2016 15th International Symposium on Parallel and Distributed Computing

Distributed Safety Verification Using Vertex Centric Programming Model

Adhish Singla*, Krishnaji Desai†, Suresh Purini* and Venkatesh Choppella*
* International Institute of Information Technology, Hyderabad, India.
† Hitachi India Private Limited, Bangalore, India.

***Abstract*—Software is finding place in deeply embedded systems to large scale distributed systems of cloud service providers such as Amazon and Google. Due to the concurrent and distributed nature of this software, it is hard to test for correctness of such systems in a foolproof manner. Explicit state model checking is an approach in which we build a model of the system and specify the properties it should hold. Then we construct a state transition system from the model and check if it satisfies the specified properties. There are two kinds of properties of interest: safety and liveness. In this paper, we focus our attention on safety verification, which involves checking if the states that are generated in the transition system satisfy some predicate formulae specified in the form of assertions. The main problem here is that the number of states in the transition system grows exponentially with the number of bits required to store the state of a model at any given point time. So the available main memory even in a server class machine is not sufficient to model check non-trivial practical models. One approach to address this problem is by using resources from a distributed collection of machines. In this paper, we adopt this approach, by proposing a distributed safety property verification algorithm using the vertex centric programming model.**

I. Introduction

Since the past decade, there has been widespread adoption of ICT technologies in domains such as health care, automobiles, mass transit systems etc. The hardware and software systems developed for such domains need to be reliable, as any unexpected behaviour potentially leads to both monetary and human loss. We can use rigorous software engineering principles during the design, development and testing phases in order to increase the reliability of the built systems. In spite of this, due to the inherent complexity of these systems and human involvement in building such systems, no guarantees can be made about their correctness. However, we can use automated formal verification and model checking tools to identify bugs early in the design cycle and thereby gain more confidence on the system with exhaustive coverage. Such tools are not only useful in real time embedded system applications but also in large enterprise and cloud services companies such as Amazon which use complex distributed algorithms in their backend infrastructure [1].

A. Explicit State Model Checking

In model checking, we first build a model of the system under consideration using a modeling language such as Promela [2]. The system can be a distributed algorithm [3], a network protocol [4] or a hardware design [5]. Then we specify the property to be verified either within the model description itself as *assertions* or separately with Linear Temporal Logic (LTL) formulae. Then a model checker such as SpinJa [6] takes the model description and the property to be checked to generate a verifier. The generated verifier, in the case of SpinJa, is a Java file which when interpreted checks if the model satisfies the specified property by systematically exploring the underlying state transition system corresponding to the model. In case of any violation, it generates a trace which can be used to do a guided simulation to identify the cause of violation. Figure 1 depicts the complete verification process that is typically followed.

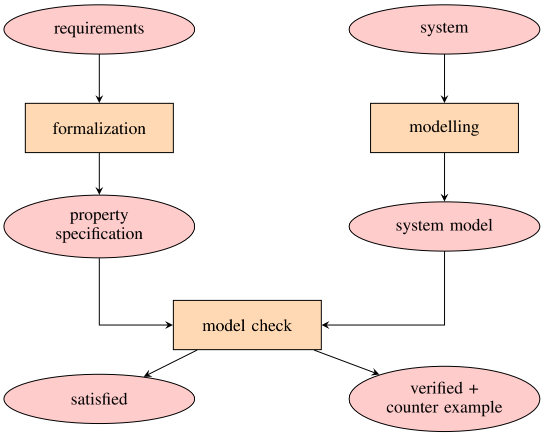

Fig. 1: Overview of the model checking process.

There are broadly two types of properties that we are interested in model checking: *safety* and *liveness* checks. Intuitively, safety properties say that nothing bad ever happens and liveness properties indicate that something good eventually occurs. For example, in an algorithm for mutual exclusion, a safety property would be that at any point of time, no more than one process is in the critical section. Whereas a liveness property would be that every process which attempts to enter the critical section will eventually do so. Figure 2 shows the Promela model specification for the Peterson's algorithm. The safety properties are specified using either inline assertions or global state invariants using simple LTL formula such as $\Box(mutex!{=}2)$. This means in every state of the transition system the predicate $(mutex!{=}2)$ has to be true. Specification of liveness properties usually requires more complex LTL formulae and is out of the scope of this paper.

