Analysis of Commutativity with state-chart graph representation of concurrent programs.

2016

Kishore Debnath
University of Central Florida

ANALYSIS OF COMMUTATIVITY WITH STATE-CHART GRAPH REPRESENTATION OF CONCURRENT PROGRAMS

by

KISHORE DEBNATH
B.Engg. Sathyabama University, 2012

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering & Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2016

ABSTRACT

We present a new approach to check for Commutativity in concurrent programs from their run-time state-chart graphs. Two operations are said to be commutative if changing the order of their execution do not change the resultant effect on the working object. Commutative property is capable of boosting performance in concurrent transactions such that transactional concurrency is comparable to a non-blocking linearizable version of a similar data structure type. Transactional concurrency is a technique that analyses object semantics, as object states, to determine conflicts and recovery between conflicting operations [7]. Processes that commute at object level can be executed concurrently at transaction level without conflicting with one another. In our approach we generate graphs by tracking run-time execution of concurrent program and representing object states in all possible thread interleavings as states and transitions. Using state-chart notations, we capture the object states at each execution step and compare their states before and after transitions as processed by a known set of operations reordered in different ways. Considering the non-deterministic nature of concurrent programs, we exhaustively capture program states during method invocation, method response, atomic instruction execution, etc., for determining commutativity. This helps user to examine state transitions at a thread level granularity, across all possible interleavings. With this methodology user can not only verify commutativity, but also can visually check ways in which functions commute at object level, which is an edge over current state-of-the art tools. The object level commutative information helps in identifying faulty implementations and performance improving considerations. We use a graph database to represent state nodes that further assists to check for other concurrency properties using cypher queries.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In the past decade, there has been a paradigm shift in high-performance programming as we saw Moore's law started to diminish. The quest for high performance scalable solutions has compelled developers to make better use of the new class of processors that comes with multiple cores. As developers started focusing on multithreaded programming models, non-blocking programming soon became a popular design choice over poorly scalable blocking solutions. Software Transactional Memory (STM)[2][3], an alternative to traditional mutual exclusion constructs, emerged as a solution that seemed to provide scalable solutions to concurrent objects. This research is motivated by the need for developers to check if their implementations can achieve a higher degree of concurrency by allowing commutative operations to execute simultaneously as transactions.

Herlihy proposed Transactional Boosting [2] as a technique for transforming concurrent linearizable objects into concurrent transactional objects. From a concurrent program perspective, transactions can be considered as a collection of one or more concurrent operations or functions. When two operations work on non-conflicting memory locations, performance gets improved by running them simultaneously over a synchronized execution due to the elimination of unnecessary synchronization. Applying this logic to transactions, the allowance of non-conflicting transactions to execute simultaneously also improves performance better than their strictly synchronized counterparts. Herlihy points out that a set of concurrent operations that can commute with each other can be executed as transaction concurrently and hence can scale better than synchronized transactions.

Two operations are commutative to one another if they leave same resultant effect on the semantic data-structure or object at abstract level[6], no matter how they reorder themselves during program run-time. For example, suppose there is a Set data-structure at abstract level, built on top of a linked-list at concrete level. If there are two add() operations that inserts elements X and Y into

the Set, then irrespective of their run-time ordering, the resultant effect on the Set data-structure is same. Also in the same resultant state, it do not matters how X and Y are linked with one another (if X is following Y or if Y is following X) in the underlying concrete linked-list. By the virtue of commutative property, execution of multiple transactions can take effect simultaneously without any synchronizing between themselves. In this paper, we present a new way of verifying commutativity by representing concurrent programs as state-chart graphs. With state-chart graph representation of concurrent program executions, users can exhaustively check ways in which functions commute at thread level granularity which further assist developers to find faulty operations and performance considerations. For this purpose, we designed a tool called *Concordant* that runs on top of any model-checker, and converts concurrent run-time model data into states and transitions. Model-data features like thread-create, thread-finish, atomic-read, atomic-write, method-invoke, method-return etc. are represented as action node in state-chart graphs. Here we use the terms *Action node* or *States*, and *Transitions* or *relationships* interchangeably.

Design and implementation of correct concurrent program is a challenging task. What is more challenging is to verify proven scalable properties like commutativity during design phase. Action node ordering in our state-chart conveys information such as how freely atomic operations execute across multiple execution traces. Additionally, by probing object memory locations of atomic operations and representing object states into nodes, we help developer compare program run-time behavior across multiple traces and investigate error in the code.

Fig. 1.1 is an example scenario of a faulty remove() method on a Set object, built on top of a linked-list structure. Fig. 1.2 shows the fault remove method implementation for the concurrent linked-list from [15] in C++. We use this linked-list as an underlying structure for a Set abstract state. Although this remove() operation commutes with any ordering of an add() operation, the commutative case here is conceptually flawed due to wrong implementation of remove() operation. Consider a linked-list is initially empty and an add() operation trying to insert X and a remove() op-

eration competing to remove *X* concurrently during execution. If the add() operation precedes the remove() operation in an execution, then the linked-list remains empty in the final state. Whereas, if the remove() precedes the add() operation then the linked-list will contain element *X* at the end of the execution, shown in fig. 1.4. Evidently, these two operations cannot commute at run-time as final state do not remains same at object-level across all execution trace. Using state-chart graph, we verify this fallacy by evaluating atomic action nodes in our state-chart graph. The memory locations of shared objects from action node help us deduced this error in variable *curr* (current node of linked-list), which was not atomically marking logic-bit for deletion. We fixed this error in fig. 1.3 by using an intermediary *temp* variable to logically mark the current node for deletion, and to ensure all other operations gets notified about the change in an atomic instance.

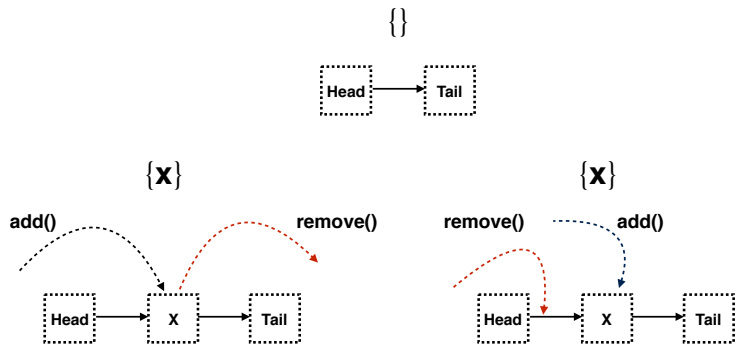


Figure 1.1: Faulty remove() function on an abstract Set object, with concurrent linked-list at concrete level.

```
bool remove(int item)
{
    bool snip;
    int key = item;
    while (true)
    {
        Window window = find(head, key);
        node *pred = window.pred;
        node *curr = window.curr;
        if (curr->key != key)
        {
            return false;
        }
        else
        {
            node *succ = curr->next.load();
            curr = SET_NODE_MARKED(curr);
            snip = ALIGN_NODE_ADDRESS(curr)->next.
                compare_exchange_strong(succ,
                ALIGN_NODE_ADDRESS(curr)->next.load());
            if (!snip) continue;
            ALIGN_NODE_ADDRESS(pred)->next.
                compare_exchange_strong(temp, succ);
            return true;
        }
    }
}
```

Figure 1.2: Faulty remove() function for Concurrent linked list.

```
bool remove(int item)
{
    bool snip;
    int key = item;
    while (true)
    {
        Window window = find(head, key);
        node *pred = window.pred;
        node *curr = window.curr;
        if (curr->key != key)
        {
            return false;
        }
        else
        {
            node *succ = ALIGN_NODE_ADDRESS(curr)->next.load();
            node* temp = curr;
            SET_NODE_MARKED(temp);
            snip = ALIGN_NODE_ADDRESS(pred)->next.
                compare_exchange_strong(curr, temp);
            if (!snip) continue;
            ALIGN_NODE_ADDRESS(pred)->next.
                compare_exchange_strong(temp, succ);
            return true;
        }
    }
}
```

Figure 1.3: Rectified remove() function for Concurrent linked list.

