IOWA STATE UNIVERSITY
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2014

Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers

Sneha Bankar
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Bankar, Sneha, "Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers" (2014). *Graduate Theses and Dissertations*. 14052.
https://lib.dr.iastate.edu/etd/14052

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Automatic verification of interactions in asynchronous systems with unbounded buffers

by

Sneha Bankar

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Samik Basu, Major Professor

Andrew S. Miner

Hridesh Rajan

Iowa State University

Ames, Iowa

2014

Copyright © Sneha Bankar, 2014. All rights reserved.

DEDICATION

I would like to dedicate this work to Dr. Samik Basu and to my parents: Mr Anil Bankar and Mrs. Ujwala Bankar for their constant moral support throughout my Master's education and while writing this Thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank firstly to Dr. Samik Basu for his constant support, guidance and most importantly being patient while going through different phases of this research and thesis. Also special thanks to members in my POS committee, Dr. Andrew Miner and Dr. Hridesh Rajan for giving their valuable inputs and giving insights on some future work of this thesis. This work is supported by US National Science Foundation grant CCF1116836.

ABSTRACT

Model Checking of distributed systems which communicate via message exchanges is an open research problem. Such communication results in asynchronous interaction between senders and receivers, one where the communicating entities, do not move in lock-step. Model Checking is only possible for systems which can be represented as finite state systems. Distributed Systems which communicate asynchronously cannot be represented as finite state systems due to undefined bound on the message buffers meant for message exchanges. Thus in general model checking such systems is undecidable. In this thesis, we present a technique to automatically identify asynchronous systems whose interactions can be represented by some finite state systems. This will allow us to automatically model check the asynchronous systems. We also present a prototype implementation and discuss the application of our technique on several case studies from existing literature.

CHAPTER 1. INTRODUCTION

The paradigm of distributed computing is becoming extremely popular with the growth of high performance computing, distributed service infrastructure and distributed consumer's scenario. This paradigm has become a basis for a wide variety of applications from data-centric services to processor intensive tasks, from systems built from very small and relatively primitive sensors to those incorporating powerful computational elements, from embedded systems to ones that support a sophisticated interactive user experience, and so on. According to G. Couloris, a distributed system is "a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing" Coulouris and Dollimore (1988). As can be seen, distributed systems encompass many of the most significant technological developments of recent years and have a widespread use. Thus, it becomes imperative to have a deep understanding of distributed systems and further analyze such systems to avoid system failures.

In order to enable interaction between different components of a distributed system, message based communication is widely used. Message based communication allows for concurrency and distributed computing. Important applications of message based distributed systems are process isolation in Singularity OS Fähndrich et al. (2006) and distributed message based services Armstrong (2002); Banavar et al. (1999). Singularity is an experimental operating system built by Microsoft Research. It is designed as a highly-dependable OS in which the kernel, device drivers, and applications were all written in managed code. Process isolation is an important feature for Singularity OS

and to accomplish this, processes are not allowed to share memory. The inter-process communication, therefore, occurs over channels via message passing. Web Services are a good example of distributed message based services. Web Services basically are formed of web-accessible software applications that communicate over the Web. The communication between services takes place through XML based messages. It is necessary that services should be able to tolerate pauses in availability of other services and slow data transmission. This is achieved through asynchronous communication.

We consider distributed systems communicating via message exchanges. These message exchanges are performed through communication channels. The different components of the distributed system are often referred to as peers. Communication over channels can incur delays (or even loss) in the delivery of messages to the receiving peer. A simple view of such a channel is that of a first in first out (FIFO) queue where the maximum size of the queue corresponds to maximum delay a channel can incur. This is referred to as the size of the channel. Other representations can include random-access queue, where the ordering of the messages being exchanged via the channels is not maintained. In either case, the communication results in asynchronous interaction between senders and receivers, one where the communicating entities, do not move in lock-step. We consider FIFO ordering of messages. We refer to message buffers present at any peer's end as receive queues. In this communication model, same peer can act as a sender as well as a receiver. The systems which follow the interaction behavior described above are called as asynchronous systems. In asynchronous systems, communicating peers can continue their operation without getting blocked as they do not have to wait for other peers to complete their actions. For instance, after a sender sends a message to the receiver, it does not have to wait for the receiver to consume the message. Once the message it sent, it can safely resume its actions.

1.1 Automatic Verification of Asynchronous Systems

Given a representation of dynamic behavior of a system, model checking enables to exhaustively and automatically verify whether the behavior is a *model* of the desired temporal specification; in short, whether the behavior satisfies the specification. The primary constraint in applying model checking is that the behavior of the system should have finite state-space; otherwise, the problem of model checking becomes undecidable. This makes model checking asynchronous systems challenging, if not impossible. The reason stems from the fact that the size of the communication channels, i.e. the size of the queues, is not known a priori. As a result, model checking an asynchronous system involves model checking the system with any finite size channel. In fact, Brand and Zafiropulo (1983) states that asynchronous systems can be viewed as communicating finite state machines, for which automatic reachability analysis (a sub-goal of model checking) is undecidable in general.

1.1.1 Existing Approach

Many different approaches have been undertaken to address the problem of verification of asynchronous systems where peers communicate via channels whose size is not specified. Since automatic verification for such systems is undecidable, researchers have focused on sub-classes of these systems for which verification can be decidable.An example of an asynchronous system communicating via message buffers is shown in figure 1.1. In this example, the system is made up of three peers. The communication channel is modeled as an unbounded receive queue for every single peer. When a peer wants to communicate with any other peer, it can perform a write to the receive queue of the other peer.Also, a peer can read from its own receive queue whenever it is ready to do so. Thus communication between the peers is asynchronous.

One line of research considers verification of asynchronous systems by imposing re-

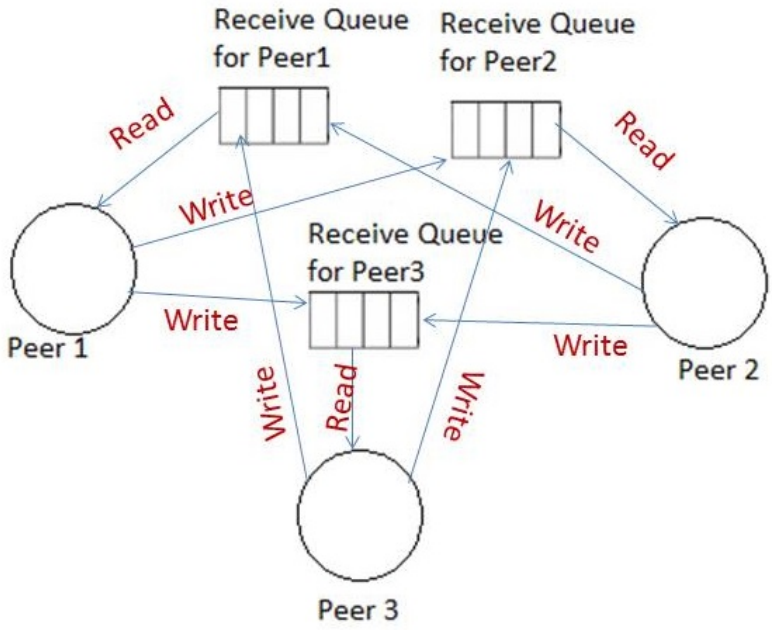

Figure 1.1 Peers with unbounded buffers

strictions on them. For example, Cecea and Finkel (2005) considers verification of half-duplex systems. This work proves that reachability set for half duplex systems can be calculated in polynomial time and determining if a system is half duplex is decidable. The half-duplex property for two machines and two channels (one in each direction) states that each reachable configuration has at most one non-empty channel. This is a clear restriction on the communication channel of peers in an asynchronous system.

A more general class of asynchronous systems is handled in Fu et al. (2004) and Fu et al. (2005), where there could be more than two peers in the system and communication channel need not be restricted. In this line of work, verification of such systems is decidable if interaction between the peers is synchronizable i.e. the behavior of asynchronously interacting peers can be mimicked by the behavior of the same peers interacting synchronously (i.e., with receive queue size 0).Yoshida and Vasconcelos (2007) and Honda et al. (2008) focus on session types, used in the context of structured

communication-centered programming. However, the drawback is that the systems have to conform to the behavior of session types, where sends and receives from the same state is not possible.Thus, there is a restriction on the behavior of communicating peers.

Siegel (2005) and Vakkalanka et al. (2010) target the domain of parallel programs that employ message passing interface. In this line of research, the send actions and receive actions are restricted such that it leads to a deadlock free interaction.Thus here the focus is on the local behavior of the system. In contrast we focus on global interaction behavior.

We do not put any limitation on the behavior of message channels or do not make any assumptions with respect to system specifications. Our work is built upon the results in Fu et al. (2005) but does not require for asynchronous systems to be synchronizable. Thus our solution is more generic. In conclusion, our approach targets a broader class of asynchronous systems. We represent unbounded buffered asynchronous systems in the form of finite state systems and check if any bounded interaction behavior mimics the given unbounded behavior. After such confirmation, traditional model checking algorithms can be used to verify different properties for the given unbounded system.

1.1.2 Proposed Solution

We give a solution which can be applied to a more general class of asynchronous systems and does not have any limitation on the behavior of the communication channel. We present necessary and sufficient conditions under which interactions between peers with unbounded receive queues in an asynchronous system are identical to interactions between the same peers with some bounded receive queues. Bounded buffered systems exhibit finite state space and thus verification is decidable for them. The intuition behind our approach is that if interaction between a particular set of peers with unbounded buffer system can be mimicked by a buffered system, then the buffered system can be used as a representation of the unbounded counterpart.

The root cause of infinite state behavior of asynchronous systems is that, the sender can send messages without any blockage, however the receiver cannot consume the messages at the rate they are being sent. The FIFO queue for the receiver grows in an unbounded fashion. However, if the receiver's behavior is such that it can consume the messages in its buffer infinitely often, then a bounded buffer is sufficient to serve the purpose of the unbounded buffer. As a result, the bounded buffer implies finite state-space representation of the asynchronous unbounded buffered system. We identify the condition that exactly captures this behavior of the receiver. We also show that the condition can be automatically verified by exploring and analyzing finite number of states in the system. Once the condition is successfully verified, which ensures the existence of the finite bound (say k) on the receive queue size, one can automatically compute such a bound. The process is based on iteratively checking whether the peer interactions with i size receive queues are identical to peer interactions with i + 1 size receive queues, starting from i = 1. This iteration is guaranteed to terminate when i = k. This process is based on the fact that the interactions between peers using i size receive queues include the same peers interactions using $\leq$ i size receive queues Basu and Bultan (2011).

1.2 Contributions

1. ***Theoretical Results*** As discussed earlier, all the existing work targets verification of specific classes of asynchronous systems. Our approach identifies a super-set of these classes and makes verification decidable for them. We prove that our algorithm always terminates whether or not any bounded behavior is detected in the given asynchronous unbounded system.

2. ***Prototype Tool*** We have built a tool which is based on our proposed approach, thus proving that our approach is realizable in practice. Different modules of the tool perform separate functions and communicate with each other via static

objects. This modularization makes future extensions to our approach easy. For example, the tool can be further extended to include additional parsers to accept additional specification languages other than XML. It can also be coupled with standard Model Checkers like SPIN to enable Model Checking of asynchronous systems for which verification is decidable.

3. ***Case Studies*** We evaluate the tool against standard communication protocols like Alternating Bit Protocol, Sliding Window Protocol and Snoopy Cache Protocol. We also perform a case study on protocols like TCPContract Protocol, TPMContract Protocol, KeyBoardContract Protocol from Singularity OS and Stock Broker Protocol, MetaConversation Protocol, Reservation Session Protocol from Web Services. We evaluate the tool on randomly generated unbounded buffer systems and special cases of sliding window protocol with increasing window size to check for scalability. Preliminary case studies prove that our approach is feasible.

1.3 Outline

The rest of the thesis is organized as follows. In Chapter 2, we discuss previous work related to solving the problem similar to ours. Chapter 3 gives detailed explanation of two conditions to be checked to detect bounded behavior in unbounded buffer systems. It also gives an overview of Algorithms to be used in order to check the two conditions. Chapter 4 gives an overview of the Tool Architecture and implementation of tool components. Chapter 5 presents case studies used for evaluation and experimental results. In Chapter 6 we summarize our approach and contributions.

CHAPTER 2. RELATED WORK

Many different approaches have been undertaken to address the problem of verification of asynchronous systems where peers communicate with unbounded message buffers. Since automatic verification for such systems is undecidable, researchers have focused on sub-classes of these systems for which verification can be decidable.

Cecea and Finkel (2005) considers verification of Communicating Finite State Machines with Unbounded Channels. More specifically it considers analysis of Half-Duplex Systems made of finite state machines that communicate over unbounded channels. Half Duplex Property says that two machines and two channels, the composition of this system each configuration should have at most one non-empty channel. The authors provide that is it possible to decide in polynomial time whether a system is half duplex. Further exact Representation of reachability set of half duplex communicating systems is possible to calculate in polynomial time. This paper discusses about properties to be checked on model of communicating finite state machines. How to check them with recognizable representation of their reachability set. Then it talks about a recognizable representation for half duplex systems. This representation helps in finding symbolic reachability graph of half duplex system. It gives a theorem to show to decide whether a system of two machines is half duplex in polynomial time. Also it talks about extensions of this half duplex communication and how it becomes Turing Powerful. Another finding here is that Propositional linear temporal logic and Computational Tree Logic analysis is undecidable for half duplex systems.

First the authors identify the possible verification properties for communicating finite

state machines (CFSM). For example: reachability problem is to determine if a particular configuration is reachable in the CFSM; deadlock problem is to check if a deadlock configuration exists in the CFSM. They talk about seven such problems, refer to Cecea and Finkel (2005) for more details. The authors say that Systems of CFSMs have the power of Turing machines. Thus verification is undecidable for them. However for a system S whose reachability set is channel recognizable and for which an effective description is possible, the first 6 problems are decidable. Channel recognizable means while states of a communicating system are considered while calculating the configurations we also consider the possible states of the queue. Next the authors discuss about half duplex systems which contains only two machines communicating with each other. They show that each reachable configuration of this system is possible to calculate using a method used to calculate channel recognizable reachability set. The basic idea is all the transitions which are possible to reach different configurations can be split into two sets of transitions. The first set is a 1-Bounded exectution i.e it takes care that the number of messages possible in both the channels is not more than one. The second set consists of only send actions which only use one machine and one channel. Thus both these sets are realizable in a half duplex system with two machines. Thus it is channel recognizable and reachability set can be calculated for this half duplex system. The authors prove that this can be done in polynomial time and that the seven verification problems for half duplex systems are decidable.

Next, it is important to decide if a given CFSM is actually a half duplex system in order to apply the above results. The authors prove that it is decidable in polynomial time. After talking about basic verification problems, the paper focuses on model checking of half duplex machines against temporal logics like *Propositional Temporal Logic* and *Computational Tree Logic*. Also another result of this paper is that for half duplex systems with more than two machines, verification is undecidable as such systems can simulate Turing Machines. Thus reachability analysis is not possible for them.

Next we consider work which is related to systems which satisfy the 'Synchronizability property'. If the interaction behavior for a set of communicating peers remains the same when asynchronous communication is replaced with synchronous communication. This is called the synchronizability problem. Basu and Bultan (2011), Fu et al. (2004) and Fu et al. (2005) focus on such system and below we discuss these works.

Basu and Bultan (2011) focuses on Choreography conformance. Specification and analysis of message-based interactions has been an important research area in service oriented computing in the last several years. Choreography languages enable specification of such interactions. A choreography specification corresponds to a global ordering of the message exchange events among the peers participating to a composite service, i.e., a choreography specification identifies the set of allowable message sequences for a composite web service. Choreography conformance problem is identifying if a set of given services adhere to a given choreography specification. In general this problem is undecidable when asynchronous communication is used. This work identifies a subclass of asynchronous systems for which choreography conformance can be checked effectively. For any system to belong to this class, this system has to satisfy 'Synchronizability Property'. The authors prove that synchronizability can be determined by comparing the behavior of the peers with synchronous communication and with bounded asynchronous communication where each message queue is restricted to a queue of size 1. Once a set of peers are determined to be synchronizable, then choreography conformance can be easily checked using existing finite state verification tools.

Initially the authors discuss about Language Equivalence, Bisimulation Equivalence and Simulation Pre-order. Refer to Basu and Bultan (2011) for the definitions. Further these equivalence are talked with respect to synchronizability. In summary, bismulation equivalence demands that branching behavior with respect to send actions between two composite systems is same. For satisfaction of the 'Synchronizability Property', satisfaction of bisumlation equivalence is more than enough. Since our work is built upon this

research. The relevant proof for Syncronizability also called as language synchronizability is mentioned in 3.2.1.

Bisimulation synchronizability allows for verifying conformance of a composite system to choreography specification expressed in any temporal logic; Simulation synchronizability allows for verifying conformance of composite system to choreography specifications expressed in universal fragment of temporal logic; and finally, language synchronizability allows for verifying conformance of composite system to choreography specification expressed as FSA and in LTL temporal logic. Bisimulation synchronizability implies simulation synchronizability which, in turn, implies language synchronizability.