Given a Promela model M, a model checker constructs a state transition system $T_M = (S, Act, \rightarrow, I, L, AP)$. The transition system can start in any of the initial states from

978-1-5090-4152-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ISPDC.2016.23

I and move from one state to another in S as a result of the actions performed from the set Act in accordance with the edge transition relation $\rightarrow \subseteq S \times Act \times S$. AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a labeling function which associates with every state the set of atomic propositions valid in that state. We say that a state $s \in S$ satisfies a predicate ϕ, denoted as $s \models \phi$, if and only if $L(s) \models \phi$. Each state of the transition system is a snapshot of the Promela model in execution and the transition system completely captures all possible behaviours of the Promela model across various execution runs. For example, in the Peterson's mutual exclusion algorithm [7] from Figure 2, each state contains the current values of the global variables, program counter values for the processes A and B, and the value of their local variables.

A safety check algorithm systematically generates all the possible reachable states in a transition system either in a breadth-first search or a depth-first search manner. Safety assertion checks are performed as and when they are encountered or on all the states of the system based on user specification. If the number of bits required to encode a state is b, then there are 2^b distinct possible states, although all of them may not be reachable. So the total number of states in the underlying state transition system can grow exponentially with the increase in the model complexity. This is the main bottleneck in the practical application of model checking. For real models of practical interest, the number of generated states are so large that they cannot be stored in the main memory of even high end machines. This is called as the state space explosion problem.

In this paper, we show how we can address this problem using the compute and memory resources in a distributed cluster of machines. Although there are prior efforts in this direction [8], [9], [10], [11] the novelty of our work comes from using distributed vertex centric graph processing frameworks such as Giraph [12]. The programming model provided by Giraph makes the design of our distributed model checker easier. Further, other necessary properties required while solving large distributed problems such as scalability and fault-tolerance are naturally provided by such frameworks as against custom built distributed applications using primitive tools such as Message Passing Interface.

The following is the brief outline of the rest of the paper. In Section II, we provide the necessary background on vertex centric graph processing frameworks. The architecture and the implementation details of our distributed safety checking algorithm is presented in Section III. In Section IV, we present experimental results and conclude in Section VI.

II. Vertex Centric Computing Model

A graph is a versatile mathematical object which can be used to represent diverse real world structures such as World Wide Web, social networks, geographical maps etc. In the context of model checking, a state transition system can be viewed as a directed graph. We can infer many application specific properties from its corresponding graph representation by performing algorithms such as reachability, shortest path, max-flow etc. However, the size of graphs such as web graph is so large that they do not fit into the main memory of even a server class machine and has to be stored using distributed secondary storage structures. Even if we can do streaming computation to overcome the main memory limitations, moving data across network is way too expensive. So it makes sense to use a map-reduce kind of computational framework for scalable distributed processing of large graphs. However, graph algorithms are not easily amenable to map-reduce framework.

```
#define A_TURN 0
#define B_TURN 1

bit x, y;
byte mutex;
byte turn;

active proctype A() {
        x = 1;
        turn = B_TURN;
        y == 0 || (turn == A_TURN);
        mutex++;
        assert(mutex != 2);
        mutex--;
        x = 0;
}

active proctype B() {
        y = 1;
        turn = A_TURN;
        x == 0 || (turn == B_TURN);
        mutex++;
        assert(mutex != 2);
        mutex--;
        y = 0;
}
```

Fig. 2: Promela model specification for the Peterson's mutual exclusion algorithm.

In this context, vertex centric computing frameworks such as Pregel [13] and Giraph [12] are useful from algorithmic design perspective because they are accompanied with suitable programming models. Since, we used Giraph in our experimental work, we refer to it more specifically in subsequent discussion. In Giraph, computations on a graph happen in rounds. In each round, every active vertex performs local computations independent of others. The input to a local computation is the current local state at the vertex and the incoming messages from its predecessor nodes in the graph. The local computations can change the state of the vertex and potentially send some outgoing messages to the successor nodes. Each round of computation is called a super-step and computations across all the vertices are synchronized at the end of each super-step. The computations come to a halt when there are no more active vertices. An abstract Giraph computation is realised by a collection of worker threads running on a cluster of machines. Every vertex in the graph is assigned to a worker and does its computation in the corresponding worker context. This induces a canonical partition on the set of all vertices.

