Behavioral Programming in BPJ

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Pintu Kumar with Katravath Manoj
(120101050,120101031)

under the guidance of

Purandar Bhaduri

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Behavioral Programming in BPJ**" is a bonafide work of **Pintu Kumar with Katravath Manoj (Roll No. 120101050,120101031**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Purandar Bhaduri**

Professor,

May, 2016
Guwahati.

Department of Computer Science & Engineering,
Indian Institute of Technology Guwahati, Assam.

Acknowledgements

We thank to the guide for being patient to answer our obvious questions and doubts.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 Behavioral Programming 2

1.1.1 Common Terminologies 2

1.1.2 Theory and Tools for Reactivity 3

1.1.3 Why use BP? 4

1.2 BPJ Introduction 5

1.3 Organization of The Report 7

2 Review of Prior Works **9**

2.1 Incremental Development Of Behavioral Program with Model-Checking . . 11

2.2 Conclusion 13

3 Examples Programs: Task Ordering and Deadlock Detection **15**

3.1 Car Manufacturing Problem 15

3.1.1 Solution 16

3.1.2 States and Priorities of BThreads 19

3.1.3 Sample Execution 19

3.2 Dining Philosophers Problem 20

3.2.1 Events and BThread 20
3.3 Conclusion 22

4 Some lighter Examples **23**
4.1 Producer-Consumer Problem 23
4.1.1 General Solution 23
4.1.2 Producer-Consumer Our Solution 24
4.2 Generate Prime Numbers from 1 to n 26
4.3 Conclusion 29

5 Conclusion and Future Work **31**

References **33**

List of Figures

List of Tables

Abstract

We illustrate the usage of BPJ Library for Behavioral Programming in Java Language through four examples. Car Manufacturing illustrates how constraints like order of the tasks can be met in the BPJ programs. Dining Philosophers problem shows how multiple behaviors coexist independently of each other, with very simple rules for each and how anti-scenario can help in finding fault in the program in this paradigm. Producer-Consumer problem's one possible execution is programmed in the paradigm illustrating that BPJ can be used in programs where for handling concurrency and mutual exclusion we require semaphores and mutexes. Finally printing primes from 1..n is a simple program to illustrate how this can be programmed in this paradigm. We failed to use model-checking and anti-scenario was not properly working, probably due to a bug in BPJ, where it gives null-pointer exception while running program with anti-scenario included.

Chapter 1

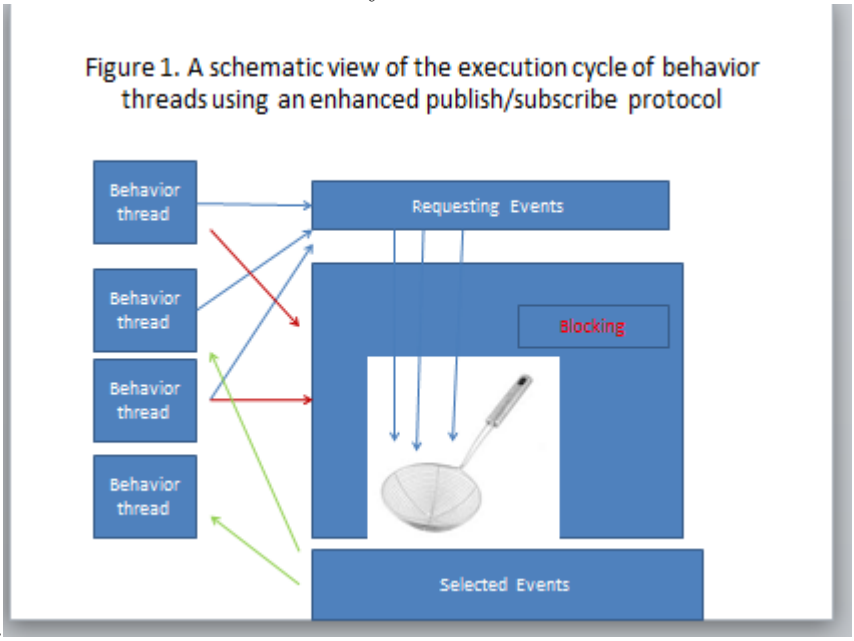
Introduction

There has been decades of advances in languages, tools and methods in software engineering for reactive systems. Also gradually criticality and demand of software systems are growing rapidly due to its widespread usage in industries like finance, business and service sectors.Yet software development is still hard, expensive and risky due to the phases it has to go through like Requirements Gathering, System Design, Testing and Verification and Maintenance.Behavioral Programming tries to make these steps easier through its unique paradigm where requirements are specified in aligned with human thinking about the system behaviors. Maintenance requires little or no modification in the existing system and adding new behaviors almost independently.[HMW12]

This paradigm has been implemented as Live Sequence Charts[DH01][HM03][MH06][HMSB10] which is improved version of Message Sequence Charts[AHP96][BAL97][BHKS97][LL95] and UML diagrams[WK98] and requirements are specified using visual diagrams by the programmer.Several Tools are available to automatically translate those requirements specifications into executable programs like PlayEngine[HM03][CHK08] [MHK02] and PlayGo[HMSB10]. BPJ[HMW10][HMW12] in Java programming languages facilitate this style of programming by exploiting it's concurrency and mutual exclusion mechanisms.Other libraries include BPC in C++[HKK13] and also are availabe in JavaScript, Blockly, Erlang.

1.1 Behavioral Programming

Behavior Programming is a language-independent paradigm[1] for programming reactive systems[2] centered on natural and incremental specification of modular behavior. This paradigm allows coding applications as multi-modal scenarios. Each scenario corresponds to an individual requirement. The requirement specifies what can, must or may not happen following certain sequences of events. To facilitate full behavioral modularity via the independent coding of separate facets of behavior, all scenarios run simultaneously and all are consulted at every decision point during execution.



