Static Analysis of Distributed Systems : An Approach

Awadhesh Kumar Singh[1] Umesh Ghanekar[2]

Abstract

The paper introduces an approach to static analysis of distributed systems. It is motivated by Dijkstra's weakest precondition calculus. Dijkstra developed it originally for reasoning about the correctness of the sequential programs. We propose to extend the proof technology into the realm of distributed systems. Another goal is to reason formally about the possible behaviors of a system consisting of distributed components. The contribution of the paper is the development of a style of modeling and reasoning about the properties that allows for a straightforward and thorough analysis of distributed systems. The well-known dining philosophers problem serves as an illustration for the notation.

Keywords: distributed systems, weakest precondition, weakest co-operation, correctness

1. INTRODUCTION

1.1 Background

Distributed system is a collection of processor–memory pairs connected by a local area network or distributed over a large geographical area. The processors communicate in various unpredictable ways, because distributed systems are inherently concurrent, asynchronous and non-deterministic. These characteristics make them more complex than sequential systems [2]. The set of tasks running concurrently make the environment more complicated. Therefore, CORRECTNESS is a major consideration in the design of distributed systems. Normally system specification depicts operational requirements and it does not include the properties like liveness, fairness, deadlock freedom, mutual exclusion, etc. Implementer's goal should be to include these properties in order to achieve the correct system design. The design of distributed systems is known to be a complex task, because the behavior of a distributed system results from interactions between concurrent processes of which the system consists [17]. Hence, the modeling tools are needed for helping in specifying the systems and in reasoning about the correctness of above-mentioned properties.

In order to guarantee the correct behavior of an implemented system, it is very important to start the system design with a correct specification. The use of formal techniques has shown to be highly desirable to aid the whole process of handling these problems, as they can produce descriptions without ambiguities, duplicity or lack of information [13]. The formal techniques are the applied mathematics of computer systems engineering, providing a means of calculating and hence predicting what the behavior of a system will be prior to its implementation. Moreover, formal verification has long been promised as a means of reducing the amount of testing required to ensure correct systems. They can be

[1]Department of Computer Engineering, National Institute of Technology, Kurukshetra 136119, India
Email: aksinreck@rediffmail.com

[2]Department of Electronics and Communication Engineering, National Institute of Technology, Kurukshetra 136119 India,
Email: umesh_ghanekar@rediffmail.com

used for proper modeling and thorough analysis of properties of distributed systems. Attempting to overcome these problems, two main categories of formalisms may be identified: the static analysis and the dynamic one. The dynamic analysis relies on the limited number of test runs to make observations about the behavior of the system. A couple of tools are available to perform this task. However, since the dynamic techniques do not take in to consideration all possible executions, therefore, they are not sound [15]. Unlike dynamic analysis – where "confidence" comes from running an arbitrary number of test cases through a design – static analysis uses mathematical techniques to examine a specified design property. The verification of a property, using static analysis, is valid under all conditions and there is no element of uncertainty whereas dynamic analysis can guarantee a property only under those test cases for which it has been verified. Thus, while dynamic analysis is open-ended and uncertain, static analysis removes uncertainty, increasing designer confidence and reducing verification time [12]. Moreover, static analysis attempts to establish universal properties independent of any particular set of inputs. It uses rigorous formalized reasoning to prove statements [10]. Therefore, we resort to static analysis, though it involves considerable amount of human effort.

1.2 Motivation

Although, a large number of static analysis techniques are available in the literature for specification and verification of concurrent/distributed systems. The static analysis is difficult to carry out even for a simple distributed system. If we consider a complex system, the complicacy of the task is enhanced further, in absence of some formal and precise method for reasoning [5]. Hesselink [16] regards Hoare triples as the most adequate way to specify the systems. He adds further, one can use Hoare logic to define derivability of Hoare triples, but weakest preconditions form a more convenient semantic formalism that is sufficiently close to Hoare triples. With this in mind we feel that weakest precondition logic with suitably designed predicates would be a suitable scheme for system specifications.