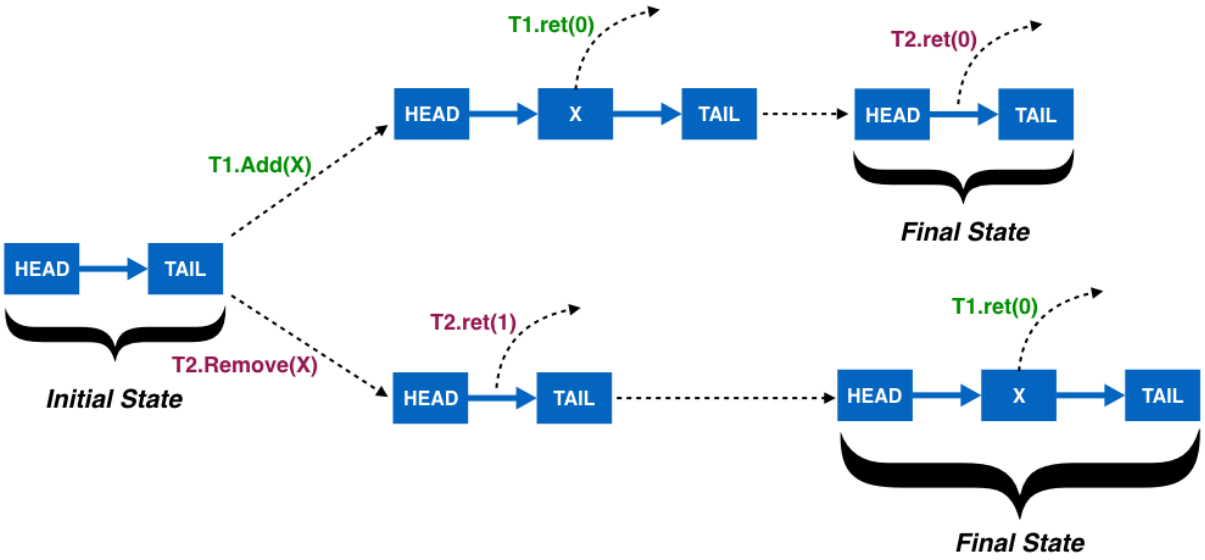


Figure 1.4: Linked-list object initially empty, gets concurrently executed by add() and remove() operation. In one trace, add() operation precedes remove(), and in final state linked-list remain empty. Conversely, in another trace where remove() preceded add(), leaves linked-list with one element in it.

Our idea of state-chart diagram is derived from the notion of trace theory concepts like dependency-graphs and concurrent histories [4][5], which forms mathematical model to evaluate properties of concurrent systems. Using trace theory, we explore different commutative instances using abstract symbols, which we then apply in our state-charts to derive the concept of *preconditional* and *post-conditional states* that we discuss in section 2. Preconditional and postconditional states are verification points to check commutativity in functions. We use graph database Neo4j [13] to represent concurrent model states as nodes and model-checker properties as node properties. This technique for commutativity checks are not done in any state-of-the art tools before. The advantages of using graph database is that we can further analyze each of the commutative functions across all traces using Cypher [14] query language. Cypher query language is a SQL alternative for graph databases. In later sections of this paper we present ways to use cypher query language (hereafter will be referred as cypher, cypher query) for searching different program properties related to com-

mutativity. State-chart graph representation of concurrent program opens a new research discourse to analyze different characteristics of non-blocking programs, some of which are presented in this paper.

This paper makes the following contributions:

- Analyzing commutative conditions on individual trace basis and compare them pair-wise to check ways in which atomic operations interleave to commute. Atomic operations are represented as action nodes in a state-chart graph, which we further discuss in section 2 and section 3.
- Verifying commutativity in data-structure classes: Contiguous and Non-contiguous memory discussed in section 3. This is done by defining *preconditional* and *postconditional states* for state-chart graphs to capture run-time semantics of concurrent objects to check commutativity, discussed in section 2.
- Representing concurrent programs in state-chart graphs using graph database Neo4j.
- Finally, presenting ways to use Cypher query language to query over state-chart concurrent programs to study complex commutative structures and concurrency characteristics, like trace analysis, per thread execution analysis and execution ordering of atomic actions, presented in section 3.

CHAPTER 2: BACKGROUND - ASYNCHRONOUS TRANSACTION EXECUTION

In this section, we discuss in detail the rudimentary concepts that are required to understand our methodology for the representation of concurrent programs as graphs. We first discuss how transactions are associated with concurrent objects and understand the need for commutativity. In the later chapters we then present trace theory for understanding preconditional and postconditional states in state-chart graphs. Finally, we present our tool, *Concordant*, which generates graphs from execution model data of concurrent programs.

In this part we discuss how commutative operations on linearizable objects relates to transactional boosting. Any transactional objects are required to perform two types of operations to synchronize between each other: Synchronization and Rollback/Abort [2] [3]. Any two conflicting transactions that needs to synchronize their executions require calling a mutual exclusion construct, for example a transactional lock. This is a strategy of avoiding collision by forecasting conflicting points. Rollback/Abort is an opposite approach, where some collisions or conflicts cannot be speculative, and upon collision a rollback operation is performed to undo the incomplete transactional changes.

In terms of concurrent programming, transactions are like a collection of concurrent functions, grouped in one indivisible set. For two transactions, each with a set of functions that collectively commutes with one another, can execute concurrently at run-time without any synchronization primitive. The commutative property is a important criteria to deduce speculative collision in concurrent transactions. An API designer can use tools like *Concordant* to check which operations commute and under what circumstances while designing interfaces. Thus, use of transactional locks can be greatly reduced by speculatively avoiding transactional collisions using object-level commutative property.

CHAPTER 3: LITERATURE REVIEW - COMMUTATIVITY IN TRACE MONOIDS

Although commutativity was formally represented in Trace thoery [4], its benefits in transactional boosting is a later discovery. We present a brief overview on the Theory of Traces that are used in formal analysis of concurrent computations using abstract symbols. This is useful to understand ways in which concurrent operations can commute with one another. Going forward, we then apply the same notion of commutativity into our example program to can determine all possible commutative states. Commutative states are basis for finding the *preconditional* and *postconditional states* in traces that helps in verifying if functions are commutable or not.

If all possible behavior of a concurrent system can be represented as a set of strings, then individual string represent a particular trace result of the concurrent framework. Each string representation is a collection of letters (say all lower case English letters) abstracting actual events or actions to cause change in a concurrent system. In a schematic concurrent string, some letters can commute representing the portion of a concurrent program that can execute independently, while the synchronizing points that are not independently executable are represented with non-commuting letters. With this idea, in trace theory all letters are broadly classified to be *Dependency* (represented as *D*) and *Independency* (represented as *I*). Dependency is any finite, symmetric and reflexive relation. Considering a finite order pair set *(a,b)* is in *D*, then *(a,a)* and *(b,a)* are also in *D* [4]. A domain of D, represented as $\sum_D$, is defined as set of all alphabets used to define *D*. For example, if *D* is represented by all english lower case letters, then $\sum_D$ is {a,..,z}. For simplicity, we would confine our domains to only few english lower-case characters. For a given dependency *D*, the relation $I_D = (\sum_D \times \sum_D)$ - *D* is called *independency*, which is symmetric and irreflexsive [4]. All traces are primarily defined on the basis of *dependencies*.

Let $\sum$ be set of alphabets, then $\sum^*$ represents set of all possible strings drawn from $\sum$. Here $\sum^*$ is called free monoid and * represents Kleene's operator.

A binary relation $\sim$ is also defined, such that relation $u \sim v$ denotes $u = xaby$ and $v = xbay$, where $x, y \in \sum^*$ and $(a,b) \in I$. I here stands for an independency relation that influences binary relation $\sim$. Here, x and y are like preconditional and postconditional states, such that all commutative ordering of letters a and b happens in between.

Now let us consider the following example, let alphabet $\sum = \{a,b,c\}$. Assuming one possible dependency relation is

$$D = \{a, b\}^2 \cup \{a, c\}^2$$

$$= \{a,b\}\ x\ \{a,b\} \cup \{a,c\}\ x\ \{a,c\}$$

$$= \{(a,a), (a,b), (b,a), (b,b)\} \cup \{(a,a), (a,c), (c,a), (c,c)\}$$

$$= \{(a,a), (a,b), (b,a), (b,b), (a,c), (c,a), (c,c)\}$$

The above example dependency relation creates dependency restrictions between letter a and b, and letter a and b. The letters which are not restrictively related here are b and c; clearly, letters b and c commutes. The corresponding independency I_D imposed by dependency D, is $I_D = \{(b,c), (c,b)\}$. Thus a trace over independency I over string $aabbca$ is $[aabbca]_D = \{$aabbca, aabcba, aacbba$\}$.

The notion of commutative operations through independency relation between letters depicts ways in which methods invocations, response and atomic operations can reorder with one another at run-time. Fig. 3.1 represents the graph representation for the above 3 traces. Fig. 2 shows all commutative relations between node for each trace pair. Nodes which are highlighted in fig. 3.2 represents *preconditional state* followed by *postconditional state*. Across all traces, these

conditional state nodes remains constant, no matter how independent letter commutes. In the later section we check for similar preconditional and postconditional states in actual code execution to verify commutative operations.