1.1.1 Common Terminologies

There are common terms like Events, Behavior Threads (Allowed and Forbidden),Priorities of Behavior Threads, Requested EventSets, Watched EventSets, and Blocked EventSets,Scenario, Anti-Scenario, Model-Checking which are used in Behavior Programming.

- Events: An execution of a program in this paradigm is a sequence of events. Events can be generated by either user or environment.

[1]Paradigm refers to a framework containing the basic assumptions, ways of thinking, and methodology that are commonly accepted by members of a scientific community

[2]system that responds to external or internal events, e.g. operating systems, various computer games etc.

- Behavior Threads: The system's modules are behaviors specifying governing rules for execution of the system. These modules are called behavior threads. Each behavior thread run independently of others.

- Priorities of Behavior Threads: Priorities provide behavior threads to rule over other behavior thread as event requested by one Behavior Thread can be blocked by other higher priority-holding Behavior Thread.This enables forbidding of some Behavior Thread execution.

- Requested, Watched and Blocked EventSets : Each Behavior Thread has these three sets of events through which it can request some event to be triggered, wait for some event to be triggered and block some events for forbidding the behaviors which are requesting or watching that event.

- Scenario: Scenario represents a required behavior, can be considered a synonym for behavior thread, is a sequence of events which is desired.Where as anti-scenario refers to a behavior which is not desired. If an anti-scenario is marked as hot, the system halts by showing the trace of how system reached that anti-scenario.

- Model-Checking: Model-checking is used to improve system behavior, it is a system verification technique in this paradigm. It can suggest by simulating all possible sequences behaviors of the system where system can go wrong, and also how that scenario can be avoided. The most generic approach for avoiding the system to reach a bad state is to specify that scenario as an anti-scenario in the system[HLMW11].

1.1.2 Theory and Tools for Reactivity

The Behavioral Programming paradigm is supported by tools for debugging.TraceVis [OnV] and Live Sequence Charts(LSC)[Rob05] are useful for debugging and program comprehension.

[EGH+11] paper addresses the problem of cause and effect relationships between relativley

independent behavior threads of an application.They provided a tool which shows the decisions of the collective execution mechanism provided by the LSC or BPJ library. It shows the behaviors and events that were executed at each decision point, and those that were delayed or abandoned. The tool also shows the causes and reasons behind these runtime choices.

[HMSB10] presents PlayGo, a comprehensive tool for this programming paradigm which takes natural language as input and produces the executable. [MHK02] provides user to "play-in" requirements through system's GUI or some abstract version thereof, and full behavior is then "played out". The tool is called play-engine which is relevant to many stages of system development, including requirements engineering, specification, testing and implementation. [HLMW11] describes a model-checking tool for verifying behavioral Java programs, without having to first translate them into a specific input language for the model-checker. The method facilitates early discovery of conflicting or under-specified scenarios, which can be resolved by adding new scenarios rather than by changing existing code. Also, counterexamples provided by the tool are event sequences that can serve directly for refinements and corrections.

1.1.3 Why use BP?

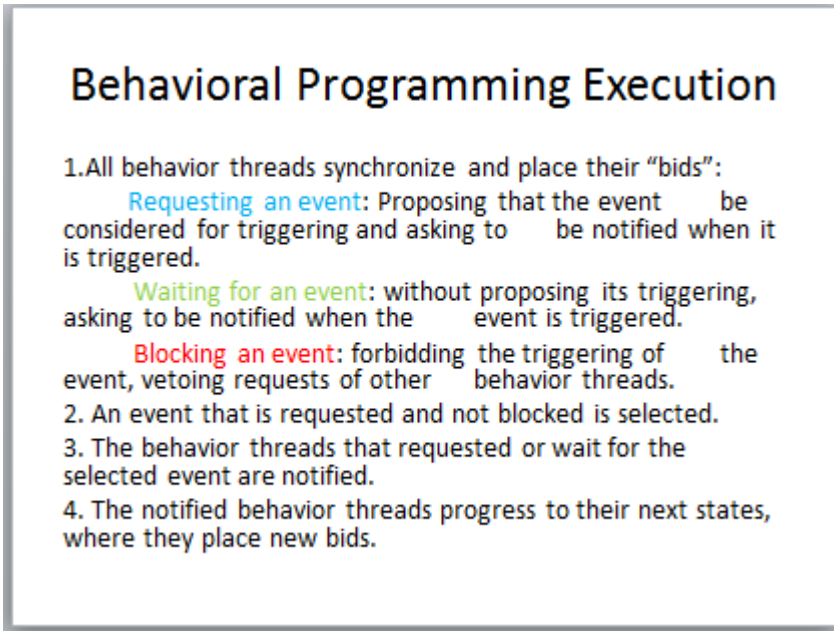
Behavioral Programming is aligned with human thinking about system behaviors. LSC provide programmer to program by visual diagrams. PlayGo facilitates the progammer to specify the requirements in natural language. Incremental system evolution is fascilitated by this paradigm as new requirements, enhancements and patches, affecting overall system behavior, can be added with little or no change to existing modules.So software maintenance is relatively less costly. Behavioral Programming is amenable to automated smart execution, verification and synthesis. Formal executable semantics enable direct use of the tools. e.g. to find conflict in the original requirement. Application-agnostic composition can enable efficient inference of system properties from module properties. Some programs

in BP can be built from parallel modules that are exponentially smaller, in terms of number of states in the transition system, than with the other programming idioms. This paradigm offers some of the succinctness advantages of cooperating automata (state-charts) using interfaces that preserve more encapsulation.

BP may not be needed or desired when we have a simple reactive decision or when incrementality is not needed. In this paper we have provided simple examples for demonstrating how we can program simple applications using BPJ library.

