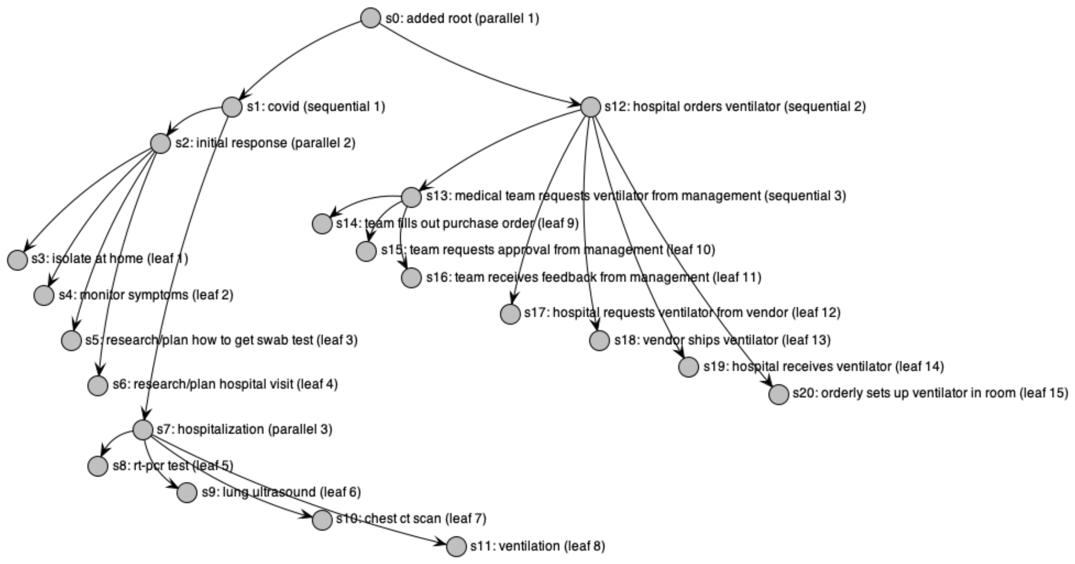

s0: added root (parallel 1)
s1: covid (sequential 1)
s2: initial response (parallel 2)
s3: isolate at home (leaf 1)
s4: monitor symptoms (leaf 2)
s5: research/plan how to get swab test (leaf 3)
s6: research/plan hospital visit (leaf 4)
s7: hospitalization (parallel 3)
s8: rt-pcr test (leaf 5)
s9: lung ultrasound (leaf 6)
s10: chest ct scan (leaf 7)
s11: ventilation (leaf 8)
s12: hospital orders ventilator (sequential 2)
s13: medical team requests ventilator from management (sequential 3)
s14: team fills out purchase order (leaf 9)
s15: team requests approval from management (leaf 10)
s16: team receives feedback from management (leaf 11)
s17: hospital requests ventilator from vendor (leaf 12)
s18: vendor ships ventilator (leaf 13)
s19: hospital receives ventilator (leaf 14)
s20: orderly sets up ventilator in room (leaf 15)

Figure 8. COVID-19 XML Graph

You can see the XML Kripke structure of the process checks file below in Figure 9. It is identical in structure to the Little-JIL diagram in Figure 6. The visual display of the Kripke structure in Figure 9 was created using the JUNG Java graphing library, which is great for creating graphs [10]. After translation from the XML to the Kripke, the Little-JIL steps are states in a finite state machine. Each state has a name like *s0*, where zero is the number of that step. Each state also has its description shown after the name and in parentheses after that, its Little-JIL control structure type and number. Including the Little-JIL type and number is helpful for future comparison after this graph is translated into its Little-JIL status steps, since at that point each Little-JIL type (*sequential*, *leaf*, etc.) will actually be four or more nodes (*posted*, *started*, *terminated*, etc.). Since the Kripke structure of the graphs in Figures 9 and 10 are identical to the

Little-JIL structure of the graph in Figures 6 and 7 and since I described those graphs at great length above, I will omit further description of them here.

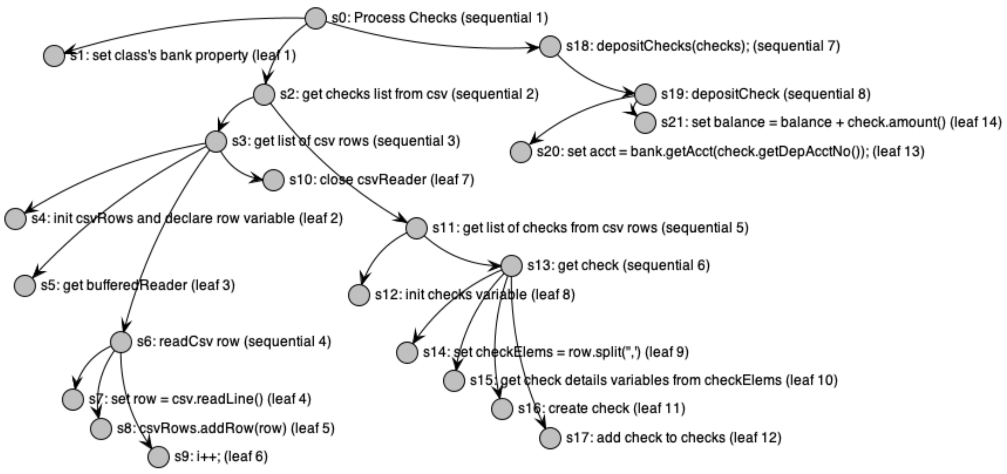

s0: Process Checks (sequential 1)
s1: set class's bank property (leaf 1)
s18: depositChecks(checks); (sequential 7)
s2: get checks list from csv (sequential 2)
s19: depositCheck (sequential 8)
s21: set balance = balance + check.amount() (leaf 14)
s3: get list of csv rows (sequential 3)
s20: set acct = bank.getAcct(check.getDepAcctNo()); (leaf 13)
s10: close csvReader (leaf 7)
s4: init csvRows and declare row variable (leaf 2)
s11: get list of checks from csv rows (sequential 5)
s13: get check (sequential 6)
s5: get bufferedReader (leaf 3)
s12: init checks variable (leaf 8)
s6: readCsv row (sequential 4)
s14: set checkElems = row.split(",") (leaf 9)
s15: get check details variables from checkElems (leaf 10)
s7: set row = csv.readLine() (leaf 4)
s16: create check (leaf 11)
s8: csvRows.addRow(row) (leaf 5)
s17: add check to checks (leaf 12)
s9: i++; (leaf 6)

Figure 9. Process Checks XML Graph

Just like how the XML graph in Figure 8 is semantically equivalent to the Little-JIL diagram in Figure 6, the process transfers XML graph in Figure 9 is semantically equivalent to the Little-JIL process transfers diagram in Figure 7. Now the Little-JIL steps are represented in a directed graph using node names, node descriptions and in the Little-JIL type and number in parentheses. The structure of the graph in Figure 9 is identical to the graph in Figure 7, which is described at great length above in that section.

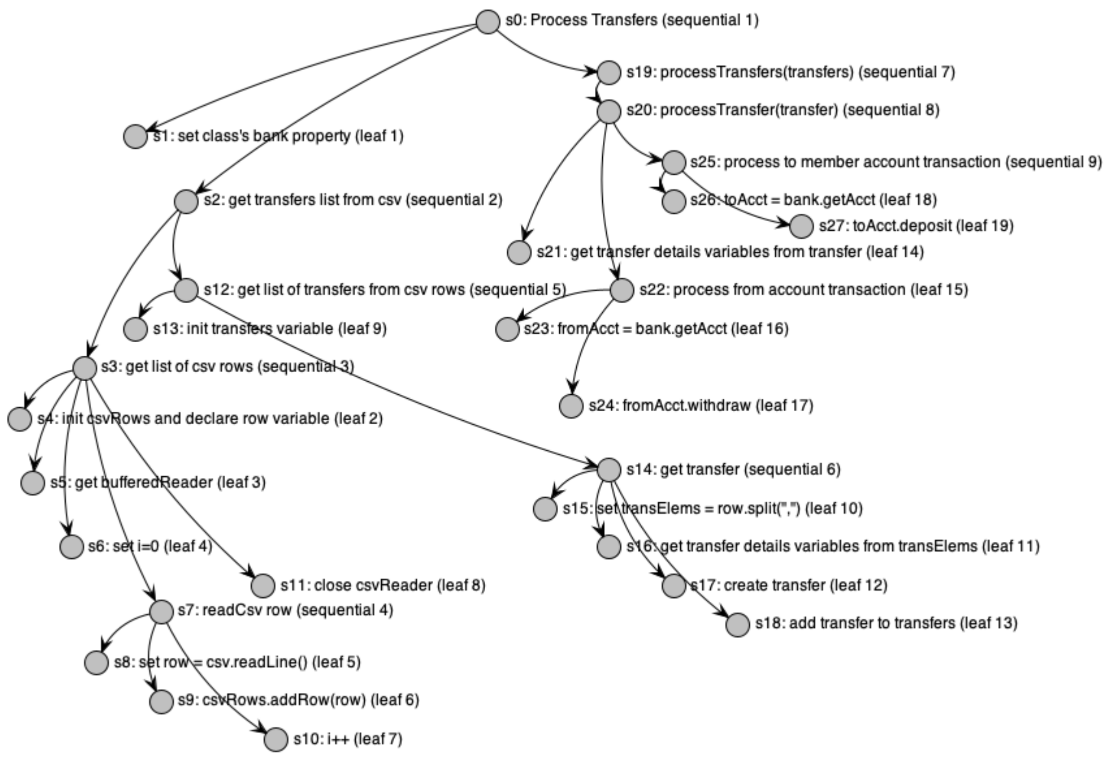

s0: Process Transfers (sequential 1)
s1: set class's bank property (leaf 1)
s2: get transfers list from csv (sequential 2)
s3: get list of csv rows (sequential 3)
s4: init csvRows and declare row variable (leaf 2)
s5: get bufferedReader (leaf 3)
s6: set i=0 (leaf 4)
s7: readCsv row (sequential 4)
s8: set row = csv.readLine() (leaf 5)
s9: csvRows.addRow(row) (leaf 6)
s10: i++ (leaf 7)
s11: close csvReader (leaf 8)
s12: get list of transfers from csv rows (sequential 5)
s13: init transfers variable (leaf 9)
s14: get transfer (sequential 6)
s15: set transElems = row.split(",") (leaf 10)
s16: get transfer details variables from transElems (leaf 11)
s17: create transfer (leaf 12)
s18: add transfer to transfers (leaf 13)
s19: processTransfers(transfers) (sequential 7)
s20: processTransfer(transfer) (sequential 8)
s21: get transfer details variables from transfer (leaf 14)
s22: process from account transaction (leaf 15)
s23: fromAcct = bank.getAcct (leaf 16)
s24: fromAcct.withdraw (leaf 17)
s25: process to member account transaction (sequential 9)
s26: toAcct = bank.getAcct (leaf 18)
s27: toAcct.deposit (leaf 19)

Figure 10. Process Transfers XML Graph

Both the COVID-19 and the banking problems can be expressed in logical terms with CTL temporal qualifiers as well as in plain English. The proposition "someone who is a high risk to others will never be at home" is represented as $AG(\neg(s \wedge q))$, where s is *high risk to others* and q is *at home*. The plain English expression, "If a ventilator is ever requested, it will always eventually be available to use" is $AG(u \rightarrow AF(v))$ where u is *ventilator requested* and v is *ventilator available to use*. These are the models to check to see if their properties hold for the whole graph. Checking if someone who is high risk will never be home is a safety property. We check if, under certain condition, the event never occurs. Likewise, checking if a requested ventilator will eventually be available for use is a liveness property. This is because we are checking if, under a certain condition, an event does occur.

The initial banking problem in English, “Transfers are never processed before all the checks are deposited” can be rephrased more technically as, “The transfer buffer is always null until the checks array size is zero and the checks buffer is closed.” In CTL, this can be expressed as is $A[\neg t\ U(s\wedge\neg q)]$, where t is the transfer buffer, s is the condition that the checks array size equals zero and q is the checks buffer.

V. 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* [12]. 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. Most of the utility and helper functions have been omitted here for brevity.

The three main subroutines for my translation strategy are *translate(), translateChildrenRecursively()*, and *swapInTemplate()*, all shown below in Figure 12. Also key to my translation strategy are the leaf (Figure 13), sequential (Figure 14), and parallel (Figure 15) template subroutines. All these functions are presented here in pseudocode, but the full Java implementation code is available in Appendex A. The functions mention the *VertexList*, which is an array-like data structure I've created which stores all the vertices in the graph along with information like which vertex is the root, the total number of steps in the translation process along with a variable keeping track of how many steps the user would like to see. Being able to stop the translation process at specific steps helps with verification that the translation algorithms are working as intended.

I checked the correctness of the translator by checking the semantics of the produced Kripke structures against the Little-JIL and against the Little-JIL translation rules presented in the paper *Verifying Properties of Process Definitions* by Cobleigh, Clake and Osterweil [12]. I have reproduced below the figures from that paper which explain how a leaf, sequential and parallel step should be translated in Little-JIL. These are the exact figures from that paper, presented exactly as they were there.

In Figure 11 you can see the Little-JIL translation of a leaf, sequential, and parallel nodes. A leaf node translates into four leaf status nodes: *leaf posted*, *leaf started*, *leaf completed* and *leaf terminated*. The translation template starts with *leaf posted*, which has one child, the *leaf started* node. The *leaf started* has two children, the *leaf completed* and *leaf terminated* nodes. A leaf node never has children, so there are no substeps to be considered like with sequential and parallel.

The sequential translation template in Figure 11 shows that a single sequential node with two or more substeps translates into four sequential status nodes with the substeps embedded in between the *sequential started* and *sequential completed*. The four sequential status steps are *sequential posted*, *sequential started*, *sequential completed* and *sequential terminated*. The *sequential posted* has one child, the *sequential started*. The child of the *sequential started* node is the first substep. Then the child of that substep is the next substep. If there are more than two substeps, the chain of substeps continues, with one substep having one child, the next substep. The final substep only has one child, *sequential completed*. All substeps besides the last always have two children, the next substep and also the *sequential terminated* node. The substeps being one after another and only the first substep connecting to *sequential started* is what creates the sequential control flow.

The parallel translation template in Figure 11 shows that a single parallel node translates into four parallel status nodes plus all the substep nodes. *Parallel posted* has one child, *parallel started*. *Parallel started* has each substep as a child. The substep children of parallel nodes are the most complicated part of Little-JIL translation. Each substep has two parallel substep children, *parallel completed* and *parallel terminated*. Each substep also has every other substep besides itself as a child. The parallel control flow in this translation template comes from the fact that every substep is a child of parallel started and a child of every other child. It is understood that the children execute at the same time or at least overlap in execution, assuming a multi-threaded model. In parallel execution using a single thread model the execution order might appear to be random and could switch back and forth between substeps before finishing the prior substep. If a substep is started and then control is given to another substep, control would eventually return to the unfinished step to be completed.

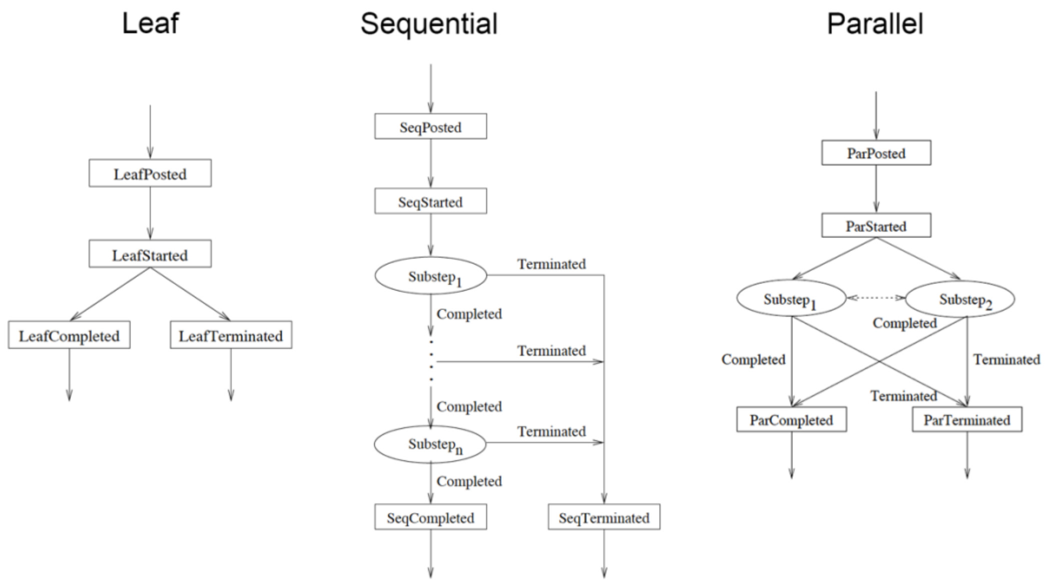


Figure 11. Little-JIL Translation Templates For Leaf, Sequential and Parallel

Below are the translation algorithms, in pseudocode, that I wrote to translate Little-JIL XML to Kripke structures with Little-JIL status nodes. In the case of parallel processes, interleavings have the option of being shown. First is the main translate function, the first function below in Figure 12. The *translate()* function takes in a array-like list of Little-JIL process nodes, already linked in a tree-like structure, to translate. It also takes in a Boolean flag which is true if the function should determine and include all the possible interleavings. There is a switch statement which checks the Little-JIL node type (*leaf*, *parallel*, etc.) of the root and swaps in the appropriate template. Then the function runs translateChildrenRecursively root. Since there are no loops in the *translate()* function, the time complexity of the *translate()* function alone is constant.

The next translation function in Figure 12 is *translateChildrenRecursively()*. *translateChildrenRecursively* takes in a single node and also the Boolean flag for whether or not to show the interleavings. First the function gets the children of the node argument. Then it loops through each child and checks if it's an original child (from the original node list to be translated) and if so, then it runs *swapInTemplate()* with the appropriate template function (*leaf*, *parallel*, etc.) as an argument to *swapInTemplate()*. After all the children have been translated it recursively calls itself again on all the current children.

In terms of algorithmic complexity for *translateChildrenRecursively()*, the first for loop is called *o* number of times, where *o* is the number of children the node had when the function is started. The second loop is called *p* times, which is the number of children the node had after the first for loop. The number of possible children is different depending on the Boolean showInterleavings. If we are not showing the explicit interleavings, we know that the number of children is less than $4(n – 1)$, where *n* is the number of nodes in the original list to be translated. $n – 1$ because the node itself being translated couldn't be in the children, but theoretically all the other nodes could be. And that value is multiplied by four in case all the children have been already translated and each been replaced by four or more template nodes. Since *o* and *p* both will be less than *4(n – 1)*, together both loops will be called less than *4(n – 1) + 4(n – 1)* times or *8(n – 1)* times, which is of linear algorithmic complexity. So if we are not showing interleavings, *translateChildrenRecursively()* is of linear algorithmic complexity.

If we are showing explicit interleavings, we can say that both *o* and *p* will be equal to or less than *(n – 1)! * (n – 1)*. *n – 1* in the first term because the node itself can't be its own child and factorial because if all children are children of a parallel node there would be *(n – 1)!* permutations, each with *(n – 1)* nodes. So if we are showing explicit interleavings, *translateChildrenRecursively()* is of factorial algorithmic complexity.

When we consider how many times *translateChildrenRecursively()* calls itself compared to *n* where *n* is the number of nodes in the initial node list to be translated, we see that the complexity is linear when we are not showing interleavings and factorial when we are showing interleavings. When we are not showing interleavings, each original node results in four nodes when translated: *posted*, *started*, *terminated* and *completed*. So, the total number of nodes in the final Kripke in this case will be *4n*. Since *translateChildrenRecursively()* is called once for each node in the Kripke, it will be called a linear number of times. Since the number of inner loop calls is linear and the number of times *translateChildrenRecursively()* is called is linear, in the case where we are not showing the interleavings, the overall complexity of *translateChildrenRecursively()* is linear.

In the case where we are showing explicit interleavings and the node is a parallel started node with multiple children, we know that *o* is exponentially greater than *n* because $o <= (n - 1)!$ as explained above. So we can say the overall complexity of *translateChildrenRecursively()* is factorial when we are showing explicit interleavings.

The last translation function in Figure 12 is *swapInTemplate()*. First, it loops through all the nodes in the template which was passed in as an argument. For each template node, it adds that node to the main translated node list. Then it removes the node which is being replaced by the template (such as a parallel node, in the case of a parallel template) and it removes any children which were replaced by interleavings (since all permutations are found and added, the originals must be removed or there will be some redundancy).

The loop in *swapInTemplate* executes once for each node in the template. This will be four nodes in the case of a leaf template or a sequential template. For a parallel without explicit interleavings it will be the four status nodes plus any original children. Since the original children could not be more than $n - 1$ (the amount of nodes in the original node list to translate, minus the parallel node itself), this in the worst case scenario would be $4 + (n - 1)$ which is $n + 3$. So, the complexity in that case is linear. In the case of a parallel with interleavings, the worst case would be $4 + (n - 1)!$ which is factorial complexity.

In determining the overall complexity of our translations, we see that *translate()* has linear complexity. Both *translateChildrenRecursively()* and *swapInTemplate()* have linear complexity without explicit interleavings and factorial complexity with explicit interleavings. So, we can say that the translation functions together have linear complexity without explicit interleavings and factorial complexity with explicit interleavings.

```
translate(nodeList, showInterleavings) {
   // swap in template for root node
   root = nodeList.getRoot
   switch (root.getNodeType) {
      case leaf: root = leafTemplate(root)
      case sequential: root = sequentialTemplate(root)
      case parallel: root = parallelTemplate(root, showInterleavings)
  }
   // run translateChildrenRecursively on root
   translateChildrenRecursively(root)
}

translateChildrenRecursively(node, showInterleavings) {
   // translate all the children
   for (child in node.getChildren) {
      if (child.isOriginal) {
        switch (root.getNodeType) {
          case leaf: swapInTemplate(leafTemplate(node))
          case sequential: swapInTemplate(sequentialTemplate(node))
          case parallel: swapInTemplate(parallelTemplate(node, showInterleavings))
        }
      }
   }

   // loop through all current children and run
   for (child in node.getChildren) {
     translateChildrenRecursively(child, showInterleavings)
   }
}

swapInTemplate(template) {
  for (node in template.nodeList) { nodeList.add(node) }    // add template nodes to nodeList
  nodeList.remove(template.nodeToReplace)                    // remove the node being replaced
  nodeList.remove(template.childrenReplacedByInterleavings) // remove children replaced by interleavings
}
```

Figure 12. Translate() Pseudocode

Below in Figure 13, 14 and 15 are simplified pseudocode versions of the template functions. In Figure 13 we have the *leafTemplate()*. First the *leafTemplate()* function creates all four leaf status nodes: *leafPosted*, *leafStarted*, *leafCompleted* and *leafTerminated*. Then it hooks up the nodes of the template: *leafStarted* is a child of *leafPosted* and *leafCompleted* and *leafTerminated* are children of *leafStarted*. Then it hooks up the parent nodes by looping through all the parent nodes and adding *leafPosted* as a child to each of them. Then it hooks up the children nodes by looping through all the children nodes and checking them for terminated or completed status. If they are terminated, it adds them as a child to *leafTerminated* and if they are completed, it adds them as a child to *leafCompleted*. Then it returns a new nodeList object with *leafPosted*, *leafStarted*, *leafCompleted*, and *leafTerminated* as arguments.

In terms of complexity for the *leafTemplate()*, the two loops in the leaf template are looping through the parents and looping through the children. The number of possible parents and children are a little hard to predict because these could be node created in other translation steps. We can say, though, in a Kripke without explicit interleavings, the number of parents and children will both be less than $4n$ each, where n is the number of nodes in the original node list to translate. In a Kripke with interleavings, the number of parents and children will both be less than $4n + (n - 1)!$ The $4n$ represents how each node can be replaced by a four node template and the $(n - 1)!$

represents the maximum number of possible interleaved nodes where *n* is all the nodes in the original template list and minus 1 is because we can remove the node being translated from the list of possible children that could be used in interleavings. So without explicit interleavings, the complexity of the leafTemplate() is linear and with explicit interleavings it is factorial.

```
leafTemplate(node) {

  // create template nodes
  leafPosted = new node()
  leafStarted = new node()
  leafCompleted = new node()
  leafTerminated = new node()

  // hook template nodes to each other
  leafPosted.addChild(leafStarted)
  leafStarted.addChild(leafCompleted)
  leafStarted.addChild(leafTerminated)

  // hook up parents
  parents = node.getParents
  for (parent in parents) {
    parent.addChild(leafPosted)
  }

  // hook up children
  children = node.getChildren
  for (child in children) {
    if (child.getStatus == terminated) {
      leafTerminated.addChild(child)
    } else if (child.getStatus == completed) {
    leafCompleted.addChild(child)
```

```
      }
    }

    return new nodeList(leafPosted,leafStarted,leafCompleted,leafTerminated)
}
```

Figure 13. LeafTemplate() Pseudocode

In Figure 14 below, we have the *sequentialTemplate()* function. One of the first things it does is create the four sequential status template nodes: *seqPosted*, *seqStarted*, *seqCompleted* and *seqTerminated*. Then it hooks up the template nodes to each other: *seqStarted* is added as a child to *seqPosted* and the first substep node is added as a child to *seqStarted*. Then the parents are hooked up by looping through the parents and adding *seqPosted* as a child to each. Next the substeps are hooked up.

Hooking up the substeps is the heart of the sequential template algorithm. A for statement loops through all the substeps, using *i* as the current index. If *i* is not equal to the index of the last substep, the next substep is added as a child to the current substep. If *i* is equal to the index of the last substep *seqCompleted* is added as a child to the current substep. Regardless of whether or not *i* equals the index of the last substep, *seqTerminated* is added as a child of the current substep.

The substep template concludes by hooking up the children nodes and then returning new *nodeList* template. To hook up the children nodes, it loops through all the children and checks the status of each one. If the status is *terminated*, the child is added as a child to *seqTerminated*. Otherwise, the child is added as a child to *leafCompleted*. Then all four template nodes (*seqPosted*, *seqStarted*, *seqCompleted*, and *seqTerminated*) are passed in as arguments to a new *nodeList* objects which is returned by the *sequentialTemplate()* function.

In terms of algorithmic complexity for the *sequentialTemplate()* function, there are three loops: hooking up the parents, the substeps and the children. Just as for the loops hooking up the parents and children in *leafTemplate()*, the complexity of the loops is linear without explicit interleavings and factorial with explicit interleavings. This reasoning is explained above in the *leafTemplate()* section. This same reasoning applies to the hooking up substeps loop as well, so we can conclude that sequentialTemplate()'s complexity is linear without explicit interleavings and factorial with interleavings.

```
sequentialTemplate(node) {
   // get substeps
   substeps = node.getSubsteps

   // create template nodes
   seqPosted = new node()
   seqStarted = new node()
   seqCompleted = new node()
   seqTerminated = new node()

   seqPosted.addChild(seqStarted)  // hook up template nodes to each other
   seqStarted.addChild(substeps[0]) // hook up seqStarted to first substep

   // hook up parents
   parents = node.getParents
   for (parent in parents) {
      parent.addChild(seqPosted)
   }

   // hook up substeps
   for (i=0; i<substeps.length; i++) {
      substep = substeps[i]
      lastIndex = substeps.length - 1
      // if not last substep, hook up to next substep
      if (i != lastIndex) {
         substep[i].addChild(substep[i+1])
         // if last substep, hook up to seqCompleted
      } else {
         substep.addChild(seqCompleted)
      }
      // all substeps get hooked up to seqTerminated
      substep.addChild(seqTerminated)
   }

   // hook up children
   children = node.getChildren
   for (child in children) {
      // hook up seqTerminated to any children of status terminated
      if (child.getStatus == terminated) {
         seqTerminated.addChild(child)
         // hook up all other children to seqCompleted
      } else if {
         leafCompleted.addChild(child)
      }
   }
   return new nodeList(seqPosted,seqStarted,seqCompleted,seqTerminated)
}
```

Figure 14. SequentialTemplate() Pseudocode

In Figure 15 below, we have the *parallelTemplate()*, which is more complicated than the previous two templates. First the function creates the four parallel status nodes: *parPosted*, *parStarted*, *parCompleted* and *parTerminated*. Then it hooks up the template nodes to each other by adding *parStarted* as a child to *parPosted*. Then it loops through the parents, adding *seqPosted* as a child to each parent. Next, the function uses an if statement with a check for the showInterleavings Boolean to provide functionality both showing interleavings and not showing interleavings.

If showInterleavings is false, the *parTemplate()* hooks up the substeps, then hooks up the children and then returns a new template, *newList*. To hook up the substeps, it loops through all the substeps twice, using *i* as the index for the outer for loop and *j* as the index for the inner loop. This is to hook up every substep to every other substep. If *i* does not equal *j*, then *substep[j]* is added as a child to *substep[i]*. Then the inner for loop closes. The last thing the outer for loop does is hook *parCompleted* and *parTerminated* up to every substep.

To hook up the children, in the case where showInterleavings is false, the function loops through all the children and checks their status. If the status is terminated, the child is added as a child to *parTerminated*. Otherwise the child is added as a child to *parCompleted*. Finally, the *parTemplate* without interleavings returns a new *nodeList* object, with the following arguments: *parPosted*, *parStarted*, *parCompleted*, *parTerminated* and a new *List* object containing an argument of the original children.

In the case that *parTemplate()* has showInterleavings as true, the function gets the explicit interleavings, calling the *getInterleavings()* recursive function, shown in Figure 16 and described in that section. Then each interleaving is looped through. For each interleaving, each node of the interleaving is also looped through. Every node first has all its children removed. If it is the first node in the interleaving, it is added as a child to *parStarted*. If it is not the last node in the interleaving, then the next node in the interleaving is added as a child to the current node. Then to hook up the children, all the children which are not original children (that is, all children created by the translation functions or all the children with statuses) are looped through and have their status checked. If the child status is *terminated*, is added as a child to *parTerminated* and otherwise, it is added as a child to *parCompleted*. Finally, the *parallelTemplate()* function with interleavings returns a new nodeList object, with the following arguments passed in: *parPosted*, *parStarted*, *parCompleted*, *parTeminated* and the interleavings node list.

In terms of algorithmic complexity, the *parallelTemplate()* function has one loop before the *showInterleavings* control splits into two paths. That initial loop cycles through all the parent nodes. This loop's complexity is linear without explicit interleavings and factorial with explicit interleavings, with the same reasoning as the parent loop in the *leafTeamplate()* and *sequentialTemplate()*.