1.3 The objective

Dijkstra's weakest precondition logic [4] uses first order predicate logic that is very easy to use and apply. It is elegant because its syntax and semantics are well defined. Therefore, the objective of the present work is to extend the Dijkstra's weakest precondition logic for assertional reasoning of distributed systems. As, the weakest precondition logic was developed by Dijkstra, originally, for reasoning about the correctness of sequential systems.

2. THE MATHEMATICAL MODEL

A system S is defined by a set $S.P$ of processes, which can interact among themselves only through message transactions. A process P is defined by a set $P.X$ of states and a set $P.R$ of state transition rules. Assertion that a proces P is in state $P.x$ is made by the predicate expression $in(P.x)$. Each process has a predefined initial state denoted by $initial(P.x_0)$. The state set SX of the system S is the collection of states of all the processes belonging to $S.P$.

A process can move from one state to the other by the action of a state transition rule. For a transition rule $P.r$ and a post condition Q there exist a weakest precondition $wp(P.r, Q)$ such that if the system state satisfies this condition then the execution of $P.r$ will eventually establish the truth of Q. This guarantee however cannot be given unless $wp(P.r, Q)$ is true before the execution of $P.r$. We divide this condition into two parts. The first part is related to the process P itself, where the second part includes the co-operation requirements from other

processes. With this idea we define two parts, viz., (i) weakest self pre-condition denoted by $wsp(P.r, Q)$ and (ii) weakest co-operation requirements, denoted by $wcr(P.r, Q)$. The total weakest pre-condition is then given by $wp(P.r, Q) = wsp(P.r, Q) \wedge wcr(P.r, Q)$

The post-condition Q and the weakest pre-condition $wp(P.r, Q)$ is used to describe a transition rule $P.r$.

A non-deterministic state transition rule $P.r$ may include number of different sub-rules each of which requires a definite pre-condition to be satisfied for its execution. These preconditions will be called guards. Execution of a sub-rule will change the state of P as well as the state of one of the co-operating processes whose active co-operation is necessary for this execution. State transition in the co-operating process will be achieved by simultaneous execution of a state transition rule. If the pre-condition for more than one sub-rule are satisfied then one of them is selected for execution. Selection procedure is non-deterministic and therefore, it is necessary to pass this information to the relevant co-operating process to produce the required state transition. The weakest pre-condition for a non-deterministic transition rule $P.r$ is obtained as follows.

Let there be n number of sub-rules denoted by $P.r^i : i = 1,\ldots,n$. On top of these sub-rules we assume a selector procedure, denoted by *select*, which makes the required non-deterministic selection .The post condition space for this procedure should therefore include a number of boolean variables denoted by

$s_i : i = 1,\ldots,m$. At each invocation the selector makes one such variable true. If a sub-rule $P.ri$ has a post condition Qi then

$s_i \Rightarrow wp(P.r^i, Q_i)$

Let Bi denotes the required guard for $P.ri$, and then the truth of this condition should ensure the selectability of s_i, i.e.,

$B_i \Rightarrow wr(select, s_i)$

Where, $wr(select, si)$ is the weakest requirement that the procedure *select* may produce s_i. Using equations for s_i and B_i the rule $P.r$ is described as follows:

$P.r$ ::

$Q \equiv \exists\, i : Q_i$;

$wp(P.r, Q) = (\exists\, k : B_k) \wedge$

$(\forall\, i \bullet B_i \Rightarrow wr(select, s_i)) \wedge$

$(\forall\, j \bullet s_j \Rightarrow wp(P.r^j, Q_j))$;

end of $P.r$;

3. THE PROPERTY OF A SYSTEM

The operational model of a system can be described by state transition rules. These rules can be described completely by their weakest pre-condition, post condition pairs. However, only operational specification may not be sufficient to describe the system requirements. In order to specify a system completely, along with the state transition rules the system properties must also be explicitly described. Best way to do this is to define a system invariant, which must remain true before and after the execution of each state transition rule. That is, there must exist a condition Q such that

$\forall\, i, \forall\, m \bullet \{wp(P_i \cdot r_m, Q_{i,m}) \Rightarrow Q\} \wedge (Q_{i,m} \Rightarrow Q)$

Similarly for a guarded command we have

$\forall\, i \bullet (B_i \Rightarrow Q) \wedge (B_i \Rightarrow wp(P \cdot r^i, Q_i)) \wedge (Q_i \Rightarrow Q)$

4. ILLUSTRATION

The complete scheme is being illustrated with the help of a well-known prototypical resource allocation problem involving allocation of pair wise shared resources in a ring of processors, that is – dining philosophers problem.