1.2 BPJ Introduction

[HMW10] have presented a general approach to software development in Java following the scenario-based approach to programming centered around live sequence charts(LSCs). A program consists of modules called behavior threads (b-threads), each of which independently describes a scenario. A protocol and coordination mechanism is given to fascilitate behavioral programming. Runs of programs are sequences of events that result from three kinds of b-thread actions: requesting that events be considered for triggering, waiting for triggered events, and blocking events requested by other b-threads. The coordination mechanism synchronizes and interlaces b-threads execution yielding composite, integrated system behavior. The protocol idioms and the coordination mechanism of b-threads are implemented as Java Library called BPJ.



The basic programming unit for implementing behavioral programming in Java is the coding of behavior threads as Java threads. BPJ provides BThread class, whose each instance is associated with a thread whose progress embodies a run of the behavior thread. The logic of b-thread is implemented in a method runBThread provided by the programmer. The events triggered by the b-threads are represented as objects of the class Event. Each b-thread object has three sets of events: requested events(requestedEvents) , watched events (watchedEvents), and blocked events (blockedEvents). Requested events are events which b-thread wants them to be considered for triggering. Watched events are those events which when triggered b-thread wants to be considered to be notified about that.Through these events b-thread does subscription for notification. Blocked events are those events which b-thread wants to block and forbid other b-thread's request for triggering the same set of events. This can be used for removing some features from the application. We don't need to remove the module rather add another b-thread with higher priority and block the events which are subscribed by the b-thread corresponding to that feature.

Each b-thread calls the behavior synchronization method bSync passing to it as parameters the events it requests, wait for, and/or blocks. The method bSync synchronizes the b-thread with all other b-threads and invokes the control mechanism that chooses and trig-

gers the next event and notifies the b-threads waiting for that event.

To assist model-checking BPJ provides a function labelNextVerificationSate("state") to label different states of the program at different bSync calls inside a BThread. Actually a BThread can go through multiple states throughout the execution cycle and program state is the cartesian product of all the b-thread states at any decision point, i.e. bSync call inside a b-thread. To make a b-thread as anti-scenario which is not the desired behavior of the system and system should halt on reaching that state, we can mark that bthread as hot by using the function markNextVerificationStateAsHot().We use this function inside runBThread() method. To wait for any instance of a class of Event(here Resourcefull class),we use EventsOfClass method provided in bp.EventSets. Also if no event is there to request, wait for, or to be blocked then we specify none as corresponding argument. labelNextVerificationState("W"); bp.bSync(none, new EventsOfClass(ResourceFull.class), none);

1.3 Organization of The Report

You can write the about otganization of your report in the following manner.

This chapter provides a background for the topics covered in this report. We provided a brief description of tools and technologies in behavioral programming, and common features of the paradigm. Then we introduced the BPJ Library in which all the four example problems have been programmed. In this chapter it is shown the coordination mechanism as publish/subscribe protocol by which different behaviors progress throughout the application execution. We explained how anti-scenarios can help detect undesired system behavior and gave references to model-checking which facilitates early detection of conflicting behavior specifications. The rest of the chapters are organised as follows: next chapter we provide review of prior works. In Chapter 3 and 4, we discuss our four examples of behavioral programming in BPJ,viz.,Car Manufacturing, Dining Philosopher, Producer-Consumer and Generating Prime Numbers. And finally in chapter 6, we conclude with some future works.

Chapter 2

Review of Prior Works

LSC Examples inside a paper Hello World From PlayEngine This site [bpE] contains six example programs which are done using BPJ Library.

- Alternating Taps Example: This program has two events addHot and addCold, three b-Threads addHotThreeTimes, addColdThreeTimes and Interleave. addHotThreeTimes requests addHot 3 times, addColdThreeTimes requests AddCold three times and Interleave causes alternation between addHot and addCold events by blocking one while waiting for the other and vice versa.

  This program's purpose is to show how we can enforce alternation in tasks in programs.

- Tic-Tac-Toe: we see its incremental development in the next section.

- helicopter Flight Examples: This is an complex example where we can see incremental development and various alternative ways to program it in BPJ. Following design patterns can be seen throughout the example.

  - Events can be distinguished from each other by

    * Class- different classes of events

    - Name - multiple instances of an event of the same class distintinguished by their names (e.g. the color change events).
  - One activity can interface with another by using
    - priority- relative priority among b-thread
    - blocking the events of the other activity
  - Distinguishing internal and external events - those generated by the system/application and those generated by the environment (which may be reported by sensors)

- Simulating Flight of a Flock of Birds: Simulated the flight of a flock of birds, with each bird having a simple set of rules implemented as b-threads. The example highlights:
  - Multi-agent activity
  - Emergent group behaviors - emanating from individual agent behavior
  - Implementation of animation with BPJ- and handling assoiciated design issues.

- Sudoku: This example illustrates two properties of behavioral programming
  - How different heuristics and pieces of knowledge about the problem domain can be used separately to program an application
  - How model-checking can be used to help find a desired scenario

- Dining Philosophers

  In this example we can see
  - How multiple behaviors coexist independently of each other, with very simple rules for each and
  - how model-checking can help finding conflicts and other desired and undesired properties.

2.1 Incremental Development Of Behavioral Program with Model-Checking

In [HLMW11], authors have introduced a model-checker. Incremental development of Tic-TacToe using that model-checker can be understood as follows.This example is borrowed from [HMW10][HMW12][HLMW11]. In classical game of tictactoe there are two players, X and O, alternatively mark on squares on a 3x3 grid. Squares are identified by ¡row,column¿ pairs: ¡0,0¿,¡0,1¿,..,¡2,2¿. The player who is able to form a full horizontal, vertical or diagonal line with three of his marks. If no player is able to form a line and entire grid is marked, the result is a draw.

In this example player X is human and O is application. Each move (marking a square by a player) is represented by an event, X(row,col) or O(row,col). The events XWin, OWin, and draw represent human's win, application's win and draw respectively. A play of game may be described as a sequence of events, e.g., the sequence X(0,0),O(1,1),X(2,1),O(0,2),X(2,0),O(1,0),X(2,2 describes a play in which X wins.