We illustrate the vertex centric graph computational model of Giraph using the shortest path computation algorithm as applied on the graph in Figure 3. In this algorithm, each vertex maintains a current estimate of the shortest path from the source vertex. In Figure 4, the value in the square bracket below each vertex reflects the local state of the corresponding vertex and it shows the current estimate of the shortest path length. The vertices are divided into two partitions $\{v_0, v_1\}$ and $\{v_2, v_3\}$. Each partition is assigned to a worker thread.

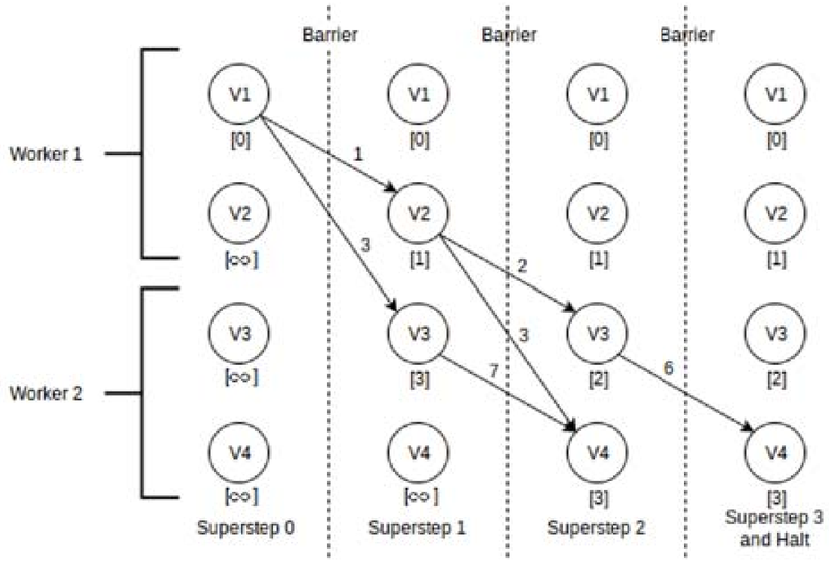

Fig. 4: Illustration of the shortest path computation from the source vertex v_1 for the graph in Figure 3 using the Giraph computational model. Labels on the arrows indicate the payload of the communication from one vertex to another.

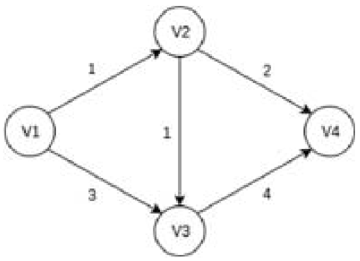

Fig. 3: Example graph to illustrate the shortest path computation from vertex v_1 using Giraph.

The worker threads can be co-located on the same machine or can be assigned different machines. In *superstep-0*, only vertex v_0 is active and it sends messages to the successor vertices v_2 and v_3. They become active in the next round and carry on the computation. Every active vertex in each stage, updates its shortest path estimate based on the incoming messages. Finally, after the *superstep-4*, no vertices are active and the computation comes to halt finishing the execution.

III. Distributed Safety Checking

Let $T_M = (S, Act, \rightarrow, I, L, AP)$ be a transition system associated with a model M. An execution run in the transition system is an infinite sequence of states $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ such that (s_i, s_{i+1}) is a valid transition for all $i \geq 0$. $L(s_0)L(s_1)L(s_2)\cdots$ is the trace obtained from such an execution run by projecting the corresponding state labels. We say that an execution run satisfies a safety property specified by a predicate formula ϕ if $\forall i \geq 0$, $L(s_i) \models \phi$. We say that a transition system satisfies a safety property ϕ if every possible execution run in the transition system satisfies the safety property. If there exists an execution run which violates the safety property, then there exists a state s_i in that run for some finite i such that $L(s_i) \not\models \phi$. Since the execution run is a valid run, the state s_i is reachable from one of the initial states in I. Based on this discussion, we can infer that a safety property ϕ can be verified by computing the set of reachable states in T_M starting from all the initial states and checking if each of the reachable states logically entails the safety predicate ϕ.