The main contribution of this work has been solving the decidability of synchronizability problem which was an open problem before.

In Fu et al. (2005) focus is on sub-class of asynchronous systems, called synchronizable systems, for which certain reachability properties (over send actions and over states with no pending receives) remain unchanged when asynchronous communication is replaced with synchronous communication. Hence, if a system is synchronizable, then the verification of these reachability properties can be done on the synchronous version of the system and the results hold for the asynchronous case. We present a technique for deciding if a given system is synchronizable. An asynchronously communicating system is synchronizable if executing that system with synchronous communication instead of asynchronous communication preserves its behaviors. Focus is on two types of behaviors: 1) the sequences of messages that are sent, and 2) the set of reachable configurations where message queues are empty, i.e., configurations with no pending receives. Before in Basu and Bultan (2011) the focus was on just the send actions of Peers. Here the authors also consider reachability of configurations with no pending receives. Such configurations are also called as Synchronized states. Thus a system is synchronizable if and only if the behaviors for the synchronous version of the system and the 1-bounded-asynchronous version of the system are equivalent with respect to sent messages and

reachable configurations with empty message queues. If a system is synchronizable, then we can check properties about its message sequences or about the reachability of its global configurations with empty message queues, using the synchronous version of the system.

The authors explain about Send traces defined as a sequence of send actions starting from the initial configuration and Synchronized trace which is a send trace and ends in a Synchronized state. In order for the systems to be synchronizable with respect to the behaviors given above, the synchronous version of the system and the 1-bounded-asynchronous version of the system are equivalent with respect to the Send traces and the Synchronized trace. Basu and Bultan (2011) already discusses about Send Traces. This work mainly proves that if synchronous version and 1-Bounded verion of a system are equivalent then there are no new Synchronizaed traces in any K-Bounded System $\forall k \geq 1$. Thus synchronizability can be decided by checking the equivalence between two finite-state systems, synchronous version and 1-Bounded version. This can be performed automatically. Then verification of the asynchronous system can be performed using the corresponding synchronous system.

In this work, the synchronizability results were applied on analysis of channel contracts in the Singularity OS. The experimental results show that almost all channel contracts in the Singularity OS are synchronizable, and, hence, their properties can be analyzed using synchronous communication semantics. These results are also applicable to domains including verification and analysis of interactions among processes at the OS level, coordination in service-oriented computing and interactions among distributed programs. Even though there could be bounds on the actual physical memory on the buffers of Singularity OS, these results are still useful as the message queues are completely removed and model checking on this system works well as the state explosion problem for systems with message queues is solved. Another important aspect of this work is that it implemented Synchronizability and applied it to Singularity OS Channel

Contracts.

Finite State Verification technique one of which is model checking suffers usually from the state explosion problem. This is even more a serious problem with concurrent systems. Approaches to solve this problem taken are partial order reduction (POR) methods, data abstraction, program slicing, and state compression techniques. This observation has led to interest in more domain-specic approaches. In Siegel (2005) authors focus on the domain of parallel programs which employ Message Passing Interface. The idea is to leverage knowledge of the restrictions imposed by a particular programming domain, or of common idioms used in the domain, in order to gain greater reductions than the generic algorithms allow. With respect to verification the focus is on halting properties which include freedom from deadlock and assertion of values of certain variables after program termination.

This paper explains the basic MPI semantics which is explained below. The basic MPI function for sending a message to another process is MPI_ SEND. To use it, one must specify the destination process and a message tag, in addition to other information. The corresponding function for receiving a message is MPI_ RECV. In contrast to MPI_ SEND, an MPI_ RECV statement may specify its source process, or it may use the wildcard value MPI_ ANY_ SOURCE, indicating that this statement will accept a message from any source. Similarly, it may specify the tag of the message it wishes to receive, or it may use the wildcard value MPI_ ANY_TAG. A receive operation that uses either or both wildcards is called a wildcard receive. The use of wildcards and tags allows for great exibility in how messages are selected for reception. The state explosion problem occurs when message channels come into picture. Before this work it was already proved that if wildcards receives are ignored than if a system does not deadlock on synchronous execution it can be model checked and state explosion problem can be avoided. Thus use of MPI_ ANY_TAG becomes difficult. In this work the MPI_ ANY_TAG can be used with some relaxation on the hypothesis of wildcard receives. Also, the range of properties

is expanded to include all halting properties. It provides model checking algorithm that deals with MPI_ ANY_ SOURCE by moving back and forth between a synchronous and a buering mode as the search of the state space progresses. An MPI program consists of a xed number of concurrent processes, each executing its own code, with no shared variables, that communicate only through the MPI functions.

The basic idea behind presented in this paper is that, in most of the model checking approaches for concurrent systems, the send and receive channels are supposed to be bounded and thus the sends possible from any sender are blocked. However the MPI standard does not have any such limitation on the channel size. The sends in this case are blocked unless receives are done synchronously by the receivers. This work mainly focuses on the halting properties of MPI programs. A state in the composed graph of the Concurrent System is said to be halted when there are no receives or any synchronized receive events to be performed.

Similar to previous works discussed earlier here, this work also tries to map a potentially infinite state graph(say G) to a finite state graph(say G') by removing the states which might lead to the infinite state behavior. In this paper, the authors discuss what such states could be with respect to MPI Programs and give the formal definitions and an algorithm to select such states. In conclusion, this work is concerned with the verication of certain properties, such as freedom from deadlock, for parallel programs that are written using the Message Passing Interface (MPI). It is known that for MPI programs containing no 'wildcard receives' (and restricted to a certain subset of MPI) freedom from deadlock can be established by considering only synchronous executions. This approach is generalizes by presenting a model checking algorithm that deals with wildcard receives by moving back and forth between a synchronous and a buering mode as the search of the state space progresses. This approach is similar to that taken by partial order reduction (POR) methods, but can dramatically reduce the number of states explored even when the standard POR techniques do not apply. On similar lines Manohar and Martin (1998)

present conditions under which slack of a channel in a distributed system can be modified whitout changing its behavior. This paper suggests some program transformations for concurrent systems which can be correct only if their interaction behavior satisfy certain conditions. Vakkalanka et al. (2010) takes the work in Siegel (2005) further. It takes into consideration slack elastic programs. Slack elastic programs consider send or receive actions in a particular order and due to this property, buffering can be added to them safely. The authors say that many MPI programs are slack elastic but there are few which are not. This paper suggests a formulation of a happens before relation for MPI programs. This will help in identifying local states which cause the system to follow non-deterministic behavior. The authors provide method to efficiently check for slack elasticity of MPI Programs. They also implement this approach in a framework which help in verifying larger set of MPI programs. Thus Siegel (2005), Manohar and Martin (1998) and Vakkalanka et al. (2010) consider a specific domain of potentially infinite state space concurrent systems called as MPI programs and also identify techniques which could help verify such programs by mapping them to a finite state space system.

Yoshida and Vasconcelos (2007) and Honda et al. (2008) focus on session types where a particular system interaction has to follow certain communication protocol. These interactions are formulated into a typing problem. Many of the language primitives had constructs which would define one time interaction between Peers and this work is concerned with a structure of a series of interactions between Peers. In this work the authors basically wanted to describe complex interaction behaviors with clarity and discipline at the high-level of abstraction, together with a formal basis for verification. They gave basic communication primitives (corresponding to assignment and arithmetic operations in the imperative setting), and the structuring constructs to combine them (corresponding to 'if-then-else' and 'while'). Verification methodologies on their basis were then developed. Communicating Peers can have different modules via which they communicate with each other. This work shows how these modules and the interaction

between them can be represented as a typed language. Whenever a Peer initiates an interactions and it is called as a Session. Each session has a corresponding channel via which the Peers communicate. The main idea here is the syntax of the type constructs is defined such that there is certain compatibility of communication patterns between two processes. This compatibility is such that each Peer cannot perform a a send or a receive action from the same state. Thus in terms of the typed language, this paper defines a global interaction protocol which the communicating peers must follow. The peers local behavior should be such that they follow this global interaction protocol. The typed conversations are used as a basis for validating programs. Yoshida and Vasconcelos (2007) dealt with asynchronous systems dealing with just two peers while Honda et al. (2008) took the work further to deal with more than two Peers.

Thus Yoshida and Vasconcelos (2007) and Honda et al. (2008) consider concurrent programs communicating with each other. This is not a certain domain of concurrent systems like MPI programs or web services described before. However, they define Type constructs for interaction between communicating peers. These constructs in turn impose communication restrictions on such interactions, due to which verification of asynchronous systems with unbounded buffers (under certain restrictions) becomes feasible.

2.1 Summary

Thus we see the approach taken to solve the problem of Verification of Asynchronous systems is to concentrate on certain sub-class of Asynchronous systems or impose certain behavioral restrictions on such systems. Thus the sub-class of Asynchronous Systems becomes verifiable or at least if they satisfy certain conditions they become verifiable. In our work, we identify a new and relaxed sub class for which interactions between peers with unbounded receive queues in an asynchronous system are identical to interactions between the same peers with some bounded receive queues. We provide certain con-

ditions to determine if a particular Asynchronous System belongs to such a sub-class and provide an approach to check for these conditions. In conclusion, one line of research considers verification of asynchronous systems by imposing restrictions on them. For example, Cecea and Finkel (2005) considers verification of half-duplex systems. This work proves that reachability set for half duplex systems can be calculated in polynomial time and determining if a system is half duplex is decidable. The half-duplex property for two machines and two channels (one in each direction) states that each reachable configuration has at most one non-empty channel. This is a clear restriction on the communication pattern of peers in an asynchronous system. A more general class of asynchronous systems is handled in Fu et al. (2004) and Fu et al. (2005), where there could be more than two peers in the system and communication channel need not be restricted. In this line of work, verification of such systems is decidable if interaction between the peers is synchronizable i.e. the behavior of asynchronously interacting peers can be mimicked by the behavior of the same peers interacting synchronously (i.e., with receive queue size 0).

Our approach over decidability of verification of asynchronous systems does not lay any restrictions on the type of asynchronous system and thus handles a wider subset of asynchronous systems.

Yoshida and Vasconcelos (2007) and Honda et al. (2008) focus on session types, used in the context of structured communication-centered programming. Honda et al. (2008) was an important step from binary asynchronous systems to multi-party systems. However, the drawback is that the systems have to conform to the behavior of session types, where sends and receives from the same state is not possible. Siegel (2005) and Vakkalanka et al. (2010) target the domain of parallel programs that employ message passing interface. This approach necessitates that the systems to be checked are deadlock free which is a local behavior. In contrast we focus on global interaction behavior.

We do not put any limitation on the behavior of message channels or do not make any

assumptions with respect to system specifications. Our work is built upon the results in Fu et al. (2005) but does not require for asynchronous systems to be synchronizable. Thus our solution is more generic.

CHAPTER 3. BACKGROUND

In this section, we present the conditions which when satisfied by an asynchronous system with unbounded message queues, it can be represented by an asynchronous system with bounded message queues. We give basic syntax and semantics to represent asynchronous systems by using an example. We will further use this example to explain the necessary conditions. Most of definitions given below have already been given in Fu et al. (2005).

3.1 Peers and Systems

Peer behavior is represented as communicating finite state machines. From each state there is a send (ouput) action or receive (input) action possible. A send action would result in addition of a message to the receiver peer's message buffer. A receive action would result in consumption of a message from the Peer buffer which holds the message. Formally we define Peers in Definition 1.

Definition 1. *A peer behavior or simply a peer, denoted by P, is a Finite State Machine* (M, T, s_0, δ) *where* M *is the union of input* (M^{in}) *and output* (M^{out}) *message sets,* T *is the finite set of states,* $s_0 \subseteq T$ *is the initial state, and* $\delta \in T \times (M \cup \{ \epsilon \}) \times T$ *is the transition relation. A transition* $\tau \in \delta$ *can be one of the following three types:*

1. *a send-transition of the form* $(t_1, !m_1, t_2)$ *which sends a message* $m_1 \in (M^{out})$.
2. *a receive-transition of the form* $(t_1, ?m_1, t_2)$ *which consumes a message* $m_1 \in (M^{in})$ *from its input queue.*

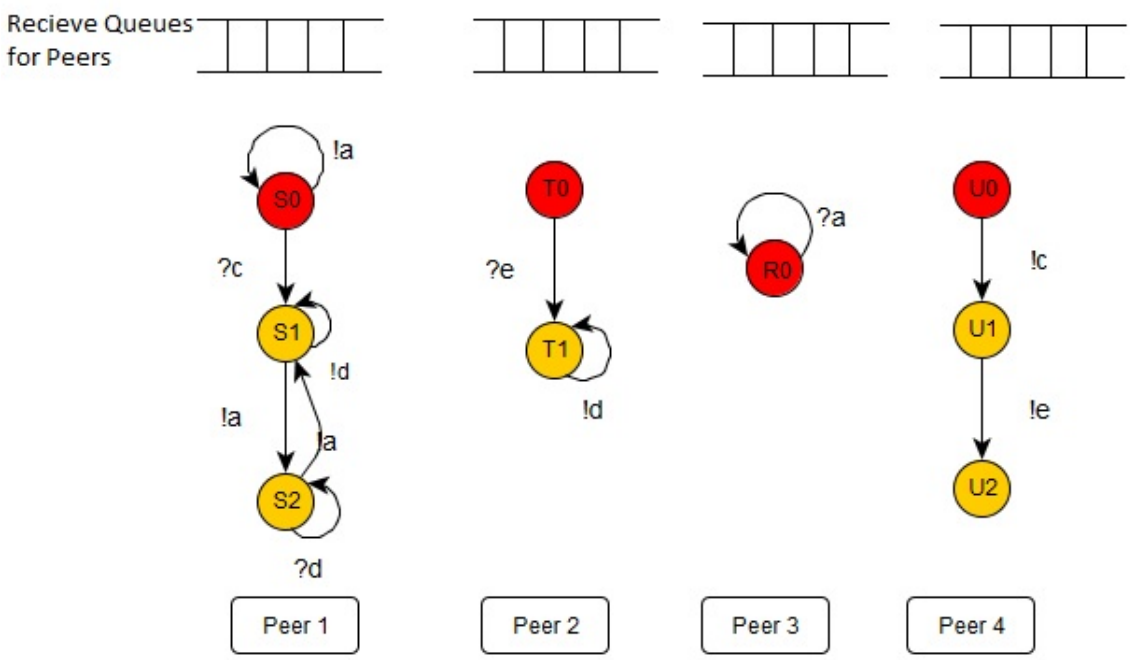

Figure 3.1 Single peers

3. *an empty-transition of the form* (t_1, ϵ, t_2).

   Any transition $(t_1, !a, t_2)$ *is written as* $t_1 \xrightarrow{a} t_2$.

We consider deterministic behavior of the Peers with respect to transitions, $\forall\,(t_1, t_2) : t_1 \xrightarrow{a} t_2 \bigwedge t_1 \xrightarrow{a} t_3 \Rightarrow (t_2 = t_3)$. Any peer behavior can be determinized following standard methods for translation of non-deterministic state machines to a deterministic one.

Example 3.1.1. *Figure 3.1, illustrates the behavior of peers. The initial states are marked in red. It also shows the send and receive transitions possible in the Peers. In the initial states the Queues of all the Peers are empty as shown in the figure. An example of a send action is at state* s_0*, Peer 1 can send a message 'a' due to send transition* $s_0 \xrightarrow{!a} s_0$*. Due to this action the message 'a' gets stored in the buffer of Peer 3. An example of a receive transition is* $s_0 \xrightarrow{?c} s_1$*, if the head of message queue of Peer 1 contains the message 'a' this action can be taken and Peer 1 moves to states* s_1.

Current Configuration		Next Configuration				
Local State	Queue	Local State	Queue	Peer Moving	Transtition	Transition Message
s0,t0,r0,u0	[ ][ ][ ][ ]	s0,t0,r0,u0	[ ][ ][a][ ]	P1	s0----->s0	!a
		s0,t0,r0,u1	[c][ ][ ][ ]	P4	u0----->u1	!c
s0,t0,r0,u0	[ ][ ][a][ ]	s0,t0,r0,u0	[ ][ ][aa][ ]	P1	s0----->s0	!a
		s0,t0,r0,u1	[c][ ][a][ ]	P4	u0----->u1	!c
		s0,t0,r0,u0	[ ][ ][ ][ ]	P3	r0----->r0	?a
s0,t0,r0,u1	[c][ ][ ][ ]	s0,t0,r0,u1	[c][ ][a][ ]	P1	s0----->s0	!a
		s1,t0,r0,u1	[][ ][ ][ ]	P1	s0----->s1	?c
s0,t0,r0,u0	[ ][ ][aa][ ]	s0,t0,r0,u0	[ ][ ][aaa][ ]	P1	s0----->s0	!a
		s0,t0,r0,u0	[ ][ ][a][ ]	P3	r0----->r0	?a
		s0,t0,r0,u1	[c][ ][aa][ ]	P4	u0----->u1	!c
s0,t0,r0,u1	[c][ ][a][ ]	s0,t0,r0,u1	[c][ ][aa][ ]	P1	s0----->s0	!a
		s0,t0,r0,u1	[c][ ][][ ]	P3	r0----->r0	?a
		s0,t0,r0,u1	[][ ][a][ ]	P1	s0----->s1	?c
s1,t0,r0,u1	[ ][ ][ ][ ]	s2,t0,r0,u1	[ ][ ][a][ ]	P1	s1----->s2	!a
		s2,t0,r0,u2	[ ][ ][ ][e]	P4	u1----->u2	!e
s2,t0,r0,u1	[ ][ ][a][ ]	s2,t0,r0,u1	[ ][ ][ ][ ]	P3	r0------>r0	?a
		s1,t0,r0,u1	[ ][ ][aa][ ]	P1	s1------>s2	!a
		s2,t0,r0,u2	[ ][ ][a][e]	P4	u1----->u2	!e

Figure 3.2 Composed system table

A composed system is a composition of Peers. This composition is a result of exchange of messages between the Peers. We formally define the Composed System in Definition 2.