The model checking tool used here is BPmc (behavioral programming model checker). Initially we may think of basic requirements like :

- SquareTaken: Once a square is marked with X or O, block its further marking.
- EnforceTurns: Alternatively block O moves while waiting for X moves, and vice versa.
- DetectXWin, DetectOWin: Wait for placement of three X marks (,three O marks) in a line and request XWin(, OWin).
- DetectDraw : Wait for nine moves and request draw event.

DetectDraw is given lower priority than DetectXWin and DetectOWin. If reverse is done, then if X wins in the ninth move, draw will be triggered instead of XWin. Since all b-threads mentioned above first wait for some event, and then proceed so we need some

b-thread which requests some event, which is needed for application to proceed. Now we add the components that request these events:

- A GUI component that translates each user-click to a corresponding X event.
- DefaultMoves: a b-thread that repeatedly requests all nine possible O moves in the order center, corner and edges: O(1,1), O(0,0), O(0,2), O(2,0),(2,2), O(0,1),O(1,0),O(1,2),O(2,1)

Now we use the model checker to gradually and incrementally enhance the program until it never loses. To verify that there is no strategy for X to win, we replace the user-driven entry of X moves with the b-thread XAllMoves, which repeatedly, nondeterministically, requests all possible X moves (this is the only nondeterminism in the program). We also need to specify that the desired safety property is that O never loses. To do this we modify the b-thread DetectXWin, where it requests the XWin even to call an API method (markNextVerificationStateAsHot()) that declares the next state of the program as bad. This causes BPmc to announce that the verification failed, and to print the relevant event sequences as a counterexample. Model-checking immediately shows that the application may loose. The counterexample trace printed by BPmc is:
X(0,0),O(1,1),X(0,1),O(0,2),X(2,0),O(2,2),X(1,0).
The victory of X could have been easily avoided if the application had played O(1,0) in its last turn preventing the completion of three Xs in a line. An obvious resolution, therefore is to add the following b-thread as a basic tactic.

- PreventThirdX: when two X's are noticed in a line, add an O in that line(and prevent an immediate loss).

Thus we see that model-checking can be used to improve our application by detecting and tracing undesired states reached by the application. The same tracing will help to find deadlock situation in case of dining philosopher problem. We need to specify all the

deadlock scenarios as safety property by above mentioned method.We also need a b-thread which requests all possible moves of the philosophers non-deterministically.

2.2 Conclusion

This chapter provided some of the existing example programs in Behavioral Programming implemented using BPJ Library. In next chapter, we discuss our four examples programmed using BPJ library.

Chapter 3

Examples Programs: Task Ordering and Deadlock Detection

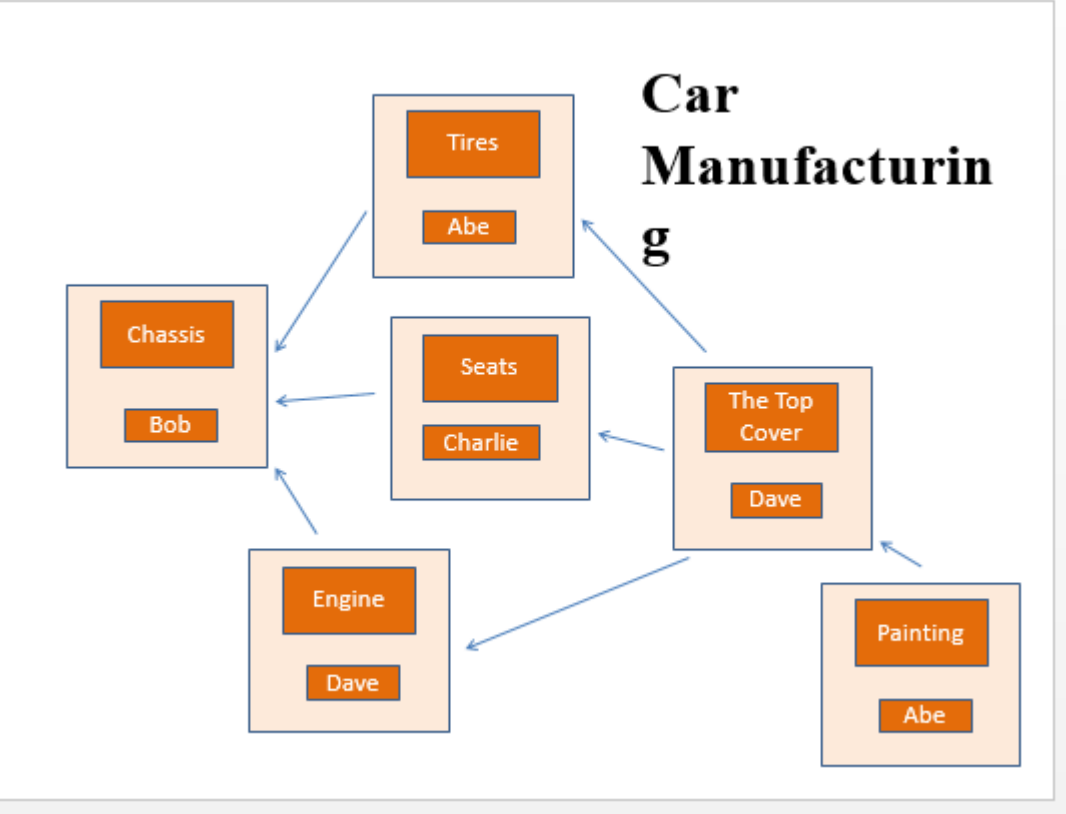
3.1 Car Manufacturing Problem

[Sou] A car is manufactured at each stop on a conveyor belt in a car factory. A car is constructed from the following parts- chassis, tires, seats, engine (assume this include everything under the hood and the steering wheel), the top cover, and painting.Thus there are six tasks involved in manufacturing a car. However, tires, seats or the engine cannot be added until the chassis is placed on the belt. The car top cannot be added until tires, seats and the engine are put in. Finally, the car cannot be painted until the top is put on. A stop on the conveyer belt in the car company has four technicians assigned to it - Abe, Bob, Charlie, and Dave. Abe is skilled at adding tires and painting, Bob can only put the chassis on the belt, Charlie only knows how to attach the seats, and Dave knows how to add the engine as well as how to add the top.