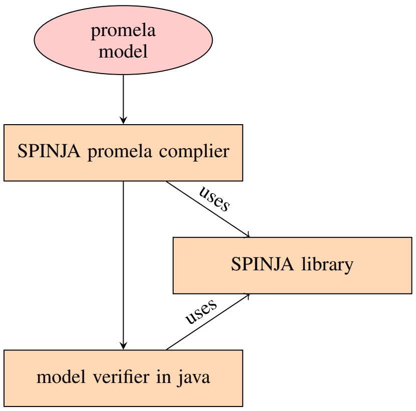

Fig. 5: Overview of the SpinJa model checker flow and architecture.

SpinJa [14], [15] is an explicit state model checker developed in Java using an object oriented approach so that it is easy to extend it by adding new algorithms without touching the main code base, that is related to, for example Promela model file parsing, next transition generation etc. Figure 5 shows the basic architecture of SpinJa. The main idea is that any extensions to the model checker can be implemented as a part of the SpinJa library. This makes it an attractive choice

```
#define NOPROC 3  /* No of processes */
#define LC 20    /* Increment from each process */
int gc = 0;
int nrpr=0;
int noproc=NOPROC;
proctype P() {
        int x=1, temp;
        do
        :: x > LC -> break;
 // Use this line assertion violation
//      :: else ->  temp=gc+1; gc=temp; x=x+1;
// Use this line for no assertion violation
        :: else ->  gc=gc+1; x=x+1;
        od;
        nrpr=nrpr+1;
}
init {
        int x=1;
        do
        :: x > NOPROC -> break;
        :: else -> x=x+1; run P();
        od;

        (nrpr==NOPROC);
        assert(gc == LC*noproc);
}
```

Fig. 6: Promela model simulating a Shared Counter.

```
public class PanModel extends PromelaModel {

        public static void main(String[] args) {
                Run run = new Run();
                run.parseArguments(args,"Pan");
                run.search(PanModel.class);
        }

        int gc;
        int nrpr;
        int noproc;
            .
            .
       public class init_0 extends PromelaProcess {
                protected int _pc;
                protected int _pid;
                protected int x;
                   .
                   .
        }
        public class P_0 extends PromelaProcess {
                protected int _pc;
                protected int _pid;
                protected int x;
                protected int temp;
                   .
                   .
       }
           .
           .
}
```

Fig. 7: Partial structure of the PanModel class.

for this project as against an industry strength tool like Spin [2] which generates the code for search algorithms also as a part of the generated verifier.

Figure 6 depicts the Promela model which is used in our experimental work and is explained further in Section IV. When such a model specification is given as input to SpinJa, it generates a Java class with the default name PanModel. The PanModel class contains inner classes which correspond to the processes and channels defined in the Promela model specification. Corresponding to the global and local variables in the model specification, there are variables declared globally in the PanModel and the inner process classes (refer Figure 7). Further, the PanModel class contains methods which capture the dynamics of the model using which transitions from one state to another state can be obtained. In the safety and liveness checking algorithms, as we explore the state transition graph, we use the PanModel object to move from one state to another state. The problem is that it is too expensive to associate an object with every state of the transition graph. So, we use serialization (encode) and deserialization (decode) functions to store and retrieve only the required information in a PanModel object. During the state transition graph exploration, for example, using the breadth first search algorithm, only the state information is enqueued and dequeued from the priority queue.