The reason for selecting the dining philosophers problem for illustration is many folds. The dining philosophers problem has become a paradigm for large class of concurrency control problems. It has achieved the status of a legend, since it captures the essence of many synchronization and resource allocation problems [8]. In fact, it provides a benchmark of the expressive power of new primitives of concurrent programming [11]. Also, the dining philosophers solutions are a basic building block for higher order synchronization problems [14] such as the drinking philosophers [8], job scheduling [1], and committee coordination [7]. Following is the problem statement for the dining philosophers problem.

1. Consider a system consisting of n number of philosopher processes and same n number of fork processes.
2. A philosopher can pick up a fork if it is already lying on the table.
3. Any philosopher will pick up first the fork lying on the table towards his left and then the right counterpart.
4. A philosopher will put down the forks in the same above sequence after finishing the eating.
5. The system should ensure the deadlock-freedom.

4.1 Formal System Specification

We first specify the system by considering only the first four conditions.

$\forall i \bullet i = 1..n$	$PHIL_i$	philosopher processes
$\forall i \bullet i = 1..n$	$FORK_i$	fork processes

(a) States for $PHIL_i$:

$PHIL_i$. *idle*	philosopher is sitting idle in the chair in the dining room
$PHIL_i$. *rtplf*	philosopher is ready to pick left fork
$PHIL_i$. *picked-ulf*	philosopher has picked up left fork
$PHIL_i$. *rtprf*	philosopher is ready to pick right fork
$PHIL_i$. *picked-urf*	philosopher has picked up right fork
$PHIL_i$. *eating*	philosopher is eating
$PHIL_i$. *rtpdlf*	philosopher is ready to put down left fork
$PHIL_i$. *pdlf*	philosopher has put down left fork
$PHIL_i$. *rtpdrf*	philosopher is ready to put down right fork
$PHIL_i$. *pdrf*	philosopher has put down right fork

(*b*) States for $FORK_i$:

$FORK_i$. *lying*	fork is lying on the dining table
$FORK_i$. *picked-up*	fork has been picked up by any one of the philosophers sitting on either side of fork

With reference to above states we can describe the processes as follows :

Process $PHIL_i$; identified by $PHIL_i$;

States : $PHIL_i$. *idle*, $PHIL_i$. *rtplf*, $PHIL_i$. *picked-ulf*, $PHIL_i$. *rtprf*, $PHIL_i$. *picked-urf*, $PHIL_i$. *eating*, $PHIL_i$. *rtpdlf*, $PHIL_i$. *pdlf*, $PHIL_i$. *rtpdrf*, $PHIL_i$. *pdrf*

Transition Rules for $PHIL_i$: $\forall i \bullet i = 1..n$

($PHIL_{i-1}$ is the philosopher sitting on the left (clockwise) of $PHIL_i$ and $PHIL_{i+1}$ is the philosopher sitting on the right (anticlockwise) of $PHIL_i$. Similarly for the $FORK_i$ also)

$phil_i$. *r*1 : :

$wsp(phil_i . r1 , in(PHIL_i . rtplf)) = in(PHIL_i . idle)$

$wcr(phil_i . r1 , in(PHIL_i . rtplf)) = true$

$phil_i$. *r*2 : :

def $Q1 = in(PHIL_i . picked\text{-}ulf) \wedge$

$in(FORK_i . picked\ up)$

$wsp(phil_i . r2 , Q1) = in(PHIL_i . rtplf)$

$wcr(phil_i . r2 , Q1) = in(FORK_i . lying) \wedge$

$in(FORK_i . r1 . s1)$

$phil_i$. *r*3 : :