The *parallelTemplate()* section where *showInterleavings* is false has two loops. The first loops through the substeps and the second loops through the children. These are similar to the substep and children loops in the *sequentialTemplate()*, with the one exception that the substep loop here is actually two nested for loops. The number of potential substeps without interleavings is $4n - 1$ where n is the number of nodes in the initial node list to be translated. The $4n$ is because each node in the *nodeList* could potentially be translated to four status nodes and the minus 1 is because the initial node will not be a substep node. Since these worst case of $4n - 1$ number of substeps is being looped through twice though in a nested fashion, it could potentially be executed $(4n - 1)^2$ times. This is an n^2 level of complexity or polynomial complexity.

The section of the *parallelTemplate()* with *showInterleavings* as true has two main loops. The first loop cycles through all the interleavings, but then also loops through all the nodes in each interleaving. We know that just the number of interleavings is already of an exponential complexity. There could potentially be *(n – 1)!* interleavings. The number of nodes would also be a factorial quantity, but there is no need to calculate it since we already know this loop has factorial complexity. The second loop cycles through the children, which we know from the *leafTemplate()* and *sequentialTemplate()* is linear without explicit interleavings and factorial with explicit interleavings. So, to summarize the complexity findings of the parallel template, without explicit interleavings it is of polynomial complexity and without explicit interleavings, the *parallelTemplate()* has factorial complexity.

```
parallelTemplate(node, showInterleavings) {

  // create template nodes
  parPosted = new node()
  parStarted = new node()
  parCompleted = new node()
  parTerminated = new node()

  // hook up template nodes
  parPosted.addChild(parStarted)

  // hook up parents
  parents = node.getParents
  for (parent in parents) {
    parent.addChild(seqPosted)
  }
```

```
// get getChildren
children = node.getChildren
childrenOriginal = node.getChildrenOriginal
childrenNotOriginal = node.getChildrenNotOriginal

// if not showing interleavings
if (!showInterleavings) {

    // hook up substeps
    substeps = node.getSubsteps
    for (i=0; i<substeps.length; i++) {
        substep = substeps[i]
        for (j=0; j<substeps.length; j++) {
            if (i != j) {
                substep.addChild(substep[j])
            }
        }
        substep.addChild(parCompleted)
        substep.addChild(parTerminated)
    }

    // hook up children
    for (child in children) {
        // hook up parTerminated to any children of status terminated
        if (child.getStatus == terminated) {
            parTerminated.addChild(child)
            // hook up all other children to leafCompleted
        } else if {
             parCompleted.addChild(child)
        }
    }

    return new nodeList(parPosted, parStarted, parCompleted, parTerminated, new List(origChildren))

    // else if showing interleavings
```

```
  } else {
    // get interleavings & store in global interleavings variable
    l=0               // index of first element to permute
    r=child.length - 1  // index of last element to permute
    getInterleavings(children, l, r) // stores permutations in interleavings global variable

    // clean up the interleavings
    for (interleaving in interleavings) {
      for (i=0; i < interleaving.length; i++) {
        thisNode = interleaving[i]
        thisNode.removeChildren()              // remove original children from each node
        if (i == 0) {
          parStarted.addChild(thisNode)          // hook up first node in each interleaving to parStarted
        }
        if (i < interleavings.length - 1) {
          nextNode = interleaving[i + 1]
          thisNode.addChild(nextNode)            // link interleaving nodes in the order of each permutation
        }
      }
    }

    // hook up children
    for (childNotOrig in childrenNotOriginal) {
      status = childNotOrig.getStatus
      if (status == TERMINATED) {
        parTerminated.addChild(childNotOrig)
      } else {
        parCompleted.addChild(childNotOrig)
      }
    }
    return new nodeList(parPosted, parStarted, parCompleted, parTerminated, interleavings)
  }
}
```

Figure 15. ParallelTemplate() Pseudocode

Lastly in our explanation of the translation functions, we have the *getInterleavings()* function below in Figure 16. The intent of this function is to find all the possible interleavings. This means finding all the possible permutations of the children in question. Permutation calculation is a common computer science exercise. I have used a recursive solution for this from the *geeksforgeeks.org* programming tutorial website [21].

The get interleavings recursive function takes in an array-like list, a left index and a right index as arguments. The base case occurs when the left and right index are the same. In that case, the current value of the list is added to a global *interleavings* array variable. The recursive case occurs when the left and right index are not the same. In that case, it runs a for loop with a start condition of left set equal to right, an end condition of index *i* being less than or equal to right and an increment of *i++*. Inside the for loop the *l* and *i* elements of the list are swapped, then *getInterleavings()* is called again, with parameters of list, *l+1* and *r*. Then the list elements l and i are swapped back before the end of the inside of the for loop. In effect this loop initially swaps elements on the left of the list and then calls getInterleavings using the new list and moves left one element to the right. This ends up cycling through all possible permutations as the recursive calls move left to the right and as the iterations of the initial *l=0* case moves *i* to the right as well. In terms of algorithmic complexity, the *getInterleavings()* function will be recursively called more than n! where n is the number of children being interleaved. So *getInterleavings()* is certainly of factorial algorithmic complexity.

```
// gets all permutations of the list parameter

// initially l is 0 and r is index of last list element
// this permutation strategy from https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/
getInterleavings(list, l, r) {
  // base case
  if (l == r) {
    interleavings.add(list) // unique permutation found and stored in interleavings global variable
    // recursive case
  } else {
    for (int i=l; i<=r; i++) {
      swap(list[l], list[i]);          // swaps l & i elements
      getInterleavings(list, l+1, r);    // l increases by one with each recursion until base case is met
      swap(list[l], list[i]);          // swaps l & i elements back
    }
  }
}
```

Figure 16. GetInterleavings() Pseudocode

Since the translation algorithms are my main contribution, it is important to be sure they are accurate and are translating correctly. To ensure this, I begin testing small translations visually against the Little-JIL translation rule diagrams and then progressively move to testing larger diagrams. The Little-JIL translation rule diagram is in Figure 11.

We start testing with perhaps the simplest diagram possible in Figure 17, a sequential node with two leaves. We'll check it against the Little-JIL rules in Figure 11 and see if it's correct. On the left of Figure 17, you see a sequential node with two leaves, which is our starting XML graph. From Figure 11, we would expect the *sequential* to translate to a *sequential posted*, a *sequential started*, then some substeps and ending with *sequential completed* and *sequential terminated*. That is the sequential template. The sequential template begins with the *sequential started* being a child of *sequential posted*. The first substep in the template in Figure 11 is a child of *sequential started*. The second substep in the template is a child of the first substep. The *sequential completed* comes next in the template and is a child of the second substep. *Sequential terminated* is child of both substeps in the template. All of what I just described about the sequential template diagram in Figure 11 is exactly what we see on the right of Figure 17, except that the leaves are also translated. So next we compare the translated leaves in Figure 17 with the leaf template in Figure 11.

Given the leaf template in Figure 11, we see that a translated leaf node has a *leaf posted*, *leaf started*, *leaf completed* and *leaf terminated*. The leaf template begins with *leaf started*, which is a child of *leaf posted*. Next in the template in Figure 11 is a *leaf completed* and *leaf terminated*, which are children of *leaf started*. Since with *leaf 1* begins with *leaf 1 posted*, *leaf 1 posted* should be a child of *sequential started*. This matches *substep 1* in the template diagram. Then *leaf 1* started is a child of *leaf 1 posted*, as shown in the template. *Leaf 1 completed* and *leaf 1 terminated* are next in the template, as children of *leaf 1 started*. For *leaf 2*, since it begins with *leaf 2 posted, leaf 2 posted* should be a child of *leaf 1 completed*. This matches *substep 2* in Figure 11 and also matches that *leaf 1* is the equivalent of *substep 1* in the sequential diagram in Figure 11 and *leaf 2* is the equivalent of *substep n*. Matching the template diagram, we can expect *leaf 2 started* to be a child of *leaf 1 posted*. Next are *leaf 2 completed* and *terminated* and they are children of *leaf 2 started*, as in Figure 11. Finally, we see in the template that *sequential completed* should be a child of *leaf 2 completed*. Also, *sequential terminated* is a child of both *leaf 1 terminated* and *leaf 2 terminated* in the template. The above descriptions of what we would expect to see with translated leaf nodes *1* and *2*, compared to the leaf template translation rule in Figure 11 is exactly what we see in Figure 17. We can thus conclude the translator is translating a sequential with two leaves correctly.

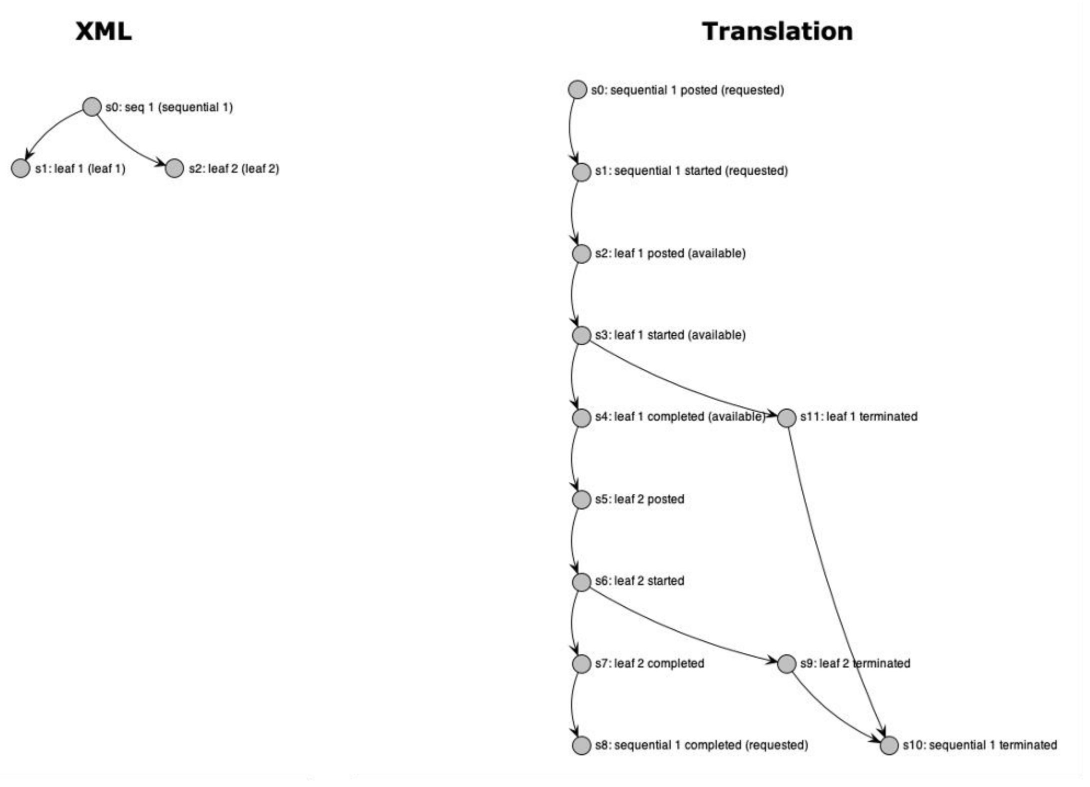

XML
s0: seq 1 (sequential 1)
s1: leaf 1 (leaf 1)
s2: leaf 2 (leaf 2)
Translation
s0: sequential 1 posted (requested)
s1: sequential 1 started (requested)
s2: leaf 1 posted (available)
s3: leaf 1 started (available)
s4: leaf 1 completed (available)
s11: leaf 1 terminated
s5: leaf 2 posted
s6: leaf 2 started
s7: leaf 2 completed
s9: leaf 2 terminated
s8: sequential 1 completed (requested)
s10: sequential 1 terminated

Figure 17. Sequential With Two Leaves XML & Translation

The sequential with three leaves, which we have in Figure 18, is similar to a sequential with two leaves in Figure 17. The three leaf version should translate exactly like the sequential with two leaves, but will have one more leaf on the far right, as the last child of *sequential 1*. Going from the leaf template rule in Figure 11, we know that a translated leaf has *leaf started* as a child of *leaf posted* and then *leaf completed* and *leaf terminated* as a child of *leaf started*. This is what we see in Figure 18 for the *leaf 3* structure. Then to see if the other connections are correct, we know that the outgoing node of the previous leaf is *leaf 2 completed*, so it should have *leaf 3 posted* as a child, which it does. We know that the outgoing node of *leaf 3* is *leaf 3 completed,* so it should have *sequential 1 completed* as its child, like in the sequential template in Figure 11. We know that the terminal node of *leaf 3* is *leaf terminated*, so this should have a child of *sequential 1 terminated*. This is exactly how the translator translated the sequential with three leaves, as seen in Figure 18, so we can conclude that the translator correctly translates a sequential with three leaves.

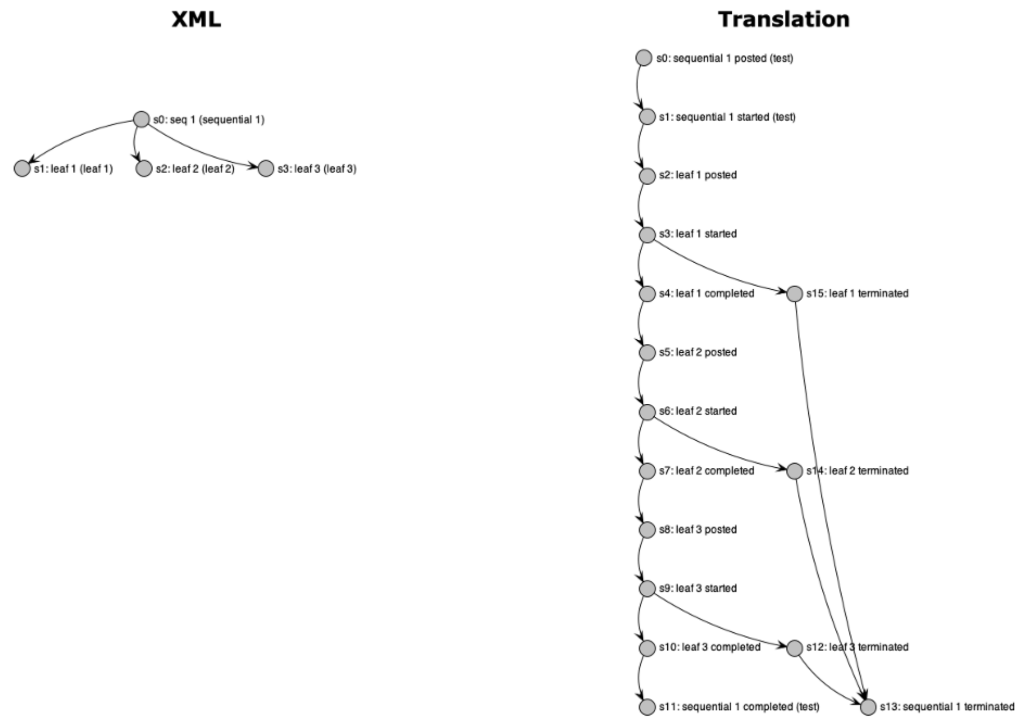


Figure 18. Sequential And Three Leaves XML & Translation

Continuing in our pattern of testing more and more complicated XML diagrams, Figure 19 is a nested sequential: a sequential node with two children, a leaf and another sequential. The second sequential has two leaf children. Comparing to the sequential template in Figure 11, we know that the translation will begin and end with *sequential 1*. The first node should be *sequential 1 posted*, followed by *sequential 1 started*. The last nodes (the two terminals) in the overall structure should be *sequential 1 completed* and *sequential 1 terminated*. Looking at the sequential template, *sequential 1's* substeps should follow after *sequential 1 started*. The two substeps of *sequential 1* in Figure 19 are *leaf 1* and *sequential 2*. Since *leaf 1* is on the left, it is equivalent to *substep 1* in the sequential template in Figure 11. Since *sequential 2* is on the right, it is the equivalent to *substep 2* in the sequential template. Since *leaf 1* is *substep 1* and *sequential 2* is *substep 2,* in order to match the template, after *sequential 1 started*, we would need *leaf 1* followed by *sequential 2*.

Leaf 1 in Figure 19 should translate like the leaf template in Figure 11: *leaf posted,* followed by *leaf started*, followed by both *leaf completed* and *leaf terminated*. Since *leaf posted* is the entry node for the leaf template, and *leaf 1* is *substep 1* of *sequential 1*, *leaf posted* should be a child of *sequential 1 started*. Since *leaf 1 completed* is the successful exit node of the leaf template, it will connect to *substep 2*, or *sequential 2*. Since *leaf 1 terminated* is the failure exit node of the leaf template, it will connect to *sequential 1 terminated*, like in the sequential template in Figure 11.

Sequential 2 in Figure 19 should match the sequential template in Figure 11 when translated. It should start with *sequential 2 posted*, followed by *sequential 2 started,* followed by *substep 1* (*leaf 2* in this case), followed by *substep 2* (*leaf 3*), followed by *sequential 2 completed*. Since *leaf 2* and *leaf 3* are the substeps of *sequential 2*, both should connect to *sequential 2 terminated*, like in Figure 11. Since *sequential 2 posted* is the template's entry node, and *sequential 2* is *sequential 1's substep 2*, *sequential posted* should be a child of *substep 1*, or in this case, *leaf 1 completed*. This is how it would match the sequential template in Figure 11. Since *sequential 2* is the last substep of *sequential 1* and *sequential 2's* successfully exit node is *sequential 2 completed*, to match the template, *sequential 2 completed* should connect to *sequential 1 completed*. To match the template, *sequential 2 terminated* should also be a child of both substeps' terminated nodes, so in this case *sequential 2 terminated* should be a child of *leaf 2 terminated* and *leaf 3 terminated*.

The only part of the translation left to explain in Figure 19 is how and where *leaf 2* and *leaf 3* will translate. We have seen how leaf substeps translate above with *leaf 1* and *leaf 2* and *leaf 3* should follow the same structure. Both leaves would match the leaf template and each have a *leaf posted*, *leaf started*, followed by both *leaf completed* and *leaf terminated*. Since the *leaf posted* nodes are the leaf template's entry node, *sequential 2 started* should have *leaf 2 posted* as a child and *leaf 2 completed* should have *leaf 3 posted* as a child (since *leaf 2* completed is the leaf template's successful exit node). Both *leaf 2 terminated* and *leaf 3 terminated* should connect to *sequential 2 terminated* to match the sequential template in Figure 11.

All the above descriptions of how the nested sequential on the left of Figure 19 should translate to match the templates in Figure 11 are exactly how the translated graph on the right of Figure 19 do in fact look. So, we can conclude that the translator is translating nested sequential structures correctly. The next graph we will check for correctness in Figure 20 is the same nested sequential in Figure 19, but with *sequential 1's* children's order reversed.

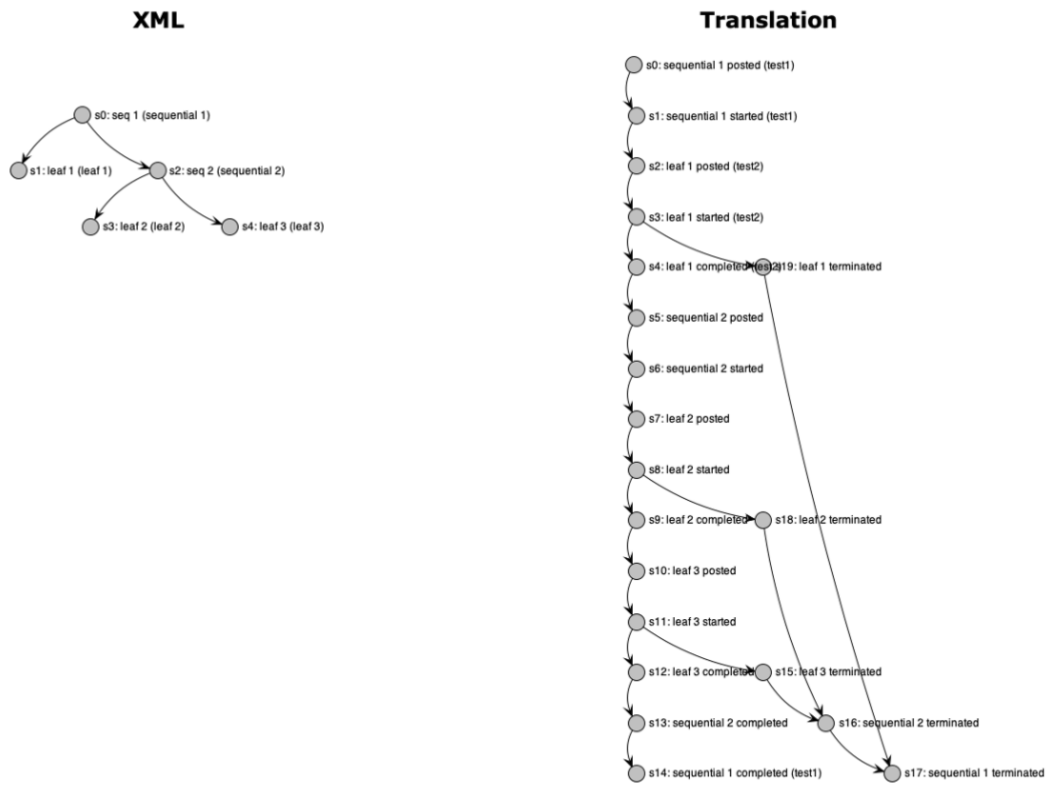


Figure 19. Nested Sequential XML & Translation

Another test of the correctness of the nested sequential nodes would be to reverse the order of *sequential 1's* children from Figure 19. So, in Figure 20 we also have a nested sequential but *sequential 2* is the first child of *sequential 1*. Here we would expect *sequential 1* started to be a child of *sequential 1 posted*, to match the sequential template in Figure 11. Then *sequential 1 posted* should be a child of *sequential 1 started*, as in the template. If the template structure continues, we would expect *sequential 1* started to be a child of *sequential 1 posted*. Then we would expect *leaf 1 posted* to be a child of *sequential 2 started*. Then to match the leaf template in Figure 11, *leaf 1* started should be a child of *leaf 1 posted* and *leaf 1 completed* and *terminated* should be children of *leaf 1 started*. Then we would expect *leaf 2 posted* to be a child of *leaf 1 completed*, since *leaf completed* is the leaf template's exit successful exit node. Then also following Figure 11's leaf template, *leaf 2 started* should be a child of *leaf 2 posted* and *leaf 2 completed* and *terminated* should be children of *leaf 2 started*. Then we would expect *leaf 3 posted* to be a child of *sequential 2 completed*. Then, if the leaf template holds again for *leaf 3*, *leaf 3 started* should be a child of *leaf 3 posted* and *leaf 3 completed* and *terminated* should be children of *leaf 3 started*. Then *sequential 1* completed should be a child of *leaf 3 completed*, since *leaf 3* is *substep 2* of *sequential 1*. We would also expect *sequential 2 terminated* to be a child of both *leaf 1 terminated* and *leaf 2 terminated*, to match the template. We would expect *sequential 1 terminated* to be a child of *sequential 2 terminated* and *leaf 3 terminated*. This is all exactly what we see in Figure 20, so we can conclude that the translator can translate a nested sequential correctly. This is all exactly what we see in Figure 20, so we can continue to maintain that the translator is translating nested sequential nodes correctly.

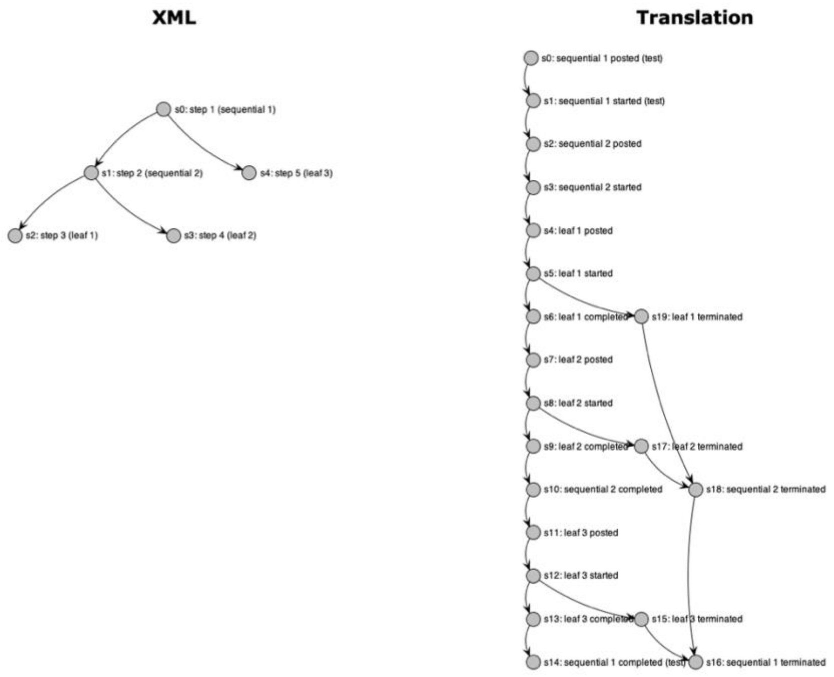


Figure 20. Another Nested Sequential XML & Translation

Continuing to test more complex diagrams against the translation rules in Figure 11, we now begin to test parallel nodes. First, we check the parallel with two leaves in Figure 21 and 22. In Figure 21 we have the XML and translation without explicit interleavings. In Figure 22 we have the translation with explicit interleavings. Then in Figure 23 and 24 we have a parallel with three leaves. Then in Figure 25 and 26 we have a parallel with four leaves. We conclude our correctness of translation of simple parallel nodes with Figure 27 and 28, of a parallel with five leaves. We finish our parallel testing with a translation of two parallel nodes, each with two leaves.

In Figure 21, we have the XML and translation without interleavings of a parallel node with two leaf children. This is exactly the same arrangement as the parallel template in Figure 11, but here the leaf substeps are also translated. So matching Figure 11, we would expect *parallel 1 posted* to be the first node and for *parallel 1 started* to be a child of *parallel 1 posted*. Since *leaf 1* in Figure 21 corresponds to *substep 1* in the parallel template, we would expect *leaf 1* to be a child of *parallel started* and then since *leaf 2* corresponds to *substep 2,* we expect *leaf 2* to be a child of *leaf 1*. When *leaf 1* and *leaf 2* are translated, we check if they match against the leaf template in Figure 11. So, we would expect *leaf 1 posted* and *leaf 2 posted* to be children of *parallel started*. We would expect *leaf 1 started* to be a child of *leaf 1 posted* and *leaf 2 started* to be a child of *leaf 2 posted*, if the template holds. We could expect *leaf 1 completed* and *leaf 1 terminated* to be children of *leaf 1* started since that's how they are arranged in the leaf template. We would expect *leaf 2 completed* and *leaf 2 terminated* to be children of *leaf 2 started*, like in the template. We would also expect *leaf 1 posted* to be a child of *leaf 2 completed* and for *leaf 2 posted* to be a child of *leaf 1 completed*, since *leaf 1* and *leaf 2* are both children of each other in the template and since *completed* is the exit node and *posted* is the entrance node. We would expect *parallel complete* and *parallel terminated* to be children of both *leaf 1 complete* and *leaf 2 complete*, like in Figure 11.

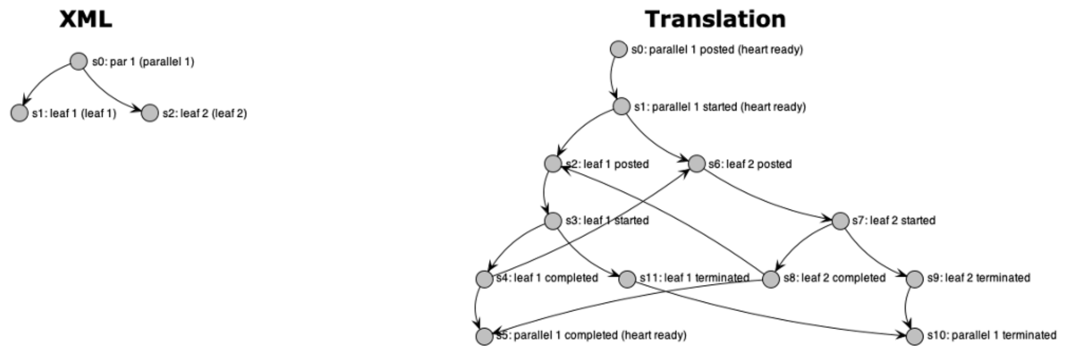


Figure 21. Parallel With Two Leaves XML & Translation

For the explicit interleavings diagram in Figure 22, we would expect this diagram to look the same as Figure 21 going down from the top until *leaf 1* and *leaf 2* come to their complete and terminate nodes. Then because we are showing the explicit interleavings, we expect to see all the permutations of *leaf 1* and *leaf 2*. There should be *2!* permutations since there are two nodes to permute and the equation for permutation is *x!* where *x* is the number of items to permute. Since *2!* equals two, we expect to see two interleavings. Spelling these out specifically, the first permutation is *leaf 1* followed by *leaf 2* and the second is *leaf 2* followed by *leaf 1*. According to the leaf template in Figure 11, all these leaves will each have *leaf posted*, followed by *leaf started*, followed by both *leaf completed* and *leaf terminated*. We would then expect, to match the parallel template, that *leaf 2 posted* will be a child of *leaf 1 completed* and *leaf 1 posted* will be a child of *leaf 2 completed*. We would expect *parallel terminated* to be a child of the terminated node of both interleavings, to match Figure 11. We would also expect *parallel completed* to be a child of the completed nodes of both interleavings. This is all exactly what we see in Figures 21 and 22, so we can conclude that the translation of parallel with two leaves is correct.