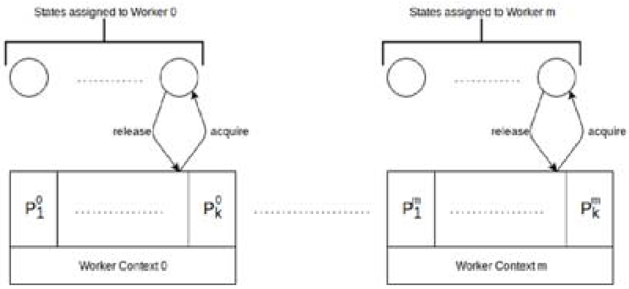

Fig. 8: Architecture of Distributed SpinJa based on Giraph.

In our architecture for distributed safety checking using Giraph, refer 9 each state of the transition system is modelled as a vertex in the graph. The computation associated with every vertex involves generating the next state vertices. This is how the transition graph gets built on-the-fly starting from one or more initial state vertices. Since a vertex gets active immediately in the next superstep of its creation, the transition graph gets explored in a breadth-first search manner. Further, since all the active vertices generate their successors simultaneously, the frontier of the transition graph gets expanded in parallel, giving rise to an extremely scalable breadth-first search exploration. During this process of graph exploration, any assertion statements that are encountered are checked with respect to the current state of the system.

In the centralized stand alone SpinJa version, the transition graph is explored by considering one vertex at a time. A vertex state is de-serialized into a PanModel object in order to generate its next state vertices. In distributed version, instead of maintaining one PanModel object, we maintain a concurrent pool of PanModel objects. This concurrent pool is shared across all the vertices assigned to the same worker using the WorkerContext mechanism provided by Giraph. Any active vertex can procure and release a PanModel object from this concurrent pool for its operation.

IV. Experimental Results

We tested the effectiveness and scalability of our distributed SpinJa [14], [15] implementation based on Giraph framework (with Hadoop 2.4.0) using the synthetic Shared Counter Promela model from Figure 6 and the Dining Philosopher's problem from the BEEM benchmark suite [16]. In the

Shared Counter Promela model,we can control the number of states in the underlying state transition system by varying the parameters NOPROC and LC. Since the only source of non-determinism in this model is the progress of each process, the parameter NOPROC gives the outdegree of each state in the state transition graph. Figure 9 shows how the number of vertices and edges in the transition system grows as the parameter LC varies from 15 to 21 with the NOPROC parameter set to 4. We are able to apply our distributed safety checking algorithm on a transition system with 10 million vertices and 40 million edges.

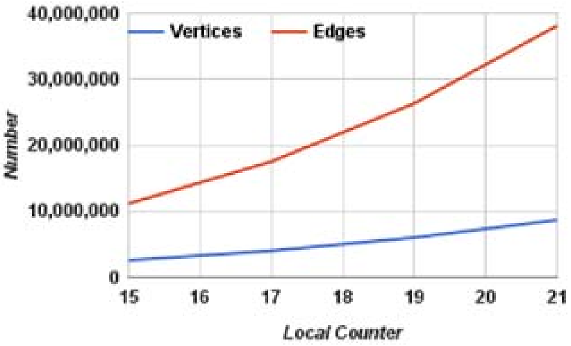

Fig. 9: Increase in vertices and edges as the LC parameters grows in the Shared Counter model when the parameter NOPROC is set to 4.

We performed our experiments using the Elastic Map Reduce (EMR) infrastructure form Amazon Web Services (AWS). Table I shows the configuration of the m3.2xlarge EC2 machines we used in our map-reduce cluster on AWS.

vCPUs	8
Memory (GB)	30
SSD Storage (GB)	2*80

TABLE I: m3.2xlarge EC2 machine Configuration.

Table II shows how the number of supersteps, time taken and memory consumed metrics grow with the number of vertices in the state transition system of the Shared Counter Promela model as controlled by the LC parameter. The statistics are obtained using 2 EC2 machines with 7 Giraph workers each with 2.5 GB memory. Figure 10 shows how the number of vertices and the time taken for safety checking grows with the parameter LC. It can be noticed that the time taken increases sub-linearly with the number of vertices due to the parallel state graph exploration.