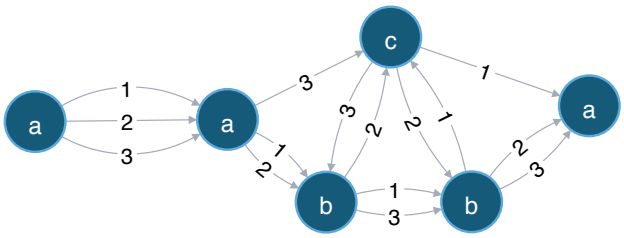
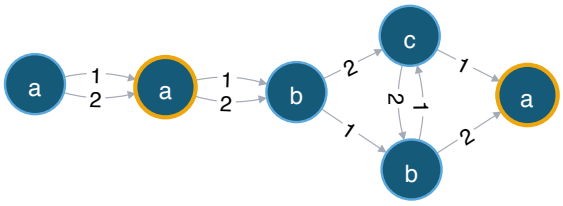


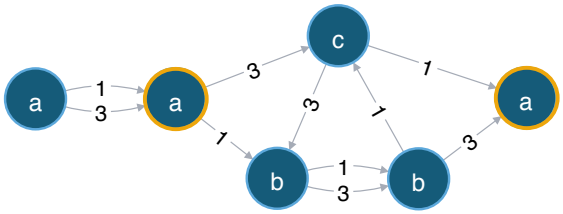
Figure 3.1: State-chart representation for *independency I*, over string *aabbca* is $[aabbca]_D$ = {aabbca, aabcba, aacbba}.

Austin Clement et al. in his paper on scalable commutativity rule [1], presented the dependency of object condition in order to affirm how functions commute during their run-time. For example, when an Set is empty, if two add() operations insert two distinct elements into the Set object, then object state at the end of two operations will remain the same irrespective of how the operations reorder with one another. But here the state of the object at the end of 2 operations are not remain the same as compared to the state of the object before the two operations processed on it. In another scenario, if we assume that a Set already has two elements in it, say (*a*,*b*), and two add() operations running concurrently tries to insert element *a* and *b* in to the Set. In this case, both add() operations fails to insert respective element in to the Set, as the values already existing in the Set. In the later case, irrespective of how these two add() operations reorder during run-time, they leave same resultant effect in the Set and hence the two operations commute with each other. What remains different in the later scenario is the state of the Set object before and after the operations processed over the object. In the later chapters, we present ways to probe object states to check if reordered functions across multiple traces leave object on same state or not. Based on this notion of object's condition, we try to define precondition and postcondition in actual objects run-time,

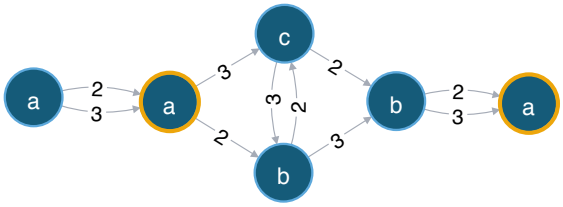
and reason the probing results across multiple interleaving in order to verify about commutativity.



(a) State-chart graph for trace 1 and trace 2



(b) State-chart graph for trace 1 and trace 3



(c) State-chart graph for trace 2 and trace 3

Figure 3.2: State-chart graph for analyzing commutativity induced by *independency I* for trace $[aabbca]_D$. Nodes highlighted with amber border represents *preconditional* and *postconditional* nodes respectively. *Preconditional* and *postconditional* state nodes are defined in context with respect to how commutativity is observed in a graph.

CHAPTER 4: METHODOLOGY

Concordant is a graph generating tool for Neo4j [13] graph database platform. For our research, we use Neo4j for state-chart representation of concurrent program executions. A model-checker is the crux of *Concordant* tool. To generate model data, *Concordant* use a verification tool, CCSpec [12], which we customized to use its underlying model-checker CDSChecker [11], to track all program states across multiple exhaustive thread interleavings. CDSChecker tracks model data properties like Action-types (Atomic Read/Write), Thread Id, Memory orders (introduced since C11/C++11), Memory address, and value associated with memory address, which are used to define states in our state chart. CCSpec [15] is capable of verifying any correctness condition specifications, running on top of CDSChecker, by collecting additional run-time information such as method invocation/response and method input/output. In this paper we reason about commutativity based on atomic action nodes in a state-chart, which is in itself a new approach. In non-blocking programming paradigm, the atomic instruction ensures changes in shared resources/objects takes place in race-free way. Thus, we assume atomic instructions are the basis of any modification in shared memory. Every recorded model instances are depicted as *action nodes* or *states* in our state-chart. Thus in this context, state refers to the action types and related properties that are captured by the model-checker during run-time. Function invocations and responses are essential for determining all memory accesses that occur in a data structure function, which is an essential to understand how two functions have ordered themselves in a trace. To check commutativity, we look for same functions in different traces, reorder in different ways. We will discuss the actual execution runs in the next chapter, and inspect the run-time properties that are tracked using *Concordant* which describe the action nodes in state-chart. Similar model instances across different traces are then merged into one single action state in the state-chart. Thus, this step reduces our search space generating a state-chart graph from independent trace results.

Concordant tool has four components in it. Fig. 4.1 explains various building blocks of Concordant. Two of the components are dynamic parts, which work with model-checkers and parser for model-checkers. A model checker is a vital part of the tool, which generates model data by exhaustively recording all object states in program run-time. In our experiment, we experimented with two sample model-checking tools: CDSChecker and CCSpec. CDSChecker is designed for the purpose of model checking, where as CCSpec is a verification tool built on top of CDSCheck with some extra features over that, which we have already discussed. Based on the model-checker in use, we use a specific model parser program that parses model data of all traces into XML format to collate run-time properties into an action tag in XML format. Specification of this XML formatting is generic to any model-checker and thus, in the later components, Merger program, this XML file gets processed to generate state-chart XML data. Merger reads through all the action components of different traces of parsed XML file generated from the Parser program, and generates hash-key based on the properties of action type, which is then assigned to each action node. Each action node with an unique hash-key id then gets appended into a XML formatted state node's id attribute value. Each hash-code that is generated also gets stored in a Set object, which ensures action nodes that are appended into the state-chart XML format are unique. Thus, common action components across multiple trace runs are referenced as single component using hash-key set. This finally generate a state-transition relation between each action nodes across multiple runs. The final component on the tool is cypher generator, which generates cypher queries based on the state-chart XML generated by the previous component. This automated cypher query generates nodes and relationships of a concurrent program execution in Neo4j graph database.

We have extended the structure of SCXML(State-chart XML)[9] to represent state-chart formatting of concurrent program, which we call eXtended State-Chart eXtensible Markup Language(xSCXML). This XML formatted state-chart file is created from Merger program. In our extended version of SCXML, we introduce new tag elements to represent action nodes/states and relevant transitions.

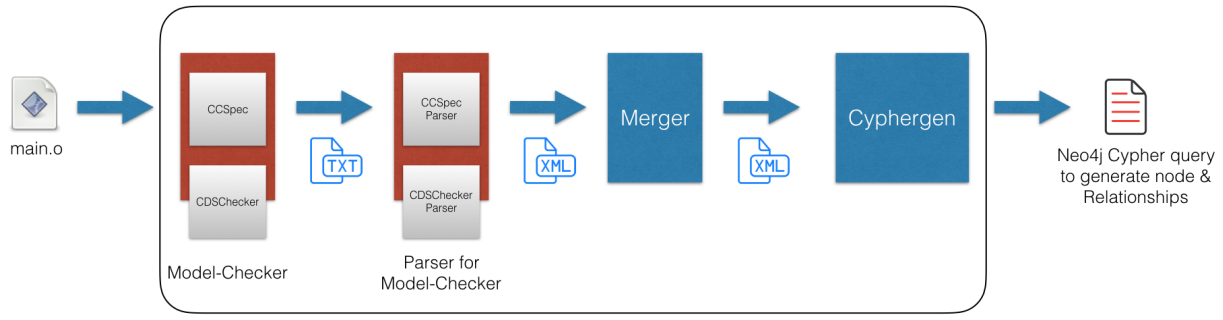


Figure 4.1: Building blocks of *Concordant* used for generating cypher queries to automate building *Nodes* and *Relationships* in Neo4j. Blocks in *Red* are dynamic part of the tool, which accommodates sample *Model-checker* and respective *Parser* program. Blocks in *Blue* are static blocks of the tool, which remains fixed and works with any given *Model-checker*.

Below table shows some of the extensions that we made on SCXML in our xSCXML.

Table 4.1: Comparison between list of tags in xSCXML and SCXML

-	Features in xSCXML	Features in SCXML
1.	Each $\langle State \rangle$ tags are associated with properties represented in $\langle Property \rangle$ sub-tags.	All $\langle State \rangle$ in SCXML are associated with same features and properties.
2.	All $\langle transition \rangle$ tags are associated with an id attribute, referring to the trace id. E.g. $\langle transitionid = "1" event = "1" target = "abcd123" \rangle$	There is no id attribute associated with $\langle transition \rangle$ tags. E.g. $\langle transitionevent = "1" target = "abcd123" \rangle$
3.	Transition tag in xSCXML represent the transition to the current state (as destination state) from the state mentioned in the *target* attribute state id (as source state)	Transitions in SCXML are opposite of that. Current state of the target super tag is the source state and the id mentioned in *target* attribute is the destination state.