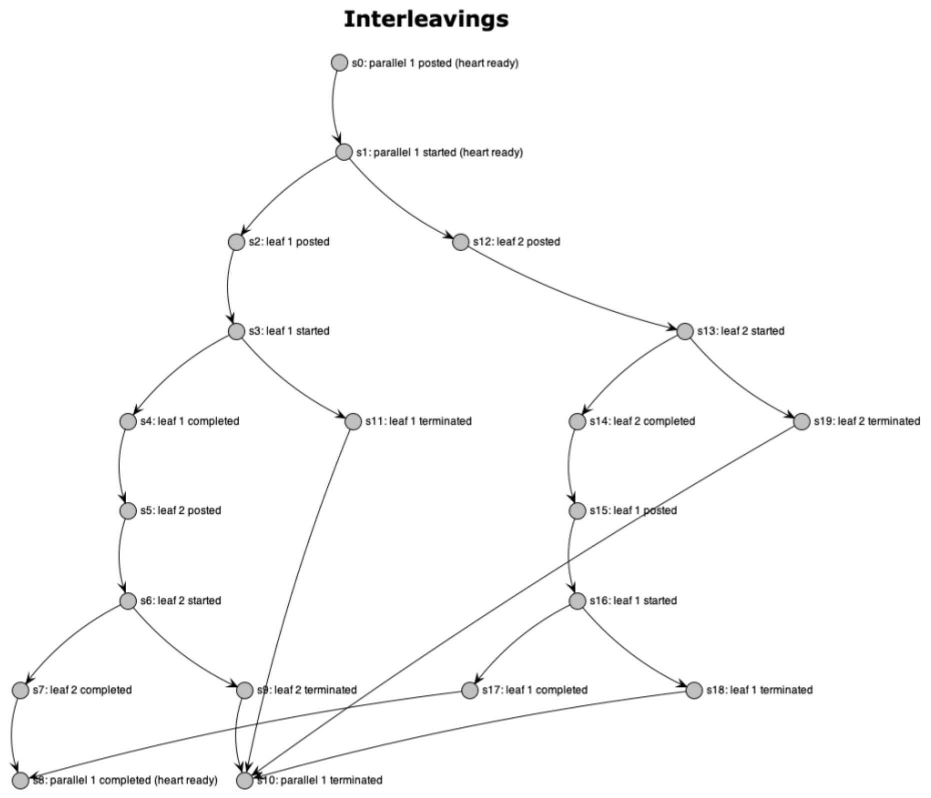


Figure 22. Parallel With Two Leaves (Explicit Interleavings)

Now we test a parallel with three leaf node children in Figures 23 and 24. Following the translation rules in Figure 11, we'd expect the right side of Figure 23 to be exactly like the right side of Figure 21, but with *leaf 3* added in. We would expect *leaf 3 posted* to be a child of *parallel started*, since like *leaf 1* and *2*, *leaf posted* is the leaf template entry node. We'd expect *leaf 3 started* to be a child of *leaf 3 posted* and *leaf 3 completed* and *leaf 3 terminated* to be children of *leaf 3 started*, according to the leaf template. We also expect *parallel completed* to also be a child of *leaf 3 completed* and *parallel terminated* to also be a child of *leaf 3 terminated*, as in the parallel template in Figure 11.

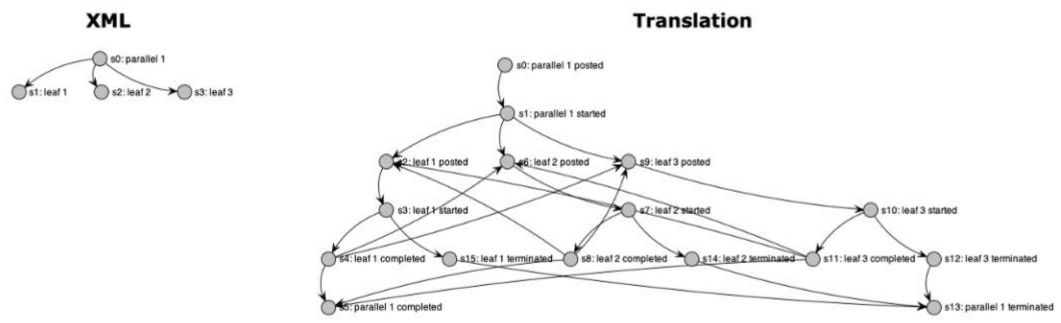


Figure 23. Parallel With Three Leaf Nodes XML & Translation (No Explicit Interleavings)

For Figure 24, since there are three children to the parallel node in the XML, we would expect there to be *3!* or six interleavings in the translation with explicit interleavings. This is what we see in Figure 24. We would expect this diagram to be quite similar to Figure 22, but with four more interleavings. Each of the interleavings should have three translated leaf nodes, each with the structure of *posted*, then *started*, then *completed* and *terminated*, according to the leaf template in Figure 11. Each final leaves' *leaf completed* in the interleaving should connect to *parallel 1 completed*, since in Figure 11, each substep connects to *parallel completed* and *leaf completed* is the template's success exit point. Each leaf's started node should connect to *parallel 1 terminated*, since *leaf started* is the failure exit point. The six permutations should cover the six possible orderings of three items: *123, 132, 213, 231, 321, 312*. This is all exactly what we see in Figure 24, so we can conclude that the translator is translating parallel nodes with three leaf children correctly.

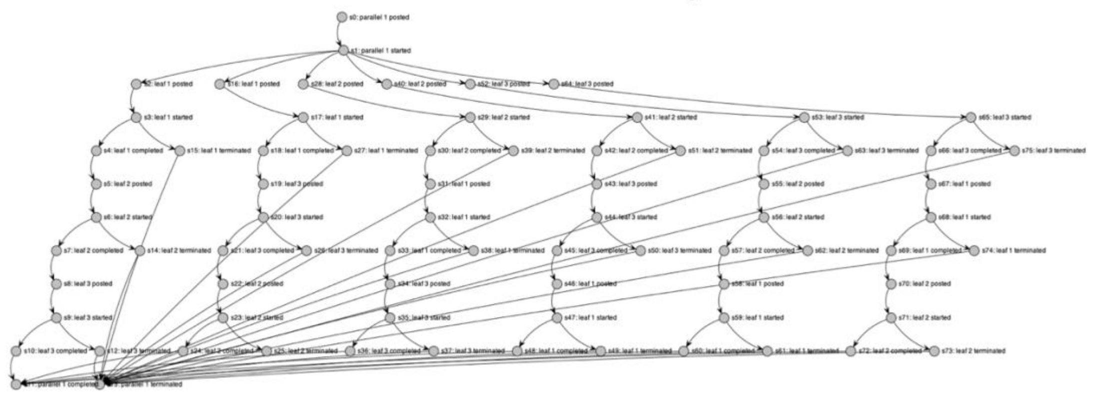


Figure 24. Parallel With Three Leaf Nodes (Explicit Interleavings)

We now test a parallel node with four leaf nodes. In Figure 25 we have a parallel with four leaf children. We would expect the right side of this diagram to be exactly like Figure 23, with the addition of the fourth leaf. Adding the fourth leaf closely follows the pattern that adding the third leaf in Figure 23 did. We would expect this new leaf to follow the same rules as the other three. When translated, the four new leaf nodes are *leaf posted*, *leaf started*, *leaf completed* and *leaf terminated*. Like *leaves 1-3*, *leaf 4 posted* should be a child of *parallel 1 started*. Like the other three, *leaf 4 completed* has a child of *parallel 1* completed as well as *leaf 1 posted*, *leaf 2 posted* and *leaf 3 posted*. Similarly, *leaf 4* terminated has a child of *parallel 1 terminated*. This all matches the parallel template in Figure 11.

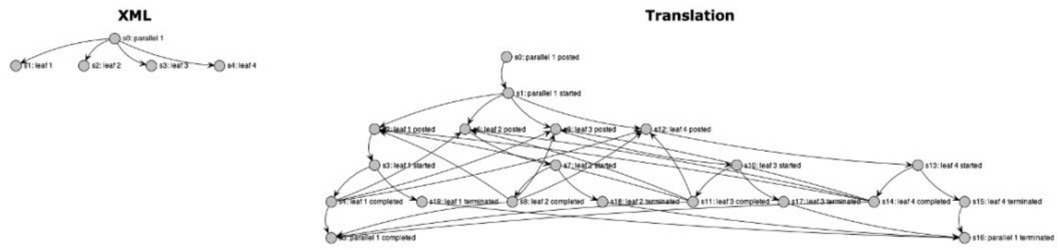


Figure 25. Parallel With Four Leaf Nodes XML & Translation (No Explicit Interleavings)

In my testing, this is the point where the graphs become so large, they begin to get more and more difficult to read. On the computer, you can zoom in and zoom out, which helps a lot if you are patient and can pan around a lot. For the parallel with four leaves with interleavings, we would expect there to be *4!* or 24 interleavings. You can see in Figure 26 the 24 leaf posted nodes in the very top row with more than one node. From this and from considering the previous parallel translations which were accurate, it seems likely the parallel with four leaves has been translated correctly.

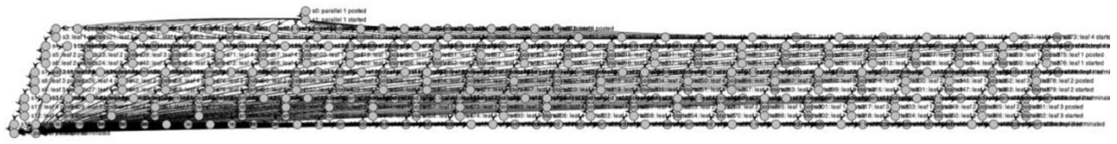


Figure 26. Parallel With Four Leaf Nodes XML & Translation (Explicit Interleavings)

In Figure 27 we have a parallel node with five leaf nodes. Following the pattern above, we can see the fifth leaf added at the far right. It is following the same rules that the other four leaf nodes are. In Figure 28, we have the translation of the parallel with five leaf children with interleavings. This graph is so large (about 2400 nodes), it is almost impossible to read here in the paper. But we know that with five children, the parallel node should have 300 interleavings. Although difficult to say with certainty, it seems likely that there are 300 interleavings in the graph. Given the correctness of the above parallel graphs, it seems likely that this graph is also correct.

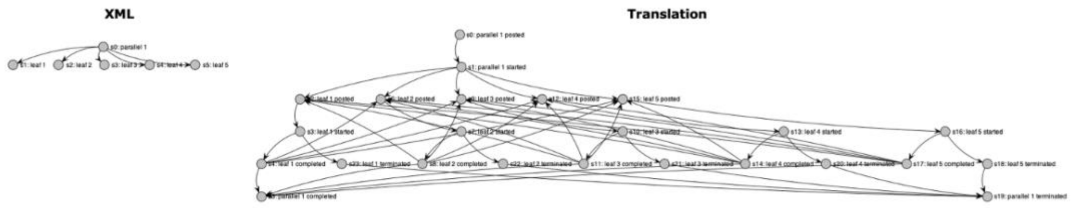


Figure 27. Parallel With Five Leaf Nodes XML & Translation (No Explicit Interleavings)

Figure 28. Parallel With Five Leaf Nodes XML & Translation (With Explicit Interleavings)

The last translation we examine for correctness is a dual parallel structure. The XML graph can be seen in Figure 29. *Parallel 1* has two children, *parallel 2* and *parallel 3*. Both *parallel 2* and *parallel 3* have two leaf children. *Parallel 2* has *leaf 1* and *leaf 2* as children and *parallel 3* has *leaf 3* and *leaf 4* as children. Looking at the parallel template in Figure 11, we expect to see the parallel pattern in Figure 29 three times, once for each parallel node. To match the template, the translation would start with *parallel 1 posted*, which would have *parallel 1 started* as a child. Since *parallel 1* has two substeps, we expect to see both substeps as children of *parallel started*, like in Figure 11. We do see *parallel 2* and *parallel 3* as children of *parallel 1*. For a moment, we can skip over *parallel 2 and 3 posted's* children. To complete the initial parallel pattern in Figure 11, we know that *parallel 2 and 3* will connect to *parallel 1* completed and *parallel 1 terminated*. We see towards the bottom of Figure 29 that *parallel 2 completed* and *parallel 3 completed* both connect to *parallel 1 completed*, which is what we would expect. Similarly, *parallel 2 terminated* and *parallel 3 terminated* both have *parallel 1 terminated* as a child, which matches the parallel template in Figure 11.

Now we can go back and focus on *parallel 2 and 3 posted's* children. Here we expect *parallel 2* and *parallel 3* to each match the parallel template in Figure 11. *Parallel 2 posted* has *parallel 2 started* as a child, just as *parallel 3 posted* has *parallel 3 started* as a child. Referring back to Figure 11, *leaf 1* and *2* correspond to *substep 1* and *2*, just as *leaf 3* and *4* also correspond to *substep 1* and *2*. So if it will match the parallel template, *substeps 1* and *2* will have each other as children, as will *substep 3* and *4* have each other as children. Then if it matches the template, *leaf 1* and *2* will both connect to *parallel 2 completed* and *parallel 2 terminated*, just as *leaf 3* and *4* will both connect to *parallel 3 completed* and *parallel 3 terminated*. We know from Figure 11's leaf template that each leaf when translated will be a *leaf posted*, which connects to a *leaf started*, which then connects to both a *leaf completed* and a *leaf terminated*. All four leaves do in fact have this structure. Since *leaf posted* is the entry node to the leaf, *parallel 2 started* has *leaf 1 posted* and *leaf 2 posted* as children. The same with *parallel 3* having *leaf 3* and *leaf 4* as children. Since *leaf completed* is the success exit node of the leaf, *leaf 1 completed* and *leaf 2 completed* both have *parallel 2 completed* as a child to match the parallel template where both substeps connect to *parallel completed*. The same with *leaf 3* and *4*. Both *leaf 3 completed* and *leaf 4 completed* have *parallel 3 completed* as a child. Also matching the parallel template, we see that *leaf 1 terminated* and *leaf 2 terminated* both connect to *parallel 2 terminated*. In the same way, *leaf 3 terminated* and *leaf 4 terminated* both connect to *parallel 3 terminated*. Thus, we see all three parallel patterns do in fact match the parallel template. So, we can conclude that the parallel nodes running themselves in parallel do translate correctly when explicit interleavings are not shown.

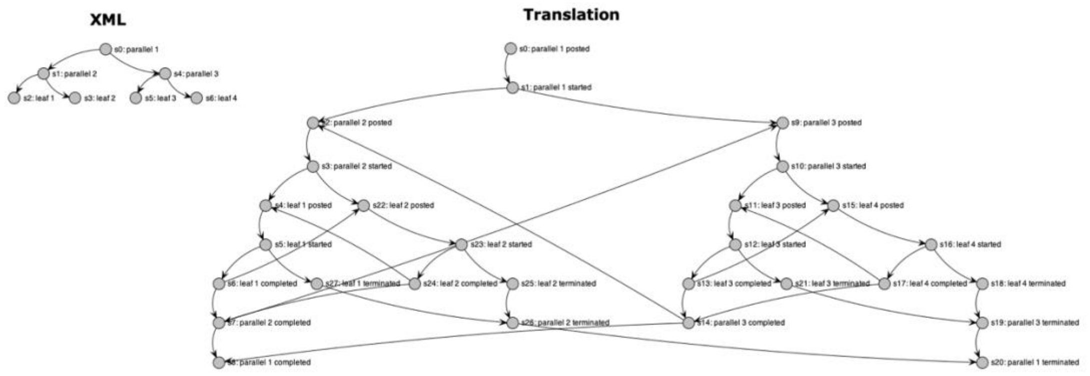


Figure 29. Dual Parallel XML & Translation (Without Explicit Interleavings)

In Figure 30 we have the dual parallel translation with explicit interleavings. This graph should match Figure 29, but have all the permutations of all parallel children showing. Since all three parallel nodes in this graph have two substeps, we expect to see *2!* or two permutations for each. Starting with *parallel 1*, we do see that *parallel 2* precedes *parallel 3* on the left and that *parallel 3* precedes *parallel 2* on the right. So, the first interleaving calculation looks correct. Now both sets of *parallel 2* and both sets of *parallel 3* should each have two interleavings of their children. We do in fact see this in

Figure 30. *Parallel 2* on the left and *parallel 2* on the right each have *leaf 1* preceding *leaf 2* and then as *leaf 2* preceding *leaf 1* under it. The same with *parallel 3*. Both the *parallel 3* on the left and the *parallel 3* on the right have *leaf 3* preceding *leaf 4* and also *leaf 4* preceding *leaf 3*. So all our permutations are accounted for.

Now that we know the interleavings are correct in Figure 30, we should also check that all the connections in Figure 30 are correct. For all eight leaf interleavings, the final node connects to the correct *parallel completed* node. And for all eight leaf interleavings, each *leaf terminated* connects to the correct *parallel terminated* node. We can conclude that the translator has translated the translation with explicit interleavings correctly.

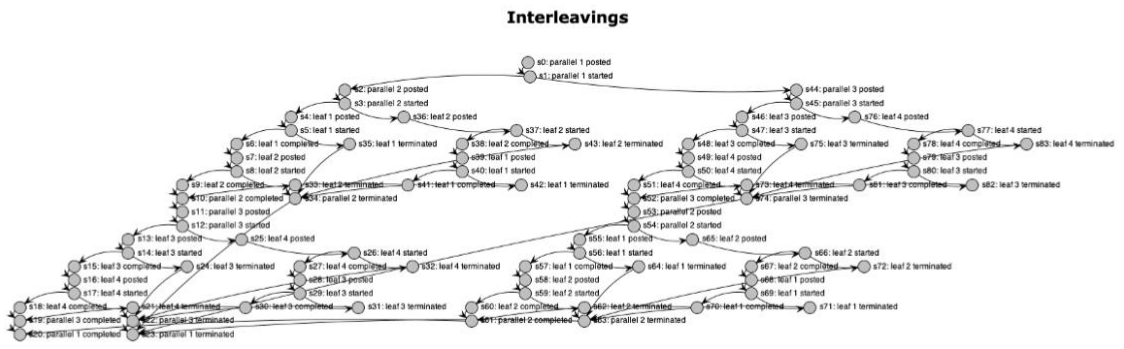


Figure 30. Dual Parallel Translation (With Explicit Interleavings)

In trying to assess the correctness of the COVID graph mentioned above, as well as the banking graph, we lean on correctness with the above graphs to infer that the COVID and banking translations are correct. The COVID and banking graphs with explicit interleavings are quite massive, so it difficult to walk through each node and explain it with certainty.

VI. RELATED WORK

Although a fair amount of work has already been done in adjacent areas, none has taken the same approach as this paper, automatically combining Little-JIL processes to run concurrently and then to check for properties. Some notable work, though, on model checking Little-JIL processes has been done by Lerner. Lerner published a paper on model checking in Little-JIL, using a prebuilt model checking tool called LTSA [4]. Their using a prebuilt tool is one way their work is different than mine. My translator and analyzer were written from scratch, which gives a kind of customizable experience not always possible with prebuilt tools. Another difference is that Lerner's approach is not one that combines processes, just runs single processes.

Chen et al. have explored detecting safety critical defects in Little-JIL processes by automatically generating fault trees. As long as the process being studied is very well defined in Little-JIL, they found automatic fault generation quite complete and beneficial in identifying errors in the process. They presented an algorithm which generated the fault trees from Little-JIL process definitions [5]. The defects they searched for had to do with wrong inputs (artifacts or resources) to a step. This is quite a different approach from mine because model checking for any given property because a property does not have to be an input. A property can just be a condition (like "the sky is blue"). Also, model checking in some ways is more powerful because of its temporal nature. Although model checking can be combined with fault tree analysis, that is not what Chen et al.'s work is about. Chen's work is also not about parallel processes.

Rura and Lerner have studied Little-JIL static analysis using a constraint checker to find semantic errors and programming anomalies in processes. They used xlinkit, a commercial constraint checker, which uses an XML type input process and compares the process to rules. Examples of defects they look for are data loss, infinite recursion and race conditions [6]. This differs from my work in a few ways. They are checking for semantic errors and process anomalies, but not for properties within the processes. They are not using concurrency or automatic combination.

Similarly, Cobleigh et al. used FLAVERS, a finite state verification system to verify properties of Little-JIL processes. FLAVERS uses a control flow graph model and for concurrent processes, a trace flow graph. They checked for race conditions, among other properties [7]. Although this work is similar to mine in that it is model checking on Little-JIL processes, there is no element of automatic process combination like in my work. Also, they use FLAVERS for model checking, whereas my model checker is built from scratch.

Some of the work model checking concurrent processes is quite sophisticated. Gupta explored the pros and cons of model checking with push down systems. Part of their work is finding concurrency bugs with model checking [8]. One difference between their work and mine is they aren't using Little-JIL and they also aren't using automatic process combination. They use VeriSol for model checking, where my model checker is written by hand. They use a variety of strategies to reduce the bottlenecks caused by interleavings such as partial order reduction and property-driven pruning.

Mongiello and Castelluccia wrote a paper on model checking properties on the BEPL business process language. BEPL stands for business execution process language and is a “graph-based description language for composition of web service that is especially suitable to model requirements of enterprise applications.” They check for properties about the message exchange protocol involved in web service communication. Like in my work, they use CTL. They use NuSMV as the input language to express the formal model [26]. The main differences between their work and mine is that I use Little-JIL and that I automatically combine different processes to model concurrency.

Cheng et al. studied formal verification of embedded software, using automated mapping from model components to the math used by verification. They use Ptolemy II for modelling embedded processes and NuSMV for model checking. They use Kripke structures as well as communicating timed automata to represent the processes in a format for model checking. They present a case study of traffic lights where there are two lights, one for cars and one for pedestrians. The property they use is whether the light is green or not. The proposition they check is whether it is impossible to have the car light and the pedestrian light at the same time. Their SMV model checker results showed that the specification did hold [27]. This work is different from mine in that it does not automatically combine two or more processes to represent concurrent operation.

Yin et al. did research around designing algorithms to help streamline issues stemming from the path explosion problem when verifying concurrent programs. Their work reduces the number or required iterations by refinement constraint and learnt clause sharing. They achieved a linear reduction of iterations. This work is only related to mine in an adjacent way. They focus in on the one problem of path explosion in model checking and offer some creative solutions. My work also explores the path explosion problem, but in relation to automatically combining one or more processes, potentially written by different people or organizations. Interestingly, in their experimental results, the times their processes took to run are much larger than those in my results. Their fastest example is 3.7 seconds and their slowest is 496 seconds. All my results were less than 1 second, although it seems that their data set (SV-COMP 2018 benchmarks) is likely running much larger examples than I ran. [28]

VII. EVALUATION & EXPERIMENTAL RESULTS

The main conclusions drawn from this paper is that both translation time and analysis time roughly increase linearly with an increase in the number of nodes in the graph. Translation time increases dramatically with interleavings. Analysis time in the general case decreases with interleavings, but when the graph is extremely large, it increases. Also, modern personal computing processor power is easily sufficient for moderately sized graphs (5-20 nodes before translation, 1,500 – 2,500 after translation)

When combining the COVID-19 experience in parallel with the hospital's ordering of the ventilator, I found that the safety property of "someone who is a high risk to others will never be at home" held for all states, but that the liveness property of "if a ventilator is requested, it will eventually be available for use" did not hold for all states. So, it is true that someone who is a high risk to others will never be at home, but it is not true that a ventilator which is requested will necessarily eventually become available. My analyzer program (about 32,000 lines of Java code) determined that one counterexample path for the liveness property was *(s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18,s19,s20,s21,s22,s23,s24,s25,s26,s27,s28,s29,s30,s31,s32,s33,s34,s35,s36,s37,s38,s39,s40,s41,s42,s43,s44,s45,s46,s47,s48,s49).* The analyzer took 0.017812 seconds to check the safety property and 0.086823 seconds to check the liveness property. By modelling the processes and graphing them in parallel, we have results which could theoretically be live saving, if this were an actual process used by a hospital. The *combined covid-hospitalization-simple.ljx* and *ventilator-order.ljx* processes, when translated with interleavings, have 1,635 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. I had feared the program would crash if the number of nodes became too big, but it actually handled them quite easily in this example.

Below are the translated graphs for the COVID-19 example 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.

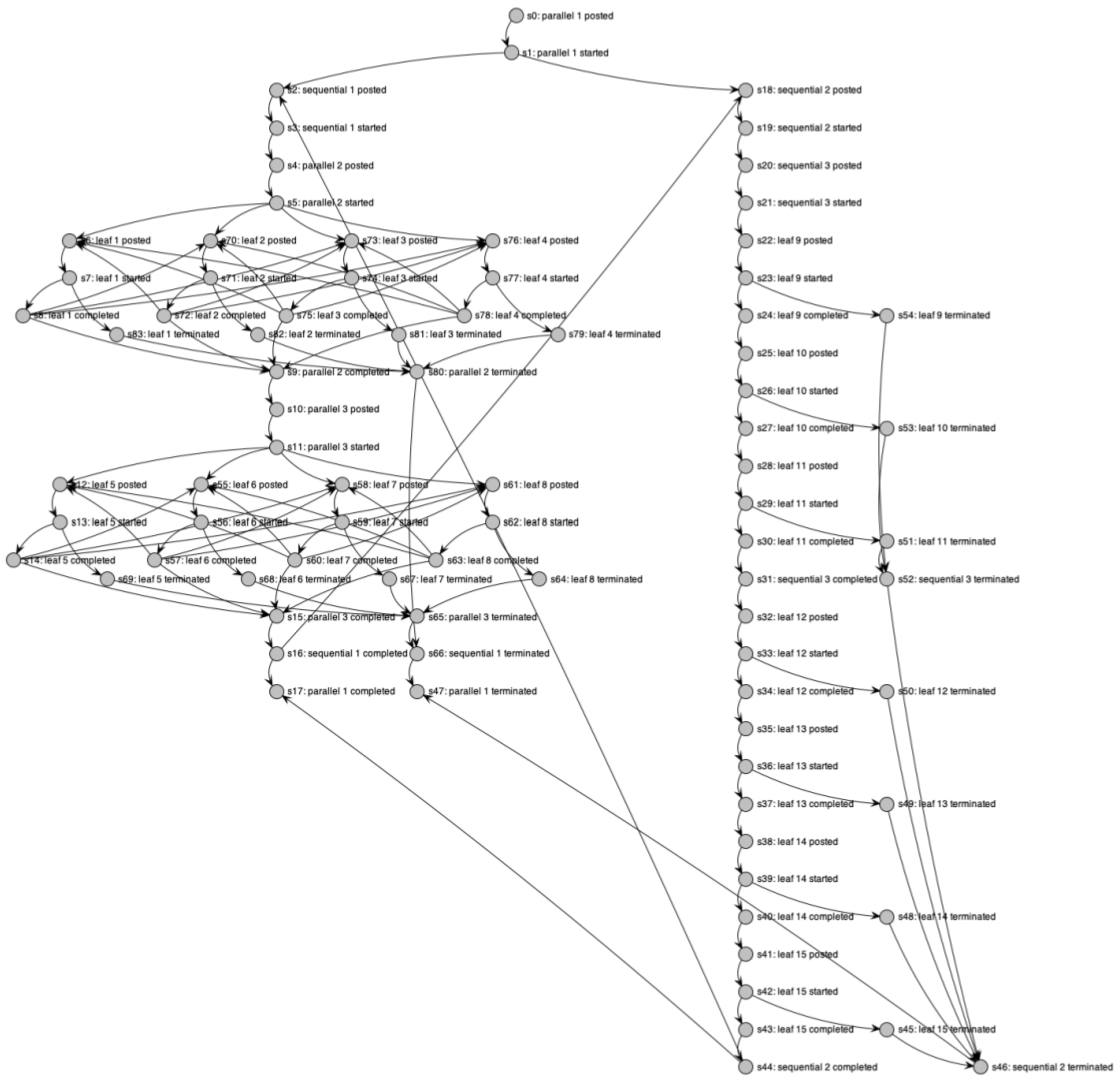


Figure 31. COVID-19 Translation Graph (Without Interleavings)

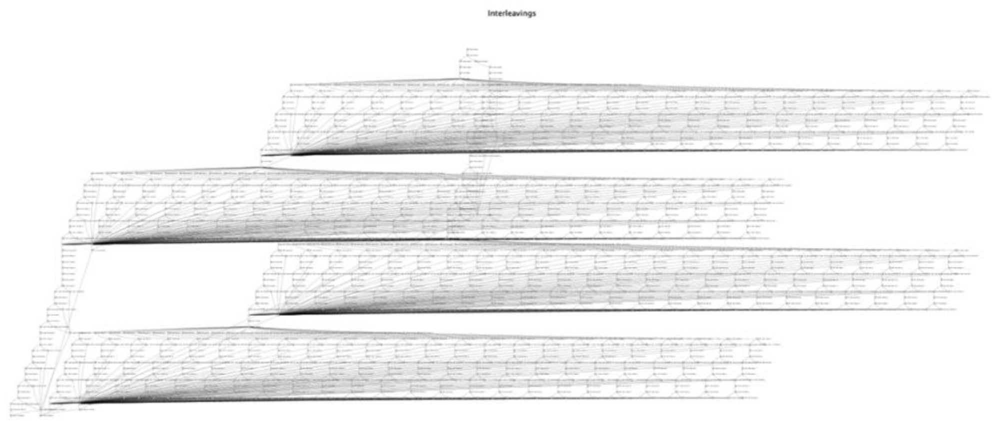


Figure 32. COVID-19 Translation Graph (With Interleavings)

Below in Figure 33 are the graphs for the process checks XML graph and the process transfers XML graph, combined and running in parallel.