Definition 2. *(Composed System Behavior) A composed system refers to a system over a set of peers* (P_1, P_2, ..., P_n) *where* $P_i = (M_i, T_1, s_{0i}, \delta_i)$ *and* $M_i = M_i^{in} \bigcup M_i^{out}$, *is denoted by a state machine (possibly infinite state)* $I = (M, C, c_0, \triangle)$ *where*

1. $M = \bigcup_i M_i$
2. $C \subseteq Q_1 \times S_1 \times Q_2 \times S_2 \times .... Q_n \times S_n$ *such that* $\forall i \in [1...n]$: $Q_i \subseteq (M_i^i n)^*$.
3. $c_0 \in C$ *such that* $c_0 = (\epsilon, s_{01}, s_{02}, s_{03}, ..s_{0n})$ *and*
4. $\triangle \subseteq C \times M \times C$, *and for* $c = (Q_1, t_1, Q_2, t_2, ...Q_n, t_n)$ *and* $c\prime = (Q\prime_1, t\prime_1, Q\prime_2, t\prime_2, ...Q\prime_n, t\prime_n)$

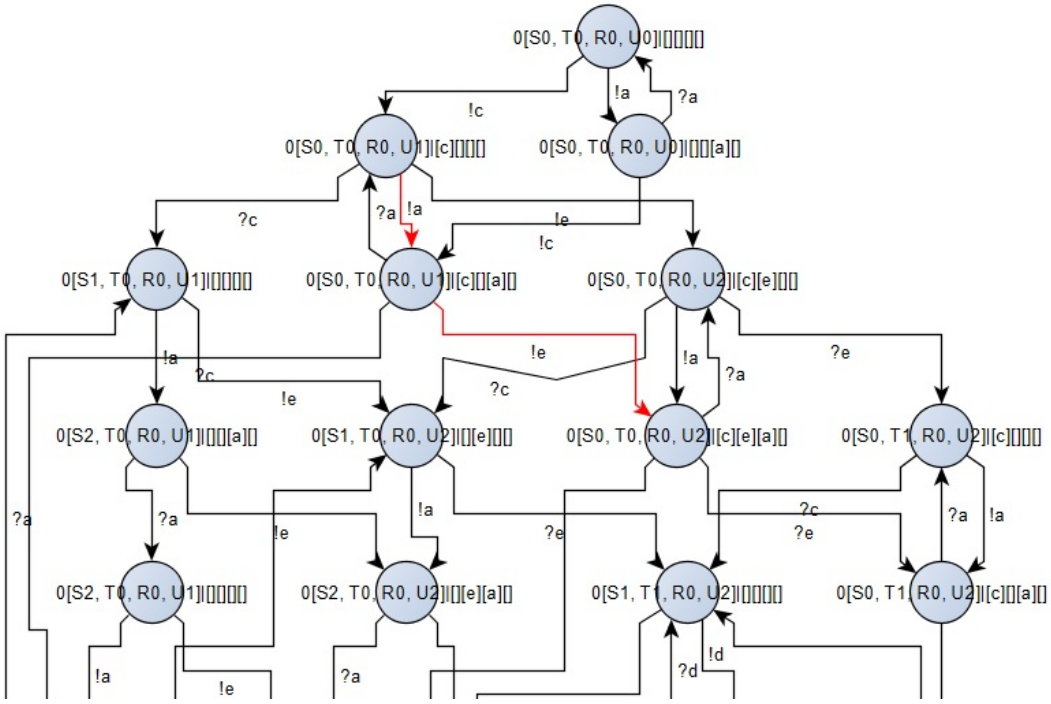

Figure 3.3 Composed System

(a) $c \xrightarrow{!m} c\prime \in \triangle$ *if* $\exists i, j \in [1..n] : m \in M_i^{out}$ *then*

i. $t_i \xrightarrow{!m} t\prime_i \in \delta_i$

ii. $Q\prime_j = Q_j m,$

iii. $\forall k \in [1..n] : k \neq j \Rightarrow Q_k = Q\prime_k$ *and*

iv. $\forall k \in [1..n] : k \neq i \Rightarrow t_k = t\prime_k$

(b) $c \xrightarrow{?m} c\prime \in \triangle$ *if* $\exists i \in [1..n] : m \in M_i$ *then*

i. $t_i \xrightarrow{?m} t\prime_i \in \delta_i$

ii. $Q\prime_j = Q_j m$ *and*

iii. $\forall k \in [1..n] : k \neq i \Rightarrow Q_k = Q\prime_k$ *and*

iv. $\forall k \in [1..n] : k \neq i \Rightarrow t_k = t\prime_k$

(c) $c \xrightarrow{\epsilon} c\prime \in \triangle$ *if* $\exists i \in [1..n]$ *then*

i. $t_i \xrightarrow{\epsilon} t\prime_i \in \delta_i,$

ii. $\forall k \in [1..n] : \Rightarrow Q_k = Q\prime_k$ *and*

iii. $\forall k \in [1..n] : k \neq i \Rightarrow t_k = t\prime_k$

Example 3.1.2. *Figure 3.2 provides a partial representation of the Composed System behavior for the Peers shown in 3.1. The 'Current Configuration' column contains Local States and Queue Status for a particular system configuration. Similarly the 'Next Configuration' is represented by the Local States and Queue Status. The 'Peer Moving' column represents the peer which performs particular transition. 'Transition' and the 'Transition Message' represent the local transition for the Peer which has moved. For example, the initial state shows that the queue for all Peers is empty. From the initial Composed State, two Peers can make the next moves, thus we have two Next Composed States. First move is possible due to a send transition from* s_0 *back to* s_0 *by sending a message 'a'. This message 'a' gets buffered in the queue of Peer 3. Also whenever a message is buffered in a Peers queue and if that Peer has a possible receive transition where it can consume that message, it becomes a possible move for a Composed State. Once a receive transition is performed, the message is consumed from the respective peer. Figure 3.3 gives a graph representation of the partial Composed System which is also described in the form of table in Figure 3.2. Each state of this graph is called as a 'configuration'. Each configuration represents the status of local states of the interacting peers and status of the queue of each Peer. The transitions represent send or receive actions.*

Definition 3. *(k-bounded System). A k-bounded system (denoted by* I_k*) is a system I where the receive queue length for any peer is at most k. The k-bounded system behavior is, therefore, realized by augmenting condition 4(a) in Definition 2 to include the condition* $|Q_j| \leq k$*, where* $|Q_j|$ *denotes the length of the queue for peer j.*

In k-bounded system I_k, unlike I, the send actions get blocked when the receive queue of the peer which is supposed to consume the message, becomes full. In other words,

the peer gets blocked when it cannot consume K number of messages which are already present in its queue. When a new message appears at the Peer it cannot buffer the new message due to limitation on the buffer size of its receive queue. I_k can be represented by a finite state machine as the bound on the queue restricts the number of all possible states to a finite number. The k-bounded for Peers shown in Figure 1 with bound 1 and bound 2 on message queues are shown in Figure 3.4 and 3.5. We can see the configuration marked in red is possible in a 2-bounded system and not in 1-bounded system as the queue size is restricted to 1 for Peer 3 in a 1-bounded system.

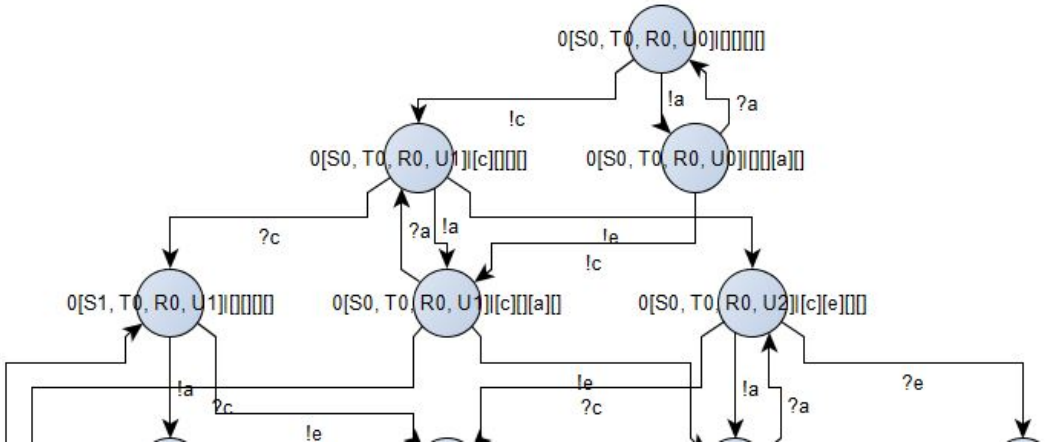

Figure 3.4 1-bounded system

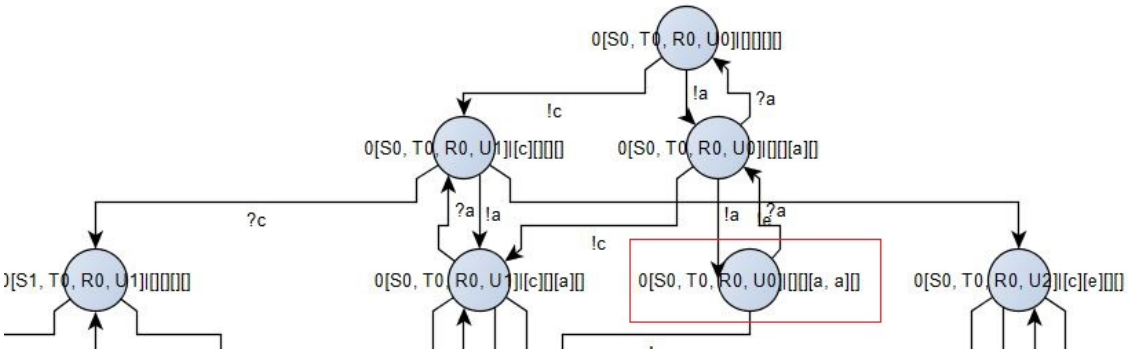

Figure 3.5 2-bounded system

3.2 System Behavior Description as Languages

The interaction behavior of systems is referred to as messages exchanged between the peers. Also note that, whenever a message is sent by a sending peer and consumed by the designated receiver, the peers go through internal state transitions. These transitions are not visible to the other peers. As mentioned before, we say that bounded behavior for I is detected, when interaction behavior of I can be mimicked by I_k. Below we define this interaction in terms of a language of a system and language equivalence (defined below) implies system interaction equivalence.

Definition 4. *(Language of Composed System). Language of a composed system I = (M, C, c_0, F, $\triangle$), denoted by $L(I)$ is defined as a set of sequences of the all the send actions possible from all the paths possible from the initial composed state c_0. (Language Equivalence). The languages of composed systems I and I_k are equivalent if $L(I) = L(I_k)$.*

Example 3.2.1. *In 3.2, some of sequences of sends that are contained in $L(I)$ are:*

1. *sequence of unbounded a's, aaaaa....aaa....*
2. *$a^*ce(a|d)$, where * represents zero or more occurrences and | represents OR.*

Proposition 3.2.1. $L(I_0) = L(I_1) \Longrightarrow \forall k \geq 1, L(I_{k+1}) = L(I_k)$

Proof. This is the proof already proved in Basu and Bultan (2011) and is proved by contradiction. An assumption is made that for some k, $L(I_1) \neq L(I_k)$. For this to be a possibility there should be partial matching send path in $L(I_1)$ and $L(I_k)$. For the rest of this path in $L(I_k)$ there are extra sends which are not possible in $L(I_1)$. One of the reasons could be that a Peer is blocked and cannot consume messages in its queue. Thus the sender which sends the extra send messages in I_k cannot do so in I_1.
Now it is given that $L(I_0) = L(I_1)$, consider the corresponding send path as above in I_0.

We still consider the extra send of messages and say the receiver for it is Peer P. As the I_0 is the synchronous system the messages are also consumed instantly and there is no Peer which is really blocked. Thus, when we talk about the extra send of messages, these sends will always be possible in I_0. As $L(I_0) = L(I_1)$, the extra sends are also possible in I_1. Thus our assumption that there is a path in I_k which contains some sends which are not possible in I_1 is wrong. Hence when a synchronous system can be represented by a 1-Bounded System, the synchronous system can be used to represent the system I.

□

3.3 Temporal Properties

We consider temporal ordering of sends expressed in the logic of Linear Temporal Logic (LTL). LTL can be used to specify temporal orderings along all paths of the system behavior: e.g., presence of a sequence (or substring) of sends, a set of sends occurring infinitely often or finitely many times.

The basic syntactic constructs in the context of our problem are as follows:

$\sigma \rightarrow true \mid \neg\sigma \mid M_{out} \mid \sigma \vee \sigma \mid X\sigma \mid F\sigma \mid G\sigma \mid \sigma \bigcup \sigma$

The above states that propositional constant true is an LTL property. If σ is an LTL property, then its negation is also an LTL property, same is true for disjunction of two LTL properties. Any element from M_{out} is an atomic/basic LTL property in our context. X σ denotes LTL property, which is satisfied in all paths starting from configuration c, if σ is satisfied in all suffix of the path starting from $s\prime$ and $s \longrightarrow s\prime$. F σ is satisfied in the set of paths if and only if σ is satisfied by some suffix of all the paths. G σ is satisfied by the set of paths if and only if σ is satisfied by all suffix of all the paths. σ_1 U σ_2 is satisfied by the set of paths if and only if σ_2 is satisfied in some suffix (say, π) in all paths, and all suffix starting before π satisfy σ_1. For instance, GFa represents the LTL property which is satisfied if along all paths of the system message a, is sent infinitely

often. On the other hand, FG¬a represents the LTL property which is satisfied if along all paths of the system message a is sent finitely many times. For detailed description of LTL properties, refer to Zakharov (2001).

Proposition 3.3.1. *Given two systems I and I', if L(I)=L(I') then for any LTL property σ over the send actions I satisfies σ if and only if I' satisfies σ.*

The above propositions (Propositions 3.2.1, 3.3.1) form the basis for verification of systems with unbounded receive queues. Asynchronous systems whose behavior can be mimicked by corresponding synchronous systems are called as Synchronizable Systems. This is possible when the interaction behavior of Peers in system I does not change even if we move to a Synchronous mode of communication between the Peers. In this thesis we extend the above work. We broaden the scope of verifiability by developing conditions under which systems which are not Synchornizable are also verifiable automatically. In the subsequent sections, we identify the condition under which one can guarantee the existence of k such that the language of a system I with unbounded receive queues is identical to the language of I_k. Once the existence of k is guaranteed, the value of k can be computed by iteratively checking for equality between L(I_i) and L(I_{i+1}) starting from i = 1 (Proposition 3.2.1). Finally, the computed I_k can be used to verify any LTL property over send actions using traditional model checking tools and the verification results will hold for I as well (Proposition 3.3.1).

CHAPTER 4. DETECTION OF BOUNDED BEHAVIOR

In this chapter we will provide the necessary and sufficient conditions for identifying the existence of a bound such that the bounded representation of the Asynchronous system can mimic the interaction behavior of the given Asynchronous system.
In 4.1, we provide the conditions for existence of a bound for an Asynchronous system. Further in 4.2 we provide the Algorithms to check these conditions. If a system satisfies these conditions, i.e., the existence of a bound is guaranteed, we will proceed to develop an algorithmic way to identify the bound 4.3. Finally, we will discuss the algorithms for finding the bound. This chapter is based on the results from Basu and Bultan (2014).

4.1 Condition for Bounded Buffer Behavior

In this section, we present a condition (φ) which when satisfied in any state in the system I guarantees that the interactions between peers participating in the system cannot be represented by interactions in any I_k. We proceed by introducing the concept of unbounded send sequence.

Definition 5. *(Unbounded Send Sequence). Given a system I = (M, T, s_0, δ) over n peers (P_1, ...P_n), consider a configuration $c \in C$. A sequence of sends starting from c is unbounded if the following holds.*

1. *A set of Peers at the configuration $c \in C$ can send any number of messages*

2. *some peer PS over the set of n Peers is the receiver of the messages being sent.*

The first condition in the above definition implies that a set of peers in the system is capable of sending unbounded number of messages. The second condition states that some peer is a receiver of the one of the messages being sent. Note that, the number of states in each peer is finite; therefore, any unbounded send sequence will require that at least one peer in PS moves in a cycle.