CHAPTER 5: FINDINGS

In this section, we describe the concepts that are used to define action nodes and transitions in our state-chart graph, based on which we verify commutativity. We then present cypher queries to comprehend other characteristics of commutative operations from graph notations.

States and Transitions

In this section, we define our states or action nodes in state-charts. This corresponds to analyze *preconditions* and *postconditions* between concurrent operations across different interleavings. As discussed earlier, we customize CCSPEC as our underlying model-checker and we track the following run-time attributes, to identify different states in our state chart:

- **Action Type** - This entity stores information of different atomic changes/actions that takes places on shared objects, for instance, ATOMIC READ, ATOMIC WRITE, ATOMIC Read-Modified-Write(RMW), etc. We also use action type to store different stages of thread life-cycle in the program: THREAD CREATE, THREAD START, THREAD JOIN, THEAD FINISH. Using CCSPEC annotations we also store Method(MET) Invocations(INVOC), Return(RET) for analyzing real-time ordering of operations to compare object states.

- **Thread id** - All action types are associated with a specific thread id. For example, a specific thread will trigger a specific method or a specific thread commits a specific atomic operation. We link each thread id to action types and this associates each action condition to a distinct state. The Thread id being associated with states will make similar preconditional or post-conditional states appear to be distinct states. In our analysis of commutativity, we compare preconditional and postconditional states across different commuting interleavings. Thus,

even though commuting states might appear to belong to different state we will explicitly consider their semantic properties to match their association, unless they are executed by threads with same ids.

- **Memory order** - All atomic operations in C11/C++11 are associated with different memory order, that defines how overlapping operations synchronizes during program execution. Memory orders range from weak orders such as the relaxed memory model (one with free fences) to stronger orders like sequential consistent (strict read/write fence). Storing memory order information has no direct connection with commutativity, but helps user deduce run-time instruction to program source code. One limitation of the underlying model-checker CDSChecker is that the user cannot map executable atomic instruction to the actual source code. In this way any run-time anomaly cannot be speculatively mapped to the actual source code. Memory order information is one extra verification feature to map run-time instruction to the source code. This speculative mapping between model-checker executable operation and source code works best when atomic instructions executed among different threads uses distinct memory orders, or distinctive atomic instruction.

- **Memory location** - While verifying commutativity among method calls, it is important to know the active memory locations of concurrent object in execution. An active memory location is a location on which an atomic instruction operates, and for other cases relates to model-checker's implementation details [11]. Memory location attribute in action node helps to index underlying object state before and after invocation of methods. Object state here refers to the semantic object state of abstract level. In our analysis of commutativity, we probe an object's semantic state at abstract level and check their representation in terms of concrete structures. In order to reason about whether any ordering of two given operations results in the same effect in the underlying data-structure, we compare the preconditional and postconditional action nodes of individual trace. In our analysis of commutativity, we

have seen that some ordering between two operations that results in differing memory locations at object level, would lead to two distinct action nodes, from which we may infer that the operations do not commute. In corollary to that, there are cases where operations that will have same active memory location at concrete object level would result in similar preconditional and postconditional action nodes when operations successfully commutes. We have distinguishably identified cases based on contiguous data-structure, where object semantic structure is directly dependent on memory location of underlying object. In the later part of this chapter we analyze commutativity in two data-structure classes: Contiguous and Non-contiguous memory.

- **Memory location value or Method response value** - Along with memory location, return value of atomic operations that memory location helps to defines a distinctive state in the state-chart. There could be some instances where scrutinizing only memory location might not be enough to infer commutativity across two or more operations. For instance, say in an queue, we are interested to check commutativity between enqueue ENQ(100) and enqueue ENQ(150). It is with the help of memory location contents 0x64(which is hex for 100) and 0x96(which is hex for 150) that conveys which operations followed the other(ENQ(100) before ENQ(150), or vice-versa).

- **Trace id** - Every unique interleaving of threads course a distinct trace which are specified by Trace id. In our state-chart, trace id is an attribute which we associate with transitions between action states. In our cypher query, we use conditions over this field to filter respective traces.

We define *preconditional* and *postconditional* states (together *conditional states*) based on action nodes in the state-chart graphs. Each action node is formed by merging trace actions across multiple exhaustive traces. These conditional states basically associated with state at which a data-

structure or object remains before and after function calls. Valid commutative functions reorder across different traces leave same resultant effect on the underlying objects. Conditional states are *dependency* nodes, while action nodes generated during function reordering are *independencies*. Thus, conditional states are the points at which commutativity is validated. In any trace analysis, definition of preconditional and postconditional states are defined based on the data-structure and its operations that are considered for commutativity checks.

For example, in a *Set* data-structure, two add() operations can commute with each other if the initial condition of *Set* is either empty or either of the added value are not present in the *Set* already. In this case our preconditional state is any action node order before *add() Method Invocation* and postconditional state could be action node placed after two *response* action nodes. There also could be instances where features in one action node is enough to define preconditional and postconditional state of a object. This is depended on the underlying model-checker that is used in *Concordant*, and features that it tracks during run-time analysis.

Commutativity in concurrent traces

We have already discussed, how memory locations directly relates to semantic structure of abstract objects when underneath data-structure is defined in contiguous memory locations. In this section we check for commutativity by analyzing the preconditional and the postconditional states in two classes of data-structure, contiguous and non-contiguous memory. Based on the type of concrete data-structure used in designing the abstract objects, we analyze conditional states in two different approaches.

Commutativity verification on contiguous memory: Array-based Queue

In an array-based queue, array is the concrete data-structure that allocates memory in contiguous locations during program run-time. This restrictive memory utilization induces action node to get identical memory maps across different traces. Our first analysis for commutativity check is on Herlihy-Wing Queue [8]. We check commutativity between two dequeue() functions, for which we consider our precondition as the queue has two elements in it. We ensure this precondition by triggering two enqueue() methods prior to forking out the dequeue() methods. To verify object state in postcondition, we trigger one more dequeue() to check if the queue has any element left in it followed by enqueue() method and check if enqueue() happens on same action node across all traces. Fig. 5.1 shows the run-time probing mechanism of contiguous memory HWQ. In the postconditional state region highlighted in amber box, across the two traces, programs flows across dequeue() method, followed by the enqueue() method. The dequeue() method is shown in atomic read action followed by two atomic read-modified-write actions (separated by two method value check actions in grey color, which are part of future works and are not discussed in this paper). The enqueue() method is consisting of atomic read-modified-write followed by an atomic-write action. Although the action node has properties of underlying concrete object, our probing mechanism works on the abstract level and thus at the postcondition we verify the semantic structure of the object to reason about commutativity.

Fig. 5.2 shows the model run of two Dequeue() operations. In the figure action nodes in preconditional and postconditional states are in concordance across the 2 traces. This shows that the change in the overall queue structure, due to two dequeue() operations reordered variably, is same across all traces.

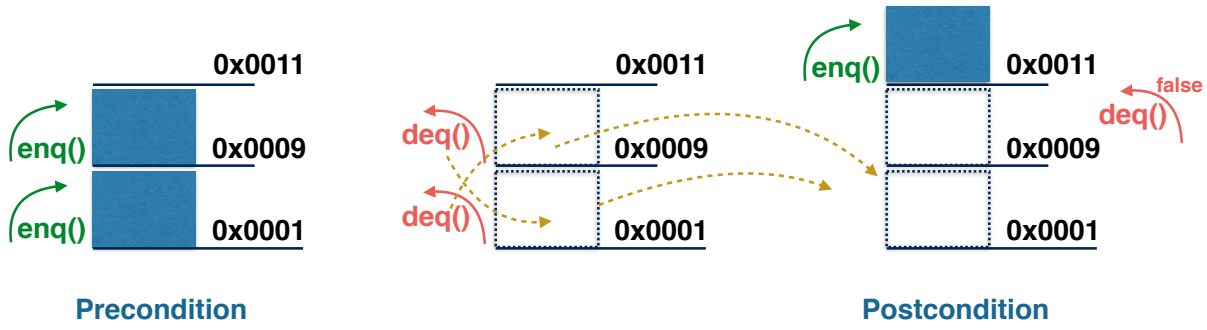


Figure 5.1: In the precondition state in an array-based queue, we ensure that the queue has 2 elements in it by enqueuing 2 elements sequentially in it. This ensure that the following 2 concurrent remove() operations finds elements to be removed. In the later postconditional state we probe these memory locations by performing dequeue and enqueue operation into the queue. If the dequeue returns false and enqueue adds an element on the same memory location across the 2 reordered traces of dequeue operations, then we concluded the operations are commutable.

A counterexample to show non-commuting operations, we rerun the same program, but with one enqueue() and one dequeue() operation. The output for model data for this scenario is shown in Fig. 5.3. The preconditional state on the object is an empty queue. The postconditional state in the queue will not be consistent across all commutative ordering between the 2 operations. We probe the postconditional state of the queue structure with a dequeue() followed by enqueue() operations executed sequentially. The enqueue() and the dequeue() operations reorder with one another in three different ways, and hence there are three traces in the figure. In trace 2 and 3, the dequeue() operation is consisted of same set of action nodes (atomic read followed by method atomic read-modified-write), where as in trace 1, action atomic read-modified-write traverses over different state of Queue object, which results in different action properties. Clearly, from the figure action states in the postconditional states are not consistent in all the traces. Therefore we infer that enqueue() and dequeue() method do not commute with each other.