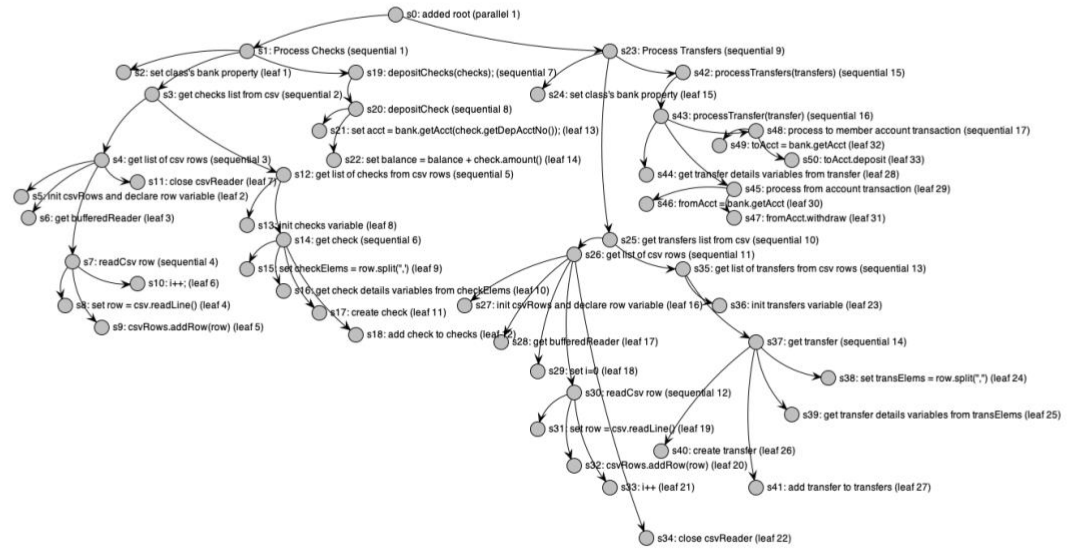

s0: added root (parallel 1)
s1: Process Checks (sequential 1)
s2: set class's bank property (leaf 1)
s19: depositChecks(checks); (sequential 7)
s3: get checks list from csv (sequential 2)
s20: depositCheck (sequential 8)
s21: set acct = bank.getAcct(check.getDepAcctNo()); (leaf 13)
s22: set balance = balance + check.amount() (leaf 14)
s4: get list of csv rows (sequential 3)
s12: get list of checks from csv rows (sequential 5)
s11: close csvReader (leaf 7)
s5: init csvRows and declare row variable (leaf 2)
s6: get bufferedReader (leaf 3)
s13: init checks variable (leaf 8)
s14: get check (sequential 6)
s7: readCsv row (sequential 4)
s15: set checkElems = row.split(",') (leaf 9)
s10: i++; (leaf 6)
s16: get check details variables from checkElems (leaf 10)
s8: set row = csv.readLine() (leaf 4)
s17: create check (leaf 11)
s9: csvRows.addRow(row) (leaf 5)
s18: add check to checks (leaf 12)
s23: Process Transfers (sequential 9)
s42: processTransfers(transfers) (sequential 15)
s24: set class's bank property (leaf 15)
s43: processTransfer(transfer) (sequential 16)
s48: process to member account transaction (sequential 17)
s49: toAcct = bank.getAcct (leaf 32)
s50: toAcct.deposit (leaf 33)
s44: get transfer details variables from transfer (leaf 28)
s45: process from account transaction (leaf 29)
s46: fromAcct = bank.getAcct (leaf 30)
s47: fromAcct.withdraw (leaf 31)
s25: get transfers list from csv (sequential 10)
s26: get list of csv rows (sequential 11)
s35: get list of transfers from csv rows (sequential 13)
s27: init csvRows and declare row variable (leaf 16)
s36: init transfers variable (leaf 23)
s28: get bufferedReader (leaf 17)
s37: get transfer (sequential 14)
s29: set i=0 (leaf 18)
s38: set transElems = row.split(",") (leaf 24)
s30: readCsv row (sequential 12)
s39: get transfer details variables from transElems (leaf 25)
s31: set row = csv.readLine() (leaf 19)
s40: create transfer (leaf 26)
s32: csvRows.addRow(row) (leaf 20)
s33: i++ (leaf 21)
s41: add transfer to transfers (leaf 27)
s34: close csvReader (leaf 22)

Figure 33. Combined XML ProcessChecks & ProcessTransfers Graphs In Parallel

I've included the banking Kripke graph (without explicit interleavings) in Appendix C because of its large size. Below are the translated parallel banking graphs with explicit interleavings.

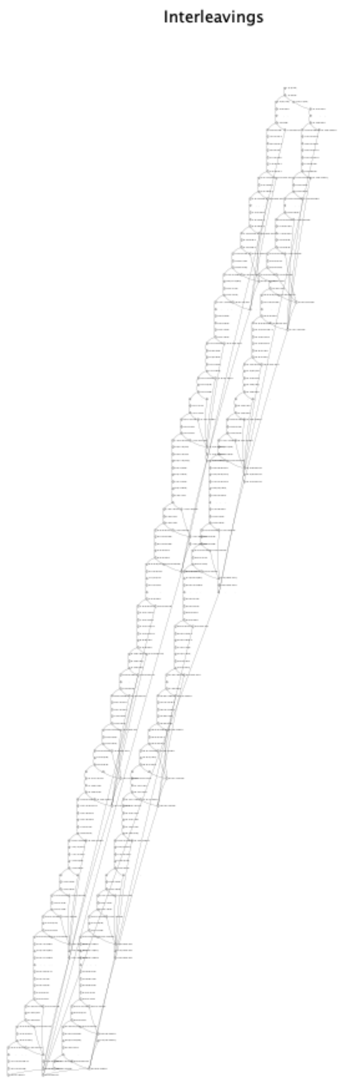


Figure 34. Bank-Parallel Graph Translated (Without Interleavings)

The banking problem, "The transfer buffer is always null until the checks array size is zero and the checks buffer is closed" or *A[¬t U(s∧¬q)]*, when evaluated, proved to not be true for all states. One counter example path the analyzer came up with is *(s0)*. In English, this is saying that although *s0* has *¬t* (the transfer buffer is null), it does not have *(s∧¬q)* as its child, so it is not true that the transfer buffer is necessarily null until the array buffer size is zero and checks buffer is null. So, it is not true that transfers are never processed until deposits have been processed. The combined ProcessChecks and ProcessTransfers graphs translated and running in parallel with interleavings has 379 nodes. The analyzer checked the *A[¬p Uq∧¬r]* model in 0.003197 seconds. This is very helpful information and could theoretically keep accounts from being incorrectly marked as overdrawn.

I also created some abstract XML graphs to translate, just to get better benchmarks on how quickly graphs of different sizes were being translated and to understand exactly how well or poorly my computer (2019 MacBook Pro with 2.6 GHz 6 core i7 processor and 32 GB RAM) was handling the exponential blowup in larger graphs with many interleavings.

Table 1. Experimental Results: Time To Translate & Analyze

Problem	# of nodes (total after translation, including interleavings)	Proposition in English	Model	Does model hold	Counter example path if does not hold	Time to translate (seconds)	Time to check model (seconds)
COVID	1,635	"someone who is a high risk to others will never be at home"	AG¬(r∧v)	Yes	-	0.050548	0.018945

COVID	1,635	if a ventilator is requested, it will eventually be available for use”	AG(p→AF(q))	No	(s0,s1,s2,s3,s4,s5,s6,s7,s8, s9,s10,s11,s12,s13,s14, s15,s16,s17,s18,s19, s20,s21, s22,s23,s24,s25,s26, s27,s28,s29,s30,s31, s32,s33, s34,s35,s36,s37,s38, s39,s40,s41,s42,s43, s44,s45, s46,s47,s48,s49)	0.047599	0.08459
Banking-Parallel	379	“The transfer buffer is always null until the checks array size is zero and the checks buffer is closed”	A[¬p Uq∧¬r]	No	(s0)	0.015728	0.002371
Parallel Node With Two Leaf Children	19	“For every path, the heart is ready in the next state”	AX(p)	Yes	-	0.003323	0.000157
Parallel Node With Three	75	“There exists a path where the hold door button	EF(p)	Yes	-	0.002486	0.000228

Leaf Children		is pressed in a future state”					
Parallel Node With Four Leaf Children	387	“For all paths, the alarm is sounding in all states”	AG(p)	No	{s0,s1,s2}	0.002652	0.000609
Parallel Node With Five Leaf Children	2403	“For all paths, decibels are greater than 100 in the next state”	AX(p)	Yes	-	0.093726	0.043702
Parallel Node With Ten Leaf Children	Throws OutOfMemoryError: Java heap space						

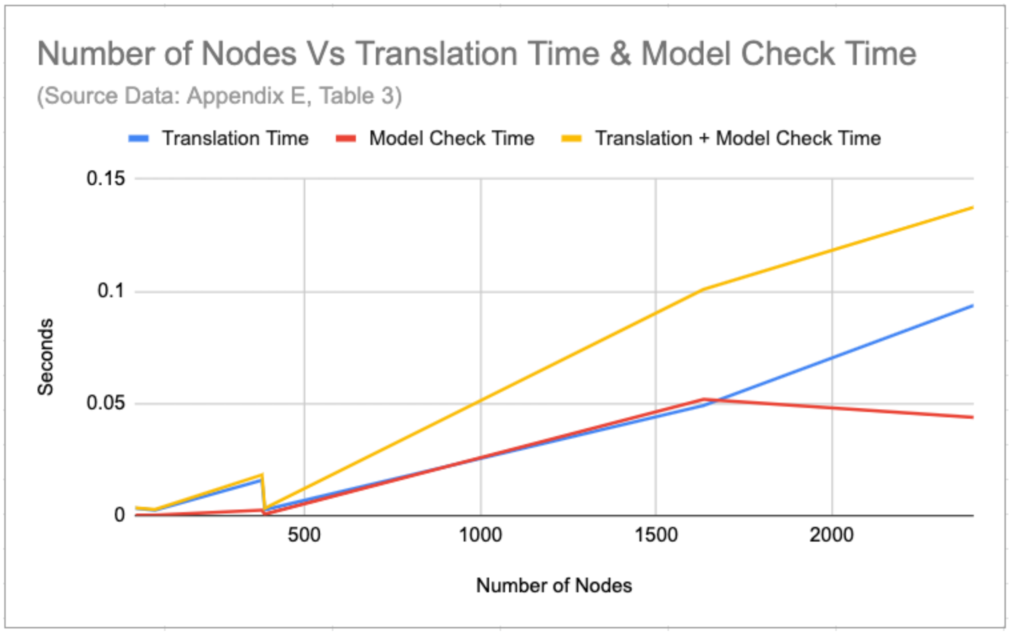


Figure 35. Numbers Of Nodes Vs Translation Time & Model Check Time

Table 2. Translation/Analysis Time With & Without Interleavings

		without interleavings			with interleavings		
Problem	Model	# of nodes	translation time (seconds)	Analysis time (seconds)	# of nodes	Translation time (seconds)	Analysis time (seconds)
COVID	AG¬(r∧v)	84	0.004577	0.028435	1,635	0.050548	0.018945
COVID	AG(p→AF(q))	84	0.004837	0.138157	1,635	0.047599	0.08459
Banking-Parallel	A[¬p Uq∧¬r]	196	0.007779	0.002238	379	0.015728	0.002371
Parallel Node With Two Leaf Children	AX(p)	8	0.001806	0.000148	19	0.003323	0.000157
Parallel Node With Three Leaf Children	EF(p)	16	0.002527	0.00039	75	0.002486	0.000228

Parallel Node With Four Leaf Children	AG(p)	20	0.002321	0.005053	387	0.002652	0.000609
Parallel Node With Five Leaf Children	AX(p)	24	0.002202	0.040588	2403	0.093726	0.043702

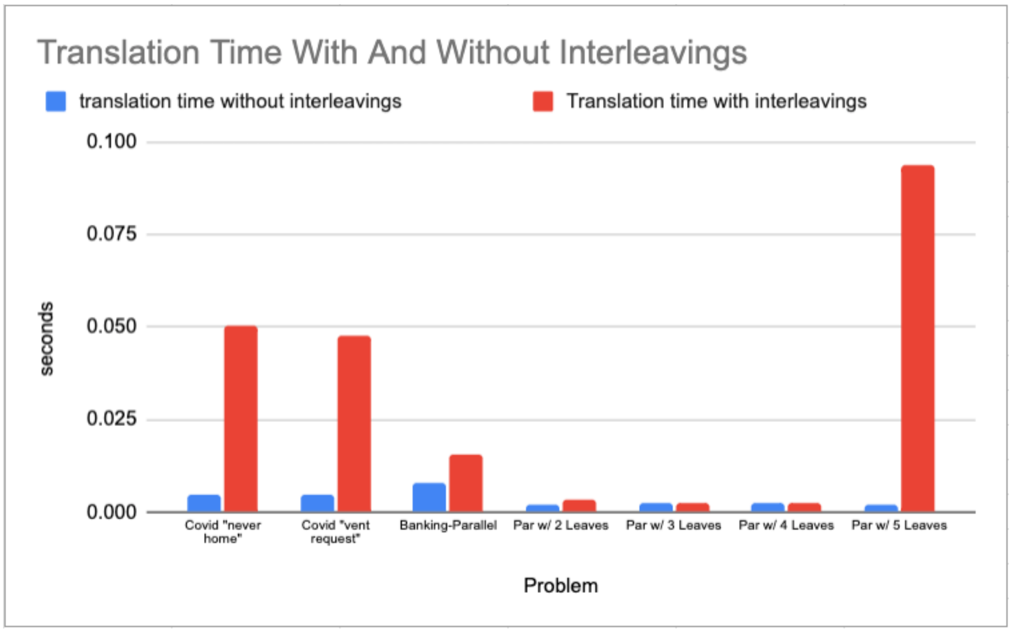


Figure 36. Translation Time With And Without Interleavings

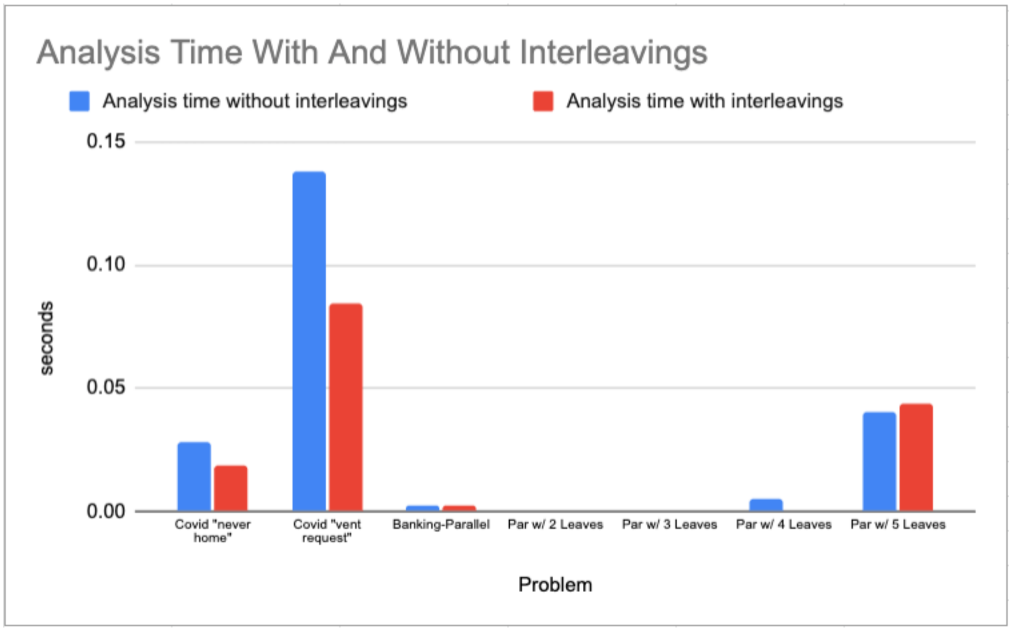


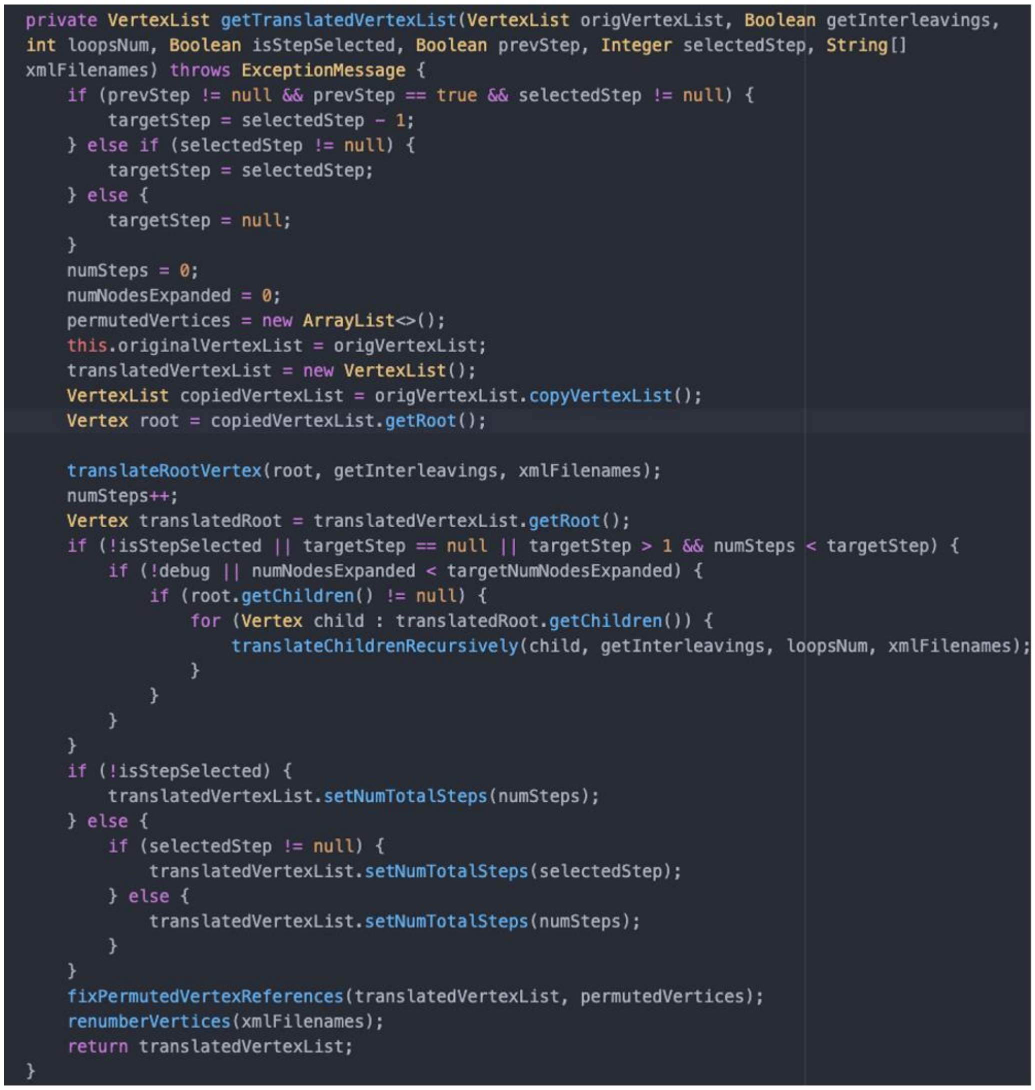
Figure 37. Analysis Time With And Without Interleavings

VIII. FUTURE WORK

More work that could be done in this area is improving the readability 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.

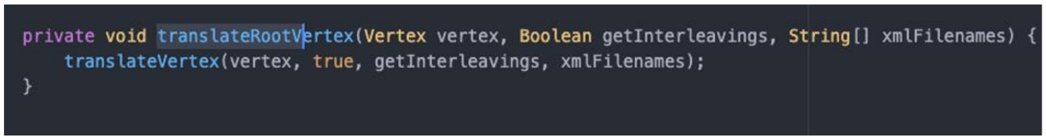
APPENDIX SECTION

Appendix A: Little-JIL Translation Code



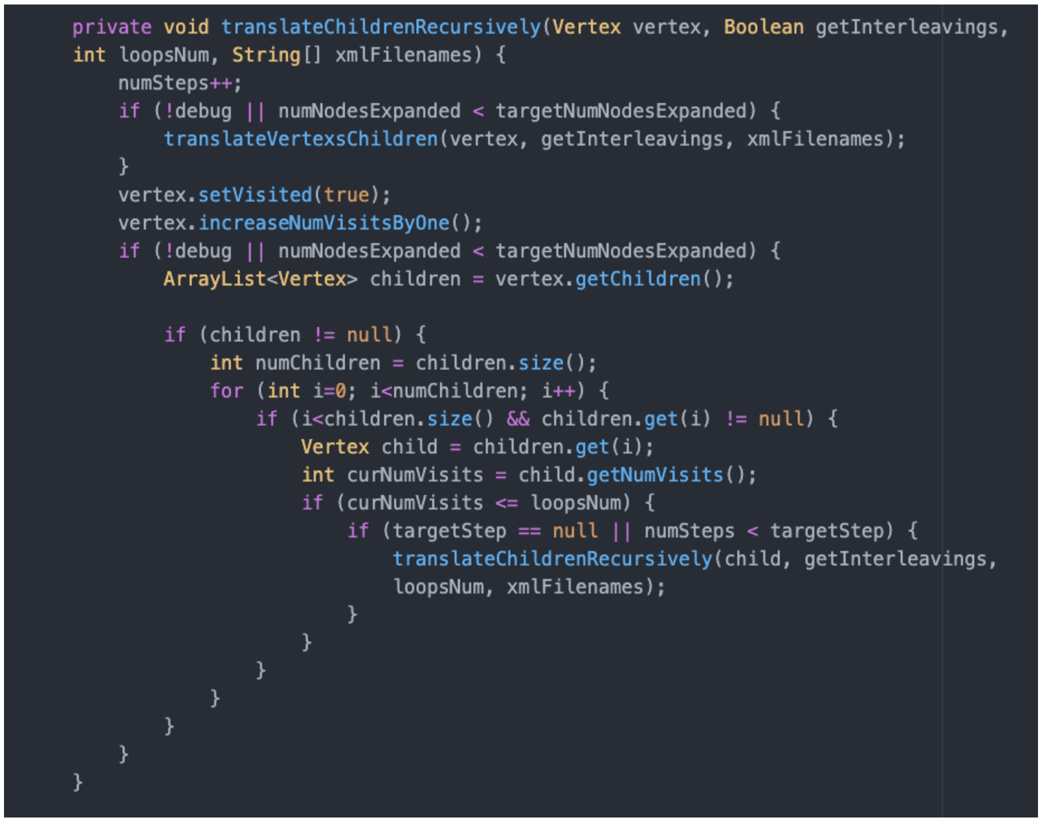
```java
private VertexList getTranslatedVertexList(VertexList origVertexList, Boolean getInterleavings,
int loopsNum, Boolean isStepSelected, Boolean prevStep, Integer selectedStep, String[]
xmlFilenames) throws ExceptionMessage {
    if (prevStep != null && prevStep == true && selectedStep != null) {
        targetStep = selectedStep - 1;
    } else if (selectedStep != null) {
        targetStep = selectedStep;
    } else {
        targetStep = null;
    }
    numSteps = 0;
    numNodesExpanded = 0;
    permutedVertices = new ArrayList<>();
    this.originalVertexList = origVertexList;
    translatedVertexList = new VertexList();
    VertexList copiedVertexList = origVertexList.copyVertexList();
    Vertex root = copiedVertexList.getRoot();

    translateRootVertex(root, getInterleavings, xmlFilenames);
    numSteps++;
    Vertex translatedRoot = translatedVertexList.getRoot();
    if (!isStepSelected || targetStep == null || targetStep > 1 && numSteps < targetStep) {
        if (!debug || numNodesExpanded < targetNumNodesExpanded) {
            if (root.getChildren() != null) {
                for (Vertex child : translatedRoot.getChildren()) {
                    translateChildrenRecursively(child, getInterleavings, loopsNum, xmlFilenames);
                }
            }
        }
    }
    if (!isStepSelected) {
        translatedVertexList.setNumTotalSteps(numSteps);
    } else {
        if (selectedStep != null) {
            translatedVertexList.setNumTotalSteps(selectedStep);
        } else {
            translatedVertexList.setNumTotalSteps(numSteps);
        }
    }
    fixPermutedVertexReferences(translatedVertexList, permutedVertices);
    renumberVertices(xmlFilenames);
    return translatedVertexList;
}
```

Figure 38. GetTranslatedVertexList()



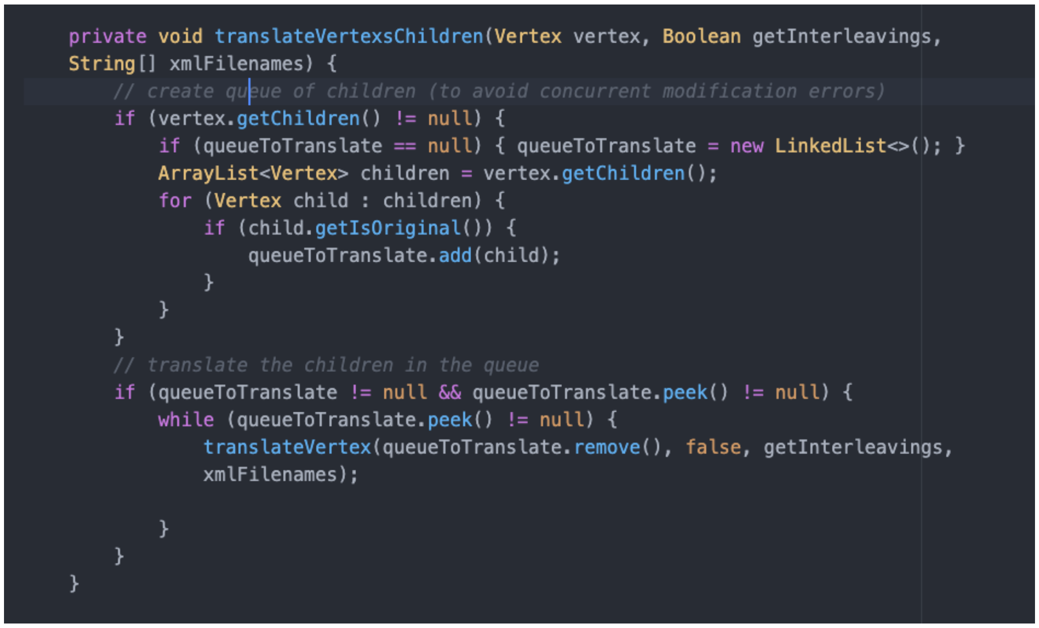
```java
private void translateRootVertex(Vertex vertex, Boolean getInterleavings, String[] xmlFilenames) {
    translateVertex(vertex, true, getInterleavings, xmlFilenames);
}
```

Figure 39. TranslateRootVertex()



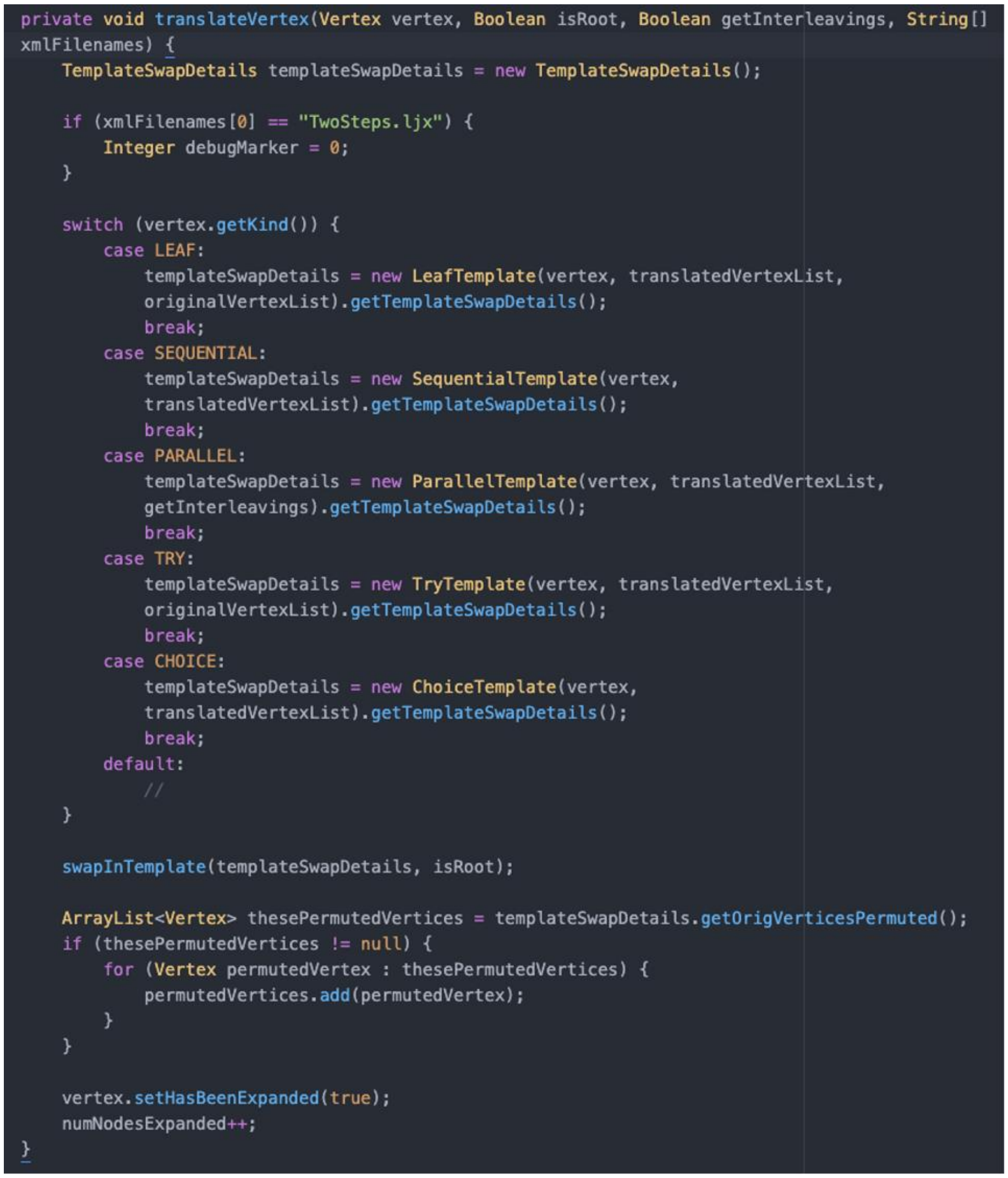
```java
private void translateChildrenRecursively(Vertex vertex, Boolean getInterleavings,
int loopsNum, String[] xmlFilenames) {
    numSteps++;
    if (!debug || numNodesExpanded < targetNumNodesExpanded) {
        translateVertexsChildren(vertex, getInterleavings, xmlFilenames);
    }
    vertex.setVisited(true);
    vertex.increaseNumVisitsByOne();
    if (!debug || numNodesExpanded < targetNumNodesExpanded) {
        ArrayList<Vertex> children = vertex.getChildren();

        if (children != null) {
            int numChildren = children.size();
            for (int i=0; i<numChildren; i++) {
                if (i<children.size() && children.get(i) != null) {
                    Vertex child = children.get(i);
                    int curNumVisits = child.getNumVisits();
                    if (curNumVisits <= loopsNum) {
                        if (targetStep == null || numSteps < targetStep) {
                            translateChildrenRecursively(child, getInterleavings,
                            loopsNum, xmlFilenames);
                        }
                    }
                }
            }
        }
    }
}
```

Figure 40. TranslateChildrenRecursively()