Given the same set of physical resources, the performance of a Giraph application can change with the number of workers and the memory allocated to each worker. Table III shows the performance of our distributed safety checking algorithm on the Shared Counter model (NOPROC=4, LC=15) as a function of the physical resources allocated, number of Giraph workers and their memory allocation. The first observation is that the total time taken for execution is decreasing with increase in the memory allocated to each worker thread although the number of worker threads are decreased. This indicates that the benefits that we get with more main memory are better than that of parallelism. Also, as we increase the number of physical machines, the execution time is increasing. This is due to the latency in communication across machines. Table V shows analogous statistics for the Dining Philosopher's problem with nine philosophers and the trends are similar to that of the Shared Counter model. Table IV shows the basic statistics like number of supersteps, number of vertices and edges in the state transition graph for the same. Overall, this indicates that using our distributed model checking algorithm we may not be able to reduce the verification time by increasing the number of physical machines due to network latencies. However, they would be useful to solve large problems involving millions of state which cannot be solved using one or more small number of machines.

LC	Supersteps	Vertices	Edges	Time (sec)	Memory (MB)
15	144	2,559,497	11,151,040	934	6222.23
17	160	4,011,013	17,537,561	1523	7267.16
19	176	6,006,454	26,337,613	2290	8598.17
21	192	8,667,750	38,096,386	3276	10913.32

TABLE II: Summary statistics on the Shared Counter model using 2 physical machines with 7 Giraph workers each of 2.5 GB memory.

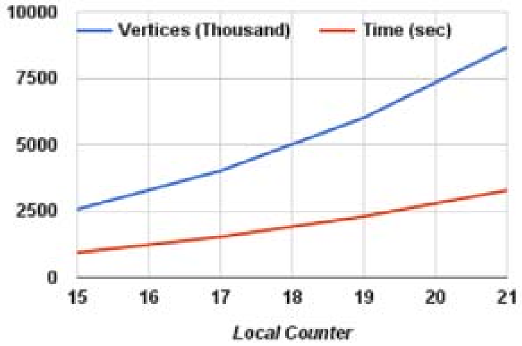

Fig. 10: Growth of the metrics Time and Vertices with the parameter LC in the Shared Counter model.

V. Related Work

State space explosion in explicit state model checking is a well studied problem in the formal verification community. The problem is worse in asynchronous paradigms due to the extra concurrency produced by unbounded delay models. To avoid this problem, efficient modeling techniques, symbolic methods, abstraction and compositional reasoning have been explored by many researchers. Clarke et al. [17] proposed state

Memory per container (MB)	2 Machines			3 Machines			4 Machines		
	Workers	Time (sec)	Memory (MB)	Workers	Time (sec)	Memory (MB)	Workers	Time (sec)	Memory (MB)
1024	20	996	8049.03	22	1857	8579.18	30	2117	8649.48
2048	10	965	6880.72	10	1826	12428.93	28	2038	13482.72
2560	7	934	6222.23	15	1801	12042.54	24	2033	16939.62

TABLE III: Performance of our distributed safety checking algorithm on the Shared Counter (NOPROC=4, LC=15) model for various resource configurations.

Memory per container (MB)	2 Machines			3 Machines			4 Machines		
	Workers	Time (sec)	Memory (MB)	Workers	Time (sec)	Memory (MB)	Workers	Time (sec)	Memory (MB)
1024	20	303	6592.61	22	498	7035.75	30	637	7962.85
2048	10	285	5163.32	10	483	11823.01	28	581	12707.62
2560	7	272	4742.17	15	461	11058.29	24	576	14283.10

TABLE V: Performance of our distributed safety checking algorithm on the Dinning Philosopher's problem with nine philosophers for various resource configurations.