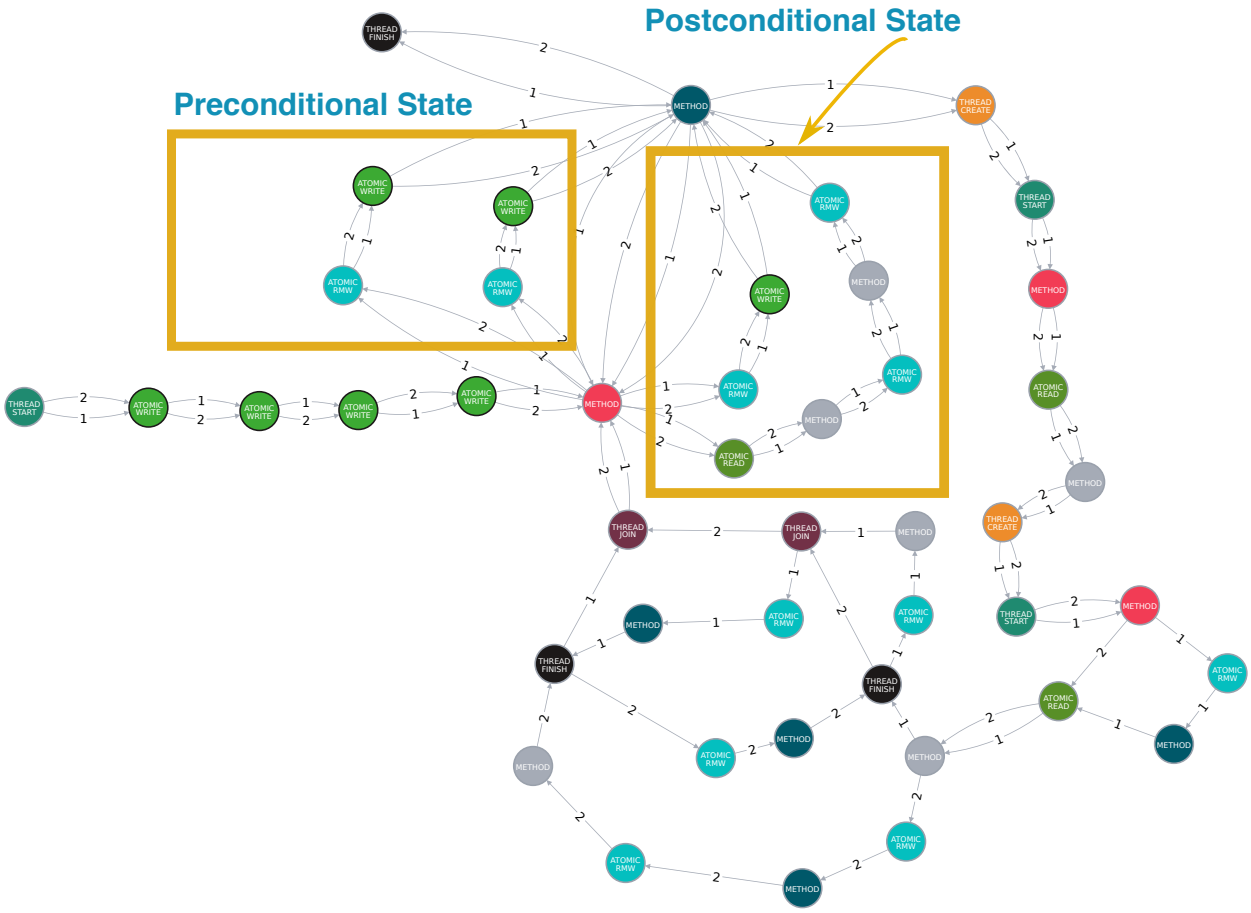


Figure 5.2: State-chart representation of Herlihy-Wing Queue [8] showing 2 Deq() operations in 2 traces. Action node with similar action-types are represented in distinct color node. Commutative action nodes have 2 traces, where thread 2's dequeue() precedes Thread 3's dequeue() in Trace 1, and Thread 3's dequeue() precedes Thread 2's dequeue in Trace 2. Each of the conditional states have same action node in both the traces.

Commutativity verification on non-Contiguous memory: Link-list based Set

For reasoning commutative property in non-contiguous memory data-structure we use a Set based on concurrent link-list [15]. In non-contiguous memory allocation, each atomic operations takes place at different memory locations and thus, will generate different action nodes in the precon-ditional and postconditional states. In order to probe the underlying object structure, we probe

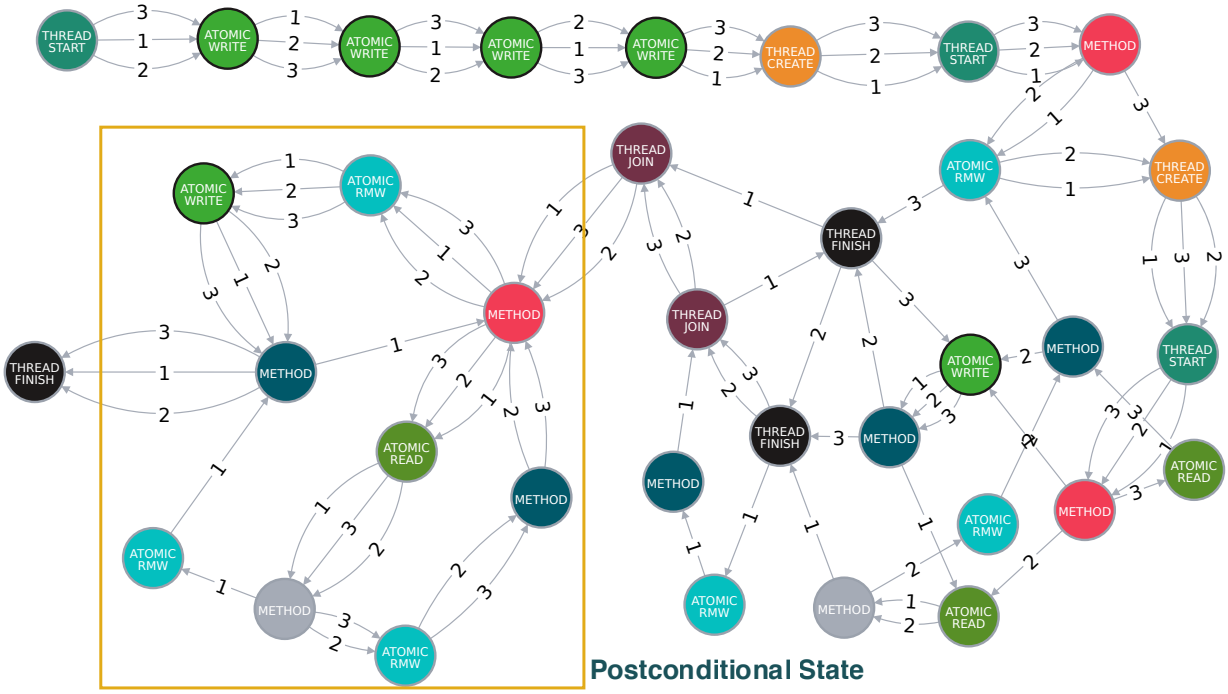


Figure 5.3: State-chart representation of Herlihy-Wing Queue [8] showing 1 deq() and 1 enq() operations in 3 traces. Since queue is empty in the precondition, we consider first THREAD START as preconditional action node. Postconditional action nodes are not consistent across all traces.

underlying link-list with an add() operation, such the argument in the add() operation is greater than any existing value in the set. Fig. 5.4 shows the probing mechanism in a linked-list based Set object. In doing this, the atomic read in the add() of concurrent link-list checks for starting from head node and stops at the tail node, where it finally performs a atomic write to insert the element into the Set. If 2 operations commute with one another, the number of read action node will remain same in the postconditional state across all traces.

Fig. 5.5 shows two add() commutative operations over a Set object. The precondition begin an empty Set and the postcondition is a add() operation with a value greater than the commutative add() parameters.