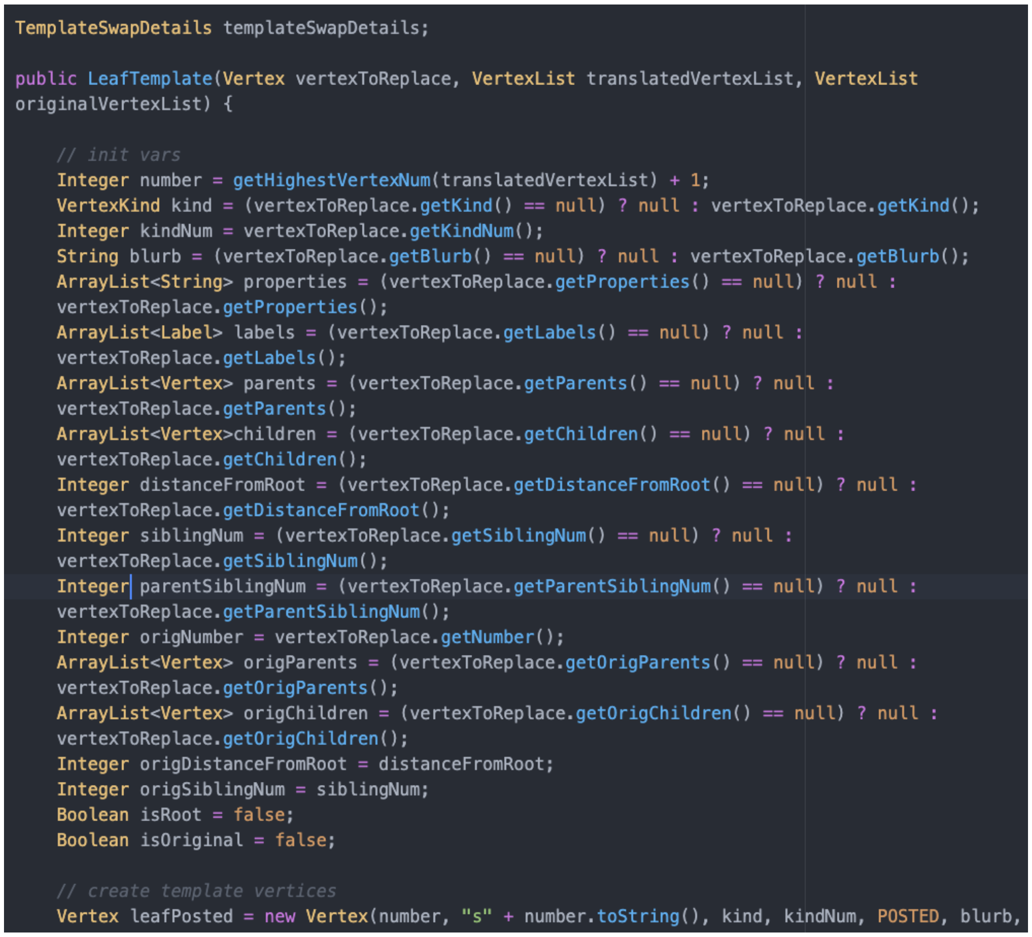
```java
private void translateVertexsChildren(Vertex vertex, Boolean getInterleavings,
String[] xmlFilenames) {
    // create queue of children (to avoid concurrent modification errors)
    if (vertex.getChildren() != null) {
        if (queueToTranslate == null) { queueToTranslate = new LinkedList<>(); }
        ArrayList<Vertex> children = vertex.getChildren();
        for (Vertex child : children) {
            if (child.getIsOriginal()) {
                queueToTranslate.add(child);
            }
        }
    }
    // translate the children in the queue
    if (queueToTranslate != null && queueToTranslate.peek() != null) {
        while (queueToTranslate.peek() != null) {
            translateVertex(queueToTranslate.remove(), false, getInterleavings,
            xmlFilenames);

        }
    }
}
```

Figure 41. TranslateVertexsChildren()



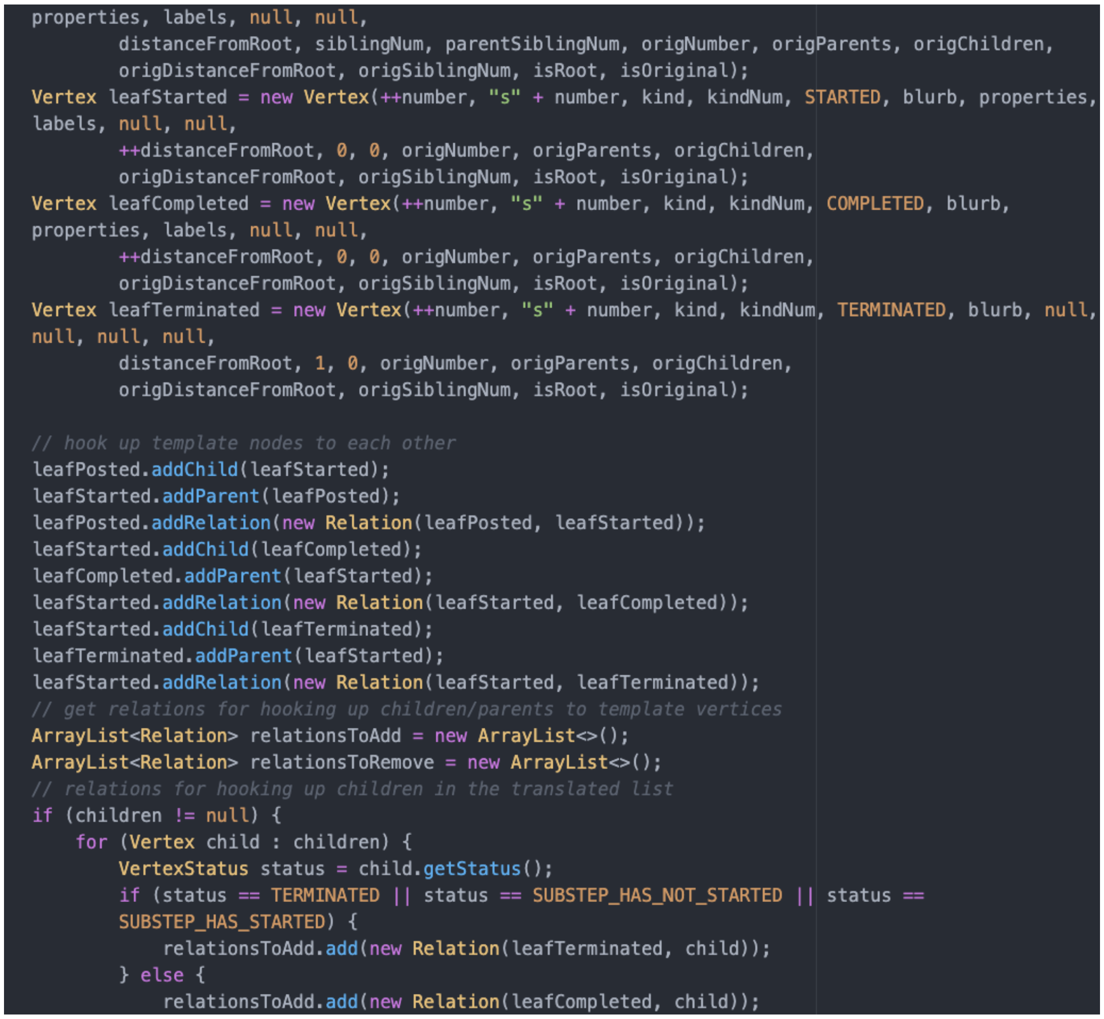
```java
private void translateVertex(Vertex vertex, Boolean isRoot, Boolean getInterleavings, String[]
xmlFilenames) {
    TemplateSwapDetails templateSwapDetails = new TemplateSwapDetails();

    if (xmlFilenames[0] == "TwoSteps.ljx") {
        Integer debugMarker = 0;
    }

    switch (vertex.getKind()) {
        case LEAF:
            templateSwapDetails = new LeafTemplate(vertex, translatedVertexList,
            originalVertexList).getTemplateSwapDetails();
            break;
        case SEQUENTIAL:
            templateSwapDetails = new SequentialTemplate(vertex,
            translatedVertexList).getTemplateSwapDetails();
            break;
        case PARALLEL:
            templateSwapDetails = new ParallelTemplate(vertex, translatedVertexList,
            getInterleavings).getTemplateSwapDetails();
            break;
        case TRY:
            templateSwapDetails = new TryTemplate(vertex, translatedVertexList,
            originalVertexList).getTemplateSwapDetails();
            break;
        case CHOICE:
            templateSwapDetails = new ChoiceTemplate(vertex,
            translatedVertexList).getTemplateSwapDetails();
            break;
        default:
            //
    }

    swapInTemplate(templateSwapDetails, isRoot);

    ArrayList<Vertex> thesePermutedVertices = templateSwapDetails.getOrigVerticesPermuted();
    if (thesePermutedVertices != null) {
        for (Vertex permutedVertex : thesePermutedVertices) {
            permutedVertices.add(permutedVertex);
        }
    }

    vertex.setHasBeenExpanded(true);
    numNodesExpanded++;
}
```

Figure 42. TranslateVertex()



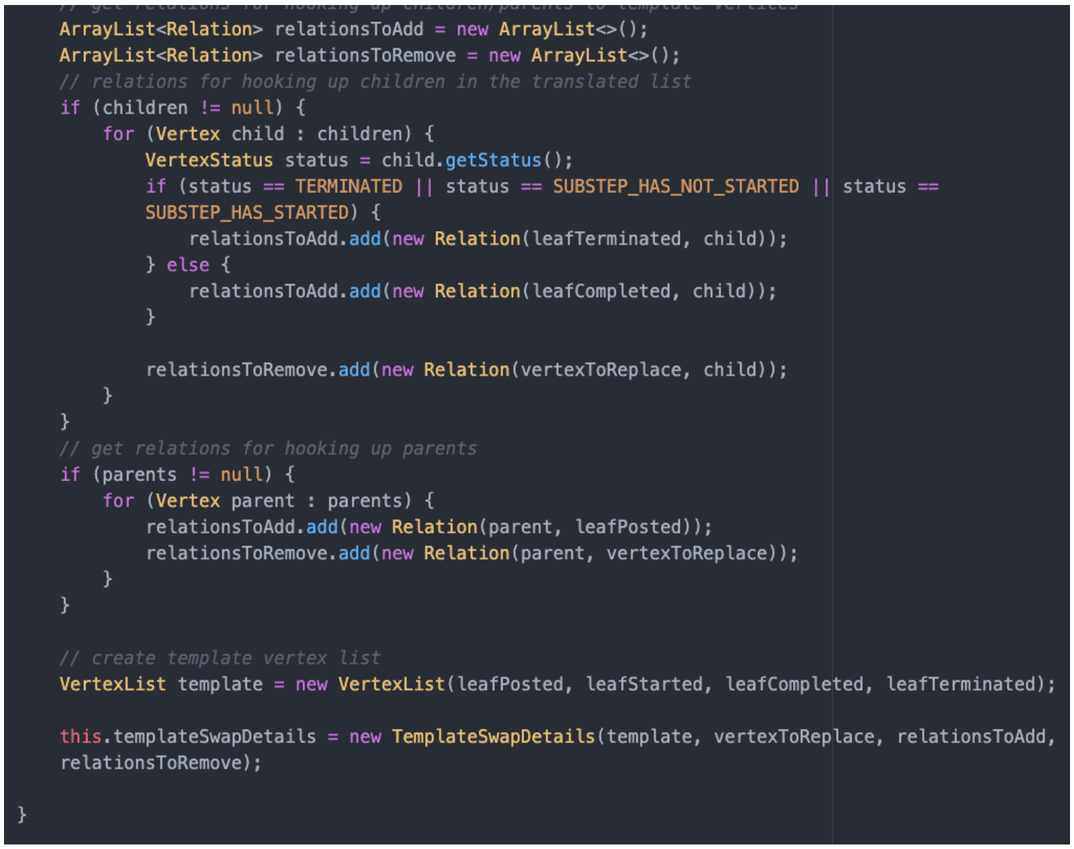
```java
TemplateSwapDetails templateSwapDetails;

public LeafTemplate(Vertex vertexToReplace, VertexList translatedVertexList, VertexList
originalVertexList) {

    // init vars
    Integer number = getHighestVertexNum(translatedVertexList) + 1;
    VertexKind kind = (vertexToReplace.getKind() == null) ? null : vertexToReplace.getKind();
    Integer kindNum = vertexToReplace.getKindNum();
    String blurb = (vertexToReplace.getBlurb() == null) ? null : vertexToReplace.getBlurb();
    ArrayList<String> properties = (vertexToReplace.getProperties() == null) ? null :
    vertexToReplace.getProperties();
    ArrayList<Label> labels = (vertexToReplace.getLabels() == null) ? null :
    vertexToReplace.getLabels();
    ArrayList<Vertex> parents = (vertexToReplace.getParents() == null) ? null :
    vertexToReplace.getParents();
    ArrayList<Vertex>children = (vertexToReplace.getChildren() == null) ? null :
    vertexToReplace.getChildren();
    Integer distanceFromRoot = (vertexToReplace.getDistanceFromRoot() == null) ? null :
    vertexToReplace.getDistanceFromRoot();
    Integer siblingNum = (vertexToReplace.getSiblingNum() == null) ? null :
    vertexToReplace.getSiblingNum();
    Integer parentSiblingNum = (vertexToReplace.getParentSiblingNum() == null) ? null :
    vertexToReplace.getParentSiblingNum();
    Integer origNumber = vertexToReplace.getNumber();
    ArrayList<Vertex> origParents = (vertexToReplace.getOrigParents() == null) ? null :
    vertexToReplace.getOrigParents();
    ArrayList<Vertex> origChildren = (vertexToReplace.getOrigChildren() == null) ? null :
    vertexToReplace.getOrigChildren();
    Integer origDistanceFromRoot = distanceFromRoot;
    Integer origSiblingNum = siblingNum;
    Boolean isRoot = false;
    Boolean isOriginal = false;

    // create template vertices
    Vertex leafPosted = new Vertex(number, "s" + number.toString(), kind, kindNum, POSTED, blurb,
```

Figure 43. Leaf Template (1/3)



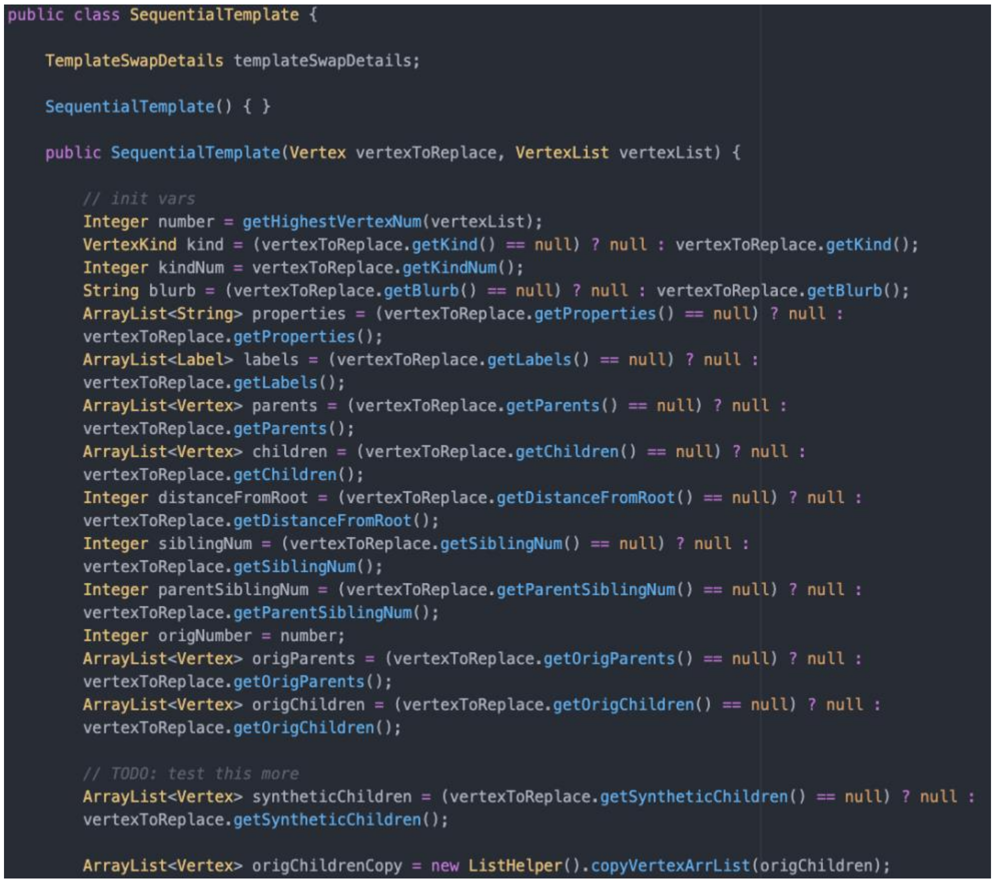
```
properties, labels, null, null,
        distanceFromRoot, siblingNum, parentSiblingNum, origNumber, origParents, origChildren,
        origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
Vertex leafStarted = new Vertex(++number, "s" + number, kind, kindNum, STARTED, blurb, properties,
labels, null, null,
        ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
        origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
Vertex leafCompleted = new Vertex(++number, "s" + number, kind, kindNum, COMPLETED, blurb,
properties, labels, null, null,
        ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
        origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
Vertex leafTerminated = new Vertex(++number, "s" + number, kind, kindNum, TERMINATED, blurb, null,
null, null, null,
        distanceFromRoot, 1, 0, origNumber, origParents, origChildren,
        origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);

// hook up template nodes to each other
leafPosted.addChild(leafStarted);
leafStarted.addParent(leafPosted);
leafPosted.addRelation(new Relation(leafPosted, leafStarted));
leafStarted.addChild(leafCompleted);
leafCompleted.addParent(leafStarted);
leafStarted.addRelation(new Relation(leafStarted, leafCompleted));
leafStarted.addChild(leafTerminated);
leafTerminated.addParent(leafStarted);
leafStarted.addRelation(new Relation(leafStarted, leafTerminated));
// get relations for hooking up children/parents to template vertices
ArrayList<Relation> relationsToAdd = new ArrayList<>();
ArrayList<Relation> relationsToRemove = new ArrayList<>();
// relations for hooking up children in the translated list
if (children != null) {
    for (Vertex child : children) {
        VertexStatus status = child.getStatus();
        if (status == TERMINATED || status == SUBSTEP_HAS_NOT_STARTED || status ==
        SUBSTEP_HAS_STARTED) {
            relationsToAdd.add(new Relation(leafTerminated, child));
        } else {
            relationsToAdd.add(new Relation(leafCompleted, child));
```

Figure 44. Leaf Template (2/3)



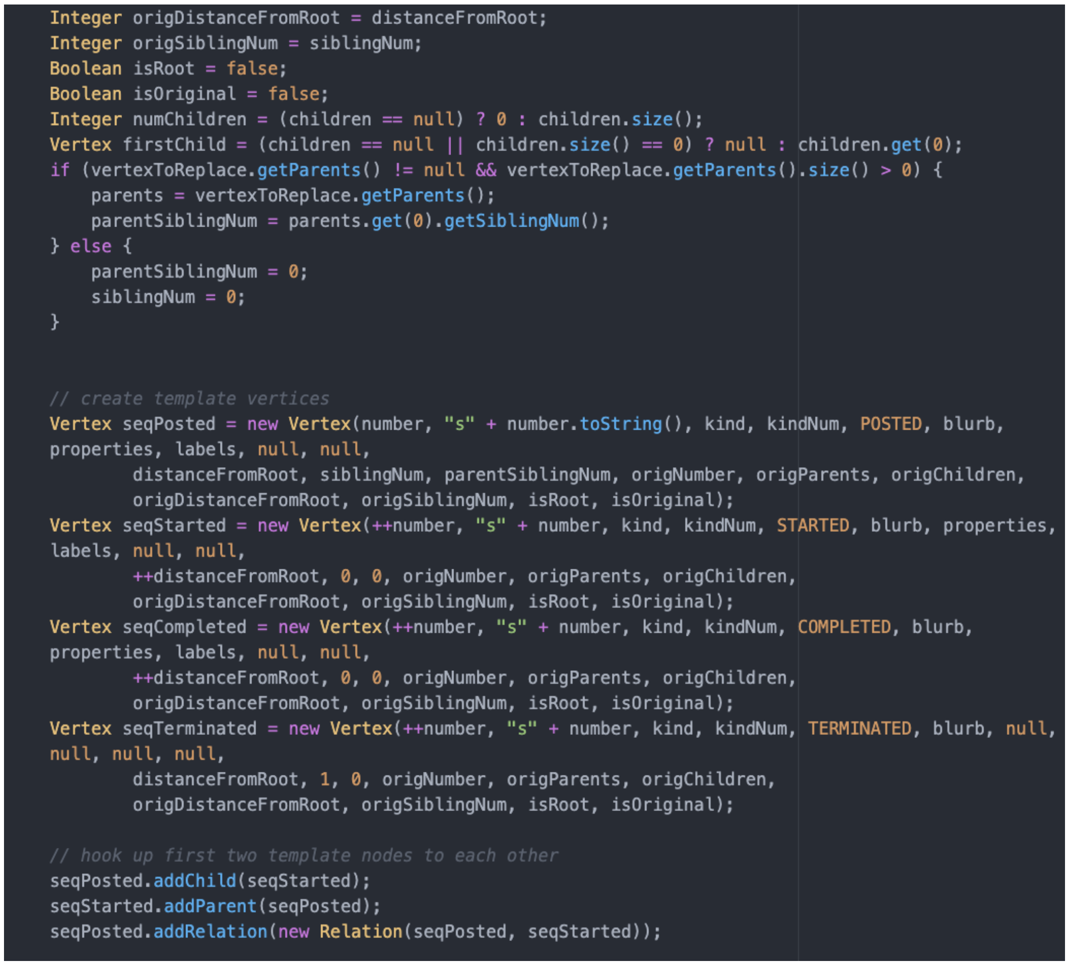
```
    // get relations for hooking up children/parents to template vertices
    ArrayList<Relation> relationsToAdd = new ArrayList<>();
    ArrayList<Relation> relationsToRemove = new ArrayList<>();
    // relations for hooking up children in the translated list
    if (children != null) {
        for (Vertex child : children) {
            VertexStatus status = child.getStatus();
            if (status == TERMINATED || status == SUBSTEP_HAS_NOT_STARTED || status ==
            SUBSTEP_HAS_STARTED) {
                relationsToAdd.add(new Relation(leafTerminated, child));
            } else {
                relationsToAdd.add(new Relation(leafCompleted, child));
            }

            relationsToRemove.add(new Relation(vertexToReplace, child));
        }
    }
    // get relations for hooking up parents
    if (parents != null) {
        for (Vertex parent : parents) {
            relationsToAdd.add(new Relation(parent, leafPosted));
            relationsToRemove.add(new Relation(parent, vertexToReplace));
        }
    }

    // create template vertex list
    VertexList template = new VertexList(leafPosted, leafStarted, leafCompleted, leafTerminated);

    this.templateSwapDetails = new TemplateSwapDetails(template, vertexToReplace, relationsToAdd,
    relationsToRemove);

}
```

Figure 45. Leaf Template (3/3)



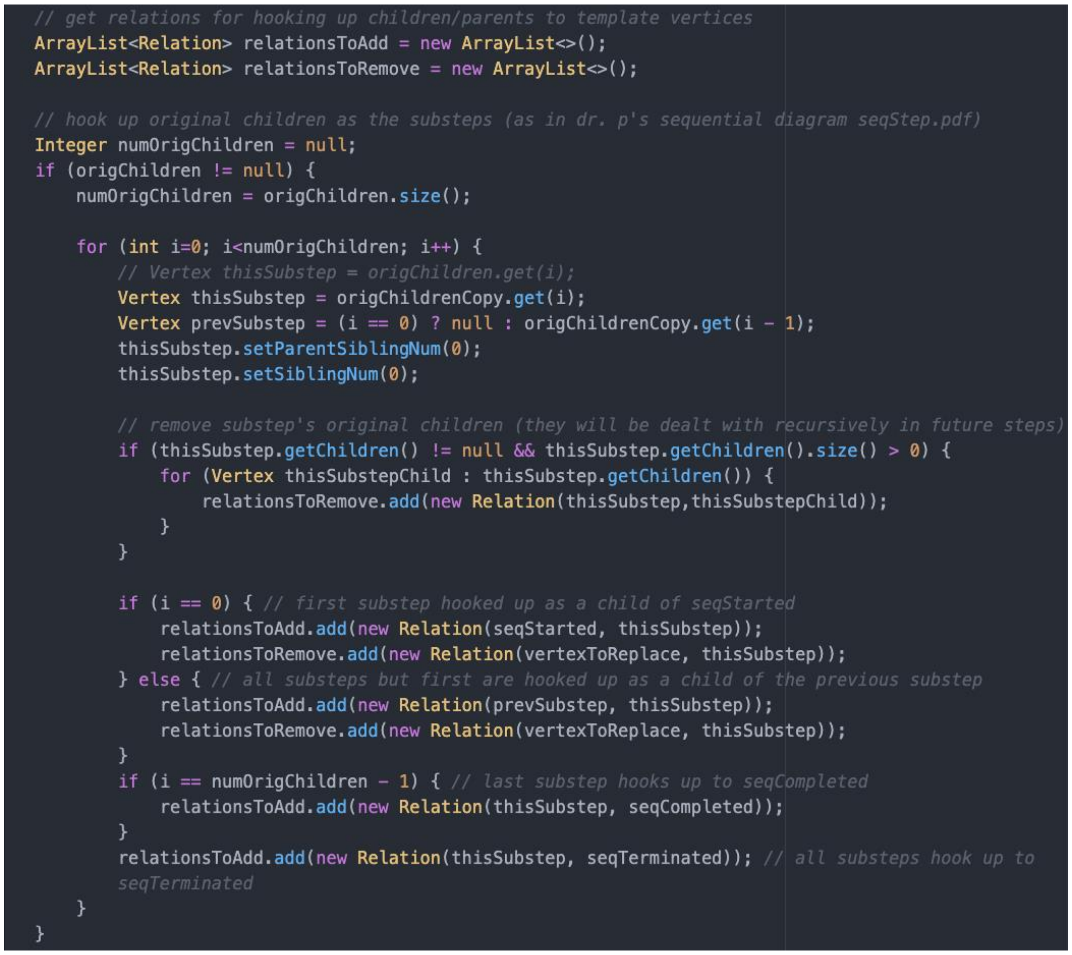
```java
public class SequentialTemplate {

    TemplateSwapDetails templateSwapDetails;

    SequentialTemplate() { }

    public SequentialTemplate(Vertex vertexToReplace, VertexList vertexList) {

        // init vars
        Integer number = getHighestVertexNum(vertexList);
        VertexKind kind = (vertexToReplace.getKind() == null) ? null : vertexToReplace.getKind();
        Integer kindNum = vertexToReplace.getKindNum();
        String blurb = (vertexToReplace.getBlurb() == null) ? null : vertexToReplace.getBlurb();
        ArrayList<String> properties = (vertexToReplace.getProperties() == null) ? null :
        vertexToReplace.getProperties();
        ArrayList<Label> labels = (vertexToReplace.getLabels() == null) ? null :
        vertexToReplace.getLabels();
        ArrayList<Vertex> parents = (vertexToReplace.getParents() == null) ? null :
        vertexToReplace.getParents();
        ArrayList<Vertex> children = (vertexToReplace.getChildren() == null) ? null :
        vertexToReplace.getChildren();
        Integer distanceFromRoot = (vertexToReplace.getDistanceFromRoot() == null) ? null :
        vertexToReplace.getDistanceFromRoot();
        Integer siblingNum = (vertexToReplace.getSiblingNum() == null) ? null :
        vertexToReplace.getSiblingNum();
        Integer parentSiblingNum = (vertexToReplace.getParentSiblingNum() == null) ? null :
        vertexToReplace.getParentSiblingNum();
        Integer origNumber = number;
        ArrayList<Vertex> origParents = (vertexToReplace.getOrigParents() == null) ? null :
        vertexToReplace.getOrigParents();
        ArrayList<Vertex> origChildren = (vertexToReplace.getOrigChildren() == null) ? null :
        vertexToReplace.getOrigChildren();

        // TODO: test this more
        ArrayList<Vertex> syntheticChildren = (vertexToReplace.getSyntheticChildren() == null) ? null :
        vertexToReplace.getSyntheticChildren();

        ArrayList<Vertex> origChildrenCopy = new ListHelper().copyVertexArrList(origChildren);
```

Figure 46. Sequential Template (1/4)



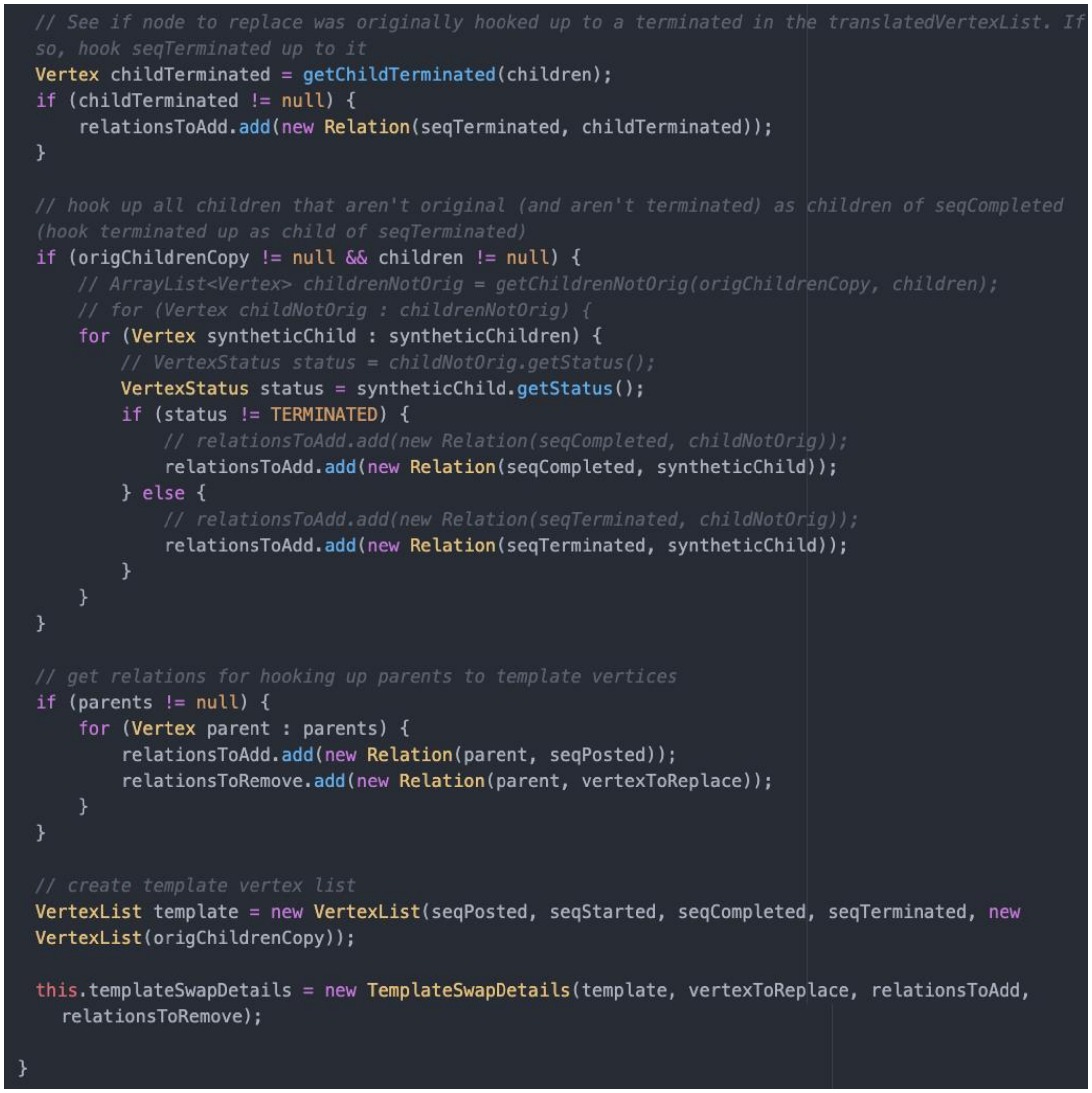
```
    Integer origDistanceFromRoot = distanceFromRoot;
    Integer origSiblingNum = siblingNum;
    Boolean isRoot = false;
    Boolean isOriginal = false;
    Integer numChildren = (children == null) ? 0 : children.size();
    Vertex firstChild = (children == null || children.size() == 0) ? null : children.get(0);
    if (vertexToReplace.getParents() != null && vertexToReplace.getParents().size() > 0) {
        parents = vertexToReplace.getParents();
        parentSiblingNum = parents.get(0).getSiblingNum();
    } else {
        parentSiblingNum = 0;
        siblingNum = 0;
    }


    // create template vertices
    Vertex seqPosted = new Vertex(number, "s" + number.toString(), kind, kindNum, POSTED, blurb,
    properties, labels, null, null,
            distanceFromRoot, siblingNum, parentSiblingNum, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
    Vertex seqStarted = new Vertex(++number, "s" + number, kind, kindNum, STARTED, blurb, properties,
    labels, null, null,
            ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
    Vertex seqCompleted = new Vertex(++number, "s" + number, kind, kindNum, COMPLETED, blurb,
    properties, labels, null, null,
            ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);
    Vertex seqTerminated = new Vertex(++number, "s" + number, kind, kindNum, TERMINATED, blurb, null,
    null, null, null,
            distanceFromRoot, 1, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);

    // hook up first two template nodes to each other
    seqPosted.addChild(seqStarted);
    seqStarted.addParent(seqPosted);
    seqPosted.addRelation(new Relation(seqPosted, seqStarted));
```