Example 4.1.1. *Consider the peers in 3.1, the start state of the system has the local states s_0, t_0, r_0, u_0. The peer P_1 is capable of sending unbounded number of a's to peer P_3. The peers P_2 and P_3 cannot perform any send operations at the start states. The peer P_4 can send finite number of messages; i.e., it is incapable of sending unbounded number of messages. When the system is at a configuration where the local states of the peers is s_1, t_1, r_0, u_2; the peers P_1 and P_2 can send unbounded number of a's and d's, respectively to P_1 and P_3.*

Note that such unbounded sends can make the size of the receiver's queues to grow in an unbounded fashion resulting in infinite state-space of the system behavior.
Our objective is to identify the condition which when satisfied guarantees the existence of k such the $L(I_k) = L(I)$. The intuition for checking when/how a finite queue size system (I_k) can replicate all behaviors of unbounded queue size system (I) is as follows. Every unbounded send sequence will result in repetition of some sequence of messages being sent. The receiver peers must be capable of consuming these messages infinitely often ensuring that the receive queues do not have to hold unbounded number of messages. Note that, due to the finiteness of each peer, consuming messages infinitely often will imply consuming the messages at regular intervals. Furthermore, the receive actions of a peer are not visible to the other peers (as the receiver consume messages from its receive queue). Therefore, it is also necessary that after consuming any subsequence of unbounded sequences of messages, receiver peers should be able to provide the same set of send sequences as they were able to before consuming the messages(ensuring that any

ordering of sends between peers that are possible in I is also possible in I_k for some finite k). Theorem 1 presents necessary and sufficient condition for guaranteeing the existence of $k : L(I_k) = L(I)$.

We proceed by first describing the simulation relation with respect to the send actions. This will be used to ensure to that peers while consuming unbounded sequences of messages will not disable any sequence of send actions.

Definition 6. *(Send-only Simulation).Given a finite state machine (M, T, s_0, δ), $t_1 \in T$ is said to be send-simulated by $t_2 \in T$, denoted by $t_1 \prec! \ t_2$, implies*
$\forall \ t_1\prime : t_1 \xrightarrow{!m} t_1\prime \Rightarrow t_2\prime : \exists \ t_2 \xrightarrow{!m} t_2\prime \wedge t_1\prime \prec! \ t_2\prime$.

Example 4.1.2. *The states s_0, s_1 and s_2 of the peer P_1 in Figure 3.1 are related to each other by the $\prec!$ relation; $s_0 \prec! \ s_1 \prec! \ s_2 \prec! \ s_0$. Each state can perform unbounded number of !a.*

Theorem 4.1.1. *Given $I = (M, T, s_0, \delta)$ over a set of n peers, $\neg[\exists k : L(I_k) = L(I)]$ if and only if there exists a configuration c = (Q_1, s_1, Q_2, s_2, ...Q_n, s_n) reachable from c_0 such that the condition $\varphi = \varphi_1 \vee (\varphi_1 \wedge \varphi_2)$ holds at c, where*

1. *there exists a set of peers which can send unbounded number of messages to some peer P_i;*
2. *P_i cannot move from s_i to any $s_i\prime$ by only consuming all the pending messages in its receive queue;*
3. *If P_i can move from s_i to $s_i\prime$ by only consuming all the pending messages in its receive queue, then $s_i \not\prec! \ s_i\prime$.*

Proof. To prove: $\varphi \Rightarrow \neg\exists k : \mathrm{L}(I_k) = \mathrm{L}(I)$ Let w be the sequence of sends leading to configuration c from c_0. As there are finite number of states in each peer, the unbounded send sequence starting from c results on unbounded repetition of a sequence (say, σ).

Consider first the case where P_i does not consume all messages in its receive queue $Q_i(\varphi_2)$. Therefore, it will require Q_i to be of infinite size to allow storing of messages resulting from unbounded repetition of send sequence σ. In other words, in the given path, the send sequence $w\sigma\sigma\sigma$ will require that Q_i size is not finite. Next consider the case where P_i can consume all messages in its receive queue Q_i to reach $s_i\prime$ from s_i and $s_i \prec! s_i\prime$(condition 3). Let $\sigma\prime$ be the sequence of sends possible from s_i that is not possible from $s_i\prime$.

Consider as before that, w is the sequence that led to c from c_0 and the unbounded send sequence results from the unbounded repetition of σ. Therefore, I can have a sequence $w\sigma\sigma...\sigma\prime...$ where the number of times σ can be repeated depends on the size of P_i queue at the state s_i. In summary, in the above paths, it is necessary for the peer P_i to have infinite receive queue size. Therefore, when φ holds, there does not exist any k such that $\mathrm{L}(I_k) = \mathrm{L}(I)$. □

Proof. To prove: $\neg\varphi \Rightarrow \exists \mathrm{k} : \mathrm{L}(I_k) = \mathrm{L}(I)$. Suppose the first condition is not satisfied i.e. there is no configuration from where peers can send unbounded number of messages to some Peer P_i. In this case, the queue size is finite in all configurations of the system. Thus, there exists a k: $\mathrm{L}(I_k) = \mathrm{L}(I)$. Here queue size will be the maximum number of messages needed to buffer by any receiver in any configuration. Now consider a case where the second condition is not satisfied. For all possible Q_i=$m_{i1}m_{i2}...m_{il}$, there exists a sub path along which all the pending messages in Q_i are consumed. Let the start and end states of this path be s_i and $s_i\prime$. Further, $s_i \not\prec! s_i\prime$. Thus, whenever an unbounded sequence of sends is generated at any configuration, there is some peer which can consume the messages and again is capable of sending the same set of messages which it could before consuming messages from its receive queues. This implies that along all paths of the system, the queues of the peers that may receive unbounded number of messages become empty at regular intervals. As the peer behaviors are represented by finite state machines, the queue size of any peer cannot grow in an unbounded fashion. Therefore,

the send sequences in I can be replicated by send sequence of I_k for some finite value of k. □

4.2 Algorithms for bounded behavior detection

Our objective is to present an algorithm that can automatically verify the condition φ in Theorem 4.1.1 for all possible configurations in I. Two problems need to be addressed to realize such an algorithm:

(a) identifying whether a set of peers in a reachable configuration can generate unbounded number of sends

(b) exploring sufficient (finite) number of configurations in the system

4.2.1 Finding Configurations with Unbounded Send Sequences

We are dealing with finite number of local states for all Peers in a communicating system. Thus even though we work with unbounded buffers we are still dealing with Finite State Machine Representations. For production of unbounded sends it is important that the Peers move in cycles while performing some send actions. For any particular configuration in the composed state, we need to check if that configuration is capable of producing unbounded send sequences. Further we also need to make sure this unbounded send sequence is consumed by some peers in the system. We present an algorithm CYCLE, which takes as parameter a configuration in a system I, and returns $CY = (CY_1, CY_2,...)$. Here CY, represents a set of send cycles generated by the input configuration in system I. CY in effect is responsible to generate unbounded send sequences as defined in Definition 5. Every CY_i represents a send cycle where the start and end states of the cycle are same. Note that this cycle belongs to a single peer.

The basic idea behind the algorithm CYCLE is, given a configuration we want to check if it produces unbounded send sequences. This can be ensured if we traverse along

the system I starting with the given configuration (with respect to local states) and come across the same configuration (with respect to local states) again. However, one restriction added to this traversal is that while moving along the configurations, the local states must belong to the same strongly connected component (SCC). For example, if there is path in I such that, $\{s_0, t_0, r_0\} \xrightarrow{!m} \{s_0\prime, t_0, r_0\}$, then s_0 and $s_0\prime$ must belong to the same SCC (It is also obvious that these two states belong to same Peer). The reason for considering states that belong to a SCC is that only such components tend to generate messages in a cycle, given a finite state system. The algorithm terminates once the above traversal is done for maximum number of possible transitions (called as Transition Threshold). Also we do not allow any messages to be consumed from the queue at the initial configuration. If along this traversal any configurations are revisited and the final queue is larger than the initial one, we say unbounded send messages are generated.

Strongly Connected Components (SCC) A strongly connected component in a FSM for a single Peer is where every state is reachable from every other state in the component. A good property of these SCCs is that they represent cycles, and if there are send transitions along the edges of the SCCs, it leads to an unbounded send sequence. Algorithm CYCLE needs states of peers which belong to the same SCC. We use the Tarjan's strongly connected components algorithm to find the strongly connected components of each Peer in system I. This algorithm takes in the finite state representation of each Peer and returns a partition of the states in the Peer which belong to the same SCC. Each state appears in exactly one SCC. We mark all the states belonging to same SCC for the system given in Figure 3.1 by unique SCC IDs, using the Tarjan's strongly connected components algorithm. For more details on the Tarjan's Algorithm refer to Tarjan (1972).

Example 4.2.1. *Figure 4.1 represents Strongly Connected Components (SCC) for the system shown in figure 3.1. All the states which belong to the same SCC are marked*

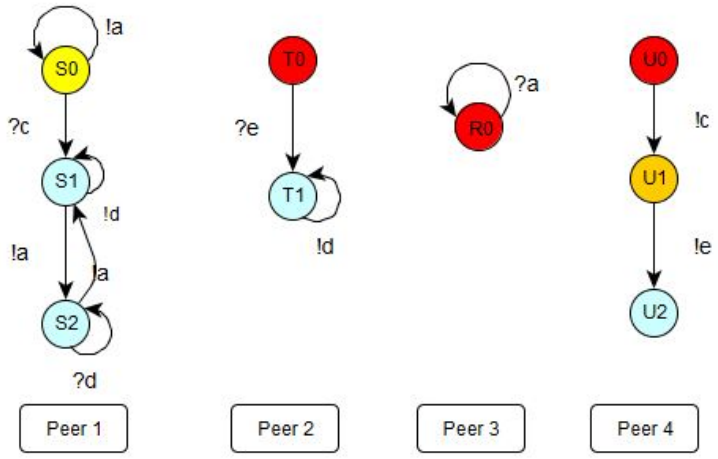

Figure 4.1 Strongly connected components

in the same color. The SCCs clearly represent cycles which could potentially produce unbounded send sequences. For instance, the following SCCs represent cycles which can potentially lead to production of unbounded sequences of message 'a'.

$s_0 \xrightarrow{!a} s_0$

$s_1 \xrightarrow{!a} s_2 \xrightarrow{!a} s_1$

Below we present the Algorithm **CYCLE**.

1. Take input as configuration of a system I, say $\{s_0, t_0, r_0, u_0\}$ and current message queue status at this configuration.

2. For this configuration, perform a depth first traversal along the system I. Also, locally Peers can move only along the states that form a part of a same Strongly Connected Components (SCC). While doing the traversal do not allow any messages to be consumed from the initial queue, but let the Peers move on a corresponding receive action.

3. If the same configuration is visited again and the message queue for any of the Peers has grown , return {Configuration,Unbounded Send Messages}. This indicates the

configuration generates Unbounded Send Messages.

Stop if the number of transitions made is greater the Transition Threshold for the configuration and indicate "No Unbounded Sequence Produced" if the Unbounded Send Messages are not generated at all (i.e No configurations are revisited). Otherwise, continue with the depth first traversal of I, i.e. make a transition along system I, such that $\{s_0, t_0, u_o\ r_0\} \xrightarrow{!m} \{s_0\prime, t_0, u_0, r_0\}$ and s_0 and $s_0\prime$ belong to the same SCC.

4. In the end return all the unbounded messages generated.

Transition Threshold The transition threshold indicates the maximum possible moves that are possible for each Peer in a configuration. If even after taking these maximum possible steps the same local states of a configuration are not visited, then it means some Peer has not moved. Thus if maximum possible moves have exceeded the Transition Threshold, it means an unbounded send cycle is not possible. Suppose a configuration represents n Peers and m_i represents a possible moves for a state(in the SCC to which it belongs) in a configuration for i^{th} Peer. Thus,
Transition Threshold = $\sum m_i$ for i=1...n

Example 4.2.2. *For a configuration, $\{s_2, t_1, r_0, u_0\}$, s_2 can make 4 transitions, t_1 can make one transition and r_0 can make 1 transition.Thus the Transition Threshold for this configuration is 6.*

Example 4.2.3. *Consider a sample configuration $\{s_2, t_1, r_0\ u_0\}$ in the composed system I and run through steps 2 and 3. The transition threshold for this configuration is 6. Thus the exploration goes through six transitions along the composed system and we see the configurations with local states $\{s_2, t_1, r_0\ u_0\}$ and $\{s_1, t_1, r_0\ u_0\}$ are visited again. Thus an unbounded send cycle for message 'a' is detected and it is reported. Also the queue size has increased for Peer 3 as compared to the initial queue status.*

Current Configuration		Next Configuration					
Local State	Queue	Local State	Queue	Peer Moving	Transtition	Transition Message	Transition Count
s2,t1,r0,u0	[][][][]	s1,t1,r0,u0	[][][a][]	P1	s2--->s1	!a	1
s1,t1,r0,u0	[][][a][]	s2,t1,r0,u0	[][][a,a][]	P1	s1--->s2	!a	2
s2,t1,r0,u0	[][][a,a][]	s1,t1,r0,u0	[][][a,a,a][]	P1	s2--->s1	!a	3
s1,t1,r0,u0	[][][a,a,a,][]	s2,t1,r0,u0	[][][a,a,a,a][]	P1	s1--->s2	!a	4
s2,t1,r0,u0	[][][a,a,a,a][]	s1,t1,r0,u0	[][][a,a,a,a][]	P1	s2--->s1	!a	5
s1,t1,r0,u0	[][][a,a,a,a,a][]	s2,t1,r0,u0	[][][a,a,a,a,a][]	P1	s1--->s2	!a	6

Figure 4.2 DFS: unbounded send cycle detection

Theorem 4.2.1. *Algorithm CYCLE computes all unbounded send sequences that are possible due to the cycles involving the local states of a configuration.*

Proof. For each configuration we allow check if the same configuration is visited again. This indicates there is a cycle and if there are any send transitions involved, we know these are Unbounded Sends. The receivers are not allowed to consume messages in the initial queue and check the queue status at the end so that we know if the queue size has increased, to take care that any Unbounded Sends that are generated are not interrupted in any way. Also we take care the single peers move along Strongly Connected States ensuring we consider transitions which would generate potential send cycles. Also this algorithm is guaranteed to terminate as the exploration is performed till the 'Transition Threshold' is reached. This ensures that the exploration stops at some point. Also the 'Transition Threshold' does take care of covering any possible Send cycles which might be generated due to the existing Strongly Connected States in a Peer. □

4.2.2 Algorithm for Exploring I and Verifying φ

In this section, we focus on exploring sufficient number of configurations for I and verify the condition φ.

Below we discuss some notations which will be used while explaining the Algorithm: NOTATIONS: For composed system $I = (M, C, c_0, F, \triangle)$, with n Peers ($P_1$, P_2, ...P_n) and configuration c = (Q_1, s_1, Q_2, s_2,...Q_n, s_n), we use the following notations,

c $\downarrow^{st}$ = (s_1, s_2, ..., s_n) Projection of local states for a configuration.

c $\downarrow^{st}_{P_i} = s_i$ Projection of P_i's local state for configuration

c $\downarrow^{st}_{\bar{P}_i} = (s_1,\ s_2,\ ...s_{i-1},\ s_{i+1},\ s_n)$ Projection of local states of all Peers except P_i for configuration

c $\downarrow^{qu}_{P_i} = Q_i$ Projection of P_i's queue for configuration

Given a sequence of messages l, $M_{in}(l)$ and $M_{out}(l)$ are the sets of input and output messages respectively. Similarly, for a set L of sequences, $M_{in}(L) := \cup_{l \in L} M_{in}(l)$and $M_{out}(L) := \cup_{l \in L} M_{out}(l)$. Finally, given a set of messages M, $R(M)$ is the set of peers can consume at least one message m $\in$ M.

We present an Algorithm **EXPLORE** which performs the depth first exploration of the configurations in the Composed System I from the initial configuration c_0. It carries following two important sets:

1. The set *Visited* of visited configurations projected onto the local states of the peers and (message type, message) which represents a transition from the previous configuration to the current configuration.

2. The set *VObl* of tuples of the form (c $\downarrow^{st}$: c$\prime$ $\downarrow^{st}$) are configurations in I. We call these tuples as 'Obligations'.

In addition to these global sets, in each depth first call the algorithm maintains the local states of the current configuration, the message queue status of all Peers for the current configuration (say c) and (message type, message) which represents a transition from the previous configuration to the current configuration. Below we present the Steps for the algorithm **EXPLORE**:

1. For the current configuration c of the Composed System I, check if it produces any Unbounded Send Cycles using the Algorithm **CYCLE**. The Algorithm **CYCLE** gives the pseduo configurations which generate these Send Cycles.

2. If Unbounded Send Cycles are generated, run the Algorithm **ObligationCheck** (explained in detail later) which returns 'True' if any Obligation is generated and

'False' if no Obligation is generated.

If no Unbounded Send Cycles are generated, mark the current configuration as Visited, i.e update the Set *Visited* with local states and (message type, message) for the current depth first call.

3. If **ObligationCheck** returns 'True', it also returns a set of Obligations(say *newObl*) generated by the Unbounded Send Cycles for the current configuration.