Write pseudo-code for Abe, Bob, Charlie and Dave to be able to work on the car, without violating the task order outlined above.[Car]

Following figure shows the dependencies between the tasks and skill-distribution of techni-

cians.



3.1.1 Solution

One solution of above problem in other paradigm is by using semaphores and mutexes or monitors. But we can very easily achieve the constraints mentioned in the problem in Behavioral Programming. We consider each task as an event, thus the program contains six events each with single instances, namely, Chassis, Engine, Painting, Seats, TheTopCover and Tires. Each person is represented by a BThread. Thus we have four BThreads,viz., Bob, Abe, Charlie, Dave. Each BThread Behaviors are as described below:

Bob{

labelNextVerificationState("RC")

request Chassis. // As applying chassis doesn't depend on any other task to be done first.

print Bob applying chassis.

}

Abe{

labelNextVerificationState("WC")

wait for chassis. //as chassis should be applied before tires

labelNextVerificationState("RT")

request tires

labelNextVerificationState("WCT")

wait for cartop //as cartop should be applied before painting

labelNextVerificationState("RP")

request painting

}

Charlie{

labelNextVerificationState("WC")

waitfor chassis

labelNextVerificationState("RS")

request seats

}

Dave{

labelNextVerificationState("WC")

waitfor chasis

labelNextVerificationState("RE")

request engine

//Now before applying cartop Dave need to ensure that tires, seats and engine are aleady applied.

//Since Bthread execution reached here that means engine has been applied. Now to make sure that if Dave waits for seats and tires before applying cartop, so if seats and tires are already triggered then Dave will keep on waiting for those events to occur.

So to make sure they are triggered after engine has been triggered, we keep the priorities of Abe and Charlie to be less (more in numeral) than Dave. Now here Dave has to wait for two events, viz., tires and seats. The order of waiting on these events and relative priorities of Abe and Charlie are dependent.Supppose Dave wait for those two events in the following order:

labelNextVerificationState("WS")

wait for seats

labelNextVerificationState("WT")

wait for tires

This implies we require first seats event to be selected for triggering and then Tires. So we have to set the priority of the BThread which is requesting Seats(i.e. Charlie) higher(lower in numeral) than that of the BThread which is requesting Tires(i.e. Abe). Thus following order of priorities of the BThreads meet the constraint:

Bob>Dave>Charlie>Abe and reverse in numerals.

labelNextVerificationState("RCT")

request cartop

}

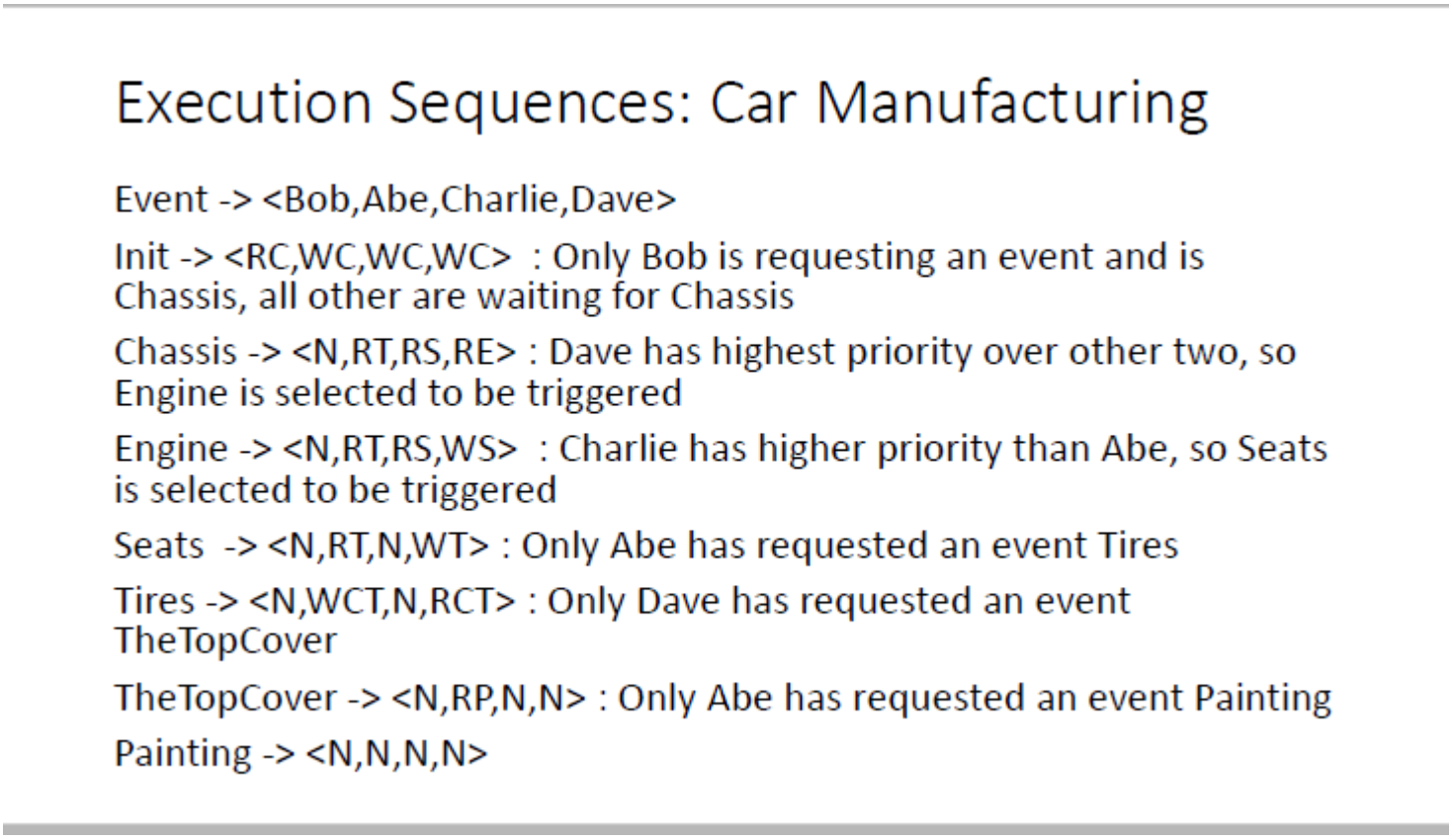
3.1.2 States and Priorities of BThreads

Following table shows the different states of the BThreads while program execution and their priorities.