Figure 47. Sequential Template (2/4)



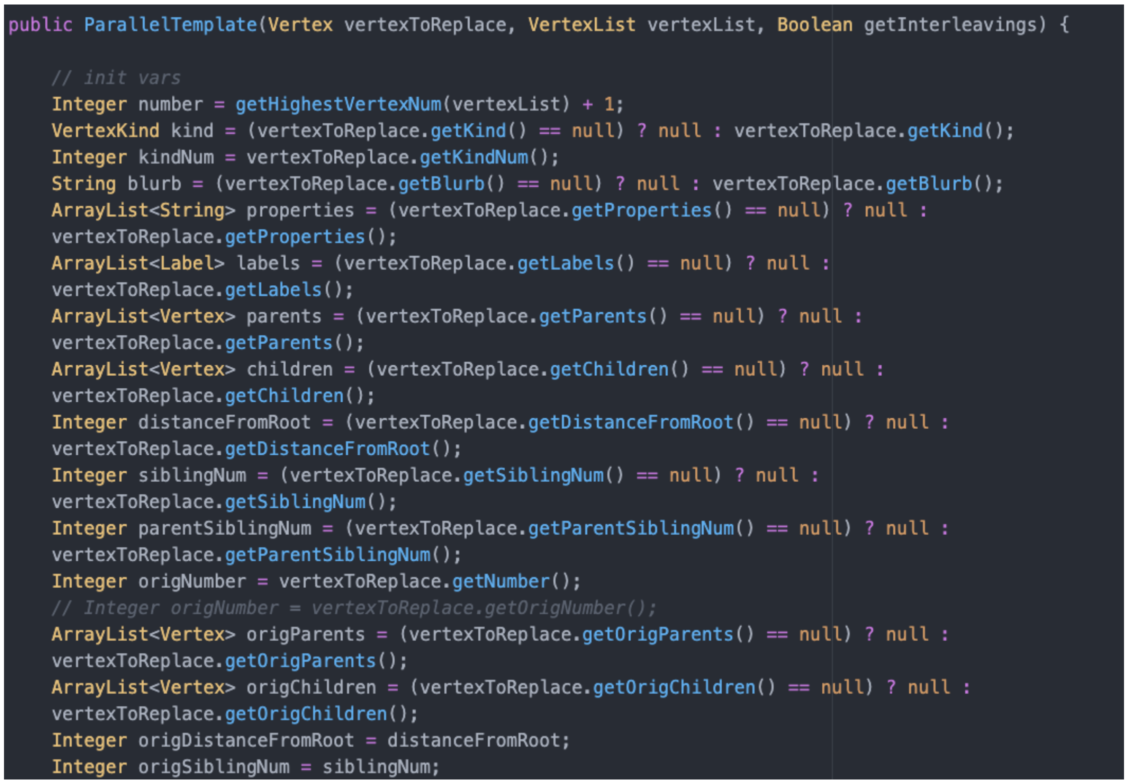
```java
// get relations for hooking up children/parents to template vertices
ArrayList<Relation> relationsToAdd = new ArrayList<>();
ArrayList<Relation> relationsToRemove = new ArrayList<>();

// hook up original children as the substeps (as in dr. p's sequential diagram seqStep.pdf)
Integer numOrigChildren = null;
if (origChildren != null) {
    numOrigChildren = origChildren.size();

    for (int i=0; i<numOrigChildren; i++) {
        // Vertex thisSubstep = origChildren.get(i);
        Vertex thisSubstep = origChildrenCopy.get(i);
        Vertex prevSubstep = (i == 0) ? null : origChildrenCopy.get(i - 1);
        thisSubstep.setParentSiblingNum(0);
        thisSubstep.setSiblingNum(0);

        // remove substep's original children (they will be dealt with recursively in future steps)
        if (thisSubstep.getChildren() != null && thisSubstep.getChildren().size() > 0) {
            for (Vertex thisSubstepChild : thisSubstep.getChildren()) {
                relationsToRemove.add(new Relation(thisSubstep,thisSubstepChild));
            }
        }

        if (i == 0) { // first substep hooked up as a child of seqStarted
            relationsToAdd.add(new Relation(seqStarted, thisSubstep));
            relationsToRemove.add(new Relation(vertexToReplace, thisSubstep));
        } else { // all substeps but first are hooked up as a child of the previous substep
            relationsToAdd.add(new Relation(prevSubstep, thisSubstep));
            relationsToRemove.add(new Relation(vertexToReplace, thisSubstep));
        }
        if (i == numOrigChildren - 1) { // last substep hooks up to seqCompleted
            relationsToAdd.add(new Relation(thisSubstep, seqCompleted));
        }
        relationsToAdd.add(new Relation(thisSubstep, seqTerminated)); // all substeps hook up to
        seqTerminated
    }
}
```

Figure 48. Sequential Template (3/4)



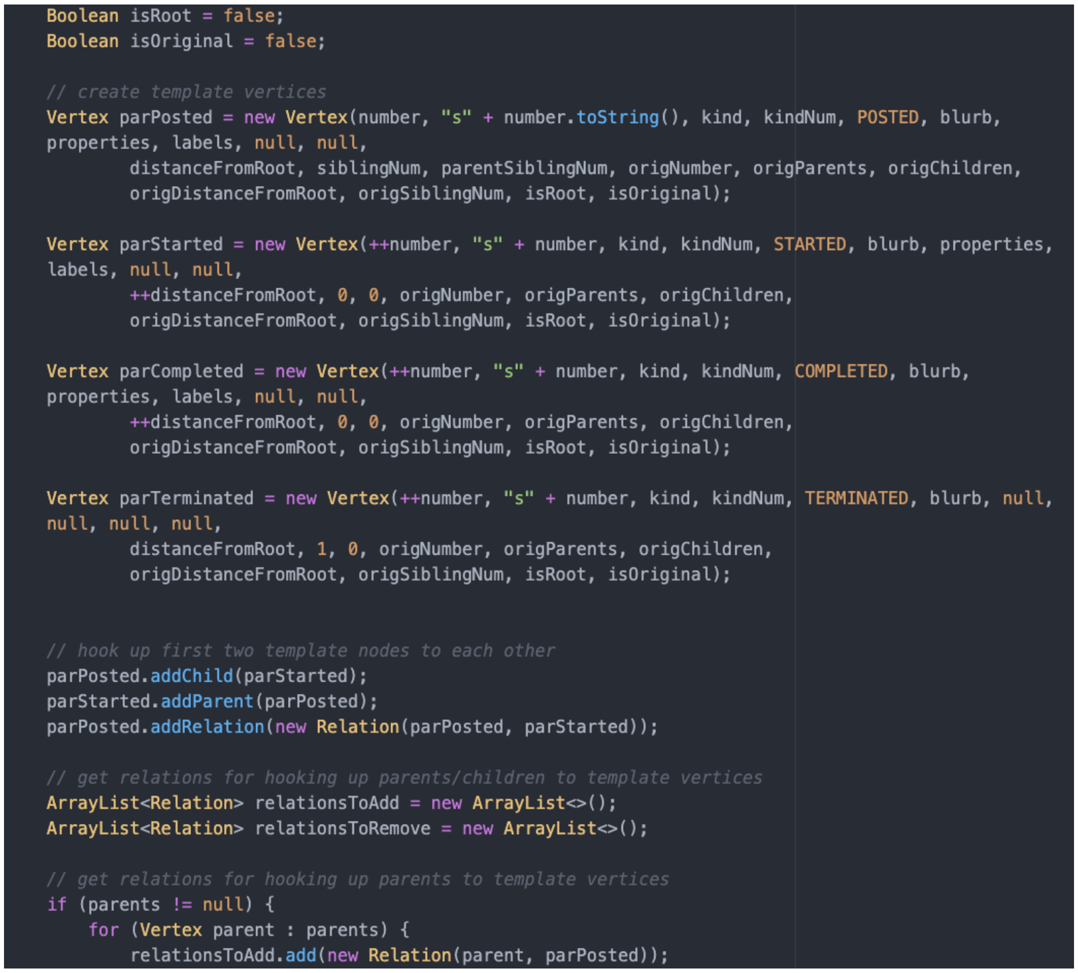
```
  // See if node to replace was originally hooked up to a terminated in the translatedVertexList. If
  so, hook seqTerminated up to it
  Vertex childTerminated = getChildTerminated(children);
  if (childTerminated != null) {
      relationsToAdd.add(new Relation(seqTerminated, childTerminated));
  }

  // hook up all children that aren't original (and aren't terminated) as children of seqCompleted
  (hook terminated up as child of seqTerminated)
  if (origChildrenCopy != null && children != null) {
      // ArrayList<Vertex> childrenNotOrig = getChildrenNotOrig(origChildrenCopy, children);
      // for (Vertex childNotOrig : childrenNotOrig) {
      for (Vertex syntheticChild : syntheticChildren) {
          // VertexStatus status = childNotOrig.getStatus();
          VertexStatus status = syntheticChild.getStatus();
          if (status != TERMINATED) {
              // relationsToAdd.add(new Relation(seqCompleted, childNotOrig));
              relationsToAdd.add(new Relation(seqCompleted, syntheticChild));
          } else {
              // relationsToAdd.add(new Relation(seqTerminated, childNotOrig));
              relationsToAdd.add(new Relation(seqTerminated, syntheticChild));
          }
      }
  }

  // get relations for hooking up parents to template vertices
  if (parents != null) {
      for (Vertex parent : parents) {
          relationsToAdd.add(new Relation(parent, seqPosted));
          relationsToRemove.add(new Relation(parent, vertexToReplace));
      }
  }

  // create template vertex list
  VertexList template = new VertexList(seqPosted, seqStarted, seqCompleted, seqTerminated, new
  VertexList(origChildrenCopy));

  this.templateSwapDetails = new TemplateSwapDetails(template, vertexToReplace, relationsToAdd,
    relationsToRemove);

}
```

Figure 49. Sequential Template (4/4)



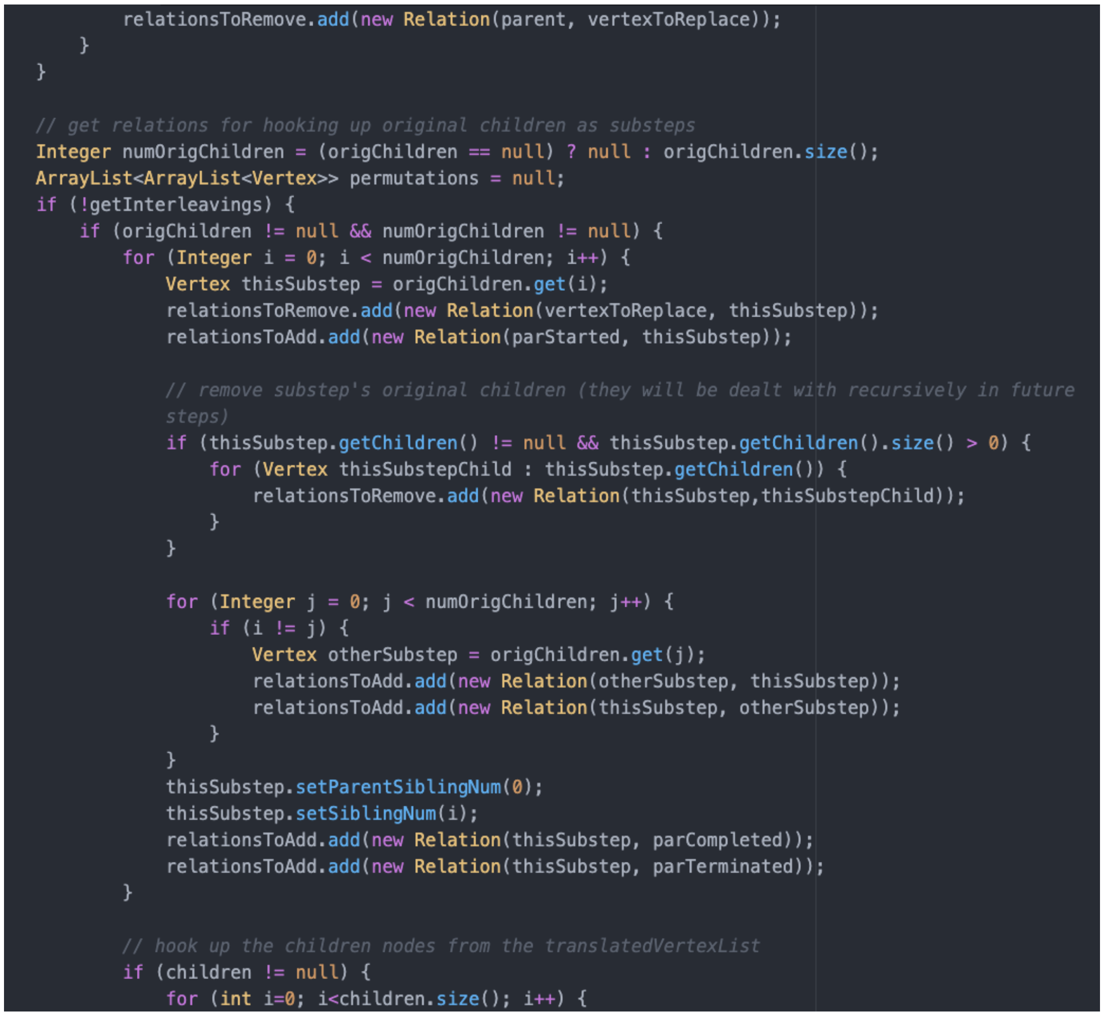
```
public ParallelTemplate(Vertex vertexToReplace, VertexList vertexList, Boolean getInterleavings) {

    // init vars
    Integer number = getHighestVertexNum(vertexList) + 1;
    VertexKind kind = (vertexToReplace.getKind() == null) ? null : vertexToReplace.getKind();
    Integer kindNum = vertexToReplace.getKindNum();
    String blurb = (vertexToReplace.getBlurb() == null) ? null : vertexToReplace.getBlurb();
    ArrayList<String> properties = (vertexToReplace.getProperties() == null) ? null :
    vertexToReplace.getProperties();
    ArrayList<Label> labels = (vertexToReplace.getLabels() == null) ? null :
    vertexToReplace.getLabels();
    ArrayList<Vertex> parents = (vertexToReplace.getParents() == null) ? null :
    vertexToReplace.getParents();
    ArrayList<Vertex> children = (vertexToReplace.getChildren() == null) ? null :
    vertexToReplace.getChildren();
    Integer distanceFromRoot = (vertexToReplace.getDistanceFromRoot() == null) ? null :
    vertexToReplace.getDistanceFromRoot();
    Integer siblingNum = (vertexToReplace.getSiblingNum() == null) ? null :
    vertexToReplace.getSiblingNum();
    Integer parentSiblingNum = (vertexToReplace.getParentSiblingNum() == null) ? null :
    vertexToReplace.getParentSiblingNum();
    Integer origNumber = vertexToReplace.getNumber();
    // Integer origNumber = vertexToReplace.getOrigNumber();
    ArrayList<Vertex> origParents = (vertexToReplace.getOrigParents() == null) ? null :
    vertexToReplace.getOrigParents();
    ArrayList<Vertex> origChildren = (vertexToReplace.getOrigChildren() == null) ? null :
    vertexToReplace.getOrigChildren();
    Integer origDistanceFromRoot = distanceFromRoot;
    Integer origSiblingNum = siblingNum;
```

Figure 50. Parallel Template (1/6)



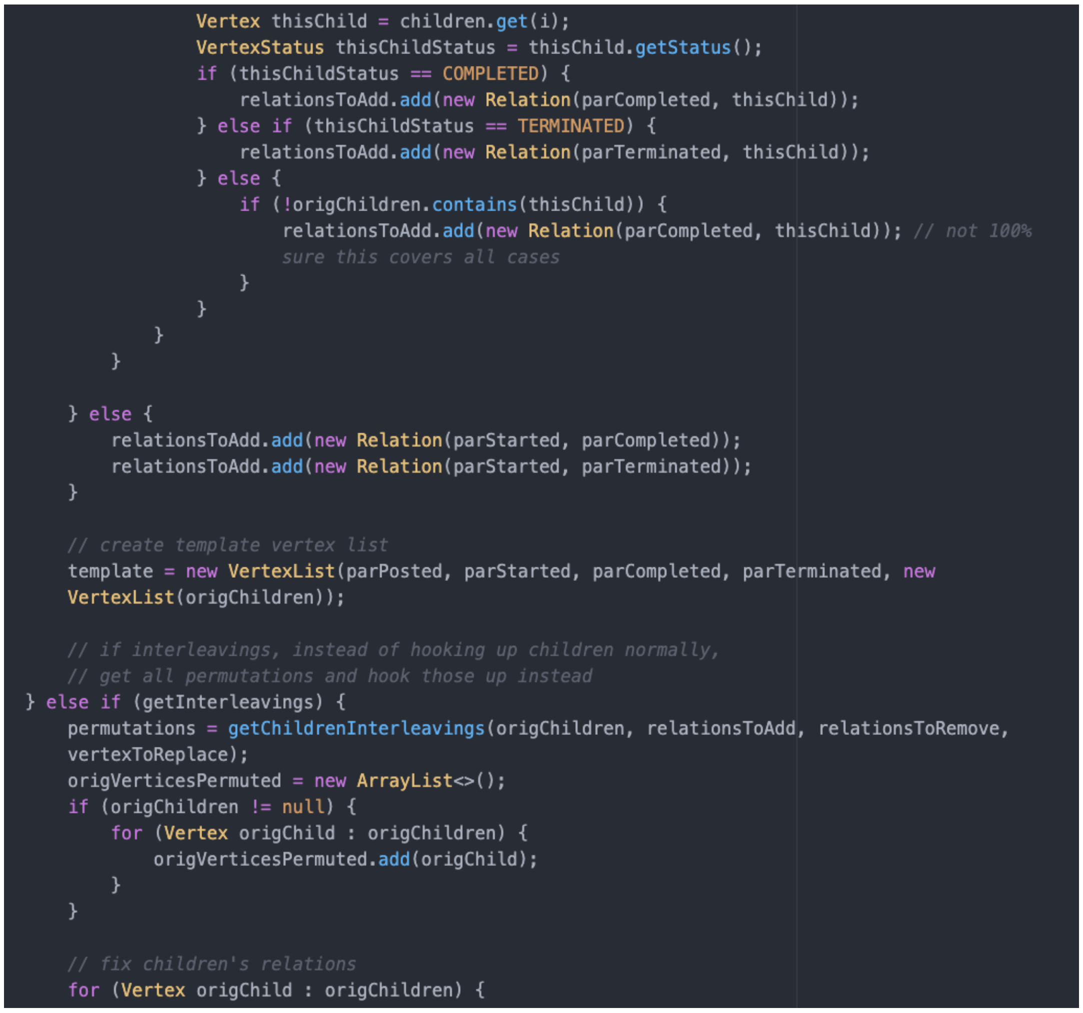
```
    Boolean isRoot = false;
    Boolean isOriginal = false;

    // create template vertices
    Vertex parPosted = new Vertex(number, "s" + number.toString(), kind, kindNum, POSTED, blurb,
    properties, labels, null, null,
            distanceFromRoot, siblingNum, parentSiblingNum, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);

    Vertex parStarted = new Vertex(++number, "s" + number, kind, kindNum, STARTED, blurb, properties,
    labels, null, null,
            ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);

    Vertex parCompleted = new Vertex(++number, "s" + number, kind, kindNum, COMPLETED, blurb,
    properties, labels, null, null,
            ++distanceFromRoot, 0, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);

    Vertex parTerminated = new Vertex(++number, "s" + number, kind, kindNum, TERMINATED, blurb, null,
    null, null, null,
            distanceFromRoot, 1, 0, origNumber, origParents, origChildren,
            origDistanceFromRoot, origSiblingNum, isRoot, isOriginal);


    // hook up first two template nodes to each other
    parPosted.addChild(parStarted);
    parStarted.addParent(parPosted);
    parPosted.addRelation(new Relation(parPosted, parStarted));

    // get relations for hooking up parents/children to template vertices
    ArrayList<Relation> relationsToAdd = new ArrayList<>();
    ArrayList<Relation> relationsToRemove = new ArrayList<>();

    // get relations for hooking up parents to template vertices
    if (parents != null) {
        for (Vertex parent : parents) {
            relationsToAdd.add(new Relation(parent, parPosted));
```

Figure 51. Parallel Template (2/6)



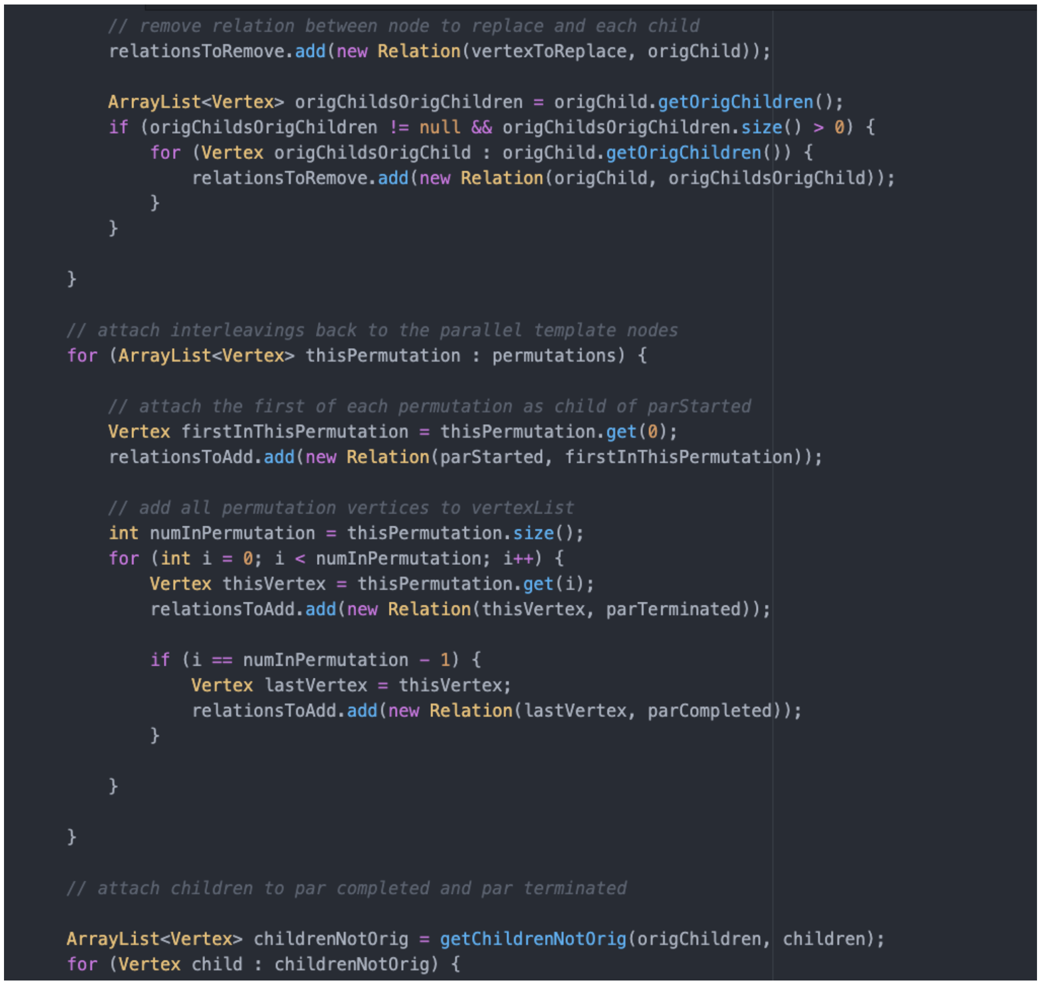
```
            relationsToRemove.add(new Relation(parent, vertexToReplace));
        }
    }

    // get relations for hooking up original children as substeps
    Integer numOrigChildren = (origChildren == null) ? null : origChildren.size();
    ArrayList<ArrayList<Vertex>> permutations = null;
    if (!getInterleavings) {
        if (origChildren != null && numOrigChildren != null) {
            for (Integer i = 0; i < numOrigChildren; i++) {
                Vertex thisSubstep = origChildren.get(i);
                relationsToRemove.add(new Relation(vertexToReplace, thisSubstep));
                relationsToAdd.add(new Relation(parStarted, thisSubstep));

                // remove substep's original children (they will be dealt with recursively in future
                steps)
                if (thisSubstep.getChildren() != null && thisSubstep.getChildren().size() > 0) {
                    for (Vertex thisSubstepChild : thisSubstep.getChildren()) {
                        relationsToRemove.add(new Relation(thisSubstep,thisSubstepChild));
                    }
                }

                for (Integer j = 0; j < numOrigChildren; j++) {
                    if (i != j) {
                        Vertex otherSubstep = origChildren.get(j);
                        relationsToAdd.add(new Relation(otherSubstep, thisSubstep));
                        relationsToAdd.add(new Relation(thisSubstep, otherSubstep));
                    }
                }
                thisSubstep.setParentSiblingNum(0);
                thisSubstep.setSiblingNum(i);
                relationsToAdd.add(new Relation(thisSubstep, parCompleted));
                relationsToAdd.add(new Relation(thisSubstep, parTerminated));
            }

            // hook up the children nodes from the translatedVertexList
            if (children != null) {
                for (int i=0; i<children.size(); i++) {
```

Figure 52. Parallel Template (3/6)



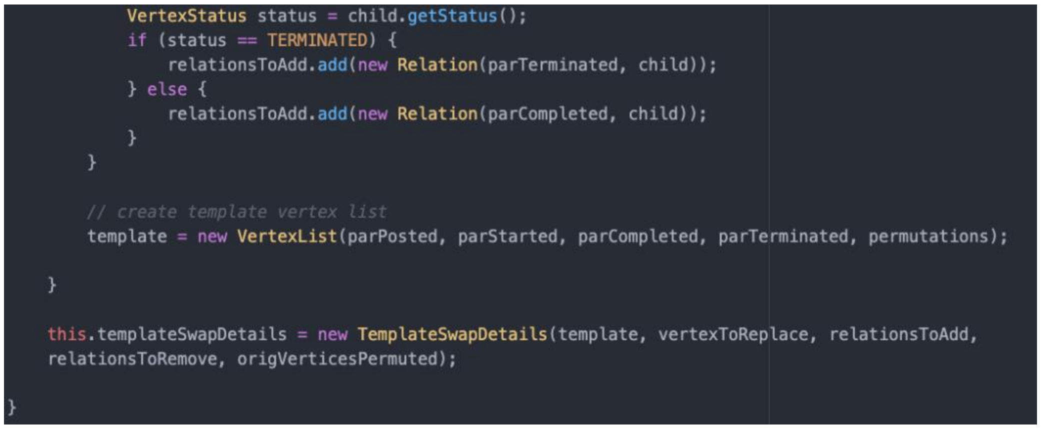
```
                Vertex thisChild = children.get(i);
                VertexStatus thisChildStatus = thisChild.getStatus();
                if (thisChildStatus == COMPLETED) {
                    relationsToAdd.add(new Relation(parCompleted, thisChild));
                } else if (thisChildStatus == TERMINATED) {
                    relationsToAdd.add(new Relation(parTerminated, thisChild));
                } else {
                    if (!origChildren.contains(thisChild)) {
                        relationsToAdd.add(new Relation(parCompleted, thisChild)); // not 100%
                        sure this covers all cases
                    }
                }
            }
        }

    } else {
        relationsToAdd.add(new Relation(parStarted, parCompleted));
        relationsToAdd.add(new Relation(parStarted, parTerminated));
    }

    // create template vertex list
    template = new VertexList(parPosted, parStarted, parCompleted, parTerminated, new
    VertexList(origChildren));

    // if interleavings, instead of hooking up children normally,
    // get all permutations and hook those up instead
} else if (getInterleavings) {
    permutations = getChildrenInterleavings(origChildren, relationsToAdd, relationsToRemove,
    vertexToReplace);
    origVerticesPermuted = new ArrayList<>();
    if (origChildren != null) {
        for (Vertex origChild : origChildren) {
            origVerticesPermuted.add(origChild);
        }
    }

    // fix children's relations
    for (Vertex origChild : origChildren) {
```