   If $newObl \subseteq VObl$ mark the current configuration as Visited.

   If $newObl \nsubseteq VObl$ then $VObl = VObl \cup newObl$.

   If **ObligationCheck** returns 'False', **EXPLORE** returns 'False'.

4. c and c$\prime$ belong to I, **NextStateGenerator** explained later generates all possible c$\prime$'s for a given c.

   For all c $\longrightarrow$ c$\prime$:

   If c is marked as Visited, visit only unvisited c$\prime$ by calling **EXPLORE** for c$\prime$.

   If c is not marked as Visited, visit all c$\prime$ by calling **EXPLORE** for c$\prime$

   If **EXPLORE** for c$\prime$ returns 'False' then return 'False' and stop the depth first exploration.

Below in Figure 4.3 we present a snapshot of the system I and show how **EXPLORE** works.

Example 4.2.4. *In the figure 4.3, exploration starts with configuration* $\{s_0, t_0, r_0, u_o\}$ *and empty queue [][][][]. This will return true if all its children return 'True'. Let us take an example path given in the 4.3, where the configurations are marked in red and the path is marked in blue. Take into consideration the configuration* $\{s_1, t_0, r_0, u_2\}$ *and queue [][e][][]. The Obligations for this configuration are already computed so it is marked as Visited. Also the next configurations generated from this configuration are already covered in a different path. Thus this configuration returns 'True' to its parent* $\{s_0, t_0, r_0, u_2\}$ *and queue [c][e][][]. All the next configurations for this configuration also return*

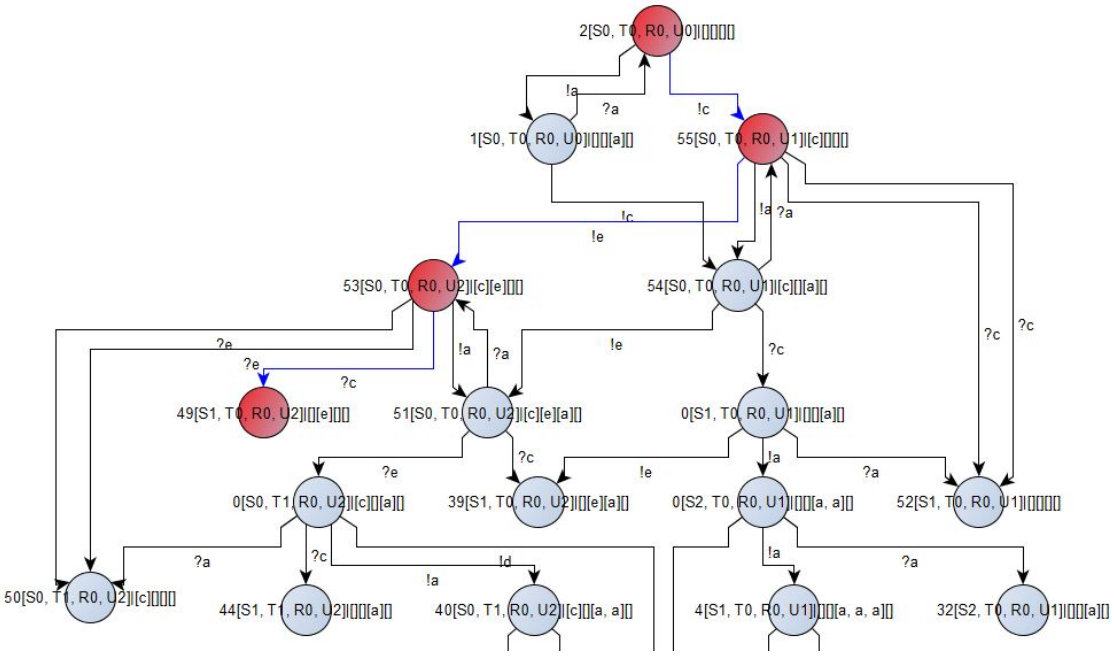

Figure 4.3 Explore I

'True'. Also the local states for this configuration are Visited in the Exploration for the first time and thus it generates new Obligations for the first time. After covering all the next configurations and paths this configuration returns 'True' to its parent $\{s_0, t_0, r_0, u_1\}$ *and queue [c][][][]. Similar to the previous configuration, this configuration generates obligations for the first time when Visited and thus the exploration goes through all the next configurations (irrespective of them being Visited or not). After all the children return 'True', this configuration returns 'True' to its parent* $\{s_0, t_0, r_0, u_o\}$ *and empty queue [][][][]. In the end when all the next configurations of* c_0 *return 'True', the algorithm returns 'True'.*

Step 1 invokes the Algorithm **CYCLE** and helps in verifying the condition φ_1 in φ. According to Step 2, if a particular configuration does not produce Unbounded Send Messages then it is marked as Visited. If there is a system I where no Unbounded Send Cycles are generated, eventually all the configurations in system I form a finite state system. Thus **EXPLORE** returns 'True' indicating φ_1 is satisfied although φ_2 and φ_3

are not satisfied.

In Step 2, input to **ObligationCheck** is current depth first exploration configuration c, current message queue and configurations from **CYCLE** which generate Unbounded Send Messages. **ObligationCheck** performs the following checks:

1. First it finds the Peers which are responsible for consuming unbounded send messages found by **CYCLE**.

2. For each such receiver Peer check if the Peer can move from its current state to some other destination state by consuming all the messages in its receive queue (φ_2).

3. If the messages in the Queue can be consumed by the receiver Peer, the destination state can send-simulate the current state (φ_3). (Algorithm **NotSendSimulate** explained later determines if a state can be send simulated by the other)

Example 4.2.5. *Let us take an example of the configuration,* $\{s_0, t_0, r_0, u_1\}$ *with queue status[c][][][]. This configuration generates unbounded sends of message 'a'. According to step 1 in* ***ObligationCheck****, the Receiver Peer is 'Peer 3' with state* r_0*. The queue for 'Peer 3' is empty and* r_0 *is send-simulated by itself. Thus the first Obligation which is generated is* $\{s_0, t_0, r_0, u_1\}$*:* $\{s_0, t_0, r_0, u_1\}$*. We also notice that this configuration can also generate another unbounded sends of message 'a' which can also be consumed by 'Peer 3'. The obligation generated for this case is* $\{s_0, t_0, r_0, u_1\}$ *: . Let us take an example of the configuration,* $\{s_0, t_1, r_0, u_2\}$ *with queue status [c][][][], This configurations generates unbounded sends of message 'a' and 'd'. According to step 1 in* ***ObligationCheck****, the Receiver Peer is 'Peer 3' with state* r_0 *for message 'a'. The queue for 'Peer 3' is empty and* r_0 *is send-simulated by itself. Thus the first Obligation which is generated is* $\{s_0, t_1, r_0, u_2\}$*:*$\{s_0, t_1, r_0, u_2\}$*.*
According to step 1 in ***ObligationCheck****, the Receiver Peer is 'Peer 1' with state* s_0 *for message 'd'. The queue for 'Peer 1' has message 'c'. After consuming message 'c', state*

s_0 moves to state s_1. We see that s_0 is send simulated by s_1. Thus the other Obligation that is generated is $\{s_0, t_1, r_0, u_2\}$:$\{s_1, t_1, r_0, u_2\}$

Obligations If the above check is successful, **ObligationCheck** returns 'True' and also returns a tuple which contains local states of the peers in the current configuration and the local states of the peers after the receiver consumes all the pending messages. We refer to such a tuple as obligation. An obligation is represented by the tuple (c $\downarrow^{st}$: c$\prime$ $\downarrow^{st}$), at c$\prime$ some peer has the obligation to consume unbounded send sequences that can be possibly sent to it by some other peers from c$\prime$.

Algorithm **NotSendSimulate**: Input to this algorithm are local states i.e. source state(say s) and destination state(say d) belonging to the receiver Peer of an Unbounded Send message(refer to Step 3 of **ObligationCheck**).

This is a recursive algorithm which returns 1 if the input destination state cannot be send simulated by the input source state. Below are the steps for this algorithm:

1. Set simulationFlag = 0
2. for all s $\longrightarrow$ s$\prime$(Consider only Send Transitions)
   Set simulationFlag = 1
   for all d $\longrightarrow$ d$\prime$(Consider only Send Transitions)
   if Send Transitions for s and d match:
   Set simulationFlag = 0
   simulationFlag = simulationFlag OR NotSendSimulate(s$\prime$,d$\prime$)
3. Return simulationFlag

Example 4.2.6. *Let us consider the configuration $\{s_0, t_1, r_0, u_2\}$ with queue status [c][][][]. From **ObligationCheck** we find that we need to check for send simulation relations between states s_0 and s_1. Thus the input to **NotSendSimulate** are states s_0 and s_1. We need to check if s_1 can send simulate s_0. The similationFlag is set to 1. The*

send transition possible from s_0 is, $s_0 \xrightarrow{!a} s_0$. The corresponding send transition for s_1 is $s_1 \xrightarrow{!a} s_2$. The simulationFlag is set to 0.

In the next recursive call for ***NotSendSimulate****, the input states will be s_0 and s_2. Again the send transitions $s_0 \xrightarrow{!a} s_0$ and $s_2 \xrightarrow{!a} s_1$ match. The similationFlag for this is set to 0. Since the call for s_0 and s_1 is already done there is no need to call* ***NotSend-Simulate*** *again. Thus the call for* ***NotSendSimulate*** *with the input states will be s_0 and s_2 and it returns 0. Thus NotSendSimulate(s_0,s_2) returns 0.*

After returning from the above call, the simulationFlag according to Step 2 in ***NotSend-Simulate*** *becomes 0.*

simulationFlag = simulationFlag OR NotSendSimulate(s_0,s_2). NotSendSimulate(s_0, s_2) is 0 and simulationFlag is 0. Thus eventually for NotSendSimulate(s_0,s_1) becomes 0 and NotSendSimulate(s_0, s_1) returns 0. Since we are checking for the negation of the condition of simulation, so when NotSendSimulate returns 0 it means there was no proof found where s_0 and s_1 are not send simulated.

Correctness of Algorithm EXPLORE

In Step 3 of **EXPLORE**, if **ObligationCheck** returns 'False', then the algorithm returns 'False'. This indicates that even though φ_1 is satisfied, either φ_2 or φ_3 are not. In other words, even though unbounded messages are generated there is no Peer which can consume these messages periodically. There is no cycle detected in the receiver's behavior. Thus no bounded behavior is detected.

We realize **EXPLORE** only returns 'True', if a particular configuration (that does not satisfy φ) is revisited (with respect to its local states and incoming (message type, message)) and if **EXPLORE** for the next configurations return 'True'.

Intuitively, **EXPLORE** does a depth first exploration along the configurations in the system I. Further a configuration is marked as *Visited* when it basically does not satisfy the condition φ. As per the logic in Step 3 of **EXPLORE**, a configuration is marked *Visited* when it does not generate any new Obligation. When no new Obligation is gen-

erated, it implies the local states for the particular configuration are visited again and all the unbounded send sequences generated by these local states can be consumed in some other configuration of system I.

EXPLORE only returns 'True', if all the configurations that lie along the path from the *Visited* configuration also do not satisfy φ. Traversing along all the paths from the *Visited* configurations is important as there could be a configuration which satisfies φ and thus a bounded representation for the system I is not possible.

In Step 4 of **EXPLORE**, if the current configuration is marked as *Visited* exploration only considers next unvisited configurations. If the current configuration is not marked as *Visited* exploration continues along all the next configurations. This means if the current configuration is marked as *Visited* there are no new Obligations which can be generated and we can skip the exploration along next *Visited* configurations. Also **EXPLORE** is guaranteed to be terminated, this is because it checks if the configurations are visited again with respect to the local states of the Peers. Since the Peers are finite state representations the algorithm is guaranteed to terminate in finite number of configurations.

Below are the Steps for the Algorithm **NextStateGenerator**: Input to this Algorithm is the current configuration c in system I, current message Queue and the state transitions in the Single Peer System shown in 1.1. This algorithm computes the next state configurations and updates the Message Queues for all Peers.

1. A configuration c is a tuple which represents projection of local states of all Peers in the system. Get all possible Transitions for the local states of all Peers in the configuration c.

2. For each Transition run the following logic:

   c$\prime$ = c

   There could be transitions of two types:

(a) If transition is Output Transition: s $\xrightarrow{!m}$ s′

Get the Peer to which s belongs, say P_{src}. For P_{src}, update the state 's' with 's′' in configuration 'c′'.

Find the receiver Peer for message m, say P_{rec}. For P_{rec}, update the message queue with message m.

return 'c′' and updated message queue.

(b) If transition is Input Transition: s $\xrightarrow{?m}$ s′

Get the Peer to which s belongs, say P_{src}. If message queue for P_{src} has message m available for consumption:

For P_{src}, update the state 's' with 's′' in configuration 'c′'.

For P_{src}, update the message queue by removing the message m.

return 'c′' and updated message queue.

Example 4.2.7. *Let us take an example configuration* ***NextStateGenerator*** $\{s_1, t_1, r_0, u_1\}$ *with message queue [][][][][a]. According to Step 1 in* ***NextStateGenerator****, get the transitions for local states in the input configuration. The table below in Figure 4.4 shows how the next configurations in the System I are generated according to Step 2 of* ***NextStateGenerator****.*

Current Configuration		Next Configuration				
Local State	Queue	Local State	Queue	Peer Moving	Transtition	Transition Message
s1,t1,r0,u1	[ ][ ][ ][a]	s2,t1,r0,u1	[ ][ ][a,a][ ]	P1	s1----->s2	!a
		s1,t1,r0,u1	[d][ ][ ][a]	P2	t1----->t1	!d
		s1,t1,r0,u2	[ ][e][ ][a]	P4	u1----->u2	!e

Figure 4.4 Next state generator

4.3 Algorithm to find bound

Proposition 3.2.1 discusses synchronizability i.e whether interaction behavior of an asynchronous system can be mimicked by a system when it interacts in a synchronous fashion. In this chapter, we discuss whether a bounded behavior can replicate the behavior of the asynchronous system even when the system is not synchronizable. In other words, we provide a more general result than synchronizability. Proceeding further, we provide the necessary proofs and further discuss the Algorithms to find the bound.

Proposition 4.3.1. $\forall k : L(I_k) = L(I) \iff L(I_{k+1}) = L(I_k)$

Proof. Given the systems X and Y, we say that L(X) $\subseteq$ L(Y), if $\forall$w $\in$ L(X), either w is sub-sequence of some $w\prime \in$ L(Y) or w $\in$ L(Y).
We know that L(I_i) $\subseteq$ L(I_{i+1}). This is because the queue size of L(I_{i+1}) is always one greater than that of L(I_i). Thus send actions possible in L(I_{i+1}) are always greater than that of L(I_i). Same holds for L(I) and thus L(I_i) $\subseteq$ L(I).
Based on $\subseteq$ relation, $\forall$k : L(I_k)= L(I) implies that L(I_{k+1})= L(I_k), as L(I_i) $\subseteq$ L(I_{i+1}) $\subseteq$ L(I). □

Proof. The reverse direction $\forall$k : L(I_k) = L(I_{k+1}) $\Rightarrow$ L(I_k) = L(I) can be directly proved by first proving $\forall$k : L(I_k) = L(I_{k+1}) $\Rightarrow$ $\forall$ i $\geq$ k : L(I_i) = L(I_{i+1}). This means that increasing receive queue size beyond k does not have any impact on the behavior in terms of send sequence.
Let us prove this by contradiction. Given L(I_k) = L(I_{k+1}), assume that there exists n $\geq$ k+1 such that L(I_{k+1}) $\neq$ L(I_n), i.e., L(I_{k+1}) $\subset$ L(I_n). For this to be a possibility there should be partial matching send path in $L(I_{k+1})$ and $L(I_n)$. For the rest of this path in $L(I_n)$ there are extra sends which are not possible in $L(I_{k+1})$. One of the reasons could be that a Peer is blocked and cannot consume messages in its queue. Thus the sender which sends the extra send messages in I_n cannot do so in I_{k+1}.
Now it is given that L(I_k) = L(I_{k+1}), consider the corresponding send path as above in

I_k. We still consider the extra send of messages and say the receiver for it is Peer P. Since we say that L(I_k) = L(I_{k+1}), it means that all the sends possible in I_{k+1} are also possible in I_k. We find that any send path which is possible in I_{k+1} does not require a message queue size of more than k. Thus for n $\geq$ k+1, there will be no send path which would require a message queue size of more than k. It is proved that there are no extra sends which are possible in In and not in I_{k+1}. The basic assumption that a path with extra sends is possible in In is proved to be wrong. □

Thus after the **EXPLORE** returns 'True', we know the interactions between peers participating in the system with unbounded buffers can be represented by a k-bounded system.

Below we represent an Algorithm **BOUNDFINDER** which finds the exact bound K such that L(I) = L(I_k). We can identify K by checking the equality of L(I_i) = L(I_{i+1}) starting from i = 1. Language equivalence can be determined by making sure all the configurations in I_i are send simulated by I_{i+1}. Checking for Language equivalence is sufficient to find K because language equivalence guarantees that all the send traces which are possible in L(I_{k+1}) are also possible in L(I_k).