Bthread	Priority	States
Bob	1.0	RC
Abe	4.0	WC,RT,WCT,RP
Charlie	3.0	WC,RS
Dave	2.0	WC,RE,WS,WT,RCT

3.1.3 Sample Execution

Sequence of <event> −> <state> denotes a sample execution of the program where <state> is the program state,i.e. cartesian product of BThread-states, which is reached after <event> is triggered.N denotes the state of a BThread when it is not selected for execution.



3.2 Dining Philosophers Problem

There are several philosophers sitting on a round table numbered 1..n There is a large bowl of spaghetti at the center There is a fork between each pair of philosopher Each philosopher may only use the fork on her right and the fork on her left A philosopher needs the two forks to eat Each philosopher when finishes eating she puts down the forks and starts thinking again We use BPJs synchronization and concurrency handling properties and program their behaviors.

3.2.1 Events and BThread

- PickUp: There are n instances of pickUpRightFork and n of pickUpLeftFork
- PutDown: There are n instances

BThread Priorities

Bthreads	Priority
Phil (n instances for n philosophers)	1.0-n.0
Fork (n instances for there are n forks for n philosophers)	(n+1).0-(2*n).0
Logger	(2*n+1).0
DeadLockScenario	(2*n+2).0
DeadLockRightHanded	(2*n+2).0

- Phil States
  - T (Thinking)
  - 1(one fork Up)
  - E (Two fork Up, Eating)
  - F (Finished Eating)
- Fork States
  - UL (Up in left hand)
  - UR (Up in right hand)
  - D (Down)
- Program states will be the Cross product of the BThread states. E.g. in case of 3 philosophers <T,D,T,D,T,D> is the initial state when all philosophers are thinking and no-one has lifted any fork.
- We have implemented Resource Hierarchy Algorithm, where, for deadlock free execution, symmetry breaking is done by enforcing last philosopher to lift right fork first in case of left-handed philosophers, and enforcing that to lift left fork first in case of right-handed philosophers.

- Starvation and fairness we have not taken in consideration, since Resource Hierarchy Algorithm is not starvation free.
- Deadlock can be detected by introducing two BThreads, one for left handed philosophers and one for right handed philosophers. Deadlock occurs only in two scenarios (The two BThreads work on these principles only):
  - Either each philosopher picks up her left fork first and waits for the right fork
  - Or each philosopher picks up her right fork first and waits for the left fork
- Model Checking can be used to detect deadlock scenarios,and for starvation and fairness checking.

```
Deadlock Scenario (n=3)
    Init-> [T,D,T,D,T,D]
    PickUp-F0-by-P0->[1,U,T,D,T,D]
    PickUp-F1-by-P1->[1,U,1,U,T,D]
    PickUp-F2-by-P2->[1,U,1,U,1,U]
[1,U,1,U,1,U] is a deadlock state
Here each line of the form <event> <State> describes a composite state of
the entire program along the path and the event whose triggering led the
program to transition from preceding state to the current.
```

3.3 Conclusion

In this chapter, we provided two examples of Behavioral Programming in BPJ which showed how multiple behaviors can coexist with little or no interaction with each other and how ordering of tasks can be achieved through exploiting the concurrency and synchronization handling feature provided by BPJ Library. Next Chapter we see another two examples of similar features with less complexities.

Chapter 4

Some lighter Examples

We see two more example problems : Producer-Consumer and Print Prime Numbers.

4.1 Producer-Consumer Problem

Two processes, producer and consumer share a common fixed size buffer used as a queue.Producer generates the data and puts into the buffer.Consumer consumes the data by removing it from the buffer one at a time.Following constraints should be met:

- Producer shouldnt add data to the buffer if it is full
- Consumer shouldnt remove data from empty buffer

4.1.1 General Solution

Add two library routines sleep and wakeup When sleep is called , the caller is blocked until another process wakes it up by using wakeup routine Add a global variable itemCount to hold the number of items in the buffer. But this solution can lead to deadlock. So we solve this using two semaphores fillCount and emptyCount:

- fillCount : number of items present in the buffer
- emptyCount:number of items spaces available in the buffer

When a new item is put into the buffer increment fillCount and decrement emptyCount Put producer to sleep if it tries to decrement emptyCount when it is 0. Wake producer up when item is consumed next time and emptyCount is incremented The solution can lead into two or more processes reading or writing into the same slot at the same time, so to overcome this execute a critical section with mutual exclusion.