Figure 53. Parallel Template (4/6)



```
            // remove relation between node to replace and each child
            relationsToRemove.add(new Relation(vertexToReplace, origChild));

            ArrayList<Vertex> origChildsOrigChildren = origChild.getOrigChildren();
            if (origChildsOrigChildren != null && origChildsOrigChildren.size() > 0) {
                for (Vertex origChildsOrigChild : origChild.getOrigChildren()) {
                    relationsToRemove.add(new Relation(origChild, origChildsOrigChild));
                }
            }

        }

        // attach interleavings back to the parallel template nodes
        for (ArrayList<Vertex> thisPermutation : permutations) {

            // attach the first of each permutation as child of parStarted
            Vertex firstInThisPermutation = thisPermutation.get(0);
            relationsToAdd.add(new Relation(parStarted, firstInThisPermutation));

            // add all permutation vertices to vertexList
            int numInPermutation = thisPermutation.size();
            for (int i = 0; i < numInPermutation; i++) {
                Vertex thisVertex = thisPermutation.get(i);
                relationsToAdd.add(new Relation(thisVertex, parTerminated));

                if (i == numInPermutation - 1) {
                    Vertex lastVertex = thisVertex;
                    relationsToAdd.add(new Relation(lastVertex, parCompleted));
                }

            }

        }

        // attach children to par completed and par terminated

        ArrayList<Vertex> childrenNotOrig = getChildrenNotOrig(origChildren, children);
        for (Vertex child : childrenNotOrig) {
```

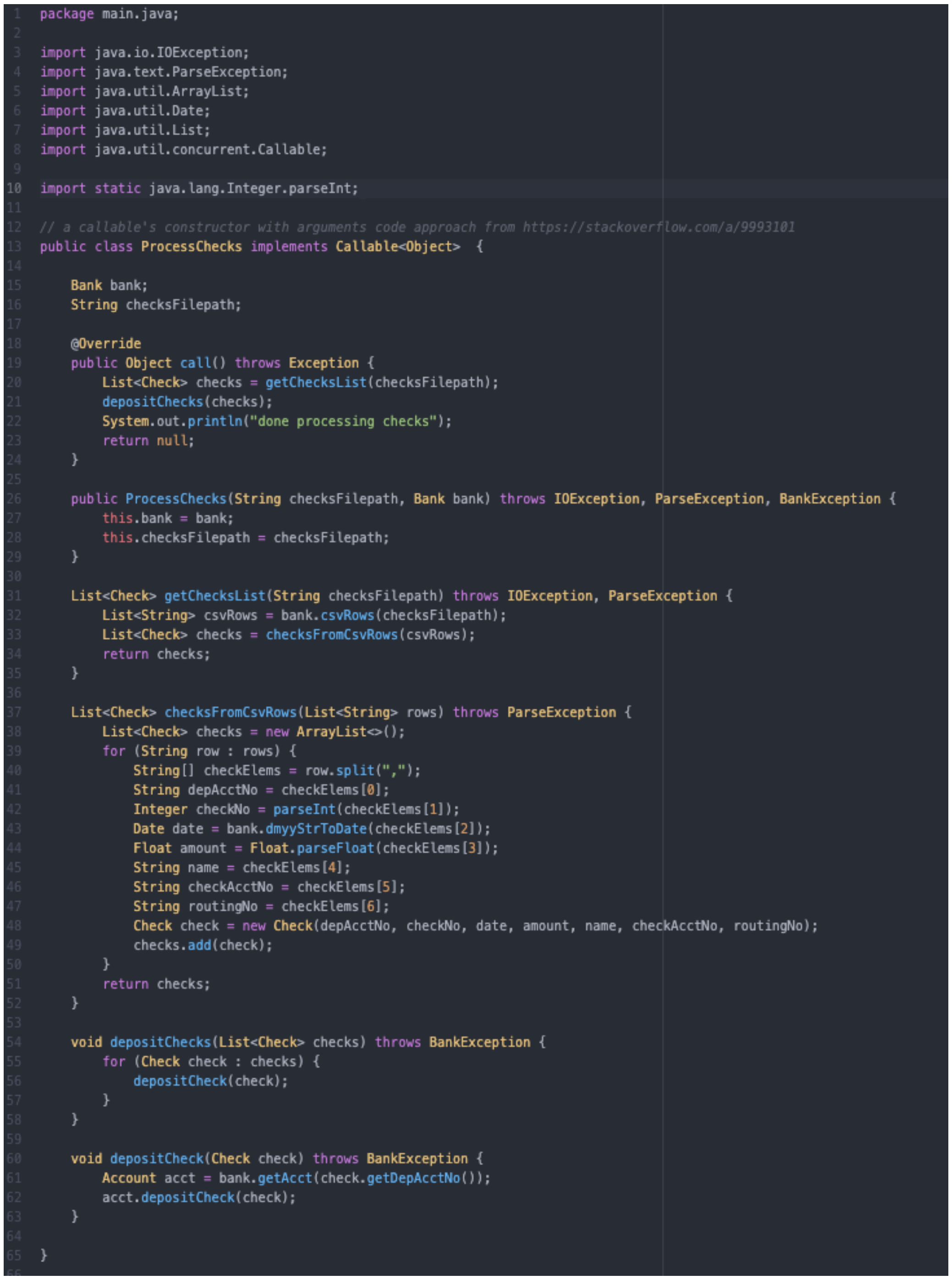
Figure 54. Parallel Template (5/6)



```
            VertexStatus status = child.getStatus();
            if (status == TERMINATED) {
                relationsToAdd.add(new Relation(parTerminated, child));
            } else {
                relationsToAdd.add(new Relation(parCompleted, child));
            }
        }

        // create template vertex list
        template = new VertexList(parPosted, parStarted, parCompleted, parTerminated, permutations);

    }

    this.templateSwapDetails = new TemplateSwapDetails(template, vertexToReplace, relationsToAdd,
    relationsToRemove, origVerticesPermuted);

}
```

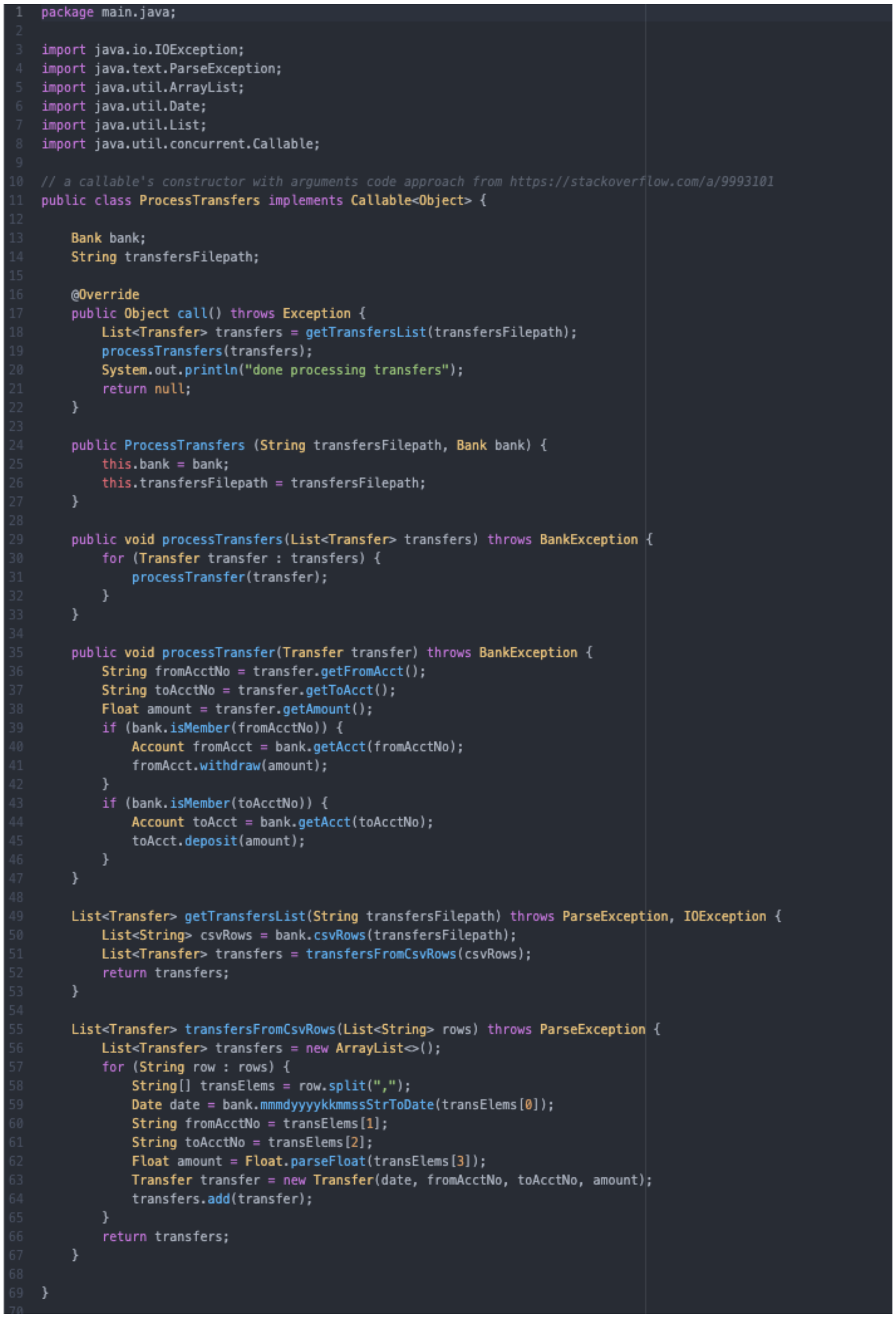
Figure 55. Parallel Template (6/6)

Appendix B: Banking Problem Code



```java
package main.java;

import java.io.IOException;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;

import static java.lang.Integer.parseInt;

// a callable's constructor with arguments code approach from https://stackoverflow.com/a/9993101
public class ProcessChecks implements Callable<Object>  {

    Bank bank;
    String checksFilepath;

    @Override
    public Object call() throws Exception {
        List<Check> checks = getChecksList(checksFilepath);
        depositChecks(checks);
        System.out.println("done processing checks");
        return null;
    }

    public ProcessChecks(String checksFilepath, Bank bank) throws IOException, ParseException, BankException {
        this.bank = bank;
        this.checksFilepath = checksFilepath;
    }

    List<Check> getChecksList(String checksFilepath) throws IOException, ParseException {
        List<String> csvRows = bank.csvRows(checksFilepath);
        List<Check> checks = checksFromCsvRows(csvRows);
        return checks;
    }

    List<Check> checksFromCsvRows(List<String> rows) throws ParseException {
        List<Check> checks = new ArrayList<>();
        for (String row : rows) {
            String[] checkElems = row.split(",");
            String depAcctNo = checkElems[0];
            Integer checkNo = parseInt(checkElems[1]);
            Date date = bank.dmyyStrToDate(checkElems[2]);
            Float amount = Float.parseFloat(checkElems[3]);
            String name = checkElems[4];
            String checkAcctNo = checkElems[5];
            String routingNo = checkElems[6];
            Check check = new Check(depAcctNo, checkNo, date, amount, name, checkAcctNo, routingNo);
            checks.add(check);
        }
        return checks;
    }

    void depositChecks(List<Check> checks) throws BankException {
        for (Check check : checks) {
            depositCheck(check);
        }
    }

    void depositCheck(Check check) throws BankException {
        Account acct = bank.getAcct(check.getDepAcctNo());
        acct.depositCheck(check);
    }

}
```

Figure 56. ProcessChecks.java



```java
package main.java;

import java.io.IOException;
import java.text.ParseException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.Callable;

// a callable's constructor with arguments code approach from https://stackoverflow.com/a/9993101
public class ProcessTransfers implements Callable<Object> {

    Bank bank;
    String transfersFilepath;

    @Override
    public Object call() throws Exception {
        List<Transfer> transfers = getTransfersList(transfersFilepath);
        processTransfers(transfers);
        System.out.println("done processing transfers");
        return null;
    }

    public ProcessTransfers (String transfersFilepath, Bank bank) {
        this.bank = bank;
        this.transfersFilepath = transfersFilepath;
    }

    public void processTransfers(List<Transfer> transfers) throws BankException {
        for (Transfer transfer : transfers) {
            processTransfer(transfer);
        }
    }

    public void processTransfer(Transfer transfer) throws BankException {
        String fromAcctNo = transfer.getFromAcct();
        String toAcctNo = transfer.getToAcct();
        Float amount = transfer.getAmount();
        if (bank.isMember(fromAcctNo)) {
            Account fromAcct = bank.getAcct(fromAcctNo);
            fromAcct.withdraw(amount);
        }
        if (bank.isMember(toAcctNo)) {
            Account toAcct = bank.getAcct(toAcctNo);
            toAcct.deposit(amount);
        }
    }

    List<Transfer> getTransfersList(String transfersFilepath) throws ParseException, IOException {
        List<String> csvRows = bank.csvRows(transfersFilepath);
        List<Transfer> transfers = transfersFromCsvRows(csvRows);
        return transfers;
    }

    List<Transfer> transfersFromCsvRows(List<String> rows) throws ParseException {
        List<Transfer> transfers = new ArrayList<>();
        for (String row : rows) {
            String[] transElems = row.split(",");
            Date date = bank.mmmdyyyykkmmssStrToDate(transElems[0]);
            String fromAcctNo = transElems[1];
            String toAcctNo = transElems[2];
            Float amount = Float.parseFloat(transElems[3]);
            Transfer transfer = new Transfer(date, fromAcctNo, toAcctNo, amount);
            transfers.add(transfer);
        }
        return transfers;
    }

}
```

Figure 57. ProcessTransfers.java

Appendix C: Large Translation and Interleavings Graphs

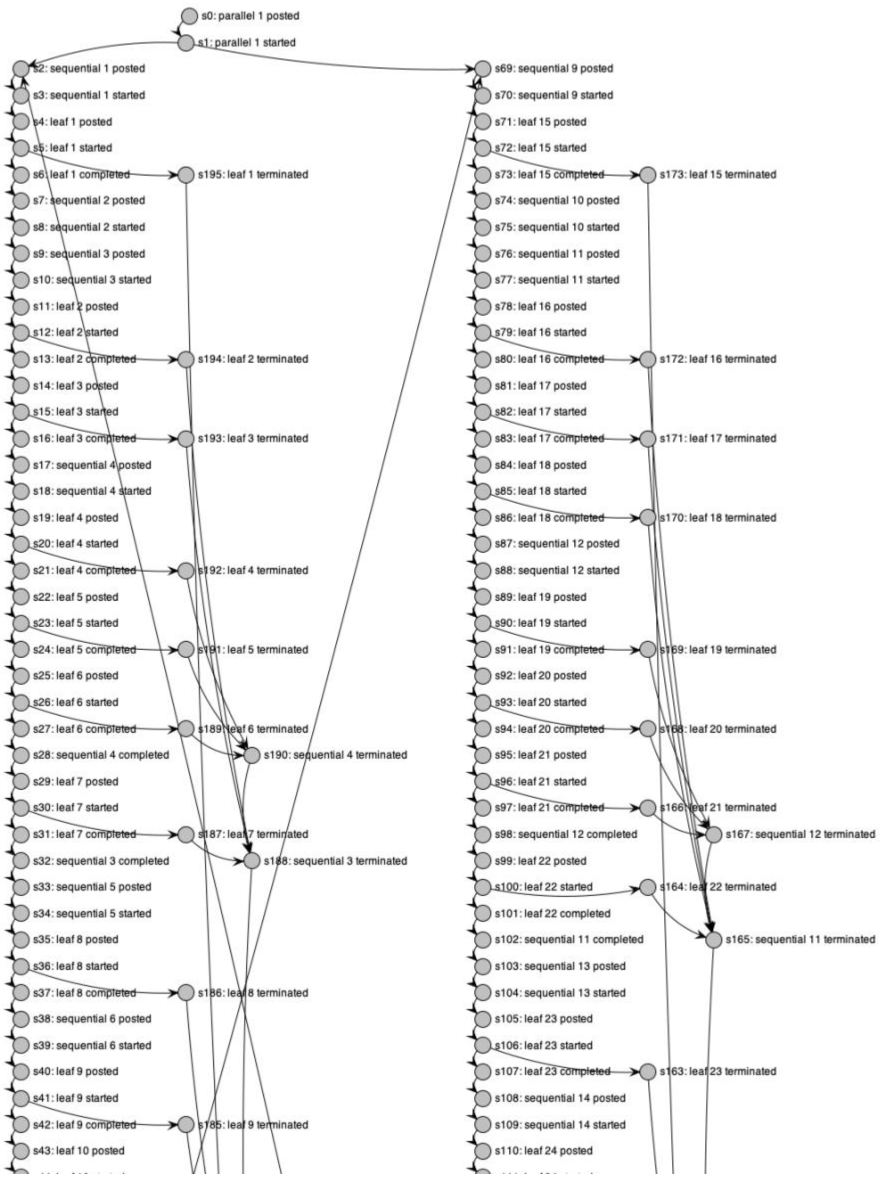


Figure 58. Banking Translation Graph (1/2)

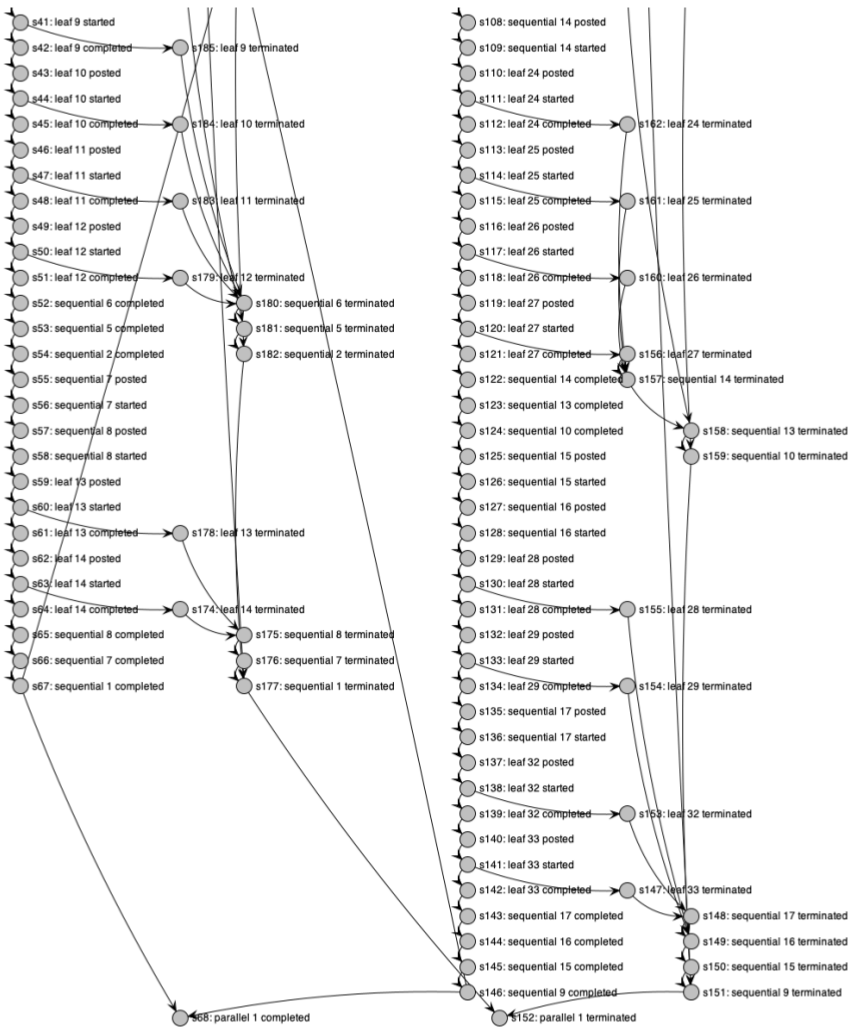
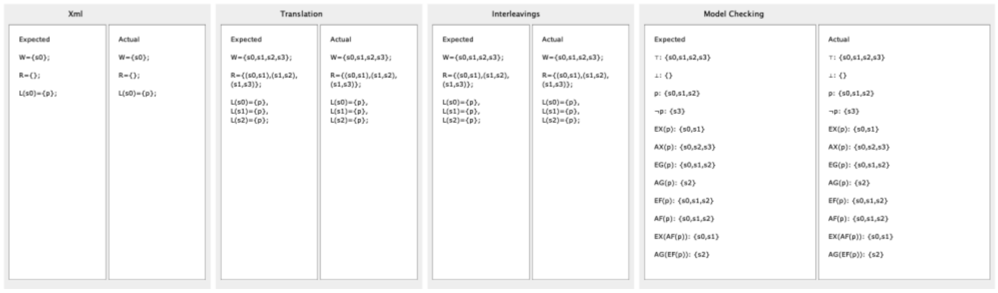


Figure 59. Banking Translation Graph (2/2)

Appendix D: Tests



Xml

```
Expected

W={s0};

R={};

L(s0)={p};
```

```
Actual

W={s0};

R={};

L(s0)={p};
```

Translation

```
Expected

W={s0,s1,s2,s3};

R={(s0,s1),(s1,s2),
(s1,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

```
Actual

W={s0,s1,s2,s3};

R={(s0,s1),(s1,s2),
(s1,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Interleavings

```
Expected

W={s0,s1,s2,s3};

R={(s0,s1),(s1,s2),
(s1,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

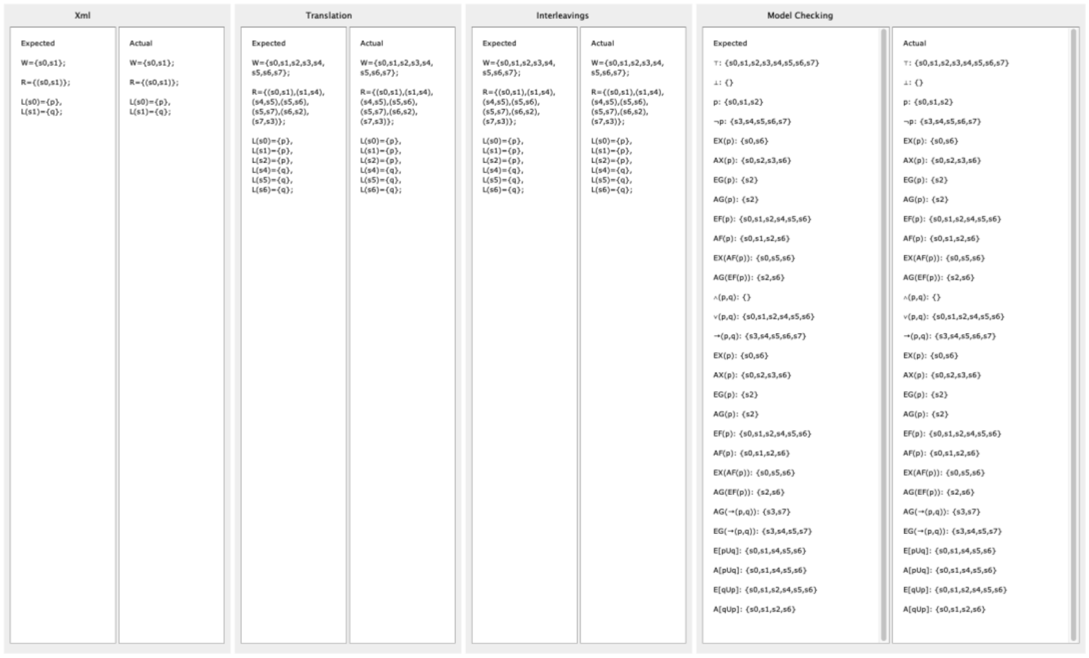
```
Actual

W={s0,s1,s2,s3};

R={(s0,s1),(s1,s2),
(s1,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Model Checking

```
Expected

⊤: {s0,s1,s2,s3}

⊥: {}

p: {s0,s1,s2}

¬p: {s3}

EX(p): {s0,s1}

AX(p): {s0,s2,s3}

EG(p): {s0,s1,s2}

AG(p): {s2}

EF(p): {s0,s1,s2}

AF(p): {s0,s1,s2}

EX(AF(p)): {s0,s1}

AG(EF(p)): {s2}
```

```
Actual

⊤: {s0,s1,s2,s3}

⊥: {}

p: {s0,s1,s2}

¬p: {s3}

EX(p): {s0,s1}

AX(p): {s0,s2,s3}

EG(p): {s0,s1,s2}

AG(p): {s2}

EF(p): {s0,s1,s2}

AF(p): {s0,s1,s2}

EX(AF(p)): {s0,s1}

AG(EF(p)): {s2}
```

Figure 60. One Step Test



Xml

```
Expected

W={s0,s1};

R={(s0,s1)};

L(s0)={p},
L(s1)={q};
```

```
Actual

W={s0,s1};

R={(s0,s1)};

L(s0)={p},
L(s1)={q};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s2),
(s7,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s2),
(s7,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s2),
(s7,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

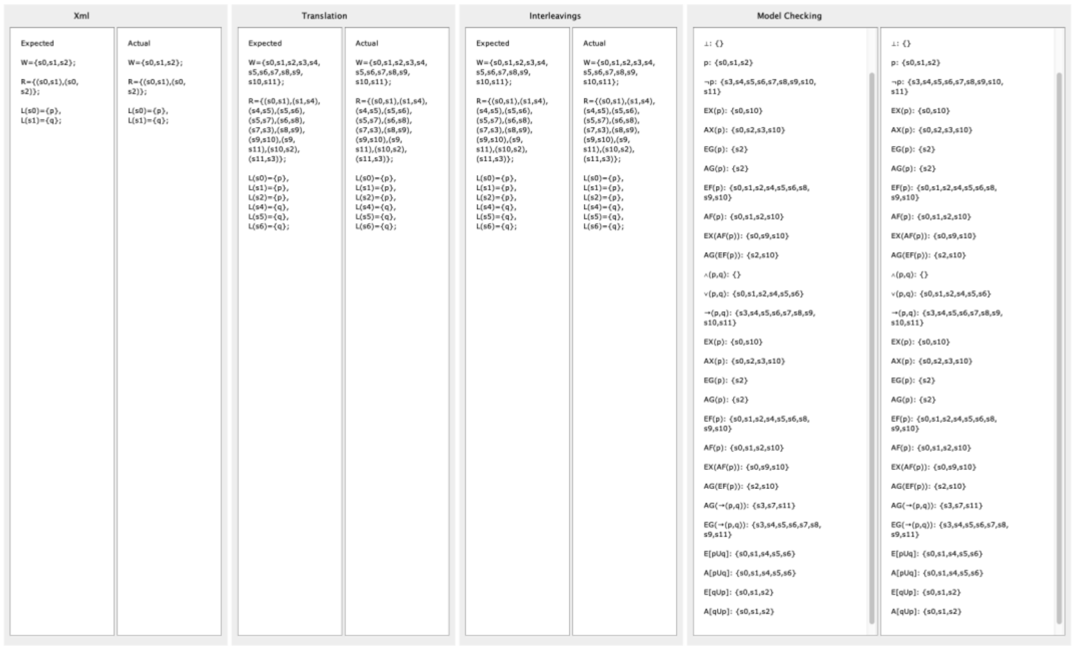
```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s2),
(s7,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Model Checking

```
Expected

⊤: {s0,s1,s2,s3,s4,s5,s6,s7}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7}

EX(p): {s0,s6}

AX(p): {s0,s2,s3,s6}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6}

AF(p): {s0,s1,s2,s6}

EX(AF(p)): {s0,s5,s6}

AG(EF(p)): {s2,s6}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7}

EX(p): {s0,s6}

AX(p): {s0,s2,s3,s6}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6}

AF(p): {s0,s1,s2,s6}

EX(AF(p)): {s0,s5,s6}

AG(EF(p)): {s2,s6}

AG(→(p,q)): {s3,s7}

EG(→(p,q)): {s3,s4,s5,s7}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2,s4,s5,s6}

A[qUp]: {s0,s1,s2,s6}
```

```
Actual

⊤: {s0,s1,s2,s3,s4,s5,s6,s7}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7}

EX(p): {s0,s6}

AX(p): {s0,s2,s3,s6}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6}

AF(p): {s0,s1,s2,s6}

EX(AF(p)): {s0,s5,s6}

AG(EF(p)): {s2,s6}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7}

EX(p): {s0,s6}

AX(p): {s0,s2,s3,s6}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6}

AF(p): {s0,s1,s2,s6}

EX(AF(p)): {s0,s5,s6}

AG(EF(p)): {s2,s6}

AG(→(p,q)): {s3,s7}

EG(→(p,q)): {s3,s4,s5,s7}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2,s4,s5,s6}

A[qUp]: {s0,s1,s2,s6}
```

Figure 61. Two Steps Test



Xml

```
Expected

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p},
L(s1)={q};
```

```
Actual

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p},
L(s1)={q};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

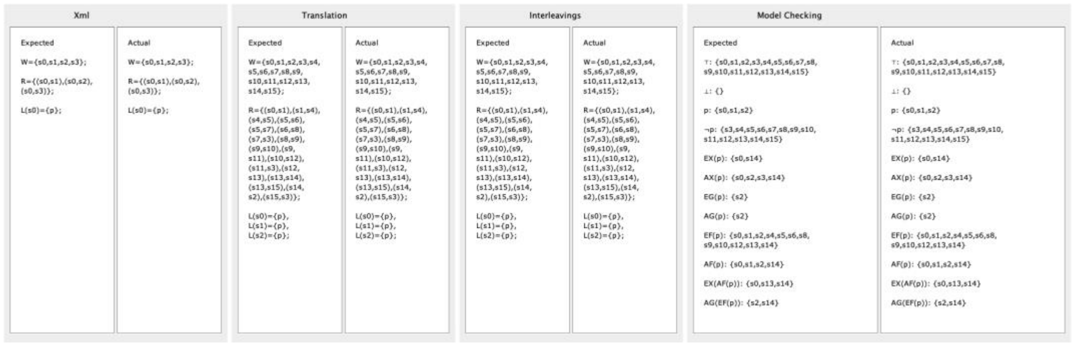
```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Model Checking

```
⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7,s8,s9,
s10,s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

AG(→(p,q)): {s3,s7,s11}

EG(→(p,q)): {s3,s4,s5,s6,s7,s8,
s9,s11}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2}

A[qUp]: {s0,s1,s2}
```

```
⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7,s8,s9,
s10,s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

AG(→(p,q)): {s3,s7,s11}

EG(→(p,q)): {s3,s4,s5,s6,s7,s8,
s9,s11}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2}

A[qUp]: {s0,s1,s2}
```