1. Set i = 1

2. Say c_i is the initial configuration for the system I_i and c_{i+1} is the initial configuration for the system I_{i+1}.
   Set KConfig = c_i and IncKConfig = c_{i+1}

3. Call K-BoundNotSendSimulate(KConfig, IncKConfig)

4. If it returns 'False', return bound K=i
   If it returns 'True':
   i = i+1
   Go To Step 2.

The basic logic for **K-BoundNotSendSimulate** is similar to that of **NotSendSimulate**. Only difference is, in **NotSendSimulate** we take into consideration states belonging to finite State Systems representing Single Peers. In **K-BoundNotSendSimulate** we consider the configurations belonging to Composed Systems I with buffers having bounds i and i+1 where i lies between 1 and K. **K-BoundNotSendSimulate** takes in input as the initial configurations of the systems I_i and I_{i+1}. It does a depth first traversal along the configurations of I_i and I_{i+1} and proceeds along both the systems if matching transitions are found. Thus I_i will send-simulate I_{i+1} if **K-BoundNotSendSimulate** returns 'False'. If along any path if a configuration in I_i cannot send-simulate a configuration in I_{i+1} i.e. a send action possible in I_{i+1} is not possible in I_i then **K-BoundNotSendSimulate** return 'False'.

Below are the steps for the Algorithm **K-BoundNotSendSimulate**:

1. Take input configurations IncKConfig (configuration for I_i) and KConfig (configuration for I_{i+1})
   Set simulationFlag = 'False'

2. for all IncKConfig $\longrightarrow$ IncKConfig$\prime$(Consider only Send Transitions)
   Set simulationFlag = 'True'
   for all KConfig $\longrightarrow$ KConfig$\prime$(Consider only Send Transitions)
   if Send Transitions for IncKConfig and KConfig match:
   Set simulationFlag = 'False'
   simulationFlag = simulationFlag OR NotSendSimulate(IncKConfig$\prime$, KConfig$\prime$)

3. Return simulationFlag

In step 2, **K-BoundNotSendSimulate** uses **NextStateGenerator** to generate next configurations in system I with buffers having bound i and i+1. Only difference is that in Step 2b of **NextStateGenerator**, skip the input transitions of type s $\xrightarrow{?m}$ s$\prime$ till we reach an output transition.

For example, if a path for s exists such that,
s $\xrightarrow{?m}$ s_1 $\xrightarrow{?m}$ s_2 $\xrightarrow{!m}$ s_3.
Condense this path to s $\xrightarrow{!m}$ s_3 and consider this as a transition in Step 2b.

In the previous Algorithm to find the bound we have used a simple approach of performing a depth first exploration of a K-Bounded and a K+1-Bounded System and checking if the states in both the systems are send-simulated or not. We can take an advantage of the fact that a K+1-Bounded System is always a larger system as compared to a K-Bounded System. Thus instead of going through all the configurations of both systems and checking for send-simulation, we can just work on the the larger system i.e K+1 Bounded System and check if all the paths covered in this system are also represented in the K-Bounded System. If this check is successful, we can say we found the correct bound i.e K.

Below we provide the steps for this approach in Algorithm **NewBOUNDFINDER**:

1. Set i=2

2. Say c_i is the Initial Configuration for the system I_i.

3. Call **BFSDeterminize**(c_{i+1})
   If it returns 'True' return Bound = i-1
   Otherwise, set i = i+1, call Step 2.

Algorithm **BFSDeterminize** generates the next configurations of the system I_i on the fly using the **NextStateGenerator** and does a breadth first exploration of I_i and then performs check to see if I_i can replicate the send behavior of I_{i+1}.

1. EnQueue c_i in a Queue Q

2. If Queue Q is not Empty
   Configuration c = DeQueue Q
   Mark c as Visited.

3. Generate all Next Configurations from c using **NextStateGenerator**. Let us say c' is one such configuration.

   In the process of generation of next configurations, we also mark configurations which can represented in both I_i and I_{i-1} with an identifier of 'i-1' and those which can only be represented in I_i with an identifier of 'i'.

4. Call **Merge** with configuration c as input

   If **Merge** returns 'False' return 'False' for all c $\longrightarrow$ c'(Consider only Send Transitions)

   If c' is not Visited, Enqueue c' in Queue Q

5. If all the state configurations of I_i are Visited and all of them are marked with an identifier for 'i-1' return 'True'

Below we present the Algorithm **Merge** where the identifiers assigned to next configurations of input configuration c which are marked in Step 3 of **BFSDeterminize** are updated. Let us say from a configuration c, we have transitions to next configurations c' of the form c $\xrightarrow{!m}$ c'. We would say two transitions match if the message type(which is always send in this case as we only consider send transitions) and message name match.

1. For all matching transitions from c, get identifiers of all next state configurations (say c' s) of these transitions.

2. If any of the identifiers is 'i-1', Update the identifiers of the all the c's to 'i-1'.

3. If such an Update is not possible, return 'False' Otherwise return 'True'.

In summary we have discussed conditions for identifying the existence of a bound such that the bounded representation of the Asynchronous system can mimic the interaction behavior of the given Asynchronous system. We discussed the Algorithms to check these conditions. If a system satisfies these conditions, i.e., the existence of a bound is

guaranteed, we discussed two different algorithms to do the same. These methods have made even the systems which are not Synchronizable verifiable. It has thus broadened the scope of verification to a larger subset of Asynchronous systems.

CHAPTER 5. TOOL DESCRIPTION

5.1 Tool Overview

Our tool is implemented in Java Programming Language. It is built using NetBeans 7.2.1 and JDK 7. The overall tool architecture is depicted in 5.1 which shows its main components. The red arrows represent the data flow while the blue arrows represent the control flow between the components of the Tool.

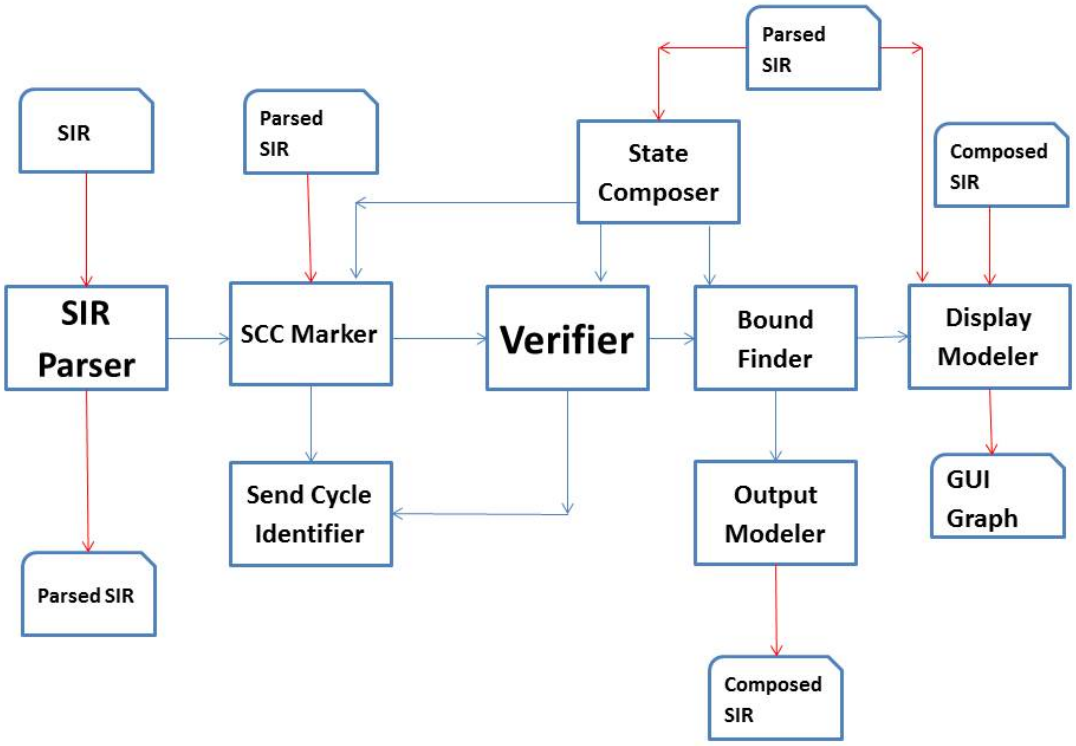

Figure 5.1 Tool architecture showing main components

1. In the first step, the tool takes as input, the system interaction representation (SIR) which is an input file that represents interaction behavior of peers in an asynchronous distributed system. After running through the SIR Parser, the input file is converted to an internal system interaction representation, referred to as Parsed SIR.

2. The Strongly Connected Component Marker runs through the Parsed SIR to mark all the states of individual peers which form a Strongly Connected Component.

3. Next, the parsed SIR goes through the Verifier, which performs composition formed out of local states of each peer, using the State Composer. The Verifier basically runs through the depth first exploration of the composed states and performs the checks necessary to capture bounded behavior in an unbounded buffered system. The Send Cycle Identifier finds all possible send cycles generated by a particular composed state.

4. If the Verifier captures a bounded behavior in the system, the Bound Finder goes ahead to find the bound on size of queue necessary to represent the unbounded asynchronous system.

5. After finding the message Queue bound, the Output modeler stores the composed bounded buffered system representation i.e the system I_k in the form of a Composed SIR. The Display Modeler handles the display of single peers as shown in Figure1.1 and of the composed system I with bound K(If the asynchronous system of Peers has bounded behavior) to the user interface.

5.2 Tool Components

1. ***System Interaction Representation(SIR)*** We need a specification language to represent an asynchronous distributed system which is shown in Figure1.1. We

use XML as the input specification language. Below we show a SIR for the system shown in Figure1.1 and explain the components of the SIR:

Listing 5.1 SinglePeers.xml

```
<root>
<process id="P1">
    <state id="S0">
        <transition id="T1" message="a" messagetype="output" target="S0"></transition>
        <transition id="T2" message="c" messagetype="input" target="S1"></transition>
    </state>
    <state id="S1">
        <transition id="T1" message="d" messagetype="input" target="S1"></transition>
        <transition id="T2" message="a" messagetype="output" target="S2"></transition>
    </state>
    <state id="S2">
        <transition id="T1" message="d" messagetype="input" target="S2"></transition>
        <transition id="T2" message="a" messagetype="output" target="S1"></transition>
    </state>
</process>
<process id="P2">
    <state id="T0">
        <transition id="T1" message="e" messagetype="input" target="T1"></transition>
    </state>
    <state id="T1">
        <transition id="T1" message="d" messagetype="output" target="T1"></transition>
    </state>
</process>
</root>
```

A single peer is represented using the tag 'Process'. Individual states are represented using the tag 'State' and transitions are represented using the tag 'Tran-

sition'. Every peer contains a peer Id and different states that make the peer. Every state contains a state Id which is unique to its respective peer. States also contain transitions. Each transition contains a transition Id which is unique to its respective state. Transitions hold information like source state, destination state, transition message and transition message type. We know every peer moves from one state to another state based on different actions it performs. In our case, these actions could be consuming messages in the buffer (input actions) or sending messages to the other peer (output actions). Thus the states are represented by 'State' in the SIR. The input and output actions are represented by 'Transitions' in the SIR. 5.2 shows the mapping between a Peer in 3.1 and its representation in the SIR XML input. In this example, 'Peer1' can go through states 'S0', 'S1', 'S2' and the state mapping is marked with red arrows. The transition mappings are marked with blue arrows. An example transition for state 'S0' is where it can output a message 'a' and come back to 'S0'. This is represented by transition id 'T1' in state 'S0'. Thus source and destination for this transition is 'S0' and message type for message 'a' is 'input.

Figure 5.2 SIR mapping

2. ***Parsed SIR*** A parsed SIR represents a static Hashmap object which stores the

input SIR and is used by different modules of the tool. Individual elements of the input XML like peers, states and transitions are stored in corresponding Java objects. Transition objects are stored in a State object. State objects are stored in Peer objects. All the peer objects are stored in a Hashmap which can then be used globally.

3. ***SIR Parser*** The main task of the SIR Parser is to parse the System Interaction representation (SIR) into a Parsed SIR format.

4. ***SCC(Strongly connected Component) Marker*** The SCC Marker takes the Parsed SIR as input and runs the Tarjans Strongly Connected Component algorithm on individual peers which are represented as finite state machines. As the name suggests, the SCC Marker assigns a component Id to each state of each peer. Thus, the states which form a part of the same strongly connected component have the same component Id. We can say input to this component is the system shown in figure 3.1 and the output is as shown in the figure 4.1. SCC Marker is just an Implementation of the Tarjan's Strongly Connected Component Algorithm for peers of asynchronous systems.

5. ***Send Cycle Identifier*** Input to this module is a configuration of a composed system I shown in figure 3.2. It finds all possible Send Cycles that are generated by the composed state configuration. These Send Cycles are cycle objects which represent Receivers of the unbounded send messages and the unbounded send messages generated. The Send Cycle Identifier is an implementation the Algorithm **CYCLE**.

6. ***State Composer*** This is an important module and is used by many other modules of the tool. The main task of this module is to generate next composed state configuration, given the current composed state configuration in composed system I. It requires the information like current local states and current message queue

of the all the peers for a given composed state configuration. To generate the next composed state, it takes in information from the Parsed SIR. The other modules dependent on this module are the Verifier, Send Cycle Identifier and the Bound Finder which need to perform a depth first exploration of the system I. State Composer is an implementation of the Algorithm **NextStateGenerator**.

7. ***Verifier*** This module is an implementation of the Algorithm **EXPLORE**. As stated in the algorithm, it performs depth first exploration of the configurations in the composed system I. The next state configurations that could appear in the depth first path of the current configuration are generated on the fly using the State Composer. The Verifier also maintains the global receive queues for individual peers which help in generating the next possible configuration. This module gets the send cycle objects using the Send Cycle Identifier module. Using these send cycle objects this module run the Algorithm **ObligationCheck** mentioned before. Based on the Algorithm **EXPLORE**, this module returns a Boolean value i.e 'True' if a bounded behavior is detected in the interaction between Peers in System I or 'False' otherwise.

8. ***Bound Finder*** This module is an implementation of the Algorithm **BOUND-FINDER**. This module is only called if the Verifier sends a signal to this module indicating that a bounded representation of the given system I exists i.e. it returns 'True'. The input to this module is the Parsed SIR. It returns the bound on message queue which can be used to represent the bounded composed system i.e the K-Bounded System. It also generates the Parsed SIR representation of the K-Bounded System. Since it involves depth first exploration of the system I, it takes help from the State Composer to generate the composed state configurations on the fly.

9. ***Output Modeler*** The output modeler takes as input the Parsed SIR represen-

tation of the K-Bounded Composed System and generates a Composed SIR. The Parsed SIR representation of the the K-Bounded Composed System is in fact representation system I_k and is a finite state system because of the bound on queue size of each Peer.

Listing 5.2 KBounded.xml

```
1 <process id="Composed Peer">-
2
3 <state id="[S0, T0, R0, U0]|[][][][][]">
4 <transition id="2" target="[S0, T0, R0, U1]|[c][][][][]" messagetype="output"
      message="c"/>
5 <transition id="1" target="[S0, T0, R0, U0]|[][][a][]" messagetype="output"
      message="a"/>
6 </state>
7
8 -<state id="[S0, T0, R0, U0]|[][][a][]">
9 <transition id="2" target="[S0, T0, R0, U1]|[c][][a][]" messagetype="output"
      message="c"/>
10 <transition id="1" target="[S0, T0, R0, U0]|[][][][][]" messagetype="input"
      message="a"/>
11 </state>
12
13 -<state id="[S0, T0, R0, U1]|[c][][a][]"><transition id="3" target="[S0, T0,
      R0, U2]|[c][e][a][]" messagetype="output" message="e"/>
14 <transition id="2" target="[S0, T0, R0, U1]|[c][][][][]" messagetype="input"
      message="a"/>
15 <transition id="1" target="[S1, T0, R0, U1]|[][][a][]" messagetype="input"
      message="c"/>
16 </state>
17
18 -<state id="[S1, T0, R0, U1]|[][][a][]">
19 <transition id="2" target="[S1, T0, R0, U2]|[][e][a][]" messagetype="output"
      message="e"/>
20 <transition id="1" target="[S1, T0, R0, U1]|[][][][][]" messagetype="input"
      message="a"/>
21 </state>
```

10. ***Composed SIR*** The Composed SIR is a simple SIR representation of a K-

Bounded Composed System. The composed SIR follow the same semantics as that of the SIR discussed before. The only difference is that in a Composed SIR every State is represented by a unique id i.e. a State Id which is a combination of the local states of the configuration in system I and the message queue status of the Peers at that particular configuration. Below we give a partial XML representation of the composed SIR.

11. ***Display Modeler*** The display modeler handles the generation of GUI graph objects for the Parsed SIR and Composed SIR. These graph objects are then used to draw to the GUI panel. Apart from the graph objects, the display modeler is also responsible for generation of other GUI panel objects like the logger, file menu and the task bar.

12. ***GUI graph*** The GUI graph is a graphical interface object required to represent the Parsed SIR and Composed SIR. The GUI graph objects related to the Composed SIR are updated on the fly as and when new states are added to the Composed SIR.