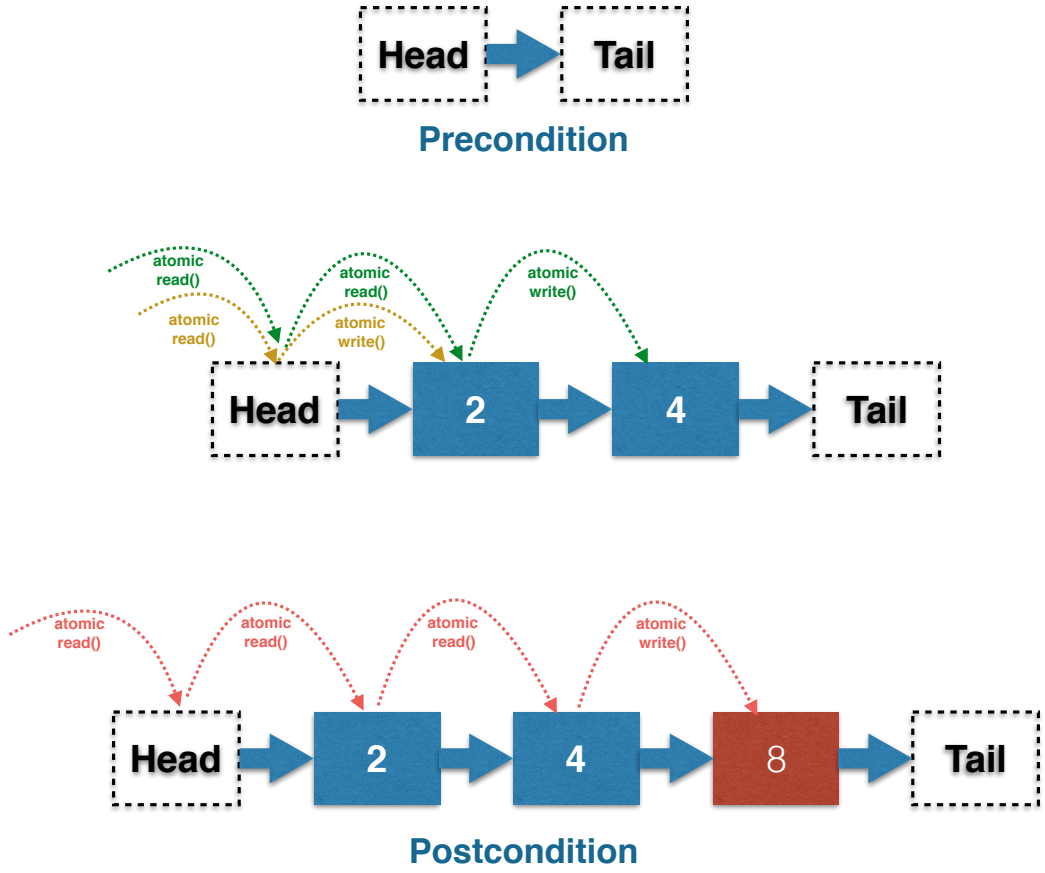


Figure 5.4: In the precondition state on a linked-list based Set, we have no elements in the Set. 2 add() operations are then concurrently executed to insert 2 elements into the Set. In the postconditional verification process, we add a third element into the linked-list, whose value is greater than the linked-list. In the process of finding the index of the element, the add operation performs read operations at each node locations in a way that it checks state of object.

A counterexample for two non-commutative methods are shown in Fig. 5.6 for 1 add() and remove() operation, where the precondition being Set is initially empty.

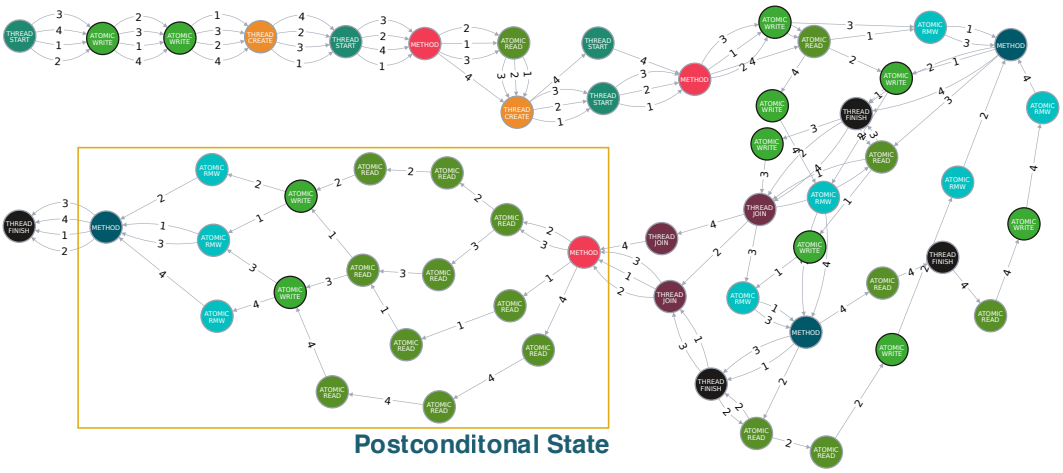


Figure 5.5: State-chart representation of concurrent link-list[15] based Set showing 2 add() operations in 4 traces. In non-contiguous memory, number of postconditional action nodes are same.

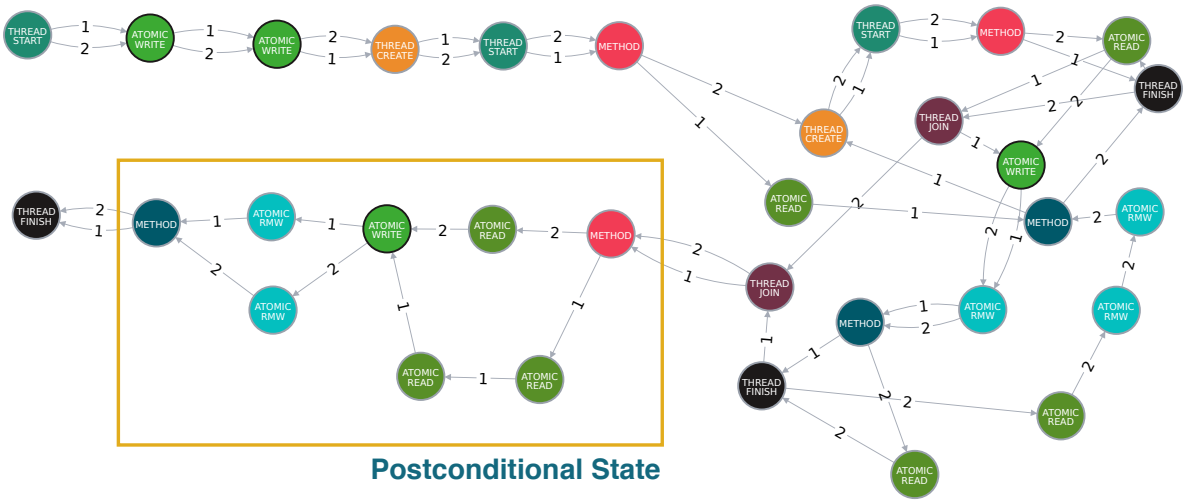


Figure 5.6: State-chart representation of concurrent link-list[15] based Set showing 1 add() and 1 remove() operations in 4 traces. In non-contiguous memory, postconditional action nodes are not in concordance for all trace results.

CHAPTER 6: BEYOND COMMUTATIVITY

In this chapter we present ways to check program characteristics using cypher query language. The advantage of using graphs in analyzing concurrent programs is that we can not only check whether functions commute, but also analyze how they commute during run-time. For our concurrent trace analysis over state-chart graph we use graph database tool, Neo4j and using Cypher query language. A preamble to Cypher query language is addressed in the appendix of this paper. Based on the mentioned run-time attributes we define properties of graph node in the graph database. We also make use of trace id to uniquely identify the properties of relationships between nodes. Here node represents states in the state-chart, and the relationships represents the directed transitions in the state-chart.

In Fig. 6.1 is a sample state-chart graph of Herlihy-Wing Queue, with one enqueue() and one dequeue() operation. Each of these operations are forked from 2 separate threads. In addition, there is a main thread that initializes atomic variables followed by spawning out threads. We use Concordant to generate related cypher query language to create action nodes and relationships for generating the corresponding graph. In this particular example there are three different ways these 2 queue operations can be reorder. In fig. 6.2, 6.3, 6.4 shows history of each interleaving. We use the following query to generate the respective graphs:

```
MATCH (a)-[r]->(b) WHERE 1 IN r.id
                   RETURN a,r,b
MATCH (a)-[r]->(b) WHERE 2 IN r.id
                   RETURN a,r,b
MATCH (a)-[r]->(b) WHERE 3 IN r.id
                   RETURN a,r,b
```