Figure 62. Three Steps Test



Xml

```
Expected

W={s0,s1,s2,s3};

R={(s0,s1),(s0,s2),
(s0,s3)};

L(s0)={p};
```

```
Actual

W={s0,s1,s2,s3};

R={(s0,s1),(s0,s2),
(s0,s3)};

L(s0)={p};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s12),
(s11,s3),(s12,
s13),(s13,s14),
(s13,s15),(s14,
s2),(s15,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s12),
(s11,s3),(s12,
s13),(s13,s14),
(s13,s15),(s14,
s2),(s15,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s12),
(s11,s3),(s12,
s13),(s13,s14),
(s13,s15),(s14,
s2),(s15,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

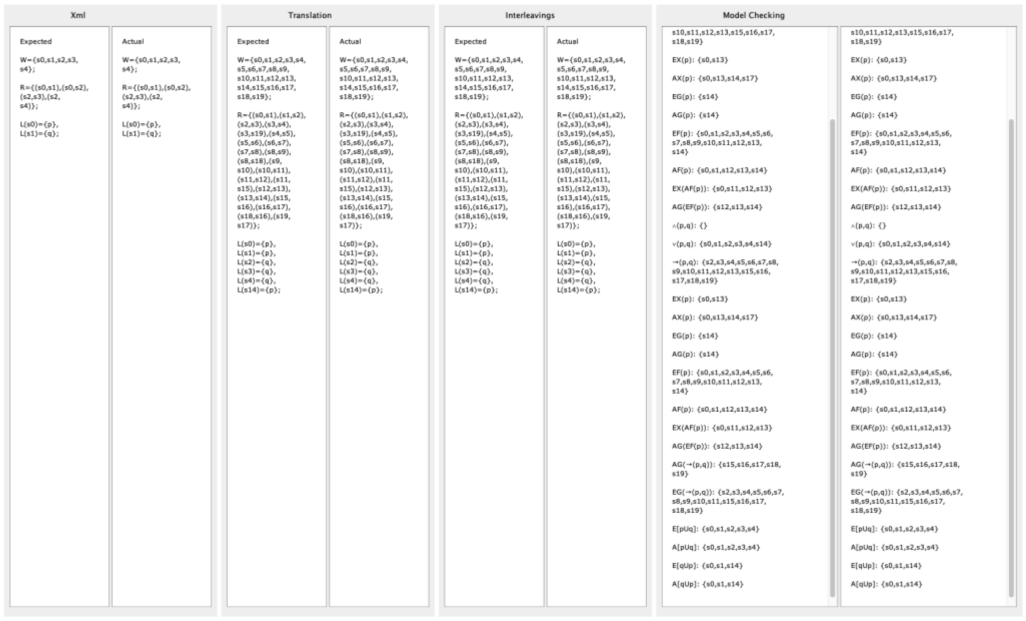
```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s12),
(s11,s3),(s12,
s13),(s13,s14),
(s13,s15),(s14,
s2),(s15,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Model Checking

```
Expected

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11,s12,s13,s14,s15}

EX(p): {s0,s14}

AX(p): {s0,s2,s3,s14}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10,s12,s13,s14}

AF(p): {s0,s1,s2,s14}

EX(AF(p)): {s0,s13,s14}

AG(EF(p)): {s2,s14}
```

```
Actual

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11,s12,s13,s14,s15}

EX(p): {s0,s14}

AX(p): {s0,s2,s3,s14}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10,s12,s13,s14}

AF(p): {s0,s1,s2,s14}

EX(AF(p)): {s0,s13,s14}

AG(EF(p)): {s2,s14}
```

Figure 63. Four Steps Test


Xml

Expected

```
W={s0,s1,s2,s3,
s4};

R={(s0,s1),(s0,s2),
(s2,s3),(s2,
s4)};

L(s0)={p},
L(s1)={q};
```

Actual

```
W={s0,s1,s2,s3,
s4};

R={(s0,s1),(s0,s2),
(s2,s3),(s2,
s4)};

L(s0)={p},
L(s1)={q};
```

Translation

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s19),(s4,s5),
(s5,s6),(s6,s7),
(s7,s8),(s8,s9),
(s8,s18),(s9,
s10),(s10,s11),
(s11,s12),(s11,
s15),(s12,s13),
(s13,s14),(s15,
s16),(s16,s17),
(s18,s16),(s19,
s17)};

L(s0)={p},
L(s1)={p},
L(s2)={q},
L(s3)={q},
L(s4)={q},
L(s14)={p};
```

Actual

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s19),(s4,s5),
(s5,s6),(s6,s7),
(s7,s8),(s8,s9),
(s8,s18),(s9,
s10),(s10,s11),
(s11,s12),(s11,
s15),(s12,s13),
(s13,s14),(s15,
s16),(s16,s17),
(s18,s16),(s19,
s17)};

L(s0)={p},
L(s1)={p},
L(s2)={q},
L(s3)={q},
L(s4)={q},
L(s14)={p};
```

Interleavings

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s19),(s4,s5),
(s5,s6),(s6,s7),
(s7,s8),(s8,s9),
(s8,s18),(s9,
s10),(s10,s11),
(s11,s12),(s11,
s15),(s12,s13),
(s13,s14),(s15,
s16),(s16,s17),
(s18,s16),(s19,
s17)};

L(s0)={p},
L(s1)={p},
L(s2)={q},
L(s3)={q},
L(s4)={q},
L(s14)={p};
```

Actual

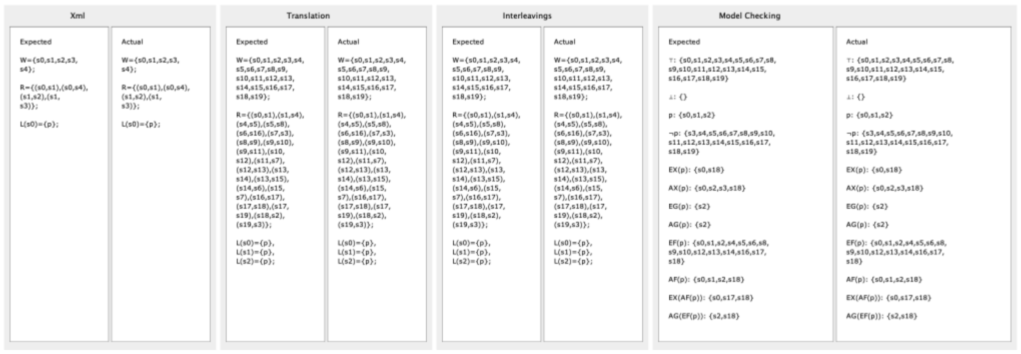
```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s19),(s4,s5),
(s5,s6),(s6,s7),
(s7,s8),(s8,s9),
(s8,s18),(s9,
s10),(s10,s11),
(s11,s12),(s11,
s15),(s12,s13),
(s13,s14),(s15,
s16),(s16,s17),
(s18,s16),(s19,
s17)};

L(s0)={p},
L(s1)={p},
L(s2)={q},
L(s3)={q},
L(s4)={q},
L(s14)={p};
```

Model Checking

```
s10,s11,s12,s13,s15,s16,s17,
s18,s19}

EX(p): {s0,s13}

AX(p): {s0,s13,s14,s17}

EG(p): {s14}

AG(p): {s14}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s9,s10,s11,s12,s13,
s14}

AF(p): {s0,s1,s12,s13,s14}

EX(AF(p)): {s0,s11,s12,s13}

AG(EF(p)): {s12,s13,s14}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s3,s4,s14}

→(p,q): {s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s15,s16,
s17,s18,s19}

EX(p): {s0,s13}

AX(p): {s0,s13,s14,s17}

EG(p): {s14}

AG(p): {s14}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s9,s10,s11,s12,s13,
s14}

AF(p): {s0,s1,s12,s13,s14}

EX(AF(p)): {s0,s11,s12,s13}

AG(EF(p)): {s12,s13,s14}

AG(→(p,q)): {s15,s16,s17,s18,
s19}

EG(→(p,q)): {s2,s3,s4,s5,s6,s7,
s8,s9,s10,s11,s15,s16,s17,
s18,s19}

E[pUq]: {s0,s1,s2,s3,s4}

A[pUq]: {s0,s1,s2,s3,s4}

E[qUp]: {s0,s1,s14}

A[qUp]: {s0,s1,s14}
```

```
s10,s11,s12,s13,s15,s16,s17,
s18,s19}

EX(p): {s0,s13}

AX(p): {s0,s13,s14,s17}

EG(p): {s14}

AG(p): {s14}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s9,s10,s11,s12,s13,
s14}

AF(p): {s0,s1,s12,s13,s14}

EX(AF(p)): {s0,s11,s12,s13}

AG(EF(p)): {s12,s13,s14}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s3,s4,s14}

→(p,q): {s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s15,s16,
s17,s18,s19}

EX(p): {s0,s13}

AX(p): {s0,s13,s14,s17}

EG(p): {s14}

AG(p): {s14}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s9,s10,s11,s12,s13,
s14}

AF(p): {s0,s1,s12,s13,s14}

EX(AF(p)): {s0,s11,s12,s13}

AG(EF(p)): {s12,s13,s14}

AG(→(p,q)): {s15,s16,s17,s18,
s19}

EG(→(p,q)): {s2,s3,s4,s5,s6,s7,
s8,s9,s10,s11,s15,s16,s17,
s18,s19}

E[pUq]: {s0,s1,s2,s3,s4}

A[pUq]: {s0,s1,s2,s3,s4}

E[qUp]: {s0,s1,s14}

A[qUp]: {s0,s1,s14}
```

Figure 64. Five Steps Test


Xml

Expected

```
W={s0,s1,s2,s3,
s4};

R={(s0,s1),(s0,s4),
(s1,s2),(s1,
s3)};

L(s0)={p};
```

Actual

```
W={s0,s1,s2,s3,
s4};

R={(s0,s1),(s0,s4),
(s1,s2),(s1,
s3)};

L(s0)={p};
```

Translation

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s8),
(s6,s16),(s7,s3),
(s8,s9),(s9,s10),
(s9,s11),(s10,
s12),(s11,s7),
(s12,s13),(s13,
s14),(s13,s15),
(s14,s6),(s15,
s7),(s16,s17),
(s17,s18),(s17,
s19),(s18,s2),
(s19,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Actual

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s8),
(s6,s16),(s7,s3),
(s8,s9),(s9,s10),
(s9,s11),(s10,
s12),(s11,s7),
(s12,s13),(s13,
s14),(s13,s15),
(s14,s6),(s15,
s7),(s16,s17),
(s17,s18),(s17,
s19),(s18,s2),
(s19,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Interleavings

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s8),
(s6,s16),(s7,s3),
(s8,s9),(s9,s10),
(s9,s11),(s10,
s12),(s11,s7),
(s12,s13),(s13,
s14),(s13,s15),
(s14,s6),(s15,
s7),(s16,s17),
(s17,s18),(s17,
s19),(s18,s2),
(s19,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Actual

```
W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s8),
(s6,s16),(s7,s3),
(s8,s9),(s9,s10),
(s9,s11),(s10,
s12),(s11,s7),
(s12,s13),(s13,
s14),(s13,s15),
(s14,s6),(s15,
s7),(s16,s17),
(s17,s18),(s17,
s19),(s18,s2),
(s19,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p};
```

Model Checking

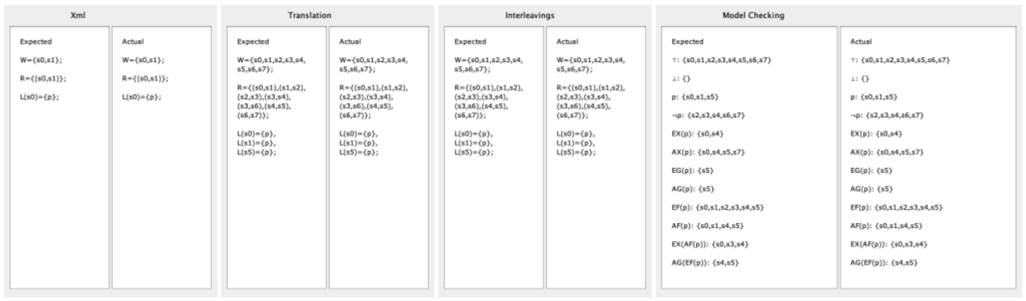
Expected

```
⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s18}

AX(p): {s0,s2,s3,s18}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10,s12,s13,s14,s16,s17,
s18}

AF(p): {s0,s1,s2,s18}

EX(AF(p)): {s0,s17,s18}

AG(EF(p)): {s2,s18}
```

Actual

```
⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s18}

AX(p): {s0,s2,s3,s18}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10,s12,s13,s14,s16,s17,
s18}

AF(p): {s0,s1,s2,s18}

EX(AF(p)): {s0,s17,s18}

AG(EF(p)): {s2,s18}
```

Figure 65. Sequential Test


Xml

Expected

```
W={s0,s1};

R={(s0,s1)};

L(s0)={p};
```

Actual

```
W={s0,s1};

R={(s0,s1)};

L(s0)={p};
```

Translation

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s6),(s4,s5),
(s6,s7)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Actual

```
W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s6),(s4,s5),
(s6,s7)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Interleavings

Expected

```
W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s6),(s4,s5),
(s6,s7)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Actual

```
W={s0,s1,s2,s3,s4,
s5,s6,s7};

R={(s0,s1),(s1,s2),
(s2,s3),(s3,s4),
(s3,s6),(s4,s5),
(s6,s7)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Model Checking

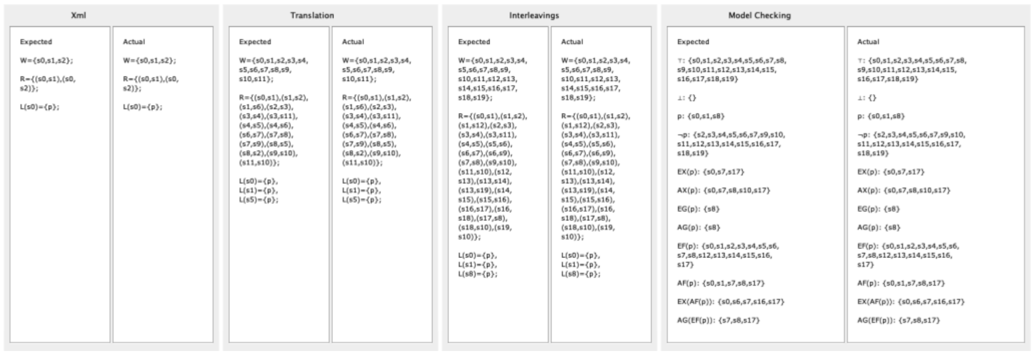
Expected

```
⊤: {s0,s1,s2,s3,s4,s5,s6,s7}

⊥: {}

p: {s0,s1,s5}

¬p: {s2,s3,s4,s6,s7}

EX(p): {s0,s4}

AX(p): {s0,s4,s5,s7}

EG(p): {s5}

AG(p): {s5}

EF(p): {s0,s1,s2,s3,s4,s5}

AF(p): {s0,s1,s4,s5}

EX(AF(p)): {s0,s3,s4}

AG(EF(p)): {s4,s5}
```

Actual

```
⊤: {s0,s1,s2,s3,s4,s5,s6,s7}

⊥: {}

p: {s0,s1,s5}

¬p: {s2,s3,s4,s6,s7}

EX(p): {s0,s4}

AX(p): {s0,s4,s5,s7}

EG(p): {s5}

AG(p): {s5}

EF(p): {s0,s1,s2,s3,s4,s5}

AF(p): {s0,s1,s4,s5}

EX(AF(p)): {s0,s3,s4}

AG(EF(p)): {s4,s5}
```

Figure 66. Parallel Two Steps Test



Xml

```
Expected

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p};
```

```
Actual

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s2),
(s1,s6),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s4,s6),
(s6,s7),(s7,s8),
(s7,s9),(s8,s5),
(s8,s2),(s9,s10),
(s11,s10)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s2),
(s1,s6),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s4,s6),
(s6,s7),(s7,s8),
(s7,s9),(s8,s5),
(s8,s2),(s9,s10),
(s11,s10)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s1,s12),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s5,s6),
(s6,s7),(s6,s9),
(s7,s8),(s9,s10),
(s11,s10),(s12,
s13),(s13,s14),
(s13,s19),(s14,
s15),(s15,s16),
(s16,s17),(s16,
s18),(s17,s8),
(s18,s10),(s19,
s10)};

L(s0)={p},
L(s1)={p},
L(s8)={p};
```

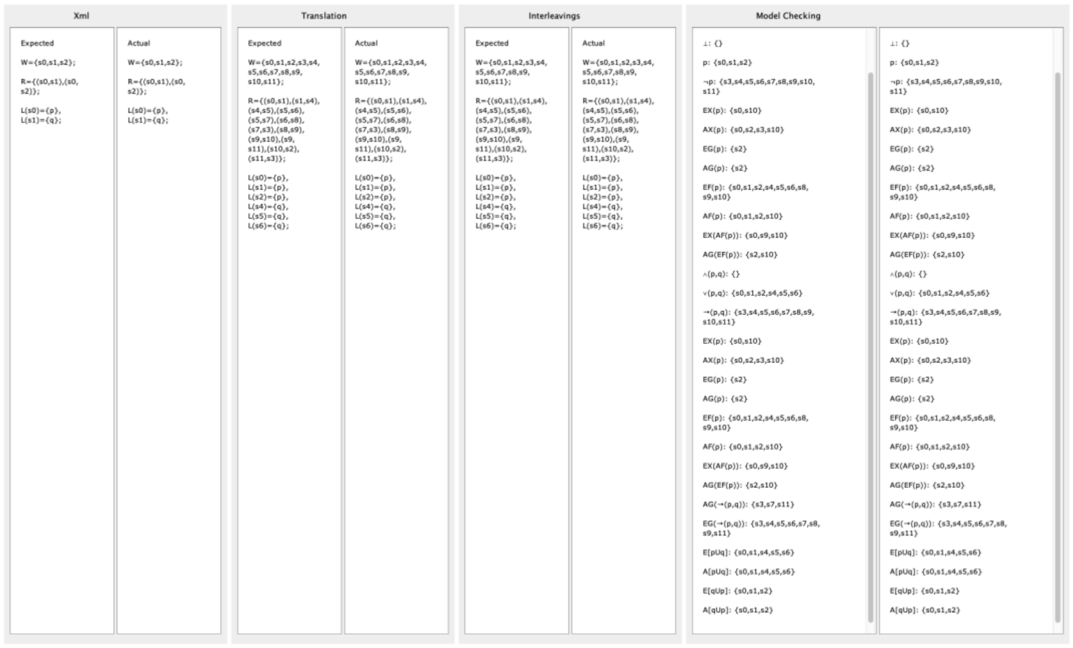
```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s1,s12),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s5,s6),
(s6,s7),(s6,s9),
(s7,s8),(s9,s10),
(s11,s10),(s12,
s13),(s13,s14),
(s13,s19),(s14,
s15),(s15,s16),
(s16,s17),(s16,
s18),(s17,s8),
(s18,s10),(s19,
s10)};

L(s0)={p},
L(s1)={p},
L(s8)={p};
```

Model Checking

```
Expected

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s8}

¬p: {s2,s3,s4,s5,s6,s7,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s7,s17}

AX(p): {s0,s7,s8,s10,s17}

EG(p): {s8}

AG(p): {s8}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s12,s13,s14,s15,s16,
s17}

AF(p): {s0,s1,s7,s8,s17}

EX(AF(p)): {s0,s6,s7,s16,s17}

AG(EF(p)): {s7,s8,s17}
```

```
Actual

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s8}

¬p: {s2,s3,s4,s5,s6,s7,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s7,s17}

AX(p): {s0,s7,s8,s10,s17}

EG(p): {s8}

AG(p): {s8}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s12,s13,s14,s15,s16,
s17}

AF(p): {s0,s1,s7,s8,s17}

EX(AF(p)): {s0,s6,s7,s16,s17}

AG(EF(p)): {s7,s8,s17}
```

Figure 67. Parallel Three Steps Test



Xml

```
Expected

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p},
L(s1)={q};
```

```
Actual

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p},
L(s1)={q};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

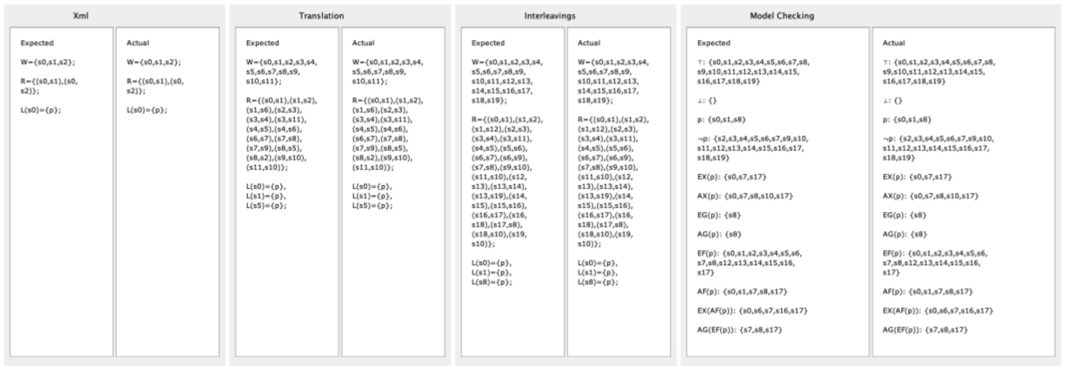
```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s4),
(s4,s5),(s5,s6),
(s5,s7),(s6,s8),
(s7,s3),(s8,s9),
(s9,s10),(s9,
s11),(s10,s2),
(s11,s3)};

L(s0)={p},
L(s1)={p},
L(s2)={p},
L(s4)={q},
L(s5)={q},
L(s6)={q};
```

Model Checking

```
⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7,s8,s9,
s10,s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

AG(→(p,q)): {s3,s7,s11}

EG(→(p,q)): {s3,s4,s5,s6,s7,s8,
s9,s11}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2}

A[qUp]: {s0,s1,s2}
```

```
⊥: {}

p: {s0,s1,s2}

¬p: {s3,s4,s5,s6,s7,s8,s9,s10,
s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

∧(p,q): {}

∨(p,q): {s0,s1,s2,s4,s5,s6}

→(p,q): {s3,s4,s5,s6,s7,s8,s9,
s10,s11}

EX(p): {s0,s10}

AX(p): {s0,s2,s3,s10}

EG(p): {s2}

AG(p): {s2}

EF(p): {s0,s1,s2,s4,s5,s6,s8,
s9,s10}

AF(p): {s0,s1,s2,s10}

EX(AF(p)): {s0,s9,s10}

AG(EF(p)): {s2,s10}

AG(→(p,q)): {s3,s7,s11}

EG(→(p,q)): {s3,s4,s5,s6,s7,s8,
s9,s11}

E[pUq]: {s0,s1,s4,s5,s6}

A[pUq]: {s0,s1,s4,s5,s6}

E[qUp]: {s0,s1,s2}

A[qUp]: {s0,s1,s2}
```

Figure 68. Sequential With Two Substeps Test



Xml

```
Expected

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p};
```

```
Actual

W={s0,s1,s2};

R={(s0,s1),(s0,
s2)};

L(s0)={p};
```

Translation

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s2),
(s1,s6),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s4,s6),
(s6,s7),(s7,s8),
(s7,s9),(s8,s5),
(s8,s2),(s9,s10),
(s11,s10)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11};

R={(s0,s1),(s1,s2),
(s1,s6),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s4,s6),
(s6,s7),(s7,s8),
(s7,s9),(s8,s5),
(s8,s2),(s9,s10),
(s11,s10)};

L(s0)={p},
L(s1)={p},
L(s5)={p};
```

Interleavings

```
Expected

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s1,s12),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s5,s6),
(s6,s7),(s6,s9),
(s7,s8),(s9,s10),
(s11,s10),(s12,
s13),(s13,s14),
(s13,s19),(s14,
s15),(s15,s16),
(s16,s17),(s16,
s18),(s17,s8),
(s18,s10),(s19,
s10)};

L(s0)={p},
L(s1)={p},
L(s8)={p};
```

```
Actual

W={s0,s1,s2,s3,s4,
s5,s6,s7,s8,s9,
s10,s11,s12,s13,
s14,s15,s16,s17,
s18,s19};

R={(s0,s1),(s1,s2),
(s1,s12),(s2,s3),
(s3,s4),(s3,s11),
(s4,s5),(s5,s6),
(s6,s7),(s6,s9),
(s7,s8),(s9,s10),
(s11,s10),(s12,
s13),(s13,s14),
(s13,s19),(s14,
s15),(s15,s16),
(s16,s17),(s16,
s18),(s17,s8),
(s18,s10),(s19,
s10)};

L(s0)={p},
L(s1)={p},
L(s8)={p};
```

Model Checking

```
Expected

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s8}

¬p: {s2,s3,s4,s5,s6,s7,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s7,s17}

AX(p): {s0,s7,s8,s10,s17}

EG(p): {s8}

AG(p): {s8}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s12,s13,s14,s15,s16,
s17}

AF(p): {s0,s1,s7,s8,s17}

EX(AF(p)): {s0,s6,s7,s16,s17}

AG(EF(p)): {s7,s8,s17}
```

```
Actual

⊤: {s0,s1,s2,s3,s4,s5,s6,s7,s8,
s9,s10,s11,s12,s13,s14,s15,
s16,s17,s18,s19}

⊥: {}

p: {s0,s1,s8}

¬p: {s2,s3,s4,s5,s6,s7,s9,s10,
s11,s12,s13,s14,s15,s16,s17,
s18,s19}

EX(p): {s0,s7,s17}

AX(p): {s0,s7,s8,s10,s17}

EG(p): {s8}

AG(p): {s8}

EF(p): {s0,s1,s2,s3,s4,s5,s6,
s7,s8,s12,s13,s14,s15,s16,
s17}

AF(p): {s0,s1,s7,s8,s17}

EX(AF(p)): {s0,s6,s7,s16,s17}

AG(EF(p)): {s7,s8,s17}
```

Figure 69. Parallel With Two Substeps Test

Appendix E: Table Data Sources

Table 3. Data Source With Both COVID Results

Problem	Model	Number of Nodes	Translation Time	Model Check Time	Translation + Model Check Time
COVID	AG¬(r∧v)	1635	0.050548	0.018945	0.069493
COVID	AG(p→AF(q))	1635	0.047599	0.08459	0.132189
COVID	average of both AG¬(r∧v) & AG(p→AF(q))	1635	0.0490735	0.0517675	0.100841
Banking-Parallel	A[¬p Uq∧¬r]	379	0.015728	0.002371	0.018099
Parallel & 2 Leaves	AX(p)	19	0.003323	0.000157	0.00348
Parallel & 3 Leaves	EF(p)	75	0.002486	0.000228	0.002714
Parallel & 4 Leaves	AG(p)	387	0.002652	0.000609	0.003261
Parallel & 5 Leaves	AX(p)	2403	0.093726	0.043702	0.137428

Table 4. Data Source With Averaged COVID Results

Problem	Model	Number of Nodes	Translation Time	Model Check Time	Translation + Model Check Time
Parallel & 2 Leaves	AX(p)	19	0.003323	0.000157	0.00348
Parallel & 3 Leaves	EF(p)	75	0.002486	0.000228	0.002714
Banking-Parallel	A[¬p Uq∧¬r]	379	0.015728	0.002371	0.018099

Parallel & 4 Leaves	AG(p)	387	0.002652	0.000609	0.003261
COVID	average of both AG¬(r∧v) & AG(p→AF(q))	1635	0.0490735	0.0517675	0.100841
Parallel & 5 Leaves	AX(p)	2403	0.093726	0.043702	0.137428

REFERENCES

[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] Cobleign and J. M. Cobleigh, "Verifying Properties of Process Definitions," *Computer Science Department Faculty Publication Series,* vol. 119, 2000.

[4] Laboratory for Advanced Software Engineering Research, "Little-JIL," University of Massachusetts Amherst, 2006. [Online]. Available: http://laser.cs.umass.edu/tools/littlejil.shtml. [Accessed 14 October 2020].

[5] S. M. Sutton and L. J. Osterweil, "The Design of a Next-Generation Process Language*," *SOFTWARE ENGINEERING - ESEC/FSE '97,* vol. 1301, pp. 142-158, 1997.

[6] 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].

[7] 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].

[8] Oracle, "About the JFC and Swing," 2019. [Online]. Available: https://docs.oracle.com/javase/tutorial/uiswing/start/about.html. [Accessed 2 August 2020].

[9] SourceForge, "JUNG Java Universal Network/Graph Framework," 2010. [Online]. Available: http://jung.sourceforge.net/. [Accessed 2 August 2020].

[10] T. Parr, "Antlr," 2014. [Online]. Available: https://www.antlr.org/. [Accessed 2 August 2020].

[11] J. M. Cobleigh, "Verifying Properties of Process Definitions," University of Massachusetts, Amherst, 2000.

[12] 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].

[13] 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].

[14] 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].

[15] 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].

[16] 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].

[17] 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].

[18] 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.

[19] 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].

[20] Geeks For Geeks, "Write a program to print all permutations of a given string," 3 December 2020. [Online]. Available: https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/. [Accessed 12 April 2021].

[21] B. Lerner, "Mount Holyoke College," July 2004. [Online]. Available: https://www.mtholyoke.edu/~blerner/papers/issta04.pdf. [Accessed 19 March 2021].

[22] A. G. C. L. O. L. Chen B., "Automatic Fault Tree Derivation from Little-JIL Process Definitions," in *SPW*, Berlin, 2006.

[23] S. Rura and B. Lerner, "Flexible Static Semantic Checking Using First-Order Logic," in *Software Process Technology*, Helsinki, 2003.

[24] J. M. Cobleigh, L. A. Clarke and L. J. Osterweil, "Verifying Properties of Process Defini," *Computer Science Department Faculty Publication Series,* p. 119, 2000.

[25] A. Gupta, "Model Checking Concurrent Programs," in *Verification, Model Checking, and Abstract Interpretation*, Savannah, 2009.

[26] M. Mongiello and D. Castelluccia, "Modelling and verification of BPEL business processes," in *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2006.

[27] C. P. Cheng, T. Fristoe and E. A. Lee, "Applied Verification: The Ptolemy Approach," Electrical Engineering and Computer Sciences University of California at Berkeley, Berkeley, 2008.

[28] L. Yin, W. Dong, W. Liu and J. Wang, "Parallel Refinement for Multi-Threaded Program Verification," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[29] 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.

[30] M. F. Atig, A. Bouajjani and T. Touili, "On the Reachability Analysis of Acyclic Networks of Pushdown Systems," LIAFA, CNRS & Univ. of Paris 7, Paris, 2008.