The main advantage of building the tool with the above approach is that it is properly modularized. The tool implements different components of the proposed Algorithms in Chapter 4 and each component is a separate module. The modules communicate via Static Objects. An important use of this approach is that due modularization, debugging becomes very easy. Another important feature is easy plug and play of modules. For example, the input language for the tool is SIR. However if the input language is to be replaced one can easily replace the SIR Parser by the required Parser and one can still run the decidability of verification. Also we have two implementations of the BoundFinder Module. Both the BoundFinder approaches can be tested and evaluated against different case studies.

CHAPTER 6. CASE STUDIES

We evaluate our proposed approach to verify asynchronous systems using a number of case studies explained later in this chapter. As seen earlier, we have important components to our algorithm like marking Strongly Connected Components, finding Unbounded Send Cycles, composing next States in system I, and finding the bound that represents the interaction behavior of the asynchronous system.

6.1 Case Study 1

This model represents an asynchronous system with four Peers communicating via message buffers. The diagram below represents this system. The **EXPLORE** in section 4.2 explores around 48 configurations. The table 6.1 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

Table 6.1 Case study 1: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.13	23.50	0.28	2.03
OBLIGATION CHECK	0.016	0.34	0.06	0.09
NEXT STATE COMPOSER	0.004	0.25	0.013	0.0194
TOTAL TIME	0.35	188.97	0.87	32.09

We also show below in table 6.2 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

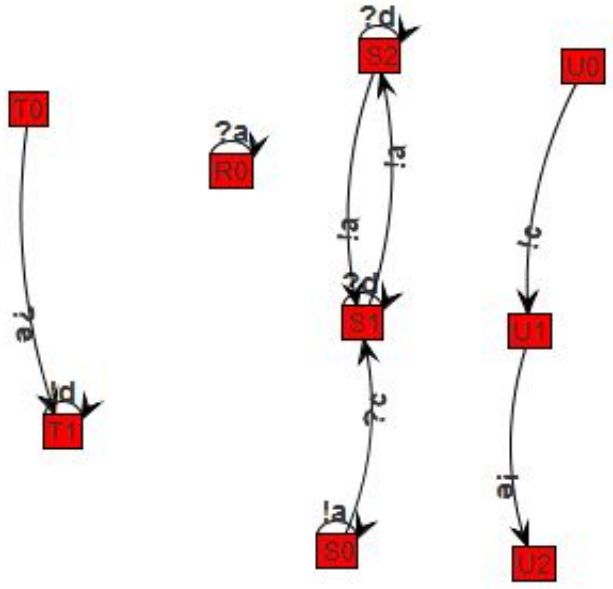

Figure 6.1 Peers: case study 1

Table 6.2 Case study 1: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	3	4	3
OBLIGATIONS	2	2	2
NEXT STATES	2	4	3

In this case, the algorithm **EXPLORE** explores all configurations with all possible local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent configuration is visited again.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 1-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 1-Bounded System is 40. This also proves that **EXPLORE** has to visit many more number of states in I even though a 1-Bounded System has less number of configurations. The table below 6.3 shows the Computation time for

methods **K-BoundNotSendSimulate** and **NewBOUNDFINDER**.

Table 6.3 Case study 1: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	246
NewBOUNDFINDER	649

6.2 Case Study 2

This model represents an asynchronous system with three Peers communicating via message buffers. The diagram below represents this system. The **EXPLORE** in section 4.2 explores around 43 configurations. The table 6.4 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

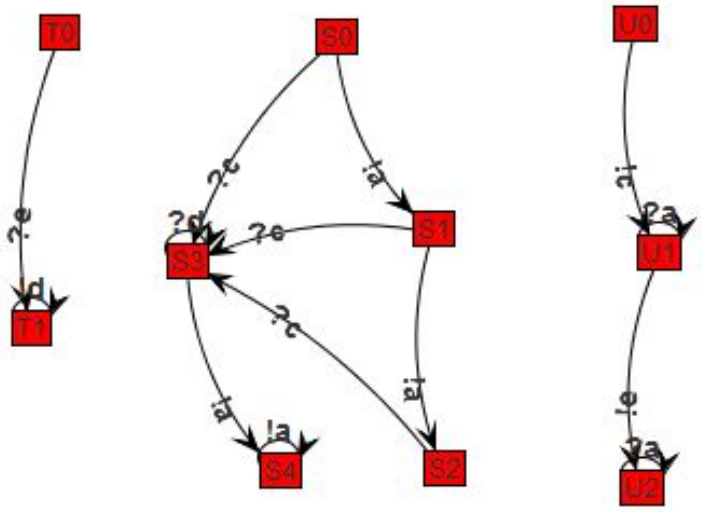

Figure 6.2 Peers: case study 2

Table 6.4 Case study 2: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.045	22.85	0.134	1.256
OBLIGATION CHECK	0.016	7.212	0.062	0.238
NEXT STATE COMPOSER	0.004	0.02	0.015	0.014
TOTAL TIME	0.19	109.78	1.18	12.01

We also show below in table 6.5 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

Table 6.5 Case study 2: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	0	3	2
OBLIGATIONS	0	2	2
NEXT STATES	1	5	3

In this case, the algorithm **EXPLORE** explores all configurations with all possible local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent configuration is visited again.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 2-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 2-Bounded System is 66. In this case, **EXPLORE** visits 43 configurations while the 2-Bounded System has more number of configurations i.e 66. The table below 6.6 shows the Computation time for methods **K-BoundNotSendSimulate** and **NewBOUNDFINDER**.

Table 6.6 Case study 2: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	489
NewBOUNDFINDER	1695

6.3 Case Study 3: Reservation Session Protocol

This model represents an asynchronous system with two Peers communicating via message buffers. It represents the 'Reservation Session Protocol'. (TODO more explaination) The diagram below represents this system. The **EXPLORE** in section 4.2 explores around 43 configurations. The table 6.7 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

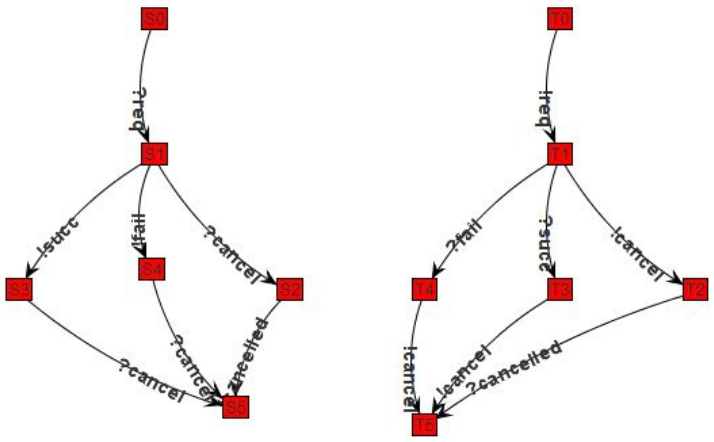

Figure 6.3 Peers: case study 3

Table 6.7 Case study 3: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.108	1.47	0.116	0.199
OBLIGATION CHECK	0	0	0	0
NEXT STATE COMPOSER 0	0.149	0.039	0.054	
TOTAL TIME	0.440	34.84	1.280	5.45

We also show below in table 6.8 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

In this case, the algorithm **EXPLORE** explores all configurations with all possible

Table 6.8 Case study 3: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	0	0	0
OBLIGATIONS	0	0	0
NEXT STATES	0	3	1

local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent configuration is visited again. In this case, there are no Unbounded Sends detected at any configuration and thus due to this, the first condition in φ is not satisfied in any of the configurations. Thus this asynchronous system can be represented by some bounded system.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 1-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 1-Bounded System is 12. In this case, **EXPLORE** visits 19 configurations while the 1-Bounded System has 12 state configurations. The table below 6.9 shows the Computation time for methods **K-BoundNotSendSimulate** and **NewBOUNDFINDER**. In this case study, the **NewBOUNDFINDER** is more efficient as compared to **K-BoundNotSendSimulate**. In this case, the Algorithm **MERGE** there are no matching transitions as per Step 1, as all the transitions from a composed state configuration are unique. Thus there is no need of any extra work of updating the bound identifiers.

Table 6.9 Case study 3: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	8
NewBOUNDFINDER	17

6.4 Case Study 4: TCP Contract

This model represents an asynchronous system with two Peers communicating via message buffers. It represents the 'Transmission Control Protocol(TCP) Contract'. The two peers here follow a protocol to establish a connection before any data transmission can start. The diagram below represents this system.

The **EXPLORE** in section 4.2 explores around 9 configurations. The table 6.10 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

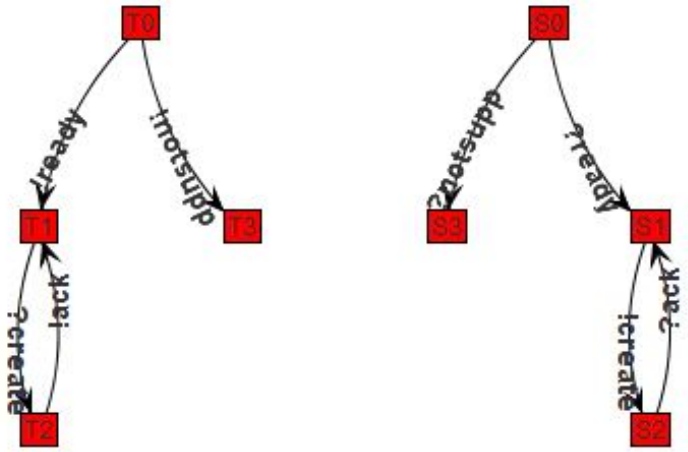

Figure 6.4 Peers: case study 4

Table 6.10 Case study 4: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.0984	6.38	0.138	0.924
OBLIGATION CHECK	0	0	0	0
NEXT STATE COMPOSER	0	0.074	0.036	0.039
TOTAL TIME	0.444	17.290	1.65	3.665

We also show below in table 6.11 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

Table 6.11 Case study 4: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	0	0	0
OBLIGATIONS	0	0	0
NEXT STATES	0	2	1

In this case, the algorithm **EXPLORE** explores all configurations with all possible local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent configuration is visited again. In this case, there are no Unbounded Sends detected at any configuration and thus due to this, the first condition in φ is not satisfied in any of the configurations. Thus this asynchronous system can be represented by some bounded system.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 1-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 1-Bounded System is 5. In this case, **EXPLORE** visits 9 configurations while the 1-Bounded System has 5 state configurations. The table below 6.12 shows the Computation time for methods **K-BoundNotSendSimulate** and **New-BOUNDFINDER**. In this case study, the **NewBOUNDFINDER** is more efficient as compared to **K-BoundNotSendSimulate**. In this case, the Algorithm **MERGE** there are no matching transitions as per Step 1, as all the transitions from a composed state configuration are unique. Thus there is no need of any extra work of updating the bound identifiers.

Table 6.12 Case study 4: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	7
NewBOUNDFINDER	4

6.5 Case Study 5: Key Board Contract - Singularity Channel

This model represents an asynchronous system with two Peers communicating via message buffers. It represents the 'Key Board Contract'. This case study analyzes channel contract specifications in Microsoft Research's Singularity operating system. A channel contract is a state machine that specifies the allowable interactions between a server and a client through an asynchronous communication channel. This contract begins in the Start state and transitions into Ready state when the server sends a Success message. Once in the Ready state, the client may send either the Get message or the Poll message. If the Get message is sent, the channel contract transitions to the Waiting state. If the Poll message is sent, the contract transitions to an implicit state. From either of these states, the server may send either the Ack message, which will transition the contract back to the Ready state.

The **EXPLORE** in section 4.2 explores around 12 configurations. The table 6.13 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

Table 6.13 Case study 5: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.151	0.597	0.208	0.247
OBLIGATION CHECK	0	0	0	0
NEXT STATE COMPOSER 0.05	0.160	0.065	0.076	
TOTAL TIME	0.73	12.47	4.49	5.464

We also show below in table 6.14 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

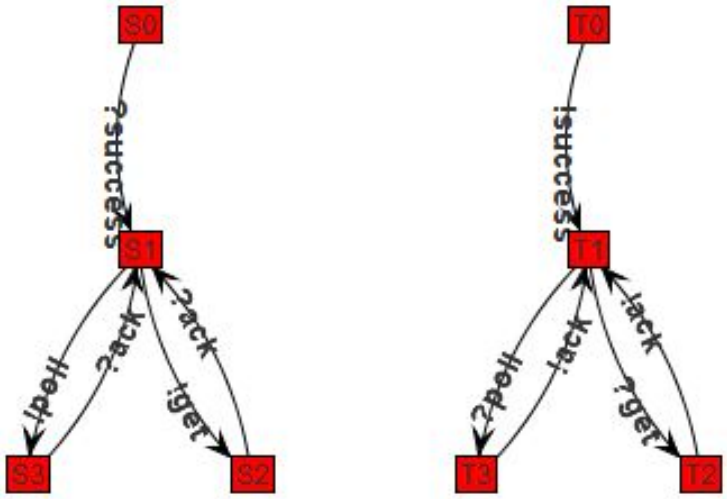

Figure 6.5 Peers: case study 5

Table 6.14 Case study 5: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	0	0	0
OBLIGATIONS	0	0	0
NEXT STATES	1	2	1

In this case, the algorithm **EXPLORE** explores all configurations with all possible local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent configuration is visited again. In this case, there are no Unbounded Sends detected at any configuration and thus due to this, the first condition in φ is not satisfied in any of the configurations. Thus this asynchronous system can be represented by some bounded system. The Unbounded Sends which are possible are already such that each Peer has to consume a message in its queue to send a message to another Peer and these actions are performed in a cycle. Thus, though messages are sent in a cycle, care is taken that they are consumed in that cycle and the queue for the Peers in the system are emptied

infinitely often.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 1-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 1-Bounded System is 8. In this case, **EXPLORE** visits 12 configurations while the 1-Bounded System has 8 state configurations. The table below 6.20 shows the Computation time for methods **K-BoundNotSendSimulate** and **NewBOUNDFINDER**. In this case study, the **NewBOUNDFINDER** is almost same as that of **K-BoundNotSendSimulate**.

Table 6.15 Case study 5: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	10
NewBOUNDFINDER	11

6.6 Case Study 6: Stock Broker Protocol (E-Service)

This model represents an asynchronous system with three Peers communicating via message buffers. It represents the 'Stock Broker Protocol'.

In each round of message exchange, Online Stock Broker i.e. Peer1 sends a list of Raw Data to Research Department i.e. Peer2 for further analysis, where for each Raw Data, one Data is generated and sent to Investor i.e.Peer3. Message classes End, Start, and Complete are intended to synchronize the three peers. Finally Investor acknowledges Online Stock Broker with Ack so that a new round of data processing can start.

The **EXPLORE** in section 4.2 explores around 3 configurations. The table 6.16 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

We also show below in table 6.17 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

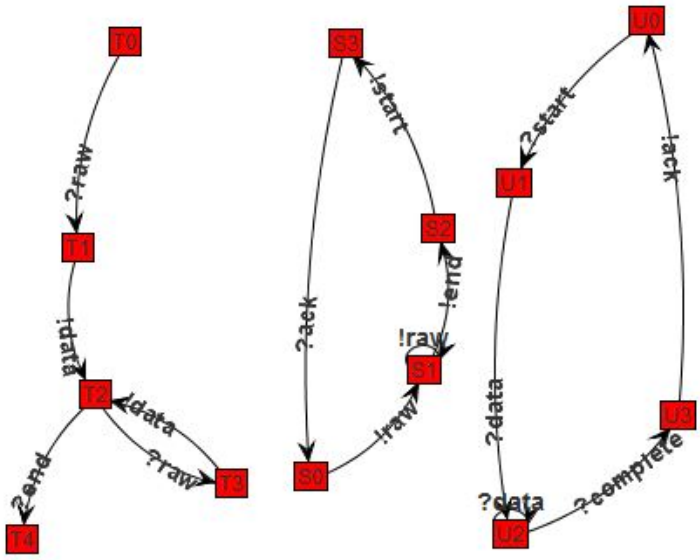

Figure 6.6 Peers: case study 6

Table 6.16 Case study 6:Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.388	1.750	0.406	0.848
OBLIGATION CHECK	0	0.191	0.168	0.119
NEXT STATE COMPOSER	0	0.03	0.025	0.02
TOTAL TIME	0	10.84	6.59	5.813

In this case, the algorithm **EXPLORE** explores the configurations in system I, it checks if the condition φ is satisfied. During this exploration a configuration with local states $\{s_1, t_0, u_0\}$ is reached where t_0 cannot consume all the messages in its queue and thus no Obligation is generated and φ is satisfied. Thus **EXPLORE** returns 'False'.

6.7 Case Study 7: TPM Contract Protocol

This model represents an asynchronous system with two Peers communicating via message buffers. It represents the 'TPM Contract'.