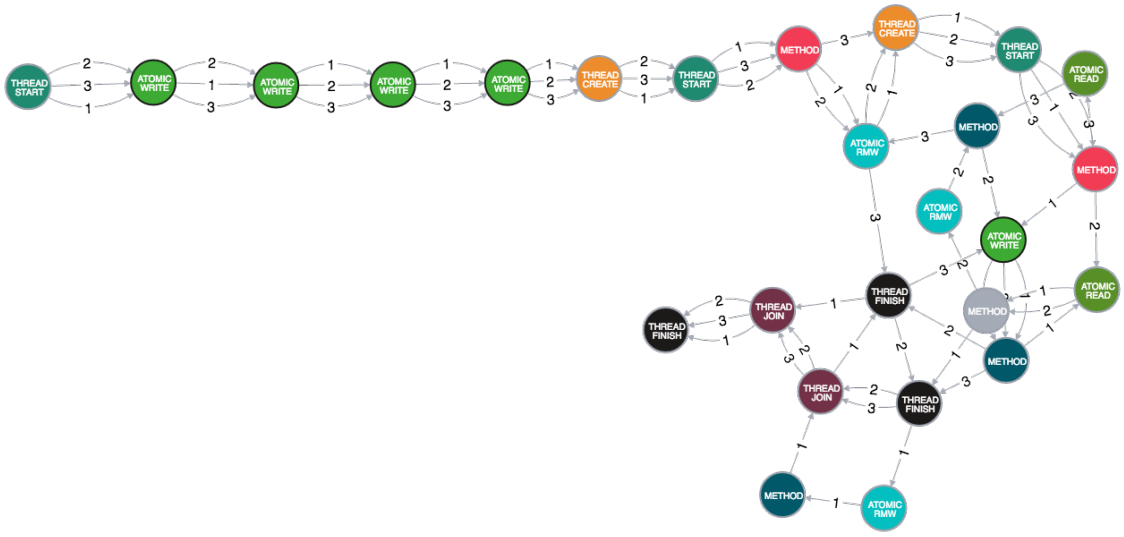


Figure 6.1: State-chart representation of Herlihy-Wing Queue [8] showing 1 Enq(), 1 Deq() operations in 2 separate threads. Action node with similar action-types are represented in distinct color node. Each transitions are labeled with trace id, to keep track of execution sequence in the program. All state-charts starts with *THREAD START* action node as initial state, which is the start of main thread in the actual program. Every complete graph that runs without an error ends at action node *THREAD FINISH*.

The *Id* property in the relationship between action nodes are used to stores the respective trace information. Each action node contains *Thread* property, that helps filter operations executed thread-wise across a trace. In a complete execution history of a concurrent program, ordering of operations carried out by multiple thread vary based on ways they could interleave. For example, in a typical execution history of a concurrent Queue execution, invocation of enqueue() operation by thread 1. may be followed by invocation and response of dequeue() by thread 2, closing with response of enqueue() by thread 1. In this example trace, thread 2 has two consecutive action states whereas thread 1 has 2 nonconsecutive action states. This is important to evaluate traces from this perspective to formulate a cypher query to fetch all specific action nodes. In the following cypher we fetch the thread specific operations using these 2 cases: The first case (a) -[r] ->b fetches all action nodes and their relationships that are executed by one thread (WHERE

a.Thread='1' AND b.Thread='1') in a particular trace (say 1 IN r.id), more than one operations without switching to another thread. The second case fetches all singly occurred action nodes (c), which were not fetched in the first case, WHERE c.Thread='1' of the same trace.

```
MATCH (a)-[r]->(b),(c)
WHERE 1 IN r.id AND a.Thread='1'
AND b.Thread='1' AND c<>a AND c<>b
AND c.Thread='1' RETURN a,r,b,c
```

```
MATCH (a)-[r]->(b),(c)
WHERE 1 IN r.id AND a.Thread='2'
AND b.Thread='2' AND c<>a AND c<>b
AND c.Thread='2' RETURN a,r,b,c
```

```
MATCH (a)-[r]->(b),(c)
WHERE 1 IN r.id AND a.Thread='3'
AND b.Thread='3' AND c<>a AND c<>b
AND c.Thread='3' RETURN a,r,b,c
```

We can do similar query for each thread analysis for other 2 traces in the graph, using the following respective cypher queries. Fig. 6, 7. are the respective graphs for per thread analysis for trace 2 and 3.

```
MATCH (a)-[r]->(b),(c)
WHERE 2 IN r.id AND a.Thread='1'
AND b.Thread='1' AND c<>a AND c<>b
AND c.Thread='1' RETURN a,r,b,c
```

```
MATCH (a)-[r]->(b),(c)
WHERE 2 IN r.id AND a.Thread='2'
AND b.Thread='2' AND c<>a AND c<>b
AND c.Thread='2' RETURN a,r,b,c

MATCH (a)-[r]->(b),(c)
WHERE 2 IN r.id AND a.Thread='3'
AND b.Thread='3' AND c<>a AND c<>b
AND c.Thread='3' RETURN a,r,b,c

MATCH (a)-[r]->(b),(c)
WHERE 3 IN r.id AND a.Thread='1'
AND b.Thread='1' AND c<>a AND c<>b
AND c.Thread='1' RETURN a,r,b,c

MATCH (a)-[r]->(b),(c)
WHERE 3 IN r.id AND a.Thread='2'
AND b.Thread='2' AND c<>a AND c<>b
AND c.Thread='2' RETURN a,r,b,c

MATCH (a)-[r]->(b),(c)
WHERE 3 IN r.id AND a.Thread='3'
AND b.Thread='3' AND c<>a AND c<>b
AND c.Thread='3' RETURN a,r,b,c
```

It order to identify the commutative states we need to map each instruction to their respective

methods. For instance, from fig. 6.1, the state-chart graph for trace 1, we see there are 5 **ATOMIC WRITE** operations, 2 **ATOMIC RMW (Read-Modified-Write)** operations and 1 atomic read operation. Referring to the same graph, we see that 4 **ATOMIC WRITE**s are common across all traces, executed by thread one that is also the main thread. Referring to our implementation of the Herlihy-Wing Queue, we see that these *WRITE*s operations refer to the initialization of the atomic variables carried out by the main thread at the start of the program. Since these steps takes effect sequentially by the main thread, in fig 6.5a, we see a consistent interleaving of action nodes across all the traces. Referring to the source code, we see that main thread (thread 1), spawns out the other threads by calling *THREAD CREATE* instruction and remains inactive till other threads finishes their tasks, and resumes back on *THREAD JOIN* instructions followed by finishing itself at *THREAD FINISH*. Referring to the second thread, in fig. 6.5b, we see action nodes *ATOMIC RMW (Read-Modified-Write)* and *ATOMIC WRITE* instructions in the thread trace. And the remaining atomic instructions are executed in trace 3, which is shown in fig. 6.5c. There could also be cases when 2 threads are spawning out the same method call. Thus, it might be tough to describe an action node to the user merely by looking at the atomic instruction, as both the operations will execute an identical set of atomic instructions, interleaved in some order. For such cases, we have *VALUE* property in each action node, to uniquely correlate states in graph to the actual method. In our example, enqueue(100) corresponds to atomic action nodes with 0x64 (hexadecimal for 100) as *VALUE* property.

Thus, for trace analysis, we make use of the following concepts:

- Thread 1 is the main thread in program and across all traces is the triggering point for forking out other threads. Main thread contains information about atomic variable initialization, pre-conditional and postconditional state of concurrent object. We reason about commutativity based on preconditional and postconditional states.

- Any thread, other than thread 1, are triggered from main thread to run concurrently and carry out operations on concurrent objects. Based on ways operations commute at run-time between different atomic instructions, different traces are generated during run-time. Using the above cypher queries developer can check and compare different traces together to check if program runs correctly and find performance tweaking points. Usually strict memory orders in atomic instruction imposes excessive restrictions on how atomic operations will reorder during run-time causing lesser interleavings between threads. Restrictive thread interleavings has serious performance drawbacks over relaxed constructs.

- Identifying action node and mapping it to a program instruction can be done using properties of action node. Each properties in an atomic action nodes between a method-invoke and method-return action corresponds to atomic operation in source code. For example, the value field in the action node is the hexadecimal conversion of actual value on which atomic action operates.

Using the above points, developer can deduce different run-time properties in a concurrent program while verifying its commutative property between different concurrent method calls.

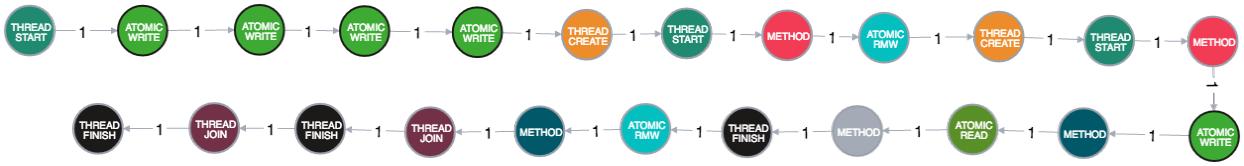


Figure 6.2: Trace 1 of state-chart representation of 1 Enq(), 1 Deq() Herlihy-Wing Queue.

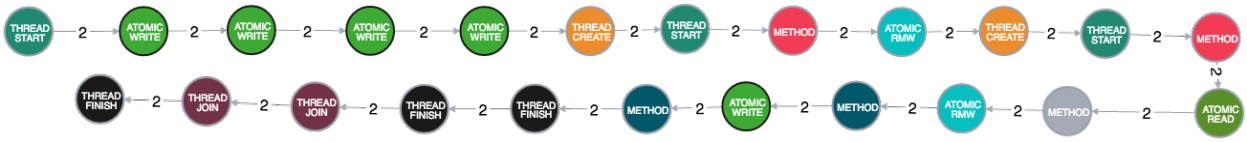


Figure 6.3: Trace 2 of state-chart representation of 1 Enq(), 1 Deq() Herlihy-Wing Queue.