$wsp(phil_i . r3, in(PHIL_i . rtprf)) = in(PHIL_i . picked\text{-}ulf)$

$wcr(phil_i . r3, in(PHIL_i . rtprf)) = true$

$phil_i . r4 : :$

def $Q2 = in(PHIL_i . picked\text{-}urf) \wedge$
$in(FORK_{i+1} . picked\ up)$

$wsp(phil_i . r4 , Q2) = in(PHIL_i . rtprf)$

$wcr(phil_i . r4 , Q2) = in(FORK_{i+1} . lying) \wedge$
$in(FORK_{i+1} . r1 . s2)$

$phil_i . r5 : :$

$wsp(phil_i . r5 , in(PHIL_i . eating)) = in(PHIL_i . picked\text{-}urf)$

$wcr(phil_i . r5 , in(PHIL_i . eating)) = true$

$phil_i . r6 : :$

$wsp(phil_i . r6 , in(PHIL_i . rtpdlf)) = in(PHIL_i . eating)$

$wcr(phil_i . r6 , in(PHIL_i . rtpdlf)) = true$

$phil_i . r7 : :$

def $Q3 = in(PHIL_i . pdlf) \wedge in(FORK_i . lying)$

$wsp(phil_i . r7 , Q3) = in(PHIL_i . rtpdlf)$

$wcr(phil_i . r7 , Q3) = true$

$phil_i . r8 : :$

$wsp(phil_i . r8 , in(PHIL_i . rtpdrf)) = in(PHIL_i . pdlf)$

$wcr(phil_i . r8 , in(PHIL_i . rtpdrf)) = true$

$phil_i . r9 : :$

def $Q4 = in(PHIL_i . pdrf) \wedge in(FORK_{i+1} . lying)$

$wsp(phil_i . r9 , Q4) = in(PHIL_i . rtpdrf)$

$wcr(phil_i . r9 , Q4) = true$

$phil_i . r10 : :$

$wsp(phil_i . r10 , in(PHIL_i . idle)) = in(PHIL_i . pdrf)$

$wcr(phil_i . r10 , in(PHIL_i . idle)) = true$

Process $FORK_i$; identified by $FORK_i$;

States : $FORK_i . lying, FORK_i . picked\text{-}up$

Transition Rules for $FORK_i$: $\forall i \bullet i = 1..n$

$fork_i . r1 : :$

def $Q5 = in(FORK_i . picked\text{-}up) \wedge$
$in(PHIL_i . picked\text{-}ulf)$

$Q6 = in(FORK_i . picked\text{-}up) \wedge$
$in(PHIL_{i-1} . picked\text{-}urf)$

$R1 = Q5 \vee Q6$

$B1 = in(FORK_i . lying) \wedge in(PHIL_i . rtplf$

$B2 = in(FORK_i . lying) \wedge in(PHIL_{i-1} . rtprf)$

$wp(fork_i . r1 , R1) = (B1 \vee B2) \wedge$
$(B1 \Rightarrow wr(select , in(FORK_i . r1 . s1))) \wedge$
$(B2 \Rightarrow wr(select , in(FORK_i . r1 . s2))) \wedge$
$(in(FORK_i . r1 . s1) \Rightarrow wp(FORK_i . r1^1, Q5)) \wedge$
$(in(FORK_i . r1 . s2) \Rightarrow wp(FORK_i . r1^2, Q6))$

$fork_i . r2 : :$

def $Q7 = in(FORK_i . lying) \wedge in(PHIL_i . pdlf)$

$Q8 = in(FORK_i . lying) \wedge in(PHIL_{i-1} . pdrf)$

$R2 = Q7 \vee Q8$

$B3 = in(FORK_i . picked\text{-}up) \wedge in(PHIL_i . rtpdlf)$

$B4 = in(FORK_i . picked\text{-}up) \wedge in(PHIL_{i-1} . rtpdrf)$

$wp(fork_i . r2 , R2) = (B3 \vee B4) \wedge$
$(B3 \Rightarrow wr(select , in(FORK_i . r2 . s1))) \wedge$
$(B4 \Rightarrow wr(select , in(FORK_i . r2 . s2))) \wedge$
$(in(FORK_i . r2 . s1) \Rightarrow wp(FORK_i . r2^1 , Q7)) \wedge$
$(in(FORK_i . r2 . s2) \Rightarrow wp(FORK_i . r2^2 , Q8))$

4.2 The Correct System Construction

If the system state satisfies pre-condition of at least one of the rules then system under test is said to be running. If system is in a state where none of the rules can be applied, no process will be able to proceed or change state. In other words, the system is said to be under deadlock. If pre-conditions of all the rules are negated and conjuncted, it will generate the combination of states of processes, which will lead to deadlock. The Occurrence of this condition shows deadlock in the system. Following is the derived deadlock condition. (In order to restrict the length of presentation, the detailed logical manipulation steps are not shown.)