Total Number of SuperSteps	30
Total Number of Vertices	435,518
Total Number of Edges	2,198,970
Total Number of Messages Sent	5,505,725

TABLE IV: Basic statistics for the Dinning Philosopher's problem with nine philosophers.

space reduction using partial order techniques by recognizing the fact that the outcome of many independent process step interleavings is the same. Bit-state hashing [18] using bloom filters allows us to control the main memory consumed but leaves parts of the state transition system unexplored due to hash collisions. By using different hash functions in parallel, Swarm [11] can perform parallel search on a cluster of machines. LTSMIN [19] uses symbolic techniques for state space reduction and also relies on distributed reachability analysis for faster verification process. Divine [8], [9] is yet another popular tool which used MPI framework for distributed liveness checking. Divine-CUDA [10] attempts to port the liveness checking algorithms onto GPU accelerators. Brim et al. [20] proposed a liveness checking algorithm wherein we discover an accepting cycle in a product Buchi automaton using a series of graph transformations. The algorithm is very much amenable for implementation in Giraph programming model as it looks almost similar to the Dijkstra's shortest path algorithm. The amount of state stored in each vertex is constant and the message size is also constant. The main problem is that the liveness checks cannot be performed on-the-fly in parallel with the graph exploration. Xie et al. [13] proposed yet another liveness checking algorithm based on Pregel framework. The main problem with their approach is that its message complexity can be very high.

VI. Conclusions and Future Work

In this paper, we proposed an architecture to extend the SpinJa model checker for distributed verification of safety properties using Giraph which supports a vertex centric programming model. This allows us to apply explicit state model checking on industry scale applications which contain huge number of states in the underlying state transition model. By using Amazon Web Services, we are able to acquire a appropriately large map-reduce cluster for our purpose and release it back after the computational job is over. This also demonstrates the financial feasibility of large scale model checking. In our future work, we would like to optimize the cost-performance ratio by dynamically scaling the map-reduce cluster based on the resources required and user specified service requirements.

References

[1] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015.

[2] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[3] M. Ben, *Principles of Concurrent and Distributed Programming, Second Edition*, 2nd ed. Addison-Wesley, 2006.

[4] "Protocol specifications, testing and verification design and validation of protocols: a tutorial," *Computer Networks and ISDN Systems*, vol. 25, no. 9, pp. 981 – 1017, 1993.

[5] T. Kropf, *Introduction to formal hardware verification*. Springer Science & Business Media, 2013.

[6] M. Jonge, "The SpinJ model checker: a fast, extensible, object-oriented model checker," 2008.

[7] G. L. Peterson, "A new solution to Lamport's concurrent programming problem using small shared variables," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 1, pp. 56–65, Jan. 1983. [Online]. Available: http://doi.acm.org/10.1145/357195.357199

[8] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček, "Divine: A tool for distributed verification," in *Proceedings of the 18th International Conference on Computer Aided Verification*, ser. CAV'06, 2006, pp. 278–281.

[9] J. Barnat, L. Brim, and P. Ročkai, "Scalable multi-core ltl model-checking," in *Proceedings of the 14th International SPIN Conference on Model Checking Software*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 187–203. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770532.1770550

[10] J. Barnat, P. Bauch, L. Brim, and M. EšKa, "Designing fast ltl model checking algorithms for many-core gpus," *J. Parallel Distrib. Comput.*, vol. 72, no. 9, pp. 1083–1097, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2011.10.015

[11] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, 2008, pp. 1–6.

[12] "Giraph," http://giraph.apache.org.

[13] M. Xie, Q. Yang, J. Zhai, and Q. Wang, "A vertex centric parallel algorithm for linear temporal logic model checking in pregel," *J. Parallel Distrib. Comput.*, vol. 74, no. 11, pp. 3161–3174, Sep. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.07.009

[14] M. de Jonge, "The SpinJ model checker," Master's thesis, University of Twente, Enschede, The Netherlands, September 2008.

[15] "Spinja," https://github.com/Hitachi-India-Pvt-Ltd-RD/spinja.

[16] R. Pelánek, "Beem: Benchmarks for explicit model checkers," in *Proceedings of the 14th International SPIN Conference on Model Checking Software*, 2007, pp. 263–267.

[17] E. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287. [Online]. Available: http://dx.doi.org/10.1007/s100090050035

[18] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, no. 5, p. 279, 1997.

[19] S. Blom, J. van de Pol, and M. Weber, "LTSMIN: Distributed and symbolic reachability," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010, pp. 354–359.

[20] L. Brim, I. Černá, P. Moravec, and J. Šimša, "Accepting predecessors are better than back edges in distributed ltl model-checking," in *Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 352–366.