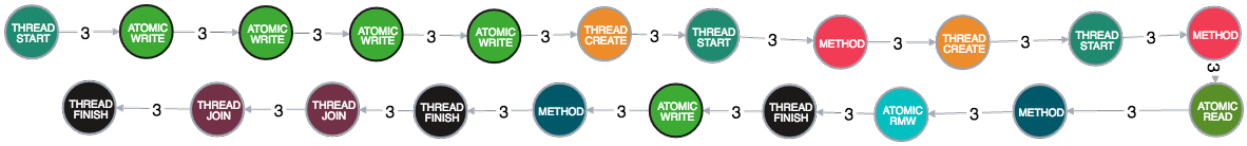


Figure 6.4: Trace 3 of state-chart representation of 1 Enq(), 1 Deq() Herlihy-Wing Queue.

(a) List of operations executed by Thread 1 in Trace 1

(b) List of operations executed by Thread 2 in Trace 1

(c) List of operations executed by Thread 3 in Trace 1

Figure 6.5: Trace 1 - Per thread analysis of 1 Enq(), 1 Deq() Herlihy-Wing Queue

CHAPTER 7: RELATED WORKS

Using commutativity as means to increase transactional concurrency has been a popular technique. Our research work on commutativity using graphs was inspired on the works done by Antoni Mazurkiewicz in Trace theory[4]. Mazurkiewicz presented concurrent processes as abstract strings that are also used in representing sequential systems. Representing concurrent strings in dependency and commuting independency characters, he devised dependency graphs to derive traces, that sowed our foundation for graphical representation of concurrent traces. In our research we used model-checker base approach to track different thread interleaving of concurrent program and represented their run-time data in the form state-transition diagram in graph database. This has not helped in identifying concurrency properties like commutativity, but also aided developers with cypher query language to investigate deeper into the concurrent programs, some which were discussed in the previous chapter.

Current understanding of transactional boosting has evolved though numerous investigations and diversified research work done towards understanding object-level commutativity. Steele et al.[16] described commutativity as conflict-free operations. Thus, operations that could be analyzed for conflict-freedom can be inferred as commutative to each other. Prabhu[17] and Rinard[18] describes ways to synchronize operations to ensure that changes occur compositely, even when memory locations are conflicting. They have emphasized on the correctness over scalability. In contrast to this approach, our focus is only on scalability, as we have relied on the underlying concrete data-structures for program correctness. Analysis of commutativity based on memory state during run-time has been used to comprehensively describe commutative operations [19]. In our case, we use a model-checker to keep track of memory locations and the return values of method calls. Object-based commutative analysis depends on observing method reordering patterns has been a

popular approach [18][20]. Previous to their work, commutativity was observed at read/write operations on memory words level. We extend object-based commutative analysis to generate graphs and interconnections between trace flows.

Clements et al. [1] presents Commuter, a conflict checking tool that lists out different combination of operations that can commute at run-time. Commuter and our tool are both based on *conditional states*, but Commuter does not tracks object-level run-time model data. Clements' tool is an inspiration to extend our method to automatically derive functions that can commute at run-time. The advantage of our approach over Commuter tool is that we present an exhaustive result of all possible interleavings to show how functions commute at atomic instruction level. In our analysis for commutativity, when have seen examples where two operations could commute in more than two way, leaving two or more trace histories. Such in-depth results are helpful in rectifying buggy executions, one example of which was presented in introduction.

CHAPTER 8: CONCLUSION

We have presented a graphical representational tool to generate graphs to check commutative operations during program run-time. By representing run-time model data in state-chart action nodes, we were able to find preconditional and postconditional states as a base to reason commutativity. Our work is the first application of graph database in analyzing concurrent programs by using cypher query language. This is not only an intuitive approach to reason about commutativity between operating functions but also an extensible approach to check other properties.

APPENDIX A: GRAPH DATABASE NEO4J

Graph database *Neo4j*[13] is a NoSQL open-source tool. There are two main components in Neo4j graph database: Nodes and relationships. Nodes are like entities that holds key-value pair like attributes on basis of which queries are executed on graph database. Relationships are the entities that connect the node entities and are directed, i.e. they are always associated with a start node and an end node. Relationships also have key-value pair an queries can be executed conditioned over relationships. All relationship links are complete in graph database, and thus, there cannot exist a directed relationship dangling to no node. Once a node gets deleted, relationships associate with that also gets deleted.

APPENDIX B: CYPHER QUERY LANGUAGE

We use *Cypher query language*[14] to fetch data associated with graph database. Like any sequential query language, there are three main commands associated with cypher query language: CREATE, MATCH and DELETE(and DETACH).

CREATE instruction is used in creating entities like nodes and relationships in a graph database. E.g.: The following cypher query creates a node with only one attribute *name* and the associated value is *"NODE NAME"*.

```
CREATE (entity:NODE {name:"NODE_NAME"}) RETURN entity
```

MATCH queries are equivalent to SELECT queries in SQL.They are used to fetch data conditioned upon node or relationship key-value pair. E.g.: The following cypher query is used in fetching movie.title (title attribute value of movie *nodes*) as "title" as header and all actor's name (name attribute in actor *node*) grouped as a collection under header as "cast", such that title attribute in movie *node* starts with "T". This query fetches first 10 queries ordered by ascending order of their movie title.

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WHERE movie.title STARTS WITH "T"
RETURN movie.title AS title, collect(actor.name) AS cast
ORDER BY title ASC LIMIT 10
```

The third kind of queries are DELETE and DETACH. The DELETE clause is used to delete nodes, relationships elements. E.g.: The following cypher query is used to match all nodes in the graph and DETACH all relationships associated with the matched nodes and delete all nodes matched in the graph.

```
MATCH (n) DETACH DELETE n
```

LIST OF REFERENCES

[1] Clements, Austin T., et al. ”The scalable commutativity rule: Designing scalable software for multicore processors.” ACM Transactions on Computer Systems (TOCS) 32.4 (2015): 10.

[2] Herlihy, Maurice, and Eric Koskinen. ”Transactional boosting: a methodology for highly-concurrent transactional objects.” Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. ACM, 2008.

[3] Herlihy, Maurice, and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. Vol. 21. No. 2. ACM, 1993.

[4] Mazurkiewicz, Antoni. ”Introduction to trace theory.” The Book of Traces (1995): 3-41.

[5] Hoogeboom, Hendrik Jan, and Grzegorz Rozenberg. ”Dependence graphs.” The Book of Traces (1995): 43-67.

[6] Weihl, William E. ”Commutativity-based concurrency control for abstract data types.” Computers, IEEE Transactions on 37.12 (1988): 1488-1505.

[7] Herlihy, Maurice, and Nir Shavit. ”The art of multiprocessor programming.” PODC. Vol. 6. 2006.

[8] Herlihy, Maurice P., and Jeannette M. Wing. ”Linearizability: A correctness condition for concurrent objects.” ACM Transactions on Programming Languages and Systems (TOPLAS) 12.3 (1990): 463-492.

[9] Barnett, Jim, et al. ”State Chart XML (SCXML): State machine notation for control abstraction.” W3C working draft (2015).

[10] Kim, Deokhwan, and Martin C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. Vol. 46. No. 6. ACM, 2011.

[11] Norris, Brian, and Brian Demsky. ”CDSchecker: Checking concurrent data structures written with C/C++ atomics.” ACM SIGPLAN Notices 48.10 (2013): 131-150.

[12] Christina Peterson, http://ucf-cs.github.io/CCSpec/

[13] http://neo4j.com/developer/graph-database/

[14] http://neo4j.com/developer/cypher-query-language/

[15] Herlihy, Maurice, and Nir Shavit. The Art of Multiprocessor Programming, Revised Reprint. Elsevier, 2012. Chapter 9, page 213 - 219.

[16] Steele Jr, Guy L. ”Making asynchronous parallelism safe for the world.” Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1989.

[17] Prabhu, Prakash, et al. ”Commutative set: A language extension for implicit parallel programming.” ACM SIGPLAN Notices. Vol. 46. No. 6. ACM, 2011.

[18] Rinard, Martin C., and Pedro C. Diniz. ”Commutativity analysis: A new analysis technique for parallelizing compilers.” ACM Transactions on Programming Languages and Systems (TOPLAS) 19.6 (1997): 942-991.

[19] Aleen, Farhana, and Nathan Clark. ”Commutativity analysis for software parallelization: letting program transformations see the big picture.” ACM Sigplan Notices 44.3 (2009): 241-252.

[20] Eberhard, John, and Anand Tripathi. ”Object-based commutativity analysis for real-time applications.” Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on. IEEE, 2005.