The **EXPLORE** in section 4.2 explores around 17 configurations. The table 6.18 below provides the minimum, maximum, median and average computation time taken

Table 6.17 Case study 6:Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	2	2	2
OBLIGATIONS	0	2	2
NEXT STATES	0	1	1

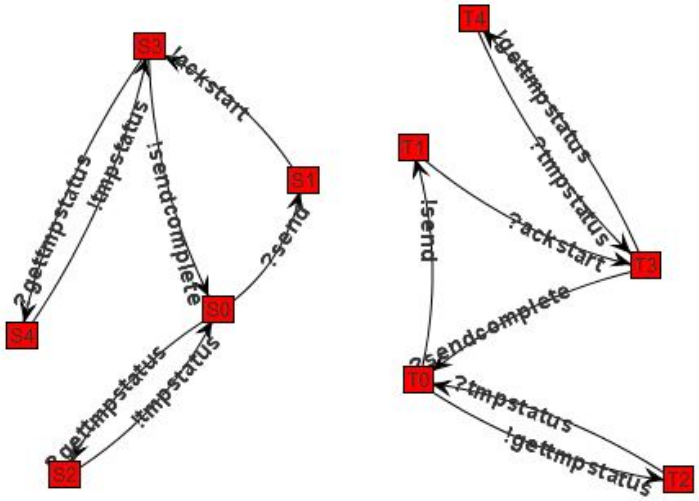

Figure 6.7 Peers: case study 7

by different methods per configuration.

Table 6.18 Case study 7: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.13	0.44	0.22	0.240
OBLIGATION CHECK	0.13	0.44	0.22	0.240
NEXT STATE COMPOSER	0.02	0.10	0.049	0.05
TOTAL TIME	0.47	60.16	6.16	23.34

We also show below in table 6.19 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

In this case, the algorithm **EXPLORE** explores all configurations with all possible local states with varying queue configurations. Overall it checks if the condition φ is satisfied along all the configurations and their next configurations until a parent config-

Table 6.19 Case study 7: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	0	0	0
OBLIGATIONS	0	0	0
NEXT STATES	0	2	1

uration is visited again. In this case, there are no Unbounded Sends detected at any configuration and thus due to this, the first condition in φ is not satisfied in any of the configurations. Thus this asynchronous system can be represented by some bounded system. The Unbounded Sends which are possible are already such that each Peer has to consume a message in its queue to send a message to another Peer and these actions are performed in a cycle. Thus, though messages are sent in a cycle, care is taken that they are consumed in that cycle and the queue for the Peers in the system are emptied infinitely often.

EXPLORE returns 'True' and thus **BOUNDFINDER** finds that 1-Bounded System can mimic the interactions for the Asynchronous System shown in Figure. The number of configurations in a 1-Bounded System is 12. In this case, **EXPLORE** visits 19 configurations while the 1-Bounded System has 12 state configurations. The table below 6.20 shows the Computation time for methods **K-BoundNotSendSimulate** and **NewBOUNDFINDER**.

Table 6.20 Case study 7: BOUNDFINDER in nanoseconds

METHOD	COMPUTATION TIME
K-BoundNotSendSimulate	15
NewBOUNDFINDER	40

6.8 Case Study 8: Alternating Bit Protocol

This model represents an asynchronous system with 4 Peers communicating via message buffers. It represents the ' Alternating Bit Protocol'.

Alternating bit protocol (ABP) is a simple network protocol operating at the data link layer that retransmits lost or corrupted messages. When the Sender sends a message, it resends it continuously, with the same sequence number, until it receives an acknowledgment from Receiver that contains the same sequence number. When that happens, Sender complements (flips) the sequence number and starts transmitting the next message. When Receiver receives a message that is not corrupted and has sequence number 0, it starts sending ACK0(a0) and keeps doing so until it receives a valid message with number 1. Then it starts sending ACK1(a1), etc.

In the model which we formulate for this case study, two Peers in the system represent the Sender and Receiver. The other two Peers represent Communication Channels used by the Sender and Receiver Channels respectively. This model formulates the interaction behavior such that the communication channel does have a scope of losing messages.

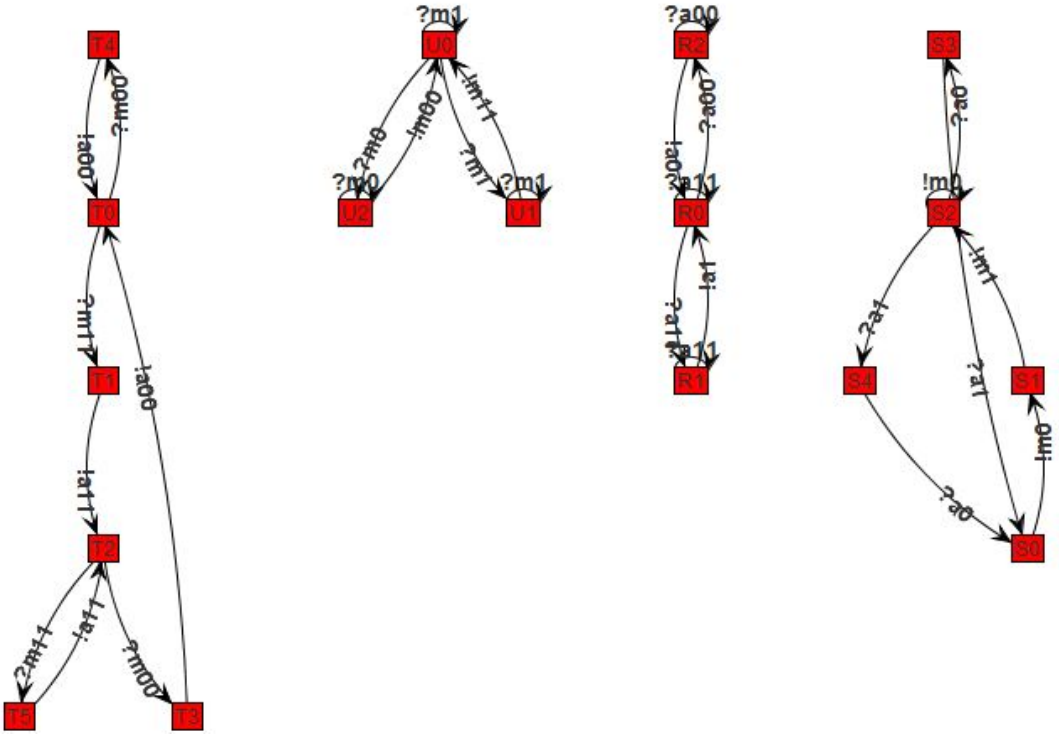

Figure 6.8 Peers: case study 8

The **EXPLORE** in section 4.2 explores around 68 configurations. The table 6.21 below provides the minimum, maximum, median and average computation time taken by different methods per configuration.

Table 6.21 Case study 8: Computation time in milliseconds

METHOD	MIN	MAX	MEDIAN	AVERAGE
SEND CYCLES	0.30	14.41	0.60	1.34
OBLIGATION CHECK	0	0.25	0.11	0.12
NEXT STATE COMPOSER	0 0.23	0.072	0.075	
TOTAL TIME	0	198.71	102.89	102.70

We also show below in table 6.22 the minimum, maximum and median values for cycles, obligations and next state Configurations generated during the depth first exploration of each configuration of system I.

Table 6.22 Case study 8: Results

OBJECTS	MIN	MAX	MEDIAN
CYCLES	3	4	3
OBLIGATIONS	0	1	0
NEXT STATES	0	5	2

In this case, the algorithm **EXPLORE** explores the configurations in system I, it checks if the condition φ is satisfied. During this exploration a configuration with local states $\{s_1, t_0, u_2\}$ is reached where u_2 cannot consume all the messages in its queue and thus no Obligation is generated and φ is satisfied. Thus **EXPLORE** returns 'False'.

6.9 SUMMARY

The table shown in below diagrams represents the summary of our results:

If we consider the Average time **EXPLORE** takes to return a decision where bounded behavior is detected or not, we can see it is proportional the number of the configurations in System I. If **EXPLORE** returns 'True', it has to cover all the configurations which contain all the local states of each Peer for sure. For each configuration it has to cover

Model	Total Peers	Total States	Total Transitions	Number of SCCS	Send Cycles	Obligations	Next States	Explored States	Bounded Graph
Demo Model1	4	9	9	8	3	2	3	65	1 & 40
Demo Model2	3	10	12	10	2	2	3	63	2 & 66
Reservation Protocol	2	12	14	12	0	0	1	19	1&12
TCP Contract	2	8	8	6	0	0	1	9	1&5
Singularity Channel	2	6	14	4	0	0	1	13	1 & 8
Stock Broker Service	2	8	11	6	2	2	1	3	No Bound
TPM Contract	2	10	14	2	0	0	1	21	1 & 12
ABP Protocol	4	16	33	4	3	0	2	7	No Bound

Figure 6.9 Summary 1

all the child configurations till a cycle in the Send and Receive behavior is detected along the path from the particular configuration.

However if the bounded behavior is absent for a System I, the Algorithm terminates at a configuration where the **ObligationCheck** fails. From the above results, there is a good chance that such a configuration is detected earlier and the **EXPLORE** returns 'False' earlier.

This explains that if there is a bounded behavior present **EXPLORE** would take more time to return as compared to when this behavior is absent.

In case of Send Cycle Detection, it involves a special depth first exploration of some configurations in System I. In this case the time for exploration is dependent on the number of strongly connected components involved in each Send Cycle Detection. In case of Send Cycle detection as opposed to **EXPLORE**, if a send cycle is present it would terminate earlier as compared to when send cycle is not present.

Let us now consider **K-BoundNotSendSimulate** and **NewBOUNDFINDER**

Model	Total Peers	Total States	Total Transitions	Number of SCCS	Send Cycles	Obligations	Next States	Explored States	Bounded Graph
Demo Model1	4	9	9	8	2.03	0.09	0.01	32.09	246
Demo Model2	3	10	12	10	1.25	0.23	0.01	12.01	489
Reservation Protocol	2	12	14	12	0.19	0.19	0.05	5.45	8
TCP Contract	2	8	8	6	0.92	0	0.03	3.66	7
Singularity Channel	2	6	14	4	0.24	0	0.07	5.46	10
Stock Broker Service	2	8	11	6	0.84	0.11	0.02	5.81	No Bound
TPM Contract	2	10	14	2	0.24	0.24	0.05	23.34	15
ABP Protocol	4	16	33	4	1.34	0.12	0.07	102.7	No Bound

Figure 6.10 Summary 2

which are two different approaches to find the K-Bounded System which represents the System I. **K-BoundNotSendSimulate** does a depth first exploration of the K-Bounded System while **NewBOUNDFINDER** does a breadth first exploration. Overall it is found that **K-BoundNotSendSimulate** outperforms **NewBOUNDFINDER**.

Another interesting insight from the results is that whenever a bounded representation for a System is detected, we can see the resulting K-Bounded System has fewer configurations as compared to the number of configurations **EXPLORE** explores to find if a bounded representation does exist.

CHAPTER 7. SUMMARY

7.1 Contributions

Our work focuses on verification of asynchronous systems with unbounded buffers and as discussed earlier this is a difficult problem to solve, as such systems lead to infinite state spaces. Previous work has been to consider subclasses of these systems like Synchronizable systems or Half-Duplex systems and other Examples mentioned in Chapter 2. Our work is built upon the results of Synchronizablility Fu et al. (2005). We have identified a more relaxed class of asynchronous systems for which verification now becomes decidable. The main idea is deciding if a bounded representation for system I exists. We check if a set of Senders can send any messages in a cycle. Further, we check if a corresponding cycle exists in the Receiver's behavior as well. Thus the messages are consumed in a cyclic fashion and they would not require a buffer of more than a specific bound 'K'.

In this thesis, we discuss algorithms for our proposed approach. Moreover once it is found that a system I can be mimicked by some K-Bounded System, we run the Bound Finder logic. This involves checking if two K-Bounded Systems are send simulated or not. We provide two different approaches to the problem of Send-Simulation. One which follows a depth first exploration and the other which follows a breadth first exploration. The new approach for Send-Simulation takes an advantage of the fact that for any i, $L(I_i) \subseteq L(I_{i+1})$. While the depth first logic runs through configurations of two systems, the breadth first logic runs through a single I_{i+1} system.

We have built a tool which is based on our proposed approach, thus proving that our approach is realizable in practice. Different modules of the tool perform separate functions and communicate with each other via static objects. This modularization makes future extensions to our approach easy. For example, the tool can be further extended to include additional parsers to accept additional specification languages other than XML. It can also be coupled with standard Model Checkers like SPIN to enable Model Checking of asynchronous systems for which verification is decidable. The output graphs are stored in standard 'Graph Modelling Language' (GML). This helps in visualization of the graphs using standard External Graph Visualizers like 'yEd', 'Prefuse' etc.

We also perform a case study on protocols like Alternating Bit Protocol, TCPContract Protocol, TPMContract Protocol, KeyBoardContract Protocol from Singularity OS and Stock Broker Protocol, Reservation Session Protocol from Web Services. We perform the run time analysis of different modules of our tool for these case studies and gain some interesting insights on the performance of the modules with respect to interaction behavior that each case study presents.

7.2 Future Work

Once we find that a System I can be represented using a K-Bounded System (I_K), we know that verification for I is decidable. Further work in this respect that remains to be done is to augment Model Checkers like SPIN to Model Check the K-Bounded System (I_K). Thus verifying I_K with respect to just the Send Language can help in verifying properties for I.

In the formalism we have used to define Asynchronous systems, we assume that it is a message based interaction between peers. In this message based interaction, we assume that every message has a unique sender and receiver. However, there could be cases where a message with same message name could be sent by multiple senders or

received by multiple receivers. If we consider a case of a message having multiple senders and unique receiver, we can still represent this interaction in form of a configuration in a composed system. If there is a case which involves multiple senders and multiple receivers non deterministic behavior of the senders (as in which Peer would act a receiver Peer for a particular message)comes into picture. This non deterministic behavior still can be represented in form of possible configurations in the composed system. All these are our insights on what are the possibilities if the formalism changes as described above. We still need to work in this area to provide proofs and check if any changes in implementation are needed.

This work currently focuses only on Verifying system properties with respect to send actions. However, there could be scenarios where Verification of properties with respect to specific set of messages is required. For example, one might need to check if a particular set of messages are sent after a certain message is sent. Thus we know that the focus is only on particular branch of the graph representing I. Thus the conditions to determine if this behavior is represented using a finite state machine will be much more relaxed. Further work needs to be done in this direction.

With respect to the Tool, more work needs to be done to improve Output Visualizations. Since our Output Graphs representing the K-Bounded System contain large number of configurations good visualization is important. The new Visualizer should make possible easy analysis and exploration of the Output Graphs. Another work that needs to be done is to check for Scalability of the Tool using randomly generated graphs. The challenging work here is to perform Choreography to generate random Communicating Finite State Systems and which map to the sub-class we target to verify.

We discussed the new approach for solving the problem of Send-Simulation, when send simulation needs to be checked for states which belong to two systems (representing the same I) and which differ in the bound of message buffer size through which the Peers communicate. The advantage here is we perform analysis on just one system I instead

of analyzing two separate systems. This method works well with distributed systems whose composed behavior is deterministic with respect to send-actions, however when non-determinism comes into picture we need to first determinise the actions performed at each composed state before applying our method.

BIBLIOGRAPHY

Armstrong, J. (2002). Getting erlang to talk to the outside world.

Banavar, G., Ch, T., Strom, R., and Sturman, D. (1999). A case for message oriented middleware. In *In Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18. Springer-Verlag.

Basu, S. and Bultan, T. (2011). Choreography conformance via synchronizability. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 795–804, New York, NY, USA. ACM.

Basu, S. and Bultan, T. (2014). Automatic verification of interactions in asynchronous systems with unbounded buffers. In *29th IEEE/ACM International Conference on Automated Software Engineering*.

Brand, D. and Zafiropulo, P. (1983). On communicating finite-state machines. *J. ACM*, 30(2):323–342.

Cecea, G. and Finkel, A. (2005). Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2).

Coulouris, G. F. and Dollimore, J. (1988). *Distributed Systems: Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J. R., and Levi, S. (2006). Language support for fast and reliable message-based communication in

singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*.

Fu, X., Bultan, T., and Su, J. (2004). Analysis of interacting bpel web services. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 621–630, New York, NY, USA. ACM.

Fu, X., Bultan, T., and Su, J. (2005). Synchronizability of conversations among web services. *IEEE Trans. Softw. Eng.*, 31(12):1042–1055.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty asynchronous session types. *SIGPLAN Not.*, 43(1):273–284.

Manohar, R. and Martin, A. J. (1998). Slack elasticity in concurrent computing. In *Proceedings of the Mathematics of Program Construction*, MPC '98, pages 272–285, London, UK. Springer-Verlag.

Siegel, S. (2005). Efficient verification of halting properties for mpi programs with wildcard receives. In Cousot, R., editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 413–429. Springer Berlin Heidelberg.

Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM Journal on Computing*.

Vakkalanka, S., Vo, A., Gopalakrishnan, G., and Kirby, R. M. (2010). Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs. In *Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 152–159, Berlin, Heidelberg. Springer-Verlag.

Yoshida, N. and Vasconcelos, V. T. (2007). Language primitives and type discipline for structured communication-based programming revisited: Two systems for higher-order session communication. *Electron. Notes Theor. Comput. Sci.*, 171(4):73–93.

Zakharov, V. A. (2001). Book review: "model checking" by e. clarke, o. grumberg and d. a. peled. 11.