4.1.2 Producer-Consumer Our Solution

Events

- Consume
- Produce
- ResourceEmpty
- ResourceFull

BThreads and Priorities

BThread	Priority
Producer	3.0
Consumer	3.1
FullResourceChecker	2.0
EmptyResourceChecker	2.1
BlockProduce	1.0
BlockConsume	1.1

Behavior of Bthreads

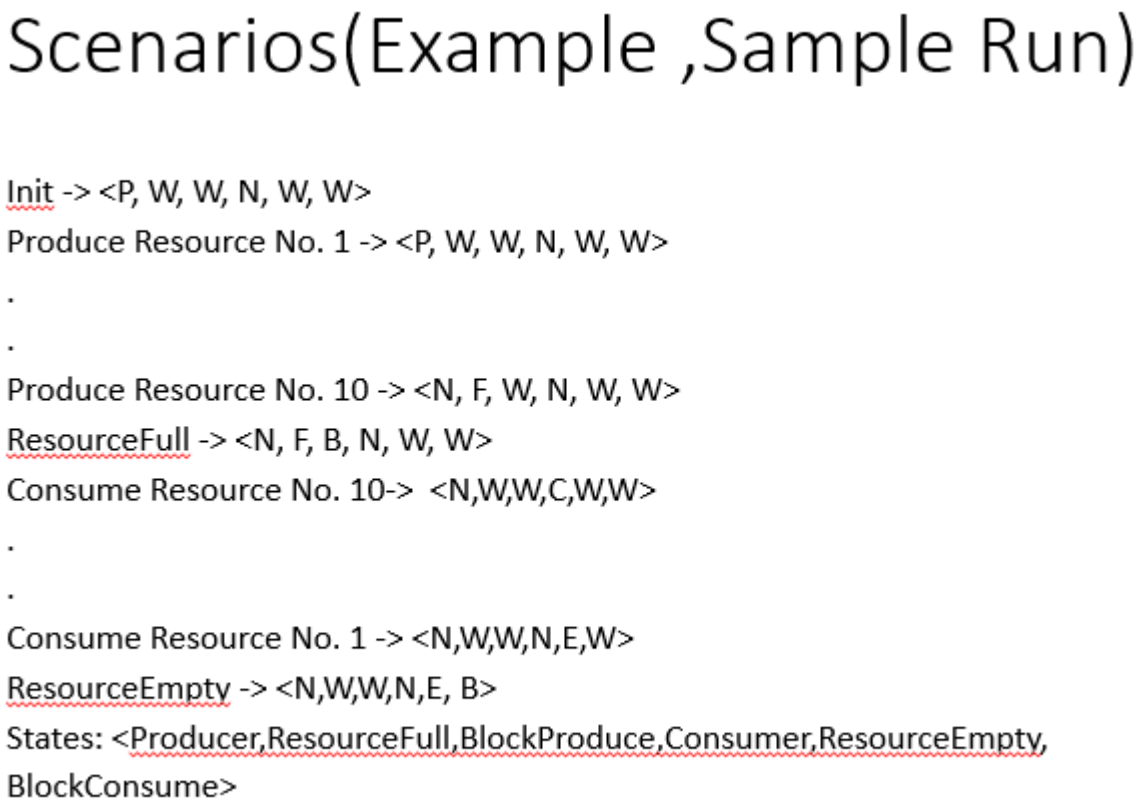
- Producer: Increment nResources and raise event Produce
- FullResourceChecker: Wait for Produce event and check if nResources equals limitRes, raise ResourceFull event.
- BlockProduce: Wait for ResourceFull event and Block Produce Event
- Consumer: Decrement nResources and raise event Consume.
- EmptyResourceChecker: Wait for Consume event and check if nResource equals 0, raise ResourceEmpty Event.
- BlockConsume: Wait for ResourceEmpty Event and block Consume event.

Here nResources keeps track of currently available number of items in the buffer, and limitRes is the number of maximum resources to be produced or buffer size limit.

BThreads & States

Bthreads	States
Producer	N, P
Consumer	N, C
FullResourceCheker	N,W, F
EmptyResourceChecker	N,W, E
BlockConsume	N,W, B
BlockProduce	N,W, B

N => Not Selected to run in current state, P => Producing, C => Consuming, W => Waiting state, F => Resource Full state, E => Resource Empty State, B => Requesting blocking of some event



Sample Run

4.2 Generate Prime Numbers from 1 to n

We implement here algorithm to generate prime numbers from 1 to n in behavioral programming paradigm. A generator BThread requests events labelled with all integers starting from 2 to n. Another BThread (PrintLabelBThread) prints the label associated with the current event. Yet another Bthread (BlockNonPrimesBThread) listens on the same event and blocks all the events labelled with the multiple of the label of the current event.
Only one event named NumberEvent is there whose n instances are used in the program.

BThreads and their Priorities and states

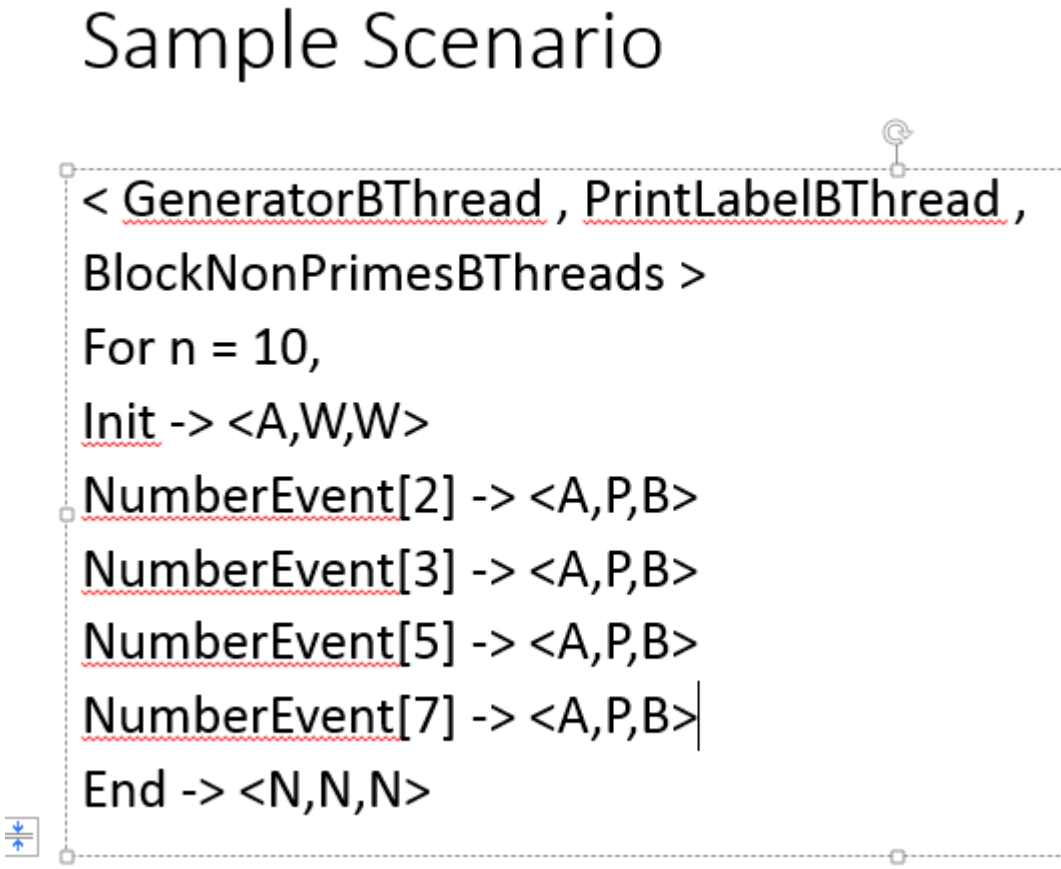
BThread	Priority	State
GeneratorBThread	3.0	N,A
PrintLabelBThread	2.0	N,W,P
BlockNonPrimesBThreads	1.0	N,W,B