$\forall i \bullet \{in(PHIL_i . rtplf) \wedge \neg in(FORK_i . lying)\} \vee$
$\{in(PHIL_i . rtprf) \wedge \neg in(FORK_{i+1} . lying)\}$

First disjunct in the result represents a not achievable system state. Since we have defined that each philosopher will try to pick up left fork first. So it is not at all feasible that all philosophers are ready to pick up left fork and no left fork is lying on the dining table. Thus system will

never be in that state. Hence we conclude at the second disjunct.

$$\forall\, i \bullet \{in(PHIL_i\,.\,rtprf) \wedge \neg\, in(FORK_{i+1}\,.\,lying)\}$$

It represents the system state, which leads to deadlock. In other words, it represents "all the philosophers are ready to pick up right fork and no right fork is lying on the dining table" which is a deadlock condition. Thus, if the specification of a system is not correct, our approach, using the weakest precondition logic, can indicate that.

On negating the above derived deadlock condition and manipulating further, using the first order predicate logic rules (details omitted), we get the following condition for deadlock freedom.

$$\exists\, i \bullet in(PHIL_i\,.\,idle) \vee \exists\, i \bullet \{in(PHIL_i\,.\,picked\text{-}urf) \wedge in(FORK_i\,.\,lying)\}$$

We conclude that our approach is not only capable of indicating faults, if any, in the initial system specifications; but it is powerful enough to derive the correct system specifications form the existing system specifications. Therefore, it can be effectively used for construction of correct distributed systems. The condition for deadlock freedom, derived above, must be incorporated in the system specifications as the system invariant. This approach is similar to the constraint satisfaction [3] where the system specifications can easily be updated when the verified system is further modified depending on needs.

5. SCOPE FOR THE MODEL VALIDATION

The approach, presented in this paper, suggests how one can be used it to handle a class of distributed algorithms in which process communication follows message passing paradigm. Though the strength of our modeling technique is simplicity, accuracy has not been compromised for the sake of simplicity. Nevertheless, our approach needs careful human effort. However, in our logic, the correctness is ensured by proving assertions that are formulas in predicate logic. These formulas must be embedded into the system during its design phase. Dijkstra [4] also mentioned this in connection with the loop invariants. The proof of these formulas requires standard predicate logic rules and also the transition rules of the system in question. Since we propose to specify a system by its transition rules, this formalization is available to us. It should therefore be possible to develop a rule-based system to evaluate the correctness of the assertions. One can also think of a proof system that may be intelligent enough to consult with the user and update its rule base.

6. WHERE THE MODEL CAN FIT ?

The proposed methodology can successfully be used to model either a component of a distributed system or an entire distributed system, using weakest preconditions. This decision is also favored by the following facts about the proposed approach.

1. The weakest precondition approach uses predicate transformer *wp* which transforms a post condition predicate characterizing the set of final states to a precondition predicate characterizing the set of initial states, this backward formulation in terms of preconditions from post conditions is preferable being goal oriented [9].
2. In our approach, we can start from the specification of the system using weakest precondition logic and work backwards to verify the existing system specifications or synthesize the correct system specifications. Therefore, it is possible to augment the specifications considerably. Using this augmentation scheme recursively, final system design, suiting to needs, can be found.

3. The weakest precondition have been divided in to two parts namely, weakest self precondition (*wsp*) and weakest co-operation requirement (*wcr*). In this approach, the co-operation requirements have been directly included in the weakest precondition. Therefore, separate proof of co-operation is not necessary, which was required in [6].

7. CONCLUSION

We have shown how our notation, which is based on weakest pre-condition logic, may represent a resource allocation problem involving communicating processes. Any distributed system consists of a collection of communicating processes, and these can be modeled in this notation. Thus any distributed system may be modeled in the notation. Finally, the approach, presented in this article, suggests how our logic can be used to handle a class of distributed systems in which processes communicate through message passing. On the similar lines one can model the other type of distributed systems in which processes communicate through shared variables. We believe that it is a rigorous and elegant means of modeling the distributed systems. The strength of our model is its simplicity. Although, the complete scheme has been illustrated by considering a well-known problem, the major contribution of this work is the methodological issue it raised and not the individual result.

References

[1] B. Awerbuch and M. Saks, A dining philosophers algorithm with polynomial response time, 31st IEEE Annual Symposium on Foundations of Computer Science, 22–24 October, vol. 1, 1990, pp. 65–74.

[2] D. Rosenblum, Specifying concurrent systems with TSL, IEEE Software, 8(3), May 1991, pp. 52-61.

[3] E. Bin, et al. Using a constraint satisfaction formulation and solution technique for random test program generation, IBM Systems Journal, 2002.

[4] E. W. Dijkstra, "*A Discipline of Programming*", Prentice Hall, 1976.

[5] I. Pau, I. Rents, and W. Schreiner, Verifying mutual exclusion and liveness properties with TLA, Technical Report 00-06, RISC-Linz, Johannes Kepler University, Austria, January 2000.

[6] K. M. Chandy and B. A. Sanders, Predicate transformers for reasoning about concurrent computation, Science of Computer Programming, vol. 24, 1995, pp 129-148.

[7] K. M. Chandy and Jayadev Misra, Parallel Program Design : A Foundation. Addison-Wesley, Reading, MA, USA, 1988.

[8] K. M. Chandy and Jayadev Misra, The drinking philosophers problem, ACM Transactions on Programming Languages and Systems, vol. 6, no. 4, October 1984, pp. 632–646.

[9] M. Ben-Ari, "*Mathematical Logic for Computer Science*", Prentice Hall, 1993.

[10] M. McFarland, Formal verification of sequential hardware: a tutorial, IEEE Tr. Computer Aided–Design of Integrated Circuits and Systems, 12(5), May 1993, pp. 633-654.

[11] O. M. Harescu and C. Palamidessi, On the generalized dining philosophers problem, 20th Annual ACM Symposium on Principles of Distributed Computing, Newport, Rhode Island, USA, August 2001, pp. 81–89.

[12]R. Kurshan, Formal verification in a commercial setting, Proc. Design Automation Conference, California, 9–13 June, 1997, pp. 258-262.

[13]S. Fialho, et al. A formal technique for the specification and verification of distributed systems and its applications in manufacturing automation, Proc. 38th IEEE Mid west Symposium on Circuits and Systems, 13–16 Aug 1995, vol. 1, pp. 27-30.

[14]S. M. Pike and P. A. G. Sivilotti, Dining philosophers with crash locality 1, 24th IEEE International Conference on Distributed Computing Systems, 23–24 March, 2004, pp. 22–29.

[15]T. Win and M. Ernst, Verifying distributed algorithms via dynamic analysis and theorem proving, Technical Report, MIT-LCS-TR-841, MIT Lab for Computer Science, MA, USA, May 2002.

[16]W. Hesselink, Predicate transformers for recursive procedures with local variables, Formal Aspects of Computing, 11(6), 1999, pp. 616-636.

[17]Y. Isobe and K. Ohmaki, A process logic for distributed system synthesis, Proc. APSEC 2000, IEEE, 2000, pp. 62-69.

Technical Biographies

Awadhesh Kumar Singh received B. E. Computer Science & Engineering degree from Gorakhpur University, Gorakhpur, India in 1988. He received M.E. and PhD(Engg) in the same area from Jadavpur University, Kolkata, India. Currently he is assistant professor in Computer Engineering Department, National Institute of Technology, Kurukshetra, India. His present research interest is distributed systems.

Umesh Ghanekar graduated from Rohilkhand University, India in 1985. He received M. Tech. degree from Indian Institute of Technology, Roorkee, India. Presently he is assistant professor in Electronics and Communication Engineering Department, National Institute of Technology, Kurukshetra, India. His research interests include computer communication networks and distributed systems.