N denotes BThread is inactive(not chosen in current run)
A denotes Bthread is active (chosen and performing some task)
W denotes BThread is waiting on some event
B denotes BThread has requested to block some event

Sample Output:

Sample output: Print Primes till n

- Application Parameters: n=20
- Event #1: 2(ID=2) requested by 2
- 2
- Event #2: 3(ID=3) requested by 3
- 3
- Event #3: 5(ID=5) requested by 5
- 5
- Event #4: 7(ID=7) requested by 7
- 7
- Event #5: 11(ID=11) requested by 11
- 11
- Event #6: 13(ID=13) requested by 13
- 13
- Event #7: 17(ID=17) requested by 17
- 17
- Event #8: 19(ID=19) requested by 19
- 19



Sample Run or Scenario

This was a simple program just to illustrate how we can do it in behavioral programming paradigm.

4.3 Conclusion

In this chapter, we provided another two examples of behavioral programming, one was producer-consumer problem to solve synchronization and concurrency issues and other was just to illustrate that such algorithms can also be implemented in BPJ.

Chapter 5

Conclusion and Future Work

We implemented four simple programs, in BPJ Library as a illustration of how to program applications in Behavioral Programming using BPJ Library, namely , dining philosopher problem(Resource Hierarchy Algorithm), Car Manufacturing Problem (To meet constraints of order of the different tasks), Producer Consumer Problem (Sample Execution) and Printing Prime Numbers till a specified limit. We couldnt verify the constraints through Programming by specifying anti-scenario (due to ,may be ,a bug in BPJ library, which is giving error as null-pointer exception while running the program with anti-scenario included). Also we couldnt do model-checking to improve our application behaviors.For example for finding deadlock and starvation scenarios and improving upon them in dining philosophers problem model-checking can be utilized. We can program more complex systems like quadrotor controller, robots which also interacts with external environment with the help of appropriate hardware supports like sensors.

References

[AHP96] Rajeev Alur, Gerard J Holzmann, and Doron Peled. An analyzer for message sequence charts. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 35–48. Springer, 1996.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. Expressing and analyzing timing constraints in message sequence chart specifications. 1997.

[BHKS97] Manfred Broy, Christoph Hofmann, Ingolf Krüger, and Monika Schmidt. *A graphical description technique for communication in software architectures.* Citeseer, 1997.

[bpE] *BPJ Programming Examples.*

[Car] *Synchronization Review.*

[CHK08] Pierre Combes, David Harel, and Hillel Kugler. Modeling and verification of a telecommunication application using live sequence charts and the play-engine tool. *Software & Systems Modeling*, 7(2):157–175, 2008.

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

[EGH+11] Nir Eitan, Michal Gordon, David Harel, Assaf Marron, and Gera Weiss. On visualization and comprehension of scenario-based programs. In *Program Com-*

prehension (ICPC), 2011 IEEE 19th International Conference on, pages 189–192. IEEE, 2011.

[HKK13] David Harel, Amir Kantor, and Guy Katz. Relaxing synchronization constraints in behavioral programs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 355–372. Springer, 2013.

[HLMW11] David Harel, Robby Lampert, Assaf Marron, and Gera Weiss. Model-checking behavioral programs. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 279–288. ACM, 2011.

[HM03] David Harel and Rami Marelly. *Come, lets play: scenario-based programming using LSCs and the play-engine*, volume 1. Springer Science & Business Media, 2003.

[HMSB10] David Harel, Shahar Maoz, Smadar Szekely, and Daniel Barkan. Playgo: towards a comprehensive tool for scenario based programming. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 359–360. ACM, 2010.

[HMW10] David Harel, Assaf Marron, and Gera Weiss. Programming coordinated behavior in java. In *ECOOP 2010–Object-Oriented Programming*, pages 250–274. Springer, 2010.

[HMW12] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[LL95] Peter B Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7(5):473–509, 1995.

[MH06] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling lscs into aspectj. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 219–230. ACM, 2006.

[MHK02] Rami Marelly, David Harel, and Hillel Kugler. Specifying and executing requirements: the play-in/play-out approach. In *Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 84–85. ACM, 2002.

[OnV] *BPJ Trace Visualizer*.

[Rob05] James Roberts. Tracevis: an execution trace visualization tool. In *In Proc. MoBS 2005*. Citeseer, 2005.

[Sou] *SourceCode for Example Programs*.

[WK98] Jos B Warmer and Anneke G Kleppe. The object constraint language: Precise modeling with uml (addison-wesley object technology series